Adaptive Model-Based Policy Translation for Usage Control

Guided Research report
Department of Informatics
Technische Universität München

Ciprian Lucaci
Examiner: Prof. Dr. Alexander Pretschner
Supervisor: M.Tech. Prachi Kumari

Abstract. Usage Control is concerned with what happens to data once access to it has been granted. In the existing frameworks, usage control policies are specified and translated based on a domain model. In real environments, domain models evolve over time. The challenge is to update the policies so that they conform to the new model. In this work we present an adaptive model-based policy translation. Two domain models are merged in an automated manner and the usage control policies are updated without the need of an user interaction. This solution includes every step from a policy specification, to translation with implementation details and runtime adaptation when a system changes.

1 Introduction

Usage control is a generalization of access control that specifies and enforces what must or must not happen to data once access to it has been granted. Some example policies include "delete data after 30 days", "don't distribute my data", "don't copy data without permission". Policies are supposed to be specified by end-users and translated using technical details provided by a more sophisticated user, whom we call the power-user [4]. A policy consists of a set of data (e.g. picture, email, document), a set of actions (e.g. print, copy, distribute, save) and constraints (e.g. maximum times, do not, after some time). These policies, specified by an end user in terms of abstract data and actions, are called the specification-level policies [5].

Specification level policies describe what must and must not happen throughout the execution of a system. For enforcement, they are translated into implementation level policies with concrete representations of the abstract data and actions achieved through model based refinement. Implementation level policies refine the specification level policies by stating how they will actually be enforced at each layer of abstraction.

In the existing work on policy translation, the refinements of data and action are defined at the level of the domain. The domain meta-model [5], shown in Figure 1, consists of three layers: the platform-independent layer (PIM), that is to be

defined by the end-user; the platform-specific layer (PSM), that is to be defined by the so called power-user; and the implementation-specific layer, also to be defined by the power-user, that reflects the technicalities of the system on which the policies are to be deployed. The domain meta-model, distinguishes among user-intelligible high-level actions on data like "copy photo" at the platform-independent (PIM) layer, refined as implementation independent technical representations (called transformers) like "take screenshot" at the platform-specific (PSM) layer, and the specific implementations of these transformers like "getImage()" function in the X11 windowing system at the implementation-specific (ISM) layer [4]. In this context, **refinement of an element means providing technical details or implementation details from a lower level of abstraction**.

One basic assumption in previous work was that the domain structure is static. This means that it was assumed that all components of the domain and their relationships with one another at various levels of abstraction are known and specified beforehand. However this assumption is too restrictive in order to reflect the real world. For example, domains can evolve over time by adding, modifying or removing software services, logical systems or physical hosts. If we consider an online social network (OSN) as an example, one can immediately observe that new components or features can be introduced or modified, e.g. outsourcing a payment service, installing a new type of browser on the client, third-party applications hooking into the network, and so on. The client's machine accessing the social network might also change in its technical configuration.

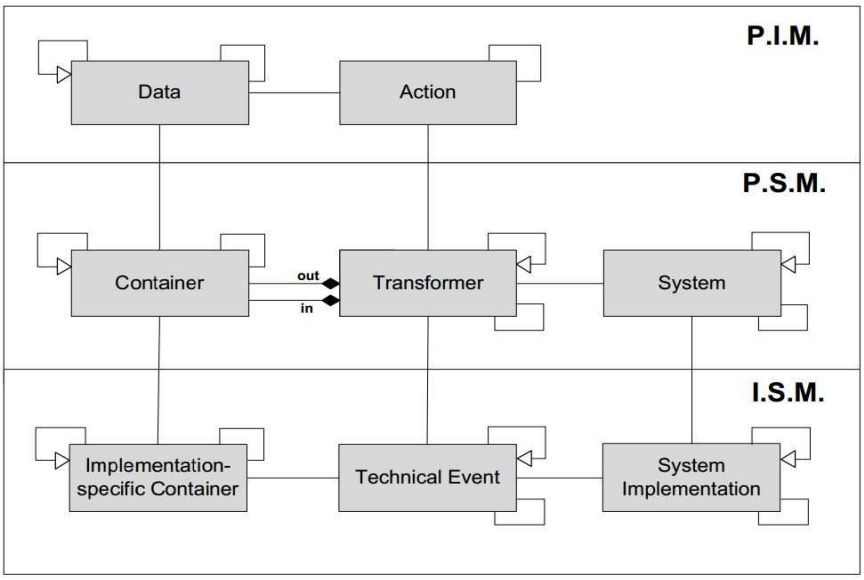

Fig. 1. Domain meta-model

Problem. Any realistic policy translation, therefore, must reflect these changes. So the first question that must be answered is: (1) how can we recognize changes in the domain structure? It is obvious that any changes in the domain structure must be reflected in the domain model that is used for data and action refinements. So the second requirement is about decision making: (2) how do we decide if the changes are to be accommodated in the policy translation and at what levels in the domain model must these changes be included? The third part of the problem is about (3) applying the changes. Intuitively, adding and removing refinements from the domain model is trivial. Updating the refinements however might not be straightforward. This subproblem therefore will be addressed by coming up with ways of merging different refinements of the same data or action which reflect the new refinements in the translation. This means that new policies must be generated or the already deployed policies need to be revoked and the new changed ones must be deployed.

Contribution. This is the first usage control adaptive solution that we are aware of. This solution consists of a conceptual but also a structural implementation. It includes every step from a policy specification in user-friendly manner, translation of the policy with implementation details and runtime adaptation when a system changes. In order to achieve each step, on the conceptual side, the change scenarios were explored and an adaptation solution adopted. Also the usage control components were extended in order to accommodate runtime adaptation. This report presents in detail the adaptation behaviour of the component which provides model-based translation.

Solution. In the following paragraphs we present the solution to the problems presented before. In the literature, a system which reacts to changes and adapts is called an adaptive system. The domain model changes and the UC infrastructure must detect or be informed of those changes and update the translation and the enforcement of the policies to conform to the updated domain model. The domain model can change by adding and removing elements. Additionally, same elements can have a similar or different structure, semantic, syntax. Therefore, for successfully overcoming the problems we employ techniques found in adaptive systems and also concepts from ontology merging. To validate our approach we conducted a case study in an online social network context.

2 Conceptual Framework

In the literature [4] were introduced two types of refinements, a transformer-based refinement and a state-based refinement. Therefore two types of adaptations need to take place, one regarding transformer-based refinement and the other one regarding the state-based operators. From a high level view, the domain model which is the base of the translation is dynamic and in time needs to be updated. New elements have to be added or the old ones updated. The changes are specified in a new domain model.

In the following paragraphs, we use the term *Base Domain Model* for the domain-model which has to be updated and the term *New Domain Model* for the model which contains the updates. After update, the Base Domain Model contains the domain model used for policy translation. The same name pattern is applied also in the context of data, containers, actions, transformers and systems. All these represent the elements of the domain model. An element of the domain model is represented by its *label*, which is the name of the element; we use label and name interchangeably, i.e. picture, album, copy, file, move, database. An element has a *type*, i.e. data, container, action, transformer, system, and it has a *level of abstraction*, i.e. PIM, PSM, ISM. When an element depends on another element from the same layer either as a refinement, aggregation, composition or specialization, we call that an *innerLink*. A new domain model can contain new elements or existing modified elements and all the modifications must be accommodated in the base domain model. In order to achieve this we employed tactics from ontology merging. For an instance of the domain model in the context of an Online Social Network, please refer to Figure 4 in Appendix A.

To better understand the requirements of an adaptive system we used the approach described in [9], and thus we identified the goals, the uncertainty factors and how to mitigate them. The top-level goal should state the overall objective of the system without being prescriptive in how to realize it. Lower-level goals refine the top-level goal by providing more information regarding the necessary steps required to achieve the top-level goal. The uncertainty factors represent sources of change (i.e. elements in the domain model) that might affect the satisfaction of the goals. Therefore, the top-level goal is represented by the achievement of the update of the enforced policies according to a changed domain model and the lower-level goals, which contribute to the high-level functionality, are to identify when the domain model has changed and what cases of changes there are. The uncertainty factors are represented by the possible changes of the properties of elements or of the relationships between elements. Uncertainty can include factors such as unknown new elements, conflicting refinements, incomplete information, unrecognised names from a lexical point of view. The mitigation of the uncertainty factors is achieved by the adaptation behaviour of the policy translation infrastructure.

2.1 Ontology merging

An ontology typically provides a vocabulary used to describe a domain of interest and a specification of the meaning of terms in that vocabulary. As presented in [16] the matching operation is the process of finding relationships or correspondences between entities of different ontologies. In our context, an entity is represented by an element of the domain model. The matching of two ontologies can be done by using different techniques applied at different levels, such as element-level or structure-level. From the matching techniques described in [16] we used both element-level and structure-level approaches. Additionally, the matching process takes into consideration linguistic properties, internal proper-

ties and semantic properties. These properties are described in detail for each type of elements in the domain model in the following sections.

The ontology merging process is the creation of a new ontology from two, possibly overlapping, source ontologies. The merged ontology is assumed to contain the knowledge of the initial ontologies, e.g. elements, relationships. This concept is closely related to that of schema integration in databases [16]. In our case, the ontology is represented by the Base Domain Model and the New Domain Model consisting of PIM, PSM, ISM elements, which have to be matched and merged.

[10] studies possible relationships between concepts in adaptive web systems. By applying these relationships between the elements in the domain model, one can easily see that there are essentially only two types of relationships that we have to consider for merging both data/containers and action/transformers: *containment* and *equivalence*.

Containment and equivalence take into consideration the following information: *syntactic information*, given by the name, signature and the type of the element, e.g. a data can be merged only with a data, an action only with an action; *structural information*, given by the refinement and its type, e.g. sequence refinement can be merged only with another sequence, and set with set; and *semantic information*, provided by the refinement elements or the lexical synonymity of terms. These three sources are all the information that we have regarding model elements.

2.2 Rules

While merging we have always employed a number of rules. The first rule, Rule 1, that we consider implicit is that **merging is done only at the same layer of abstraction**, i.e. PIM-PIM, PSM-PSM, ISM-ISM. It also means that only elements pertaining to the same system are merged. For example, when merging a file system details with another file system details at the PSM level, we never merge these details with those of a database system; similarly, at the ISM level, Firefox details are merged with Firefox, and Unix system calls are merged with Unix system calls at the same level.

The remaining rules are detailed in other parts of the report.

Rule 1 *Merging is done only at the same layer of abstraction. Only elements pertaining to the same system are merged*

Rule 2 *At PSM and ISM for the same system, the terminology follows a standard vocabulary.*

Rule 3 *The new domain model is always an update of the base one, not a replacement.*

2.3 Adaptive systems

As mentioned before, the domain model is dynamic because in real life systems change rather often. The usage control infrastructure must react in an adaptive manner to those changes. Therefore, the translation infrastructure must be adaptive and incorporate new models so that policies are correctly enforced at the implementation level. The component responsible for policy translation must act as an adaptive system, accepting requests to include a new domain model and merging it in the base domain model.

Self-adaptive systems have in common that they can reason about their state and environment. This is done via feedback processes described by a control loop which consists of four key activities: collect, analyse, decide and act [11]. The first activity, collect, implies gathering relevant information that reflects the current state. Analyse means structuring and reasoning about the raw data. Next, the system makes a decision based on the analysed data. Finally, the system acts in order to implement a decision.

These four key activities are captured by the three questions introduced in the beginning. Collect is represented by *Recognize*, analyse and decide is represented by *Decide*, and act is represented by *Incorporate*. Therefore, for each layer of abstraction the changes have to be recognized, the information must be analysed and the merging must be applied, and all in an automatic manner.

In order to accommodate changes at each layer of abstraction, each layer must have a mechanism in the translation component which acts on its behalf as an adaptive system which reacts to changes in the domain model. Therefore, to reach the top-level goal, a fully decentralized pattern seems to best fit the use case, as seen in [12]. This means that from a conceptual perspective, there is a control loop for each level of abstraction (PIM, PSM and ISM) and when adaptation is needed, the layer mechanisms inform each other of the required change.

Having considered all these, a generic algorithm which achieves the top-level goal of merging two domain models is Algorithm 1. The methods, their signatures and other details of implementation will be instantiated according to the context in which we apply the algorithm, i.e. data/containers, actions/transformers, systems.

In order to merge the elements, one has to keep account also of the dependencies between elements. Data and Containers do not depend on other types of elements. These can be merged first. Action and Transformer elements depend on Data and Container elements. Data and Containers must be already merged. Therefore, actions and transformers can be merged only after data and containers. Thirdly, systems depend on transformers, therefore they are the last elements of a domain model that are merged. Fourthly, state based transformers are defined by using data and action elements, therefore they can be merged only after these elements have been merged.

Further we will present every step (i.e. recognize, decide, incorporate) necessary to obtain the merging applied at the level of data and containers, actions and transformers and systems.

Algorithm 1: Generic merge

```
input  : baseLayer, newLayer :DomainModelLayer
output: baseLayer :DomainModelLayer
1  //Stage 1. Recognize,Decide,Incorporate inner layer
2  forall the Element newE of newLayer.elements do
3  |  baseE :Element;
4  |  /* makes use of the equivalence(a,b) relation */ baseE =
   |  searchEquivalent(newE, baseLayer);
5  |  if equivalentExists == false then
6  |  |  add newE to baseLayer;
7  |  else
8  |  |  add discovered synonyms to baseE;
9  |  end
10 |  updateInnerLinks(baseE, newE);
11 end
12 //Stage 2. Recognize,Decide,Incorporate cross layer
13 forall the elements of newLayer do
14 |  baseE = searchEquivalent(newE, baseLayer);
15 |  updateCrossLayerRefinement(baseE, newE);
16 end
17 //Stage 3. Layer Optimization
18 forall the elements of baseLayer do
19 |  /* makes use of the containment(a,b) relation */
   |  removeRedundantInnerLinks(baseE);
20 end
```

2.4 Data and Containers

In the context of data and containers, one uncertainty factor is represented by the possible changes in the domain model. Therefore the first step towards mitigation is to identify all possible changes from PIM level to ISM level. The domain model can change by addition, removal or modification of elements or of the association relations between the elements. Each element is identified by its label, i.e. its name; adding a "file" means adding an element labelled "file" with this name at a level of abstraction with refinement and associations, i.e. inner links. Unless otherwise specified, two elements are always compared based their label. In the context of data and containers, adding new elements means for the end-user the addition of new elements which can later be used as elements for the policy specification; e.g. add "profile" data at PIM, "file" container at PSM or "unixFile" at ISM. Modifying the links between elements means changing the semantics of the domain-model, e.g. before, a "profile" was an aggregation only of pictures, and after adaptation, a "profile" is an aggregation of "pictures" and "comments". These types of changes can appear at each level of the domain, and we will expand upon them. At the same time one must also investigate how the changes are affecting each other, if or how they affect multiple levels of abstraction.

Recognize Changes We have already mentioned that the Platform Independent Model (PIM) is an abstraction of Platform Specification Model (PSM), and that PSM is an abstraction of Implementation Specification Model (ISM). In other words, PSM provides technical details for the PIM layer and ISM provides implementation details for PSM. This means that the level of abstraction decreases from top to bottom. If two elements are in a generalization-specialization relation as defined by an UML diagram, a change in the implementation does not imply a change also in the more generalized element. The relation between PIM-PSM and PSM-ISM is of type generalization-specialization. Therefore, a change in a lower layer does not imply a change in the upper layer. The relation across layers for data-container and container-container is of type *generalization-specialization*, while the relation inside the same layer, inner-layer association, is of type *aggregation or composition*. If an element depends as refinement or association on other elements from the same layer and the same type, we call those dependencies an *innerLink*. The inner-layer associations between data reflect the associations between real systems, such as a database or an excel document where a workbook is composed of tables, a table is composed of rows, a row is composed of cells; or in a file system where a directory can be an aggregation of files; or a html page which is an aggregation of html elements which in their turn are refined as labels, tables, forms and these can contain other html elements. The cross-layer links between elements reflect how the data is stored in real systems, or how technical concepts are implemented in particular systems.

Next, starting at PIM, we explore what types of changes can occur and how these changes affect the domain model. At the level of Platform Independent Model (PIM) the first type of change is the addition of a *new data*. This can come in different versions.

First, if the New Data is refined in terms of other existing Base Data at the PIM level, then the adaptation affects only the PIM level, no further updates are necessary. This is a case when only an association between data changes. The associations between data and data, named *inner-layer associations*, might need to be updated, as it was in the aforementioned example of adding a "comments" element to a "profile". The associations between data can be of type *aggregation*, an album is an aggregation of pictures, and *composition*, a forum is a composition of comments. Also, some of the associations between New Data and Base Action at PIM have to be updated, meaning that there might be actions which can apply on the new data. If the new domain model consists only of modified associations or removal of elements, the same flow of update events occurs. Besides these updates, no other levels are affected.

Secondly, when a New Data needs to be added to the Base Domain Model, besides the updates presented before, also associations between the New Data at PIM and the Base Container at Platform Specific Model (PSM) might need to be updated. This is a case of *cross-layer associations* between data and container. An example of this could be an addition of "picture" at PIM which is refined as a "file" at PSM. Additionally, if the PSM container does not exist, then a new container at PSM level needs to be added first. Then, the necessary associations

at PSM level between existing transformers and the new container need to be updated in a similar manner with those at PIM. Only as the last step the cross associations between PIM data and PSM container can and must be added. This is an example where a change at one layer affects also another layer.

Thirdly, it might be the case that the New Container at PSM from the previous step is part of a new system, e.g. distributed file system. In this case, the actual implementation of this container needs to be specified also at the ISM layer because PSM layer contains no implementation details. Therefore, if the ISM containers do not exist, the necessary containers at ISM level have to be added, together with the inner and cross layer associations. This is an example when a change at PIM requires a change at PSM, which in its turn, requires a change at ISM. All the three layers are affected. An example of this could be the addition of "video" data at PIM, which requires a new container such as "htmlElement" at PSM, which in its turn is refined as "video" label in html5.

At the PSM and ISM level the same changes and effects flow occurs for the addition of containers or the update of the inner/cross layer associations between containers and transformers or the associations between containers and transformers. The flow of changes involved with adding, changing or removing a data or a container are essentially the same for all the layers.

One can immediately observe that when an element is updated the flow of changes is from top to bottom, updates at higher levels of abstraction might trigger updates at lower levels.

Decide upon Changes We have seen that in order to achieve ontology merging, one has to establish relationships between elements. These relationships can be of type *containment* and *equivalence*. To establish a relationship between two elements, one can use syntactic, structural or semantic information. We remind the reader that an element is represented by its label, which is its name and unless otherwise specified, when talking about an element in a general way we refer to an element with a particular label.

In the case of *data* and *container* elements, the ***syntactic information*** is given by type of an element. In this case, data elements are merged compared with data elements, and containers only with containers. With regard to ***structural information***, data and containers are refined only as a set of other elements. Considering the ***semantic information*** in the case of data, this is given by lexical similarity between the names of the two data. For example, in the base domain a data is labelled "picture", whereas in the new domain, the same data is labelled "photo", the two labels are in a relationship of synonymity from a natural language or lexical viewpoint. *Lexical similarity* implies that, from a natural language point of view, two words have the same meaning. We avoid the term lexical "synonym" to avoid confusion with the "synonym" property introduced before.

No other information can be used to establish the relationship between the elements except the term used to label an element for the following reasons. The first reason is that a data element is specific to a particular user's domain model.

The PSM layer containers represent only where that data can be stored. The same data from different domain models can be stored in different containers and in different implementations. This does not necessarily mean that the data is different, but only that it can be stored in different containers. The second reason is that the same label can be used by different users in two different domain models to store different information in the same container. For example, one user can label its data "project" which stores business specific information about economical impact, whereas another user can label its data with the same name but storing technical specifications of the project. Both data are refined as a file at the PSM and ISM level. Having no other information available in the domain model, one has to assume that the same term is used with the same meaning in different domain models if these domain models are to be merged. This is another uncertainty factor which is to be mitigated by an agreement of terms.

In order to achieve an *agreement of terms*, a negotiation of the terms and conditions of usage has to take place between the two parties as described in [3]. Before the merging of the domain models can take place, a negotiation of the semantics of terms needs to take place. This means that the parties will agree which terms will designate the same data. In order to allow a user to still use his specific business terminology but also allow room for agreement we introduce for each element another property: *synonym*. Each element alongside its label will have also a set of synonyms which are intuitively alternative names. For example, an element labelled "picture" can have as a synonym the name "photo". Therefore, the element can be identified by both of the terms. By using the synonym, an element of the domain model can be used inside company A with the name "picture", but when company A wants to collaborate with company B, it needs not change its business terminology but only provide a synonym for that element, for example "photo" being the common term. With regard to the set of synonyms for each element, it can be defined in two manners. First, for each element in the domain model the synonyms are statically defined, and secondly, at runtime during the merging process, the discovered lexical similar labels are added to the set of synonyms for that element.

In the case of *container* elements, the *syntactic information* and the *structural information* is the same as in the case of data. The difference comes at the *semantic information* where for PSM and ISM containers no lexical information can be established for the following reasons. When considering the PSM and ISM levels one has to keep in mind that these represent generic systems at PSM level and actual system implementations at ISM level. Additionally, to ensure a basic level of compatibility between domains one has to consider at PSM and ISM **the terminology for containers and transformers** to have terminology that **doesn't change so often for the same systems**. For example, a container named "file" remains "file" for all generic operating systems (OS). There should not be two ontologies which talk in different terms about the same OS system at PSM level, i.e. in one case there is "generic-file" and in the other one there is "template-file" for the same container. Of course, there are differences at ISM

level for different implementations of the OS, such as Windows or Unix. Therefore at PSM, for the same system, one has to use the same terminology for all domain models. At ISM level, if the names differ, this means they represent two different implementations, e.g. open and fopen, exec and execv. We consider this to be a reasonable rule for merging that one can apply, see Rule 2 (section 2.2). Although, in order to introduce a level of flexibility for the negotiation process, we allow the existence of synonyms also at this level, but these synonyms do not necessarily have a correspondence in the natural language, there is no lexical similarity at PSM and ISM level.

Further, we acknowledged that there is no *containment* relation between data-data or container-container when considering merging. Although, in the same domain model there can be aggregations and compositions defined, thus containment relationships, these relationships are defined at the structural level and not at the lexical level. These types of relationships will be automatically merged when considering the refinement or the associations defined in the domain model. For example, an "album" at the lexical level is related with a "picture", an album contains pictures, but this has to be defined in the domain model and not inferred from the terms.

We have seen that the merging of two data or container elements does not depend on the refinement. Therefore, the merged refinement is the superset of the lower level containers. If "picture" was refined only as an htmlElement at PSM and the new model refines "photo" as file, the result will be "picture" refined as htmlElement and file. As another rule for merging, see Rule 3, we consider **the new domain model to always be an update of the base one, not a replacement**.

Therefore, two data are *equivalent* if the labels are the same, or they contain the other's label as a synonym or lexical similarity can be established between the names. Two containers are equivalent if the labels are the same or there is a match between the synonym names. The merging is done based on the name of the data or the name of the container.

bool Equivalent(Data a, Data b): *Data a is equivalent to Data b IF they have the same name, OR one's name is in the set of the other's synonyms, OR the names are lexically similar.*

bool Equivalent(Container a, Container b): *Container a is equivalent to Container b IF they have the same name, OR one's name is in the set of the other's synonyms.*

Next, in order to merge two sets a relation between a set and an element needs to be defined. This is done by using the *equivalence* function defined before. We define the *containment* relation between a set and an element.

bool Contains(Set s, Element e): *Set s contains Element e IF there is an element a as a member of s which is equivalent to b, i.e. equivalent(a,b)==true.* Now, two set refinements can be merged by creating a refinement as the union set.

Incorporate Changes We have argued that the flow of changes is from top to bottom, meaning a new or modified element at PIM may require first a new or modified element at PSM, and the same order for PSM-ISM. More specifically, if one wants to add a data, the refinement elements of the lower levels must be added first. The technical and the implementation details are provided by the lower, more detailed, levels of abstraction. Each element can have as refinement other lower level elements from a more refined level of abstraction. In order to merge two elements all the details of those elements must already be available to be used for comparison. One element has as a property the refinement which references lower level elements, and in order to merge the refinement, the lower level elements must already exist in the merged lower level layer. Therefore, when incorporating the changes, one has to use a *bottom-up approach*, add the lower elements first and only after that merge the upper elements based on name and merge the refinement, because the children elements don't know of the parents. First, the new ISM layer must be merged with the base ISM layer, secondly the new PSM layer must be merged with the base PSM layer, and only lastly the new PIM layer can be merged with the base PIM layer.

Applying the generic algorithm presented before (see Algorithm 1), the adaptation process applies at each layer the following steps presented in Algorithm 2 for each element of the new domain model in the order ISM, PSM, PIM layers.

Algorithm 2: Add New Element

```
1  void addNewElement(Element newE, DMLayer baseLr)
2      if newE.isMerged then
3          return;
4      end
5      forall the Element innerE in newE.innerLinks do
6          addNewElement(innerE, baseLr);
7      end
8      /* Stage 1. inner layer update */;
9      Element baseE = searchEquivalent(newE, baseLr);
10     if baseE != null then
11         add newE to baseLr;
12     else
13         add discovered synonyms to baseE;
14     end
15     updateInnerLayerLink(baseE, newE);
16     /*Stage 2. cross layer update */;
17     baseE = searchEquivalent(newE, baseLr);
18     updateCrossLayerRefinement(baseE, newE);
19     /*Stage 3. layer optimization*/;
20     removeRedundantRefinement(baseE);
21     mark newE as merged;
```

We start by illustrating at ISM. Before doing any merging, for each container from the new layer the same procedure is applied in a recursive manner for every innerLink element to make sure that the referred elements are already merged by the time the associations need to be merged. In this manner we ensure that the lowest level containers are added first, and then the algorithm returns in a recursive manner to merge the containers.

In *Stage 1*, all the recognized new ISM containers from the new domain model are added to the base domain model. If the container is already at the lowest level, then no recursive call on its refinements can be made. For each container from the new layer, its counterpart is searched in the base domain using the function ***searchEquivalent(Element newE, DMLayer baseLayer)*** which represents the *recognize* and *decide* activities. As we have already mentioned, see Rule 2, at ISM level containers are matched by the same name or synonyms. If the base container does not exist, then the new container is added to the base. Next, new inner-layer links are added in the function ***updateInnerLayerLink(Element baseE, Element newE)***. This represents part of the *incorporate* activity. Here the *containment* relation between a set and an element is verified. For example, before adaptation the base domain had the container "file" and the container "htmlElement" which was associated with "img" container only. The new domain container "htmlElement" is associated with "label" and "media", while "media" is associated with "img" and "video". After this step is finished the base domain model will be the following: "file", "htmlElement" associated with "label" and "media", and "media" associated with "img" and "video". One can observe that the old containers were kept and the new associations have been updated. The same strategy is applied also at PSM with no difference because the rule 2 is valid there also. For a basic merging example with all three abstraction layers, please refer to Figure 5 in Appendix A.

In *Stage 2* we process the cross layer refinements. After the previous step, all containers from the new domain model are now present in the base domain model. Also, all the inner associations between containers are present. The next step is to add the cross layer refinements for each container at PSM. For each new container from the new domain and for each cross layer refinement of that container, the refinement link is added to the corresponding base domain container. This is done by the function ***updateCrossLayerRefinement(Element baseE, Element newE)***. This is the other part of the *incorporate* activity. There is no need to check if the container is present, because this was already done in the first stage. Now that the ISM and PSM layer are completely merged, we can proceed to the last layer of abstraction, PIM.

At the PIM level, data elements from the base domain model and new domain model are compared. We apply the same steps starting from stage 1 with a few differences. We will expand only on the differences. The inner-layer associations at this level can be of two types, aggregations and compositions, but this does not affect the merging. Next, for all data elements from the new domain model is searched a counterpart in the base domain model. The first step of the search is to look for an element with the same name or synonym. If there is no match,

Algorithm 3: Search Equivalent Element

```
1  Element searchEquivalent(Element newE, DMLayer baseLr)
2      forall the Element baseE of baseLr.elements do
3          if baseE.type!=newE.type then
4              continue;
5          end
6          /* implements the equivalence(a,b) relation */
7          if newE.name == baseE.name then
8              return baseE;
9          end
10         if baseE.type!=Data ||baseE.type!=Action then
11             continue;
12         end
13         /* the following only for Data or Action */
14         if newE.synonyms.contains(baseE.name) then
15             return baseE;
16         end
17         if baseE.synonyms.contains(newE.name) then
18             return baseE;
19         end
20         if isLexicalSimilar(baseE.name, newE.name) then
21             return baseE;
22         end
23         forall the String syn of baseLr.synonyms do
24             if isLexicalSimilar(syn, newE.name) then
25                 return baseE;
26             end
27         end
28         forall the String syn of newE.synonyms do
29             if isLexicalSimilar(baseE.name, syn) then
30                 return baseE;
31             end
32         end
33     end
34     /* no equivalent element found */
35     return null;
```

then a lexical library is used to compute a similarity metric between the words that form the name of the compared containers. This is done by the function ***isLexicalSimilar(String name1, String name2)***. The closest base concept to the new concept is chosen as the container for the merging. For space reasons, the presented pseudo-code takes in consideration only the first found lexical similar element. If two concepts, names are similar, then the base container will from now on also contain a synonym name. The final step is to add the cross layer refinements as described in stage 2.

In *Stage 3* after all the merging has been completed there can be a case where an optimization is required. Redundant inner layer links could have emerged, because one can specify as a refinement an element at the same layer of abstraction. For example, before merging, the base domain model had at the PSM level a "domElement" which was also refined at the same level as "img". The new domain model has the same "domElement" refined at PSM as "media", which at its turn is refined again at the same level as "img" and "video". After merging, at this step, the base element "domElement" has a redundant refinement link to "img", a direct link and one through the "media" element. Because the refinement is a generalization-specialization relation, only the link through "media" element should be maintained. Therefore at this stage, all redundant inner refinement links as presented before are removed.

One can immediately observe that there are no merging exceptions. This is because the semantic of data and containers is based only on the names of the elements. At PIM each new term is considered either a new data or a lexical similarity can be discovered between words, while at the PSM and ISM level there is a common terminology, each new container represents a new technical or implementation detail.

A final remark in this section is that if a container is added, this always implies a change also on the transformers side. Transformers, by definition act on a set of containers and represent system functions. Transformers have as an association a set of input containers and a set of output containers. For example, transformers that work on a container "file", will not work on another type of container. Transformers at PSM and ISM level are closely related with real systems. System functions have a limited set of data types on which they can be applied. Usually, a new container is related with a new system, thus, new transformers have to be added also. On the other hand, if one adds a new transformer, no new containers are usually required. But if one adds a new container than the association between transformers and containers needs to be updated because a container adds new functionality to a transformer. With this in mind, we continue to the next adaptation context, actions and transformers.

2.5 Actions and Transformers

An action is performed by the user while attempting to use the data. Usage control can act as an enforcement mechanism, to deny the realization of that usage, or can act as a detective mechanism, to notify to the owner the usage of a particular data [3].

Furthermore, as described in [4] transformers can be refined as a **set** of other transformers or a **sequence** of other transformers. A set represents a logical *or* relation between transformers. This means that if the PIM action "copy" is refined at PSM as a set of "copyFile", "copyHtml" and "copyDir", then the action "copy" is recognized when any of the low level transformers was invoked. On the other hand, a sequence induces ordering among the elements. For example, transformer "copyFile" is refined as a sequence of the transformers "open", "read", "write", "close". This means that "copyFile" can be considered

executed only after all the refinement transformers have been invoked in the specified order, "close" was the last invocation, sometime in the past a "write" has happened, before that a "read" and before that also an "open". Additionally between these elements of the sequence, any number of other transformers can be executed [4].

Another aspect one has to keep in mind is the relation between layers. As presented in [17], in UML class diagrams, generalization-specialization relation assigns to each abstract class a **set** of possible specializations. We have mentioned that the PSM layer adds technical details to the PIM layer, and ISM adds implementation details to the PSM layer. In other words, between the layers there is the relation of generalization-specialization. Deriving from these, we recognzied that across layers there are only set refinements. On the other hand, **sequence** refinements talk of system specific implementations and how the system behaves at runtime. For example, to copy a file, one must open, read, write and then close. We have already mentioned that the relationship across layers is of type set, but there is an additional reason why one cannot have a sequence across layers. The generalization-specialization is a static relation not a dynamic relation, there is no ordering or relations between specializations, e.g. relations depending on time. Also, to specify a sequence at one level of abstraction which talks of a lower level of abstraction, would mean to bind that refinement to a specific implementation. For example, "copy" at PSM might be refined in another ISM system with only simple transformer, "copyFile". Therefore, in order to avoid having a PSM transformer refined as a sequence for a particular implementation, that sequence is defined at the same level with the implementation, i.e. ISM level. The sequence thus defined acts as an intermediary element with a unique name referencing the actual transformers and being referenced by the PSM transformer just as another transformer.

Recognize Changes As in the case of data and containers, we argue that the flow of changes follows the same pattern, top to bottom. This means that a change in a more abstract layer triggers a change also in a lower level layer. In other words, if an element is added at a higher level of abstraction, then it is possible that new elements are added at lower and more concrete levels of abstraction.

The first type of change is when at the PIM level the action-data inner association changes, meaning that an existing action is updated and it now applies also on another existing data. In other words, an action is enabled or disabled upon a certain data and no other levels are affected. e.g. before, "copy" action was associated only with "text"; after, it is also associated with "picture"; the end-user can now specify policies like "do not copy picture". At PSM and ISM, the transformer elements have as a property a set of containers as "input parameters" and a set of containers as "output parameters", but the same flow of changes applies also in case case of an association update, i.e. innerLink update. In the case of transformers, the name of the transformer and its input parameters represent the *signature* of a transformer. This is analogous with the signature

of a method in the case of programming languages. Two transformers have the same signature if they have the same name or synonyms and they have the same number, same order and same input containers. Only the output containers are not considered part of the signature, that is, a transformer cannot differ from another one only by different output containers.

The second change case is when a new action is added at the PIM level. As we have already seen, one has to add the new action at PIM, add associations at PIM between the new action and existing data, and add cross refinements between the new action and the existing PSM transformers; e.g. "delete" action is added. Additionally, if the PSM transformer does not exist, then one must add the new transformer at PSM level, add inner associations at PSM level between existing containers and the new transformer, add cross associations between the new PSM transformer and existing ISM transformers or the inner refinement of the new PSM transformer. Only in the end the cross-layer refinements from PIM to PSM can and must be added. This is a case where two layers are affected by the update; e.g. add "delete" action at PIM, then add remove transformers for browsers, or delete for the operating system. If the ISM transformer does not exist, then it is a case of three layer adaptation triggered by a change at PIM. Also, if a new transformer is added, then the old transformers can be updated to be inner refined only with the new transformer. This is a good example when deprecated functions can be replaced by new and more secure ones.

The third case of change is a change in the refinement of an action or a transformer, an update of the action-action inner-layer links, action-transformer cross-layer links, or transformer-transformer cross/inner-layer links. As we have already seen, transformers can be refined as a set of other transformers or by a sequence of other transformers. These refinement transformers can be either at same level or a lower level. Therefore, one can add a new refinement, or remove a refinement, or add an element to a sequence, or remove an element from a sequence. Inside the same level of abstraction at PSM or ISM, a transformer element can also be specialized into other transformer elements. This is yet another form of inner-layer links between elements.

Decide upon Changes In the case of actions and transformers, the information for merging is provided not only by the name but also by the refinement. In a manner similar to data and containers, syntactic, semantic and structural information is used to specify the parameters of merge. With regard to ***syntactic information***, this is provided by the type of an element, i.e. action or transformer. With regard to ***semantic information***, this is provided by the lexical similarity relationship present between the names of the actions and the refinement of two elements. Finally, ***structural information*** is related to the type of the refinement, if the refinements are in a containment or equivalence relationship, and the signature.

Therefore, in the case of actions, the comparison must take into account the names of actions and the refinement. For example, two actions "copy" and

"duplicate" are going to be merged only if, the names are equivalent, as seen at containers, and if the refinement can be merged.

bool Equivalent(Action a, Action b): *Action a is equivalent to Action b IF they have the same name, OR one's name is in the set of the other's synonyms, OR the names are lexically similar OR they have the same refinement.*

In the case of transformers, the equivalence relation must additionally take into consideration that the transformers must be part of the same system and also the signature.

bool Equivalent(Transformer a, Transformer b): *Transformer a is equivalent to Transformer b IF (they have the same name, OR one's name is in the set of the other's synonyms OR they have the same refinement), AND both have the same signature, AND both of the transformers are part of the same system.*

Regarding equivalence, a first note is that the equivalence relation between two transformers does not imply that they always have the same meaning, i.e. the same refinement, it only provides a method to find a correspondent transformer from a new domain model to a base domain model, by making use of the rule 2. Secondly, we have used generic names for transformers because, for example, if one defines a domain-model for a relational database management system and installs a UC infrastructure, at the ISM level, functions could be stored procedures with the name defined by the administrator. Because of this and the rule 2, no lexical relation can be established between transformers' name at runtime by the use of a natural language dictionary. Thirdly, synonyms are allowed also at the PSM and ISM level in order to provide a mechanism for negotiation of terms, although no runtime synonyms discovery is provided for the reasons mentioned before. Fourthly, because the refinement of an action or a transformer provides the semantic of an element, only in the case when the refinement is the same, even though the label may differ the two elements will be considered as one and a synonym with the label of the new element will be added. One transformer can have indefinitely number of inner refinements, which leads to the fact that the same refinement can be expressed in multiple ways by using set or sequence refinement. Therefore the only case when it is feasible to merge the transformers only by the use of synonyms is when they have exactly the same inner and cross refinement. If they don't have the same refinement, the following logic applies in order to try and merge the refinement when two elements are equivalent.

In order to merge the refinement, we have to define merging rules for sets and sequences. We start with **sets**. Suppose a base action or transformer has a simple set refinement, meaning that the element is refined by only a few elements. Also suppose a new action or transformer has a more detailed set refinement, meaning more elements are added to the refinement. For example: a domain model defined for an older version of a mail client system considers action "copy" only "forwarding" the email. An updated domain of a newer version system considers "copy" being comprised of "forwarding" or "printing" the email. One approach is to have the merged action "copy" with refinement set composed

of "print" and "forward". In this way, after the domain models are merged, the users of the old mail client will not be able any more to print the emails, although before this was not the case. In this approach, the larger set of transformers is the one that will be enforced. The union set is obtained by making use of the *containment* relation between a set and an element introduced before. From a security viewpoint we label this a *conservative* approach to refinement. This approach ensures that the most detailed policy translation is enforced. If this policy is used as a preventive mechanism, then an increased restriction level is maintained. If a system is installed with a more detailed policy refinement, then the more detailed policy refinement will be considered the merged one. This approach also makes use of the rule 3 which affirms that the new domain model is an update of the base domain model, not a replacement.

The next challenge is related to comparing **sequences**. In the next paragraphs if a sequence element is marked as *sequenceName=<ref1, ref2, ...>*, while a set refinement is marked as *setName={ref1, ref2, ...}*. A sequence also has a property named *length* which represents the number of contained elements. To compare sequences one has to define the equivalence relation between sequences.

The first case is when two sequences use the same unique elements and the same order, but different number of elements e.g. *seq1=<open, read, write>* and *seq2=<open, read, read, write>*. One uncertainty factor which occurs at this stage is the context in which this sequence will be used. It can be used in a detective mechanism or a preventive mechanism, depending on the policy to be specified by the end-user. The main difference between the sequences is that the first one marks the execution of the refined action or transformer after one read, while the second sequence does that only after two events have happened. Let's suppose that end-user 1 had specified a policy "notify me when copy" using a refinement with seq1, while end-user 2 used seq2, knowing before hand that the notification will come only after a second read. Then if one keeps *seq2* as the merged one, end-user 1 would have its initial usage policy violated, one could read the data once without a prior owner notification. If one keeps *seq1* as the merged sequence, then end-user 2 will receive more notifications but no usage policy is violated. We remind the reader that between the elements of a sequence any number of other transformer invocations ca be recognized by the usage control infrastructure [4]. This means that *seq1=<open, read, write>* can be see also as *seq1=<open, *, read, *, write>*, where * represents any transformer recognized by the UC infrastructure. Therefore, *seq1* can be said that subsumes *seq2*. On the other hand, in the context of a preventive mechanism, this approach has at least one downside. Supposing, that the aforementioned *seq2* provides multiple writes instead of reads. Both of the sequences are used in a policy to overwrite a location so that no sensible information could ever be retrieved. If *seq2* is used as the merged one, then the policy is maintained and also the more secure definition is used. If *seq1* is used, the policy is still maintained but depending on the physical storage medium sensible information could be retrieved. Having no information about the context in which a sequence is used, for the adaptive translation system we have used for merging sequences the more generic one.

Because sequences can contain any number of identical elements one after the other, in order to compare two sequences we introduce a function named ***minimize***. The function is applied on a sequence and returns another sequence with the same elements and the same order of the input sequence but there are no pairs of identical elements, i.e. consecutive duplicated elements are reduced to one. For example, minimize(<a, a, b, b, a, a, c, c, d>) = <a, b, a, c, d>. We mark the minimized sequence of seqA as *seqA.min*.

sequence Minimize(Sequence s): *Eliminates all pairs of identical elements from input sequence s while preserving the order of the elements.*

Another function needed to compare sequences is a method which verifies if an element is contained in a sequence. This is defined in the following way.

bool Contains(Sequence s, Element e): *Sequence s contains Element e IF there is an element a as a member of s which is equivalent to e, i.e. equivalent(a,e)==true.*

Now we have all the elements necessary to define the equivalence relation between two sequences.

bool Equivalent(Sequence s1, Sequence s2): *Sequence s1 is equivalent to Sequence s2 IF for all elements e of s2.min, s1.min contains e, i.e. contains(s1.min, e)==true, AND all elements of s2.min have the same order as all elements of s1.min, AND s1.min.length==s2.min.length.*

We have seen so far that in the case of sets we always use the union set of the refinement sets. Regarding sequences, if two are equivalent then the shortest subsequence of consecutive repetitive elements from each sequence is kept. For example, seq1<a, b, b, c, a, a> and seq2<a, b, b, c, c, a> are merged to seq<a, b, b, c, a>. In a detective mechanism, this approach ensures no invalidation of policies but in the case of preventive mechanisms, there can be a downside as mentioned before. The adaptive translation system always merges two elements such that the most restrictive definition of a policy will be achieved.

A second case of merging is when one of the sequences has at least one unique extra element, e.g. *seq1=<open, read>* and *seq2=<open, read, write>*. Although every invocation sequence captured by *seq2* can also be captured by *seq1*, these two sequences cannot be considered in an equivalence relationship. In the context of a preventive mechanism, *seq1* informs of a read while seq2 informs of a write. In the context of a detective mechanism, seq1 can restrict a read while seq2 can restrict a write. This is another uncertainty factor which needs to be mitigated when two sequences with the same name, one from the base domain model and one from the new domain model need to be merged. The first approach is to keep both of the sequences as a valid refinement. The second approach would be that by triggering a merging exception, a human intervention is required to establish the definition to be used. The transformers at PSM and ISM carry only technical or implementation details. There is no a priori information about the context of the policy or action which will use a particular transformer. The refinement can be included in a preventive or a detective mechanism. Therefore, we chose to keep both of the refinements when no equivalence had been established between two sequences.

The third case of merging sequences is when the order or the elements are different. In this case, the sequences cannot be merged.

Yet another uncertainty factor when comparing two actions' or transformers' refinements definitions is a case of *partial definition*. Partial definition in this context means that one of the refinements contains an extra element, such as more transformers, compared to the other refinement. What happens if one of the systems does not have all the implementation details defined? For example, a "copy" action is refined at the PSM level in the FileExplorerGeneric system by the transformer named "copyFile". At ISM level, for FileExplorerA implementation, the refinement means "DuplicateFileA" or "MoveFileA" and for the other case, FileExplorerB, the refinement is only "DuplicateFileB". The major challenge is how to automatically generate the implementation of "move" for FileExplorerB because this case is present in FileExplorerA. A partial mitigation of this problem can be achieved by using "system ontologies" for known systems. We consider a system ontology to be a database which contains a mapping from natural language terms to specific implementation functions. One example of system ontology could be one where there is a mapping between different representations across systems of the same function. For example, "move" is also known as mv, or mov or relocate. Therefore, if the behaviour (e.g. moveFile) is implemented in different systems only by a function already known then one can search in this ontology for an implementation for another system. In the case of our example, the ontology should contain a definition of "move" for the FileExplorerB implementation if this functionality is supported. This implies that the behaviour of the systems is known beforehand.

Now that the decision points are specified, we can proceed to the adaptation algorithm.

Incorporate Changes In order to merge two domains, a similar strategy with that already presented in the case of data and containers is applied, therefore, again a bottom-up approach is necessary. First the ISM level is merged, then PSM and finally PIM and for each element a similar algorithm is applied as algorithm 2. The merging information is provided by the syntactical, structural and semantic properties of the elements.

In *Stage 1* all new transformers from the new domain model are added to the base domain via the ***addNewElement(Element newE, DMLayer baseLr)*** function. The equivalent transformers between the two domain models are found via the ***searchEquivalent(Element e, DMLayer baseL)*** which was first introduced in the context of data and containers. The method has an additional test which verifies if the compared transformers are from the same system, and another additional test which compares the signature of the transformer. If an existing equivalent transformer is found it can be the case that a transformer in the base domain is refined as sequence and another transformer with the same name in the new domain model is refined as set. Then the sequence is extracted and added as another inner-layer refinement to the super set of refinements of the merged transformer. For example: before merging the base

transformer was *deleteFile* = <*removeFile, overwriteMemory*>, the new transformer was *deleteFile* = {*purgeLocation*}; after merging, the base transformers are *deleteFile* = {*purgeLocation, deleteFileSeq1*}, *deleteFileSeq1* = <*removeFile, overwriteMemory*>. A sequence element can be seen only as an intermediary element which provides order information while its name having no direct correspondence in the definition of a real system. That is why the name of a sequence can be changed or generated automatically as seen in Figure 2. These steps are all applied at the same layer of abstraction.

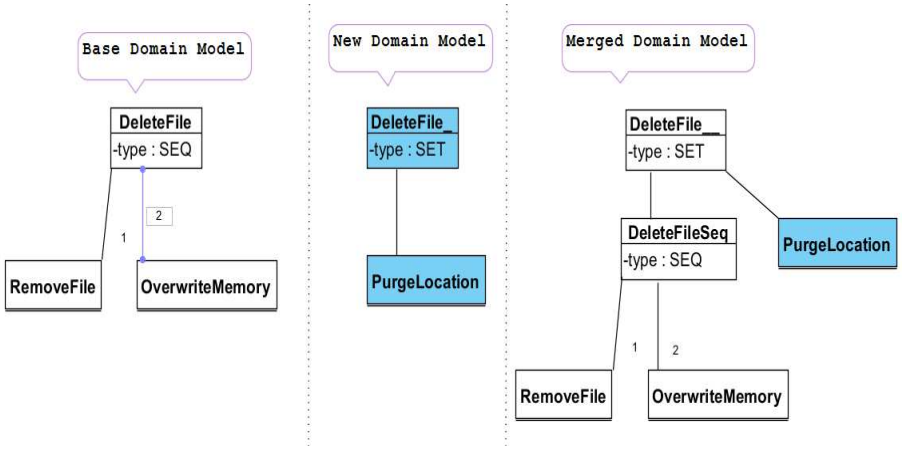

Fig. 2. Transformers merge

Next, at *Stage 1* for each transformer the inner layer refinement is updated via the ***updateInnerLayerLink(Element baseE, Element newE)*** function. In the case of set refinement, the union set is considered the merged one. In the case of sequence refinement, we apply one of the three merging cases presented before. In pair all the refinement sequences are compared and based on the equivalence relationship the merged sequence is kept. The aforementioned steps are applied in an identical manner at PIM, PSM, and ISM. In the case of transformers, the specialization of a transformer is merged in a similar manner as the set refinement, by creating the union set of specializations.

In the next stage, *Stage 2* the cross layer refinements are processed in the function ***updateCrossLayerRefinement(Element baseE, Element newE)***. We have already argued that across layer there are only set refinements. Therefore, for equivalent elements, the refinements are merged as the union set of refinements.

In the final stage, *Stage 3* a clean up process is applied in function ***removeRedundantRefinement(Element e)***. Sequences can have different names but the same elements. Therefore, at this moment, an element could end up with a set refinement composed of multiple sequences with different name, but

which could be in an equivalence or containment relation, e.g. copy={seq1, seq2}, seq1=<open, read, write> , seq2=< open, read, read, write> . Here we look only at the elements. These sequences are processed in pair and merged according to the case presented before. All redundant sequences are removed from the base domain model.

Now that the merging is complete for the actions and transformers, we can proceed to systems.

2.6 Systems

Systems are present at the PSM level specifying technical details, e.g. operating system, browser, social network, browser plugin, and at ISM level specifying implementation details, e.g. Windows, Unix, Firefox, Chrome, Excel, Outlook. A system is represented by a *set of transformers*. Transformers from different systems can operate on the same container, e.g. Firefox and Windows can write to a file. Therefore containers are not associated with any particular system.

Recognize Changes The first type of change is a completely new system, which can be seen as a *paradigm break*, informally defined as a technical innovation with new conceptual and technical details. This means that an entire stack of transformers and containers have to be added at the PSM level, but also at ISM. An example of this is a distributed file system at PSM, and a specific implementation at ISM, e.g. Hadoop, GFS. If a new system concept is added this means that new actions and data at PIM can be added to specify new policies, but at PIM level the update has to be done manually. The existing actions and data have to be associated with the new elements in order to provide policy enforcement also for the new system. When a new system is added, all three levels of abstraction can be are affected, i.e. PIM, PSM and ISM. For example, a mail client is added at PSM and as an instance at ISM, the Thunderbird application. Now, at PIM one could add a new action "emailto" because the domain model supports the action.

The second type of change is a *new implementation*; e.g.: Safari browser is added; Firefox is updated to the new version which offers new functions. At ISM level an instance of a PSM system can be updated, or a PSM system is updated. This update might consist of addition, merging, change, or removal of elements. If new elements are added to the model of the system, than also the domain model must be updated. At this step it suffices to say that, from the perspective of a system, the optimization of a transformer, e.g. a stream copy function has better performance and better security, this type of change does not affect the policy translation and enforcement process.

The third type of change is a *new instance*. For example, an application runs with a specific user profile. The point of change is at the ISM level when an application runs with a different set of parameters depending on the user which launches the application. For example, Mozilla Firefox can be run with different profiles. Would that require a change in the policy translation? Before answering

this question, another one needs to be addressed. What does it mean to run an application with different user profiles? An application has a certain set of functions which can be invoked. A user profile cannot extend the set of functions of the application, it can only restrict the functions set. An administrator profile can use the entire set of functions whereas a user profile or a developer profile can only use a subset of functions. Therefore, if the domain model captures the entire set of the functions of the application, then the policy translation will not be affected because no new entities come into picture. Another example of this type of change is when a new backup server is deployed. The implementation of it is identical to the other servers but the data flow can be different. For more details please refer to subsection 3.4.

The fourth type of change regarding a system is an *application extension*. An application can have extensions, plugins, addons attached to it. A Mozilla extension added to the browser is one type of this possible system change. We consider only this type of modules which are attached to the existing system. An extension only reuses the existent core functions of the platform system (which are defined at ISM level) to provide new functionalities. In this case the policy translation process and enforcement does not change. If a module provides new functionalities this is considered a *new implementation* and treated as such. An example of this could be a new OS library.

Decide upon Changes The merging of two systems is similar with the merging of data and containers. At this point we remind the reader that a system consists of a name and a set of transformers, and the merging is done only at the same level of abstraction, see rule 1. At PSM an OS is merged only with an OS, not an ISM Windows, or vice-versa.

Because systems and transformers represent technical and implementation details of the domain model, the ***syntactic information*** necessary to establish an equivalence relation is provided only by the name of the element, equality of terms or a match between synonyms, which are alternative names. There is no ***structural information*** involved. A generic system at PSM is refined as a collection of ISM implementations. The refinement provides no additional information regarding the properties of the system. Also, there is no ***semantic information*** because the terms are not necessarily from the natural language, and a rather stable vocabulary is already assumed at this level, see rule 2.

Equivalent(System a, System b): *System a is equivalent to System b IF they have the same name, OR one's name is in the set of the other's synonyms.*

Incorporate Changes The merging is done by following the same steps introduced at data and containers merge, see algorithm 2. Comparing two elements of type system follows the same logic as in the case of containers, see algorithm 3. Two systems from the same layer of abstraction and the same name are merged by using the union set of the refinement and the union set of the transformers pertaining to that system. The union is obtained by making use of the *containment* relation between a set and an element.

Instance level Alongside the three levels of abstraction, PIM, PSM, ISM there is also a fourth level which appears only at runtime, labelled *instance level*. This is the level of the actual running system. For example, if we have a distributed context, one can link the change at the actual running system level. In this case, there is no need for a change in the domain model.

An example mentioned before was a back-up server and after a certain amount of time a second back-up server is added and later a third server is added. Does the policy translation system need to change? If the backup servers are a replica of the main servers, then the same domain model is be used. If the backup servers, differ in the implementation from the main server, then the domain model must include the implementation details of those backup servers. This is a case when an additional instance of an existing system is added to be monitored by the usage control infrastructure. If one has specified policies in terms of "data should not be copied more than x times" then the usage control infrastructure must keep track of the flow of data. For more details please refer to subsection 3.4.

2.7 State-based formulas

So far actions have been refined in terms of events, but [4] introduced also state-based operators so that actions can be specified also in terms of states a system must or must not enter. Examples of state-based operators are "data is not in container", "data is only in container", "data is combined with container". For each high-level action, the power-user can define the resultant state of the execution of that action as a *state-based formula* using state-based operators. The authors of [4] argue that actual data and their containers are known only when a policy is deployed in a concrete system, so the resultant state formulas are expressed in terms of potential data and their containers by introducing variables. Only variable data is needed, while potential containers are specified using PSM containers. Before an action is used to specify a policy, its state based formula is statically defined. The state-based refinements are not layer specific but action specific. Hence, they are not reflected in the domain model.

Recognize Changes Between two Usage Control infrastructures with regard to state-based formulas, there can be the following differences. Firstly, the same action can be defined by using synonyms when adding the state-based formulas. Secondly, the same state-based formula from different UC infrastructures defined for the same action can differ in the number of parameters on which the state-based operator is applied. Thirdly, for the same action, different number of state-based operators can be used in order to define a state-based formula.

Because the two domain models have been merged before and actions also have synonyms, the updated state-based operators can be applied to the correspondent action even if the same action is known by different names in different domain models. Having mentioned, see rule 3, that the base domain model is always updated and no elements are removed, it cannot be the case that an element from the state-based formula is not defined.

Decide upon Changes At this level an uncertainty factor which occurs is which state-based operators are defined for each action. If an action has a state-based operator "data is not in container x" and the new action has a state-based operator "data is only in container x", then this will result in a conflicting definition. Because we have assumed that the new domain model is always an update not a replacement, we reason that this should apply also in the case of state-based operators. The conflicting definition is signaled by the usage control infrastructure and in order to mitigate the conflict, a manual intervention is necessary to negotiate the definitions; in the aforementioned example, only one of the operators can be used.

Incorporate Changes In order to update the definitions of state-based formulas for actions, alongside the new domain model, a list of updated state-based formulas for actions needs to be sent to the translation component. For each action, the component responsible with adaptation adds a new state based formula to a specific action or updates the list of containers on which that operator is applied. Firstly, if a new action together with its state-based formulas is added, then this is just the addition of new elements to the list of state-based formulas. Secondly, if new state-based operators are added for an existing action, then as long as there are no conflicts between state-based operators, they are added to the list of operators for that particular action. This is equivalent with adding an element to a set, as was in the case of containers and their refinement. Finally, if only a parameter of an state-based operator is added, then this is also equivalent to only adding an element to a set. Now, the state-based formulas have also been merged.

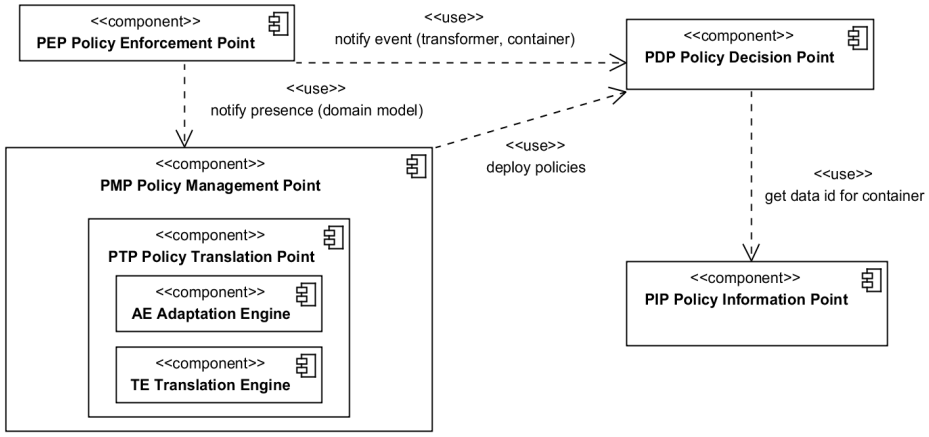

Fig. 3. Usage Control infrastructure

3 Implementation

In the literature [1], a generic usage control infrastructure consists of three main components: a *Policy Enforcement Point* (PEP), able to observe, intercept, possibly modify and generate events in the system; a *Policy Decision Point* (PDP), representing the core of the usage control monitoring logic; and a *Policy Information Point* (PIP), which provides the data state to the PDP. The generic infrastructure was extended with a *Policy Management Point* (PMP), which manages the UC policies, i.e. deploy, revoke policies.

To enable translation, the PMP was extended with another component, the *Policy Translation Point* (PTP). The main responsibility of the PTP is action refinement based on the domain model introduced before and as described in [4]. Please refer to Figure 3 for an overview of the architecture.

The PTP was also extended and now consists of two main components: a *Translation Engine* (TE) and an *Adaptation Engine* (AE). The specification language and the translation process of the TE have already been presented in [1,2,4]. In this work we introduced the Adaptation Engine (AE) which provides the adaptation and merging part for the model-based policy translation.

The domain model has been presented as containing the details of all the systems in a single location. If one translates policies only for single systems, then the details of systems can also be kept in separate locations. On the other hand, if one wants to translate policies which affect multiple systems and cross-layer scenarios, then the translation needs to have simultaneously access to all the systems because elements from different systems might need to be included together in the same refinement.

3.1 Initial conditions

In an adaptive system environment one of the initial challenges is how to recognize change in the environment, in our case how to recognize that the domain model has changed. The first approach to recognize it is in an active manner, labelled *active*, meaning that the Usage Control (UC) infrastructure detects a new component and adds its domain-model to the UC infrastructure. We recognized that this approach is not feasible because, for example, on a machine there are some components which are installed and are not subject to the UC surveillance, such as Windows updates. Other components that might be malware in nature are not detected upon installation. Also the UC infrastructure would have to broadcast a message and then listen for replies from newly added components. This can impose performance penalties. The second approach, labelled *passive*, assumes that each PEP component comes with its model and it informs the system about its presence by sending its domain model as a parameter of the notification. Upon notification, the UC infrastructure incorporates the new elements which the new domain-model might have.

When a new PEP is added which also comes with a domain model, the PEP will notify to the PMP its presence and will send the new domain model and the list of updated state-based formulas. The new domain model always contains

a three-layer model. At the PSM and ISM level, the power-user defining the domain-model must adhere to the common vocabulary for known systems or use synonyms. At PIM level one can provide synonyms for data and actions in order to more easily negotiate an agreement between terms. The core task of the Adaptation Engine, part of the PTP, is to recognize the differences between two domain models and include them in the base domain model.

One has to consider also the cases when a domain model already contains the elements presented in the new domain model. One approach is to override the existing elements with the new elements. We have already seen that this approach can lead to a loss of information during refinement. The second approach is to always include all the differences. This has the disadvantage that the base domain model will increase in size rapidly and will end up containing deprecated elements. But this approach has the advantage of backward compatibility. If policies are translated based on this model and later enforced on an older implementation, then they will still be correctly enforced if a UC infrastructure has two versions of the same system installed at the same time. This is the reason behind rule 3.

The next step is to clearly define the initial conditions. The Usage Control infrastructure can start in one of two cases regarding the PTP. First, there is no initial model. In this case, the first installed PEP that notifies its presence will also deliver its domain model and this domain model will become the Base Domain Model for the entire infrastructure. All additional PEPs when they will notify their presence to the infrastructure will have their domain model merged with the Base Model. Secondly, the UC infrastructure has an initial model. In this case, an administrator would have already defined an initial model for the infrastructure. All additional PEPs' Domain Models will be merged with the Base Model. This is the most common use case.

3.2 Adaptive behaviour

We have already mentioned that in order to provide automatic adaptation not only the domain model needs to be updated but also the policies. This can be done with two approaches, we named them *delayed* and *real-time* adaptation.

The first approach is *delayed adaptation*. This means that until the UC infrastructure is restarted, the old policies are maintained, although if new policies are translated, the new domain model is used for translation and policy enforcement.

The second approach is *real-time adaptation*. This means that after the domain model is merged, the old policies are first revoked, retranslated and deployed. New policies will be translated according to the new domain model. To achieve this the behaviour of the PMP was extended. Now, the PMP maintains a list with all the deployed policies and all the information necessary to automatically request a new translation from the PTP of a particular policy. All the policies in that list are first revoked, then they are translated according to the new model and deployed to the PDP. For each policy is maintained information related to the syntax of the policy before translation, instantiation parameters and data ids generated at deployment.

Another key aspect one has to keep in mind is the state of the PDP. For each protected data the PIP keeps a track of all the containers in which the data resides. The PDP also keeps a track of the entire history of the invoked transformers. In order to make a decision, the PDP queries the PIP regarding the protected data inside a specific container and also queries the trace of invocations in order to detect a possible violation of the policy. A deployed policy talks of implementation details and for each container there is a data id which is used to mark a container as containing sensible information. When a policy is revoked, retranslated and redeployed, this information has to be restored by the PMP as described before.

If a new policy specified with the actions and data from the new domain model is submitted to be translated, then the PTP after the merging of the two domains will be able to translate the policy according to the base domain model. The new terminology has been added to the base domain, so a new term is either a new element or a synonym of the existing element.

3.3 Lexical similarity

At PIM level data reflects entities from the user's business environment, the data one wants to protect e.g. profile, blog, comment, budget, salary, income, expenses, insurance number. In this case a simple string matching is not sufficient, because one domain can have a data named "picture" and another one "photo". The adaptation engine has to recognize these two data as one and the same. For this reason, we have used a natural language lexical library, namely RitaWordnet [13–15] to establish relations between terms. With this approach one can immediately establish relations between two terms.

The second challenge arose with those terms that talk of the same data with unrelated terms. For example, if one social network has the concept "wall" while another one has the concept of "mirror" and these two networks merge, then there is no direct linguistic relation between these terms. The updated domain model will use both of the terms.

The synonyms provide a vocabulary negotiation mechanism between two end-users who want to use the same domain model for their policy translation but still want to use the terms with which they are most accustomed to. Synonyms are either provided in a static manner as a property of an element of the domain model, or synonyms are discovered in a dynamic manner while merging by the use of a lexical similarity comparison of terms.

To compute lexical similarity between two words, RitaWordnet library offers a multitude of algorithms to choose from to obtain a numerical value between two concepts. We used the standard metric which takes into consideration the common linguistic root of a word and similar meaning to compute the value. Between two words, e.g. "picture" and "photo", a normalized distance value is computed. If the computed value is closer to 0, e.g. 0.1, this means that the words are synonyms from a natural language perspective. If the computed value is closer to 1, e.g. 0.7, this means that the words are not related.

Taking into consideration the method used to compute the distance and by using experimental methods we have also used a distance threshold of 0.2 for nouns from which the concepts can no longer be considered similar. Example values are: photo-picture 0.0, photo-car 0.44, photo-child 0.5, picture-album 0.22, document-file 0.22, directory-folder 0.16, copy-replica 0.11.

When a lexical similarity is discovered between two elements, the words are stored as synonyms for those specific elements. Therefore, in a future comparison the synonyms can provide a performance improvement by avoiding the need for a search in a dictionary.

3.4 System instance

At the instance level, there is case when the domain model is not changed but a new instance of the same system is deployed. For example, a new backup server is added to the network. The backup server has the same implementation details as the main server. The PEP of this instance will notify its presence but the domain model has no new elements to be included. The difference is where the protected data is stored. In order that the UC control works correctly, the backup server must be populated with data via a monitored channel. This means that the protected data is tracked by the UC infrastructure and the containers used on the new server to store the data are also traced by the UC infrastructure. This monitoring has to be done at the Policy Information Point (PIP). If the system has a global PIP then it can maintain a map of all the allowed backups and all the containers where data may reside. If the PIP is not global but local, then the policies which talk in terms of "do action max/min number of times" will not be enforced correctly if one access once the main back-up server, than the second back-up server and so on. This is avoided by using a centralized PIP. Another option would be to pass the state of the data from one instance to another so that the PDP can correctly verify if the conditions specified in the policies are fulfilled. If the PDP is also local, its state must be restored.

4 Evaluation

One can install a usage control infrastructure in multiple environments. The first environment is an open system. This means that numerous third-party systems can be installed and have access to the data. For example, an online social network can be extended with numerous plugins which extend the user experience, such as the case of Facebook and third-party apps. In order to restrict the access of such apps to the data, they must conform to the policies. A second case of environment is an open-closed system, such as a company. One can install software only from a repository.

From a usage control perspective there is no difference between open systems and open-closed systems. If an application needs to be submitted to usage control surveillance then it must come with a PEP module and a definition of its domain model. If the application uses another's application framework functions, e.g. a

file explorer uses operating system calls, then the enforcement can be maintained at the level of those functions.

A first is an excel application which was extended with a PEP. Bob has some sensible documents on which he has specified some policies. Alice works at another company and their companies will use a shared environment. The companies want to merge their usage control infrastructure but not change the internal terminology used to specify policies. Before merging they agree for data and actions on a common terminology specified as synonyms. After merging, when Alice logs in to Bob's computer the policies specified for Bob's file are enforced correctly. When Bob tries to access Alice's protected files on the same computer, then Alice's policies for files are enforced correctly. On the other hand, if Alice installs another version of excel which has also a PEP, then at the first request to access a sensitive data, the PEP notifies its presence, the base domain model is updated and the policies deployed by Bob are retranslated and enforced also for the new application.

The second scenario is with regard to a social network environment. Bob specified usage policies on his pictures on his profile. In order to receive access to that data, Alice must have a browser PEP in place. The first time she access Bob's data, the policies are enforced at the browser level. In the background, the PEP notified its presence, the domain model was updated to include the browser, then the policies were translated according to the new model and deployed. In the meantime, an OS PEP was also installed. In the background the same steps are taken so that when Alice logs again on Bob's profile, the deployed policies are now specified also at the OS level. For example, if she wants to access the cache where a sensitive data is stored, the PDP must grant permission according to the policies.

4.1 Performance analysis

Usage Control infrastructure solutions can cause a significant performance overhead which leads to a bad user experience. Therefore, we measured the behaviour in terms of relative computation time overhead. The machine on which the tests were run was configured as follows: Intel i7-2640M 2.8ghz, 8Gb Ram, Windows7 64bit, SSD Intel 160Gb.

The test policies were generated starting from seven templates which were instantiated by varying different parameters such as "number of times", "action", "subject", "time amount". After combing all the relevant possibilities for instantiating a template, we have came up with 28 unique policies. A template can be instantiated multiple times and the same policy can be deployed multiple times for different data. In order to replicate this real case scenario, we have duplicated the policies so that we used 160 policies as fix set of policies for evaluating the performance. We considered that 160 policies are sufficient to replicate a sufficiently realistic scenario. Using 160 policies can mean that a user has defined policies for 160 unique sensitive data.

The approach we took was first to measure the performance per component. The first evaluated component was the translation engine (TE), part of the PTP.

The time required for the translation of a policy depends on the complexity of the domain model. If an action is refined only for a browser system, then it will be translated faster than an action which is refined also for an operating system. The base domain model used for the translation of the 160 policies had defined the browser system and the operating system both at PSM and ISM. Data, containers, actions and transformers formed in total 54 elements which had to be evaluated. The time required by the TE to translate 160 policies was an average of 11.4 seconds.

The next evaluation took into consideration the translation and deployment of those policies. Deploying a policy on the PDP involves also some additional processing on the PDP side. The policy is checked for syntax correctness, then for each container is assigned a generated id which is used to tag a container as containing sensitive information. The time required by the PMP to translate and deploy 160 policies was in average 24.3 seconds. We have not considered only the time required by the PMP to the deploy the policies because of implementation details. The deployment module is multithreaded and can perform better when it needs only to deploy. On the other hand, the translation engine acts as a limiting factor and the JVM optimizes the performance in a different manner when translation and deployment are used together.

Operation on 160 policies	average time
1. Translate	11.4 seconds
2. Translate+Deploy	24.3 seconds
3. Adaptation	0.04 seconds
4. Adapt+Revoke+Translate+Deploy	24.38 seconds

Table 1. Performance overhead

The third step was to evaluate the adaptation engine (AE). The performance overhead also depends on the complexity of the domain models, because every element of the domain model has to be compared to its counterpart. For example, the base domain model consists of 42 elements, the new domain model consists of 43 elements, and the merged domain model consists of 54 elements. In total 54 elements were affected by the update and 12 elements were added to the base domain model. The average time required for this to take place is 0.4 seconds. This approach is a stress-test of the adaptation engine.

The final step, is to measure the end-to-end performance. We refer to end-to-end performance as the time perceived by an user from the moment the request to adapt the domain model is sent to the moment when the updated policies are deployed and ready to be used. In the second scenario, after Alice installed a new PEP this is the time perceived by her after she sends the first request to access a sensitive data and the time she is granted access or denied. In the background, the PEP sent an adaptation request, the deployed policies were revoked, all 160 policies were retranslated and redeployed. The average time required by the UC

infrastructure was of 24.38 seconds. This time also includes the communication overhead induced by the communication between the PEP and the PMP.

Considering the results, see Table 1, the following observations can be made. First of all, the translation process is one of the major limiting factors of the infrastructure. Optimizing the translation ends in improving the user experience. Secondly, the time required by the adaptation increases depending on the number of the elements of the domain model. If the domain model consists of thousands of elements, the time will be more significant because every element must be compared to its equivalent. On the other hand, adaptation takes place only when a new PEP is installed and new PEPs are installed less frequently compared to adding or revoking policies. This means that the time required by the adaptation can be considered the time required when a newly configured system first starts up. It is a common case that systems require more time when they first start. Thirdly, in scenario two in the context of a social network, the end-to-end time appears only at the first access of the data, not every consequent access. A good analogy would be that the end-to-end time is the time required for a ticket machine to issue a ticket, once the ticket is issued the performance overhead is only that introduced by checking the ticket at each access point, in our case the PDP verifying the policies.

In conclusion, an adaptive model-based policy translation introduces an insignificant performance overhead, whose cost is relatively minor compared to having static model-based policy translation.

4.2 Verification and Validation

The verification of the adaptation engine (AE) was made by configuring unit test cases for each of the merging cases presented in this report. Individual test cases were added for data and containers, action and transformers, systems and state-based formulas. Combinatorial testing method was used in order to generate the test cases. For example, in the case of transformers the input vector was composed of properties such as the number of input or output parameters, the type of refinement, the signature and the name of an element. For all elements in total there are more than 10 properties distributed over more than 6 main unit tests. Each test case is constructed in such a way that only merging cases are verified without the potential influences from other elements. For every test, there is a base domain model, a new domain model and an expected domain model with a known number of elements. After the base domain model is merged with the new domain model, the expected number of elements is verified for each layer. Additionally, the merged domain model is updated again with the expected domain model and the number of updated elements must be 0.

Validation was done with the following motivation behind. The adaptation engine acts as a transformation function which takes as input domain models M_i and M_j, and merges them to M_k, i.e. $M_i \oplus M_j \rightsquigarrow M_k$. We have observed that the properties that this function verifies are commutativity and associativity. Commutativity means that $M_i \oplus M_j = M_j \oplus M_i$ and associativity means that $(M_i \oplus M_j) \oplus M_j = M_i \oplus (M_j \oplus M_k)$. Another observed property is that for a series

of transformations $M_i \oplus Mj \oplus ... \oplus M_k \rightsquigarrow M_n$, there is new domain model M_y such that $M_i \oplus M_y \rightsquigarrow M_n$. The aforementioned properties are verified by using test cases which make use of one ore more of these properties and compare the merged domain model with an expected domain model. Because elements which define sequences can have different names, the merged domain model and the expected domain model need not be identical. One domain model can have an inner refinement with three inner links, while another one can have only one inner link. In the end only the final ism transformers will appear in the translated policy, therefore the translated policies must be identical. Therefore, the comparison of two domain models can be done also by comparing the translated policies.

4.3 Limitations

The data from a domain model can be specified with business domain terms which are not recognized by a lexical ontology. Therefore, a specific use case ontology should be put in place. This is a limitation introduced by the use of an external lexical library. We have not taken into consideration domain model elements which are labelled with composed terms, e.g. UserProfile, Projectportofolio, updateUserTable. A composed term cannot be recognized "as is" by the lexical library.

A second limitation is introduced by the fact that the domain model is always updated and backward compatibility is maintained. This means that the base domain model always expands. In case a domain model must be simplified, this must be done either manually or by using a special notification signed by the administrator to mark that the new domain model must be accepted as a replacement for the elements it contains.

A third implementation limitation is with regard to the signature of a transformer. A new domain model can contain a transformer with the same name in the base domain model but with a different signature. In this case, the transformer is also added to the base domain model. It is possible that in the base domain model, there are sequences which are specified in terms of the initial transformer. After another signature of the same transformer is added, then another sequence should be added to include also the new signature. For example, a transformer "open" which accepts as input a "url" was added to the domain model. A sequence s1 was already defined which used the transformer "open" that accepts as input a "file". A sequence s2 can now be defined the same as s1 with the exception of using the "open" transformer applied on a "url". On the other hand, if two transformers have different signatures it is very common that they have a different implementation, for a example addition defined for strings or integers. Therefore, the adaptation engine cannot generate automatically a similar sequence which uses an alternative signature of a transformer. The sequence which uses the new signature will also have to be specified in the new domain model.

5 Conclusions and Future Work

In this work we introduced a complete usage control framework which supports adaptive model based policy translation and enforcement. The problem which was addressed was the dynamic nature of systems and the necessity of the usage control infrastructure to react in an adaptive manner to these changes.

Furthermore, we have shown that the cost in terms of performance is acceptable compared to the need of an adaptive infrastructure. This is the first usage control adaptive solution that we are aware of. This solution consists of a conceptual but also a structural implementation. It includes every step from a policy specification in user-friendly manner, translation of the policy with implementation details and runtime adaptation when a system changes.

We have seen that one of the uncertainty factors related to merging domain definitions is the unknown context in which a refinement is used. It can be used as a preventive or a detective mechanism. In terms of future work, one step further would be a context dependent translation and merging. Another uncertainty factor was the absence of domain ontologies for different business domains, e.g. education, business, legal system. Designing such ontologies can server as a basis for using synonyms in order to negotiate the meaning across different systems. Another further step is to design system domain ontologies which contain different templates of transformers or sequences that can be used in the case of an incomplete definition of a domain model.

The contribution of this report was to show adaptive model-based policy translation for usage control from a theoretical point of view which was implemented in a real use-case scenario without the user experience being severely affected by the use of such an infrastructure.

6 Related work

The goal of this work is to provide an adaptive model based translation for usage control policies. General refinement was introduced in [20, 21] while policy refinement has been approached in [7] by goal decomposition, using process algebra in [19] and preserving security in [22]. In [6] ontology-based techniques are described for semi-automated translation of access control policies. These approaches were helpful in better understanding uncertainty factors from a security perspective.

A domain model framework for adaptive web systems was introduced in [10]. Different types of relations have been established between the elements in the domains one wants to merge. Our domain system is dynamic and the model based translation must be adaptive to the new domain models. These relationships between domain model elements have been introduced also in our context.

In [8] formal concept analysis and external lexical information is used in order to merge two ontologies. The authors use a bottom-up approach in order two merge the ontologies. Their adaptive architecture is similar with the one used by the Adaptation Engine in this work. New components are added to the usage

control inftrastructure with their own domain model, which in [8] is portrayed by the local ontology. In our context an ontology is a domain model. The merged ontology is the merged domain model which will be used by the Translation Engine.

In [23], methods that used for the definition, adoption, and utilization of element similarity measures in the context of XML schema matching are classified, reviewed, and experimentally compared. These methods employ syntactic and semantic similarity measurements. In [24] string similarity metrics for ontology alignment are introduced. In our work, we have used tactics from these approaches in order to define the equivalence relation between elements. No direct XML schema matching or ontology merging strategy could have been employed because the domain model has meaningful information both in the structure as well as in the terminology used. Therefore aspects from both matching and merging have been adopted.

The authors of this report are not aware of any repository with business specific ontologies. For example, an ontology with terms from computer industry, or legal departments or educational institutions can be employed to more easily establish the negotiation between two domain models vocabulary.

References

1. A. Pretschner, E. Lovat, M. Buechler: Representation-Independent Data Usage Control, in *Sixth International Workshopon Data Privacy Management*, pp.29, 2011
2. M. Hilty, D. Basin, C. Schaefer, T. Walter, J. Biskup, and J. Lòpez : A Policy Language for Distributed Usage Control, in *Proc. 12th European Symp. on Research in Computer Security*, vol. 4734, pp.531-546, 2007
3. Pretschner, A.: An Overview of Distributed Usage Control, in *Knowledge Engineering: Principles and Techniques Conference*, pp.25-33, 2009
4. P. Kumari, A. Pretschner: Model-based Usage Control Policy Derivation, *Proc. 5th International Symposium on Engineering Secure Software and Systems (ESSoS)*, pp.58-74, February, 2013
5. P. Kumari, A. Pretschner: Deriving Implementation-level Policies for Usage Control Enforcement, *Proc. 2nd ACM Conference on Data and Application Security and Privacy (CODASPY)*, pp.83-94, February, 2012
6. Guerrero A., Villagrá V., López de Vergara J., Sánchez-Macián A., Berrocal J.: Ontology-based policy refinement using SWRL rules for managementinformation definitions in OWL *Proc. IEEE Distributed Systems: operations and management*, 2006
7. Bandara A., Lupu E., Moffett J, Russo A.: A Goal-based Approach to Policy Refinement, *Proc. IEEE Policies for Distributed Systems and Networks*, pp.229-239, 2004
8. Stumme G., Maedche A.: Ontology Merging for Federated Ontologies for the Semantic Web, *Proc. Intl. Workshop on Foundations of Models for Information Integration*, 2001
9. Betty H. C. Cheng, P. Sawyer, N. Bencomo, J. Whittle: A Goal-Based Modeling Approach to Develop Requirements of an Adaptive System with Environmental Uncertainty, *Proceedings of the ACM/IEEE International Conference on Model Driven Engineering Languages and Systems (MoDELS 2009)*, pp.468-483, 2009

10. Wenpu Xing; Ghorbani, A.A.: Information domain modeling for adaptive Web systems, *Proc. IEEE/WIC/ACM International Conference on Web Intelligence*, pp.684-687, 2005
11. Betty H. Cheng, R. Lemos, H. Giese et al.: Software Engineering for Self-Adaptive Systems: A Research Roadmap, *Software Engineering for Self-Adaptive Systems, Springer*, pp.1-26, 2009
12. R. Lemos, H. Giese, M. Shaw et al.: Software Engineering for Self-Adaptive Systems: A Second Research Roadmap, *Software Engineering for Self-Adaptive Systems : Lecture Notes in Computer Science Volume 7475, Springer*, pp.1-32, 2013
13. George A. Miller: WordNet: A Lexical Database for English, *Communications of the ACM Vol. 38*, No. 11, pp.39-41, 1995
14. Christiane Fellbaum: WordNet: An Electronic Lexical Database, *Cambridge, MA: MIT Press.*, 1998
15. RiTa.WordNet: Java library to access to the WordNet ontology, *http://rednoise.org/rita/index.html*
16. Ontology Matching, *Springer, 2nd. ed*, 2013
17. M. Szlenk: Formal Semantics and Reasoning about UML Class Diagram, *Proc. IEEE Dependability of Computer Systems*, pp.51-59, 2006
18. Ji Zhang, Betty H. C. Cheng: Model-based development of dynamically adaptive software, *ICSE*, pp.371-380, 2006
19. Bossi A., Piazza C., Rossi S.: Action Refinement in Process Algebra and Security Issues, *Logic-Based Program Synthesis and Transformation, Springer-Verlag Berlin*, pp.201-217, 2008
20. Reeves S., Streader D.: General Refinement, Part One: Interfaces, Determinism and Special Refinement, *Electronic Notes in Theoretical Computer Science 214*, pp.277-307, 2008
21. Reeves S., Streader D.: General Refinement, Part Two: flexible refinement, *Electronic Notes in Theoretical Computer Science 214*, pp.309-329, 2008
22. Bossi A., Piazza C., Rossi S.: Preserving (Security) Properties under Action Refinement, *Convegno Italiano di Logica Computazionale, CILC Parma* , 2004
23. Algergawy A., Nayak R., Saake G.: XML Schema Element Similarity Measures: A Schema Matching Context, *On the Move to Meaningful Internet Systems: OTM*, 2009
24. Cheatham M., Hizler P.: String similarity metrics for ontology alignment, *12th International Semantic Web Conference*, 2013

A Appendix

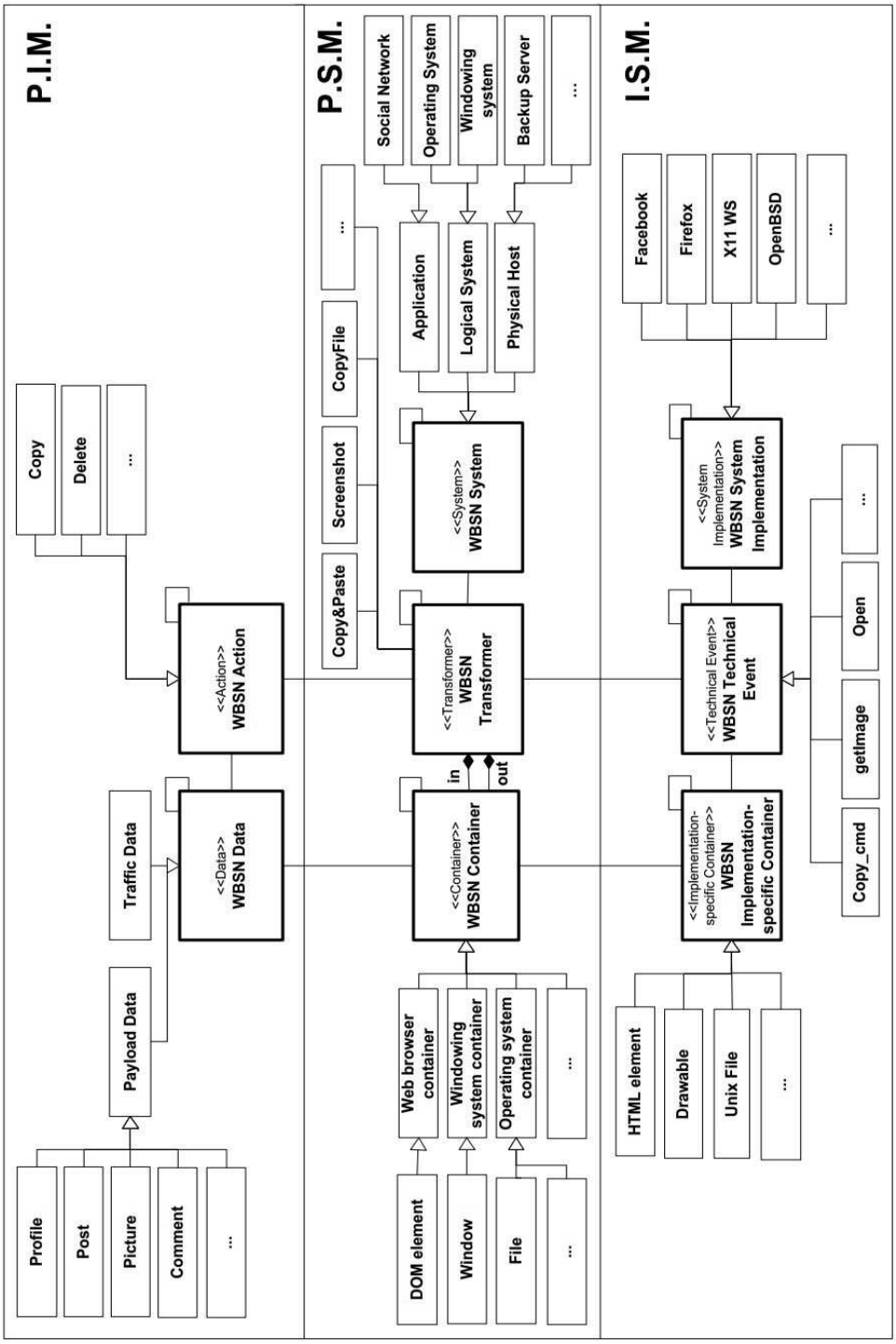

Fig. 4. Online Social Network - domain model

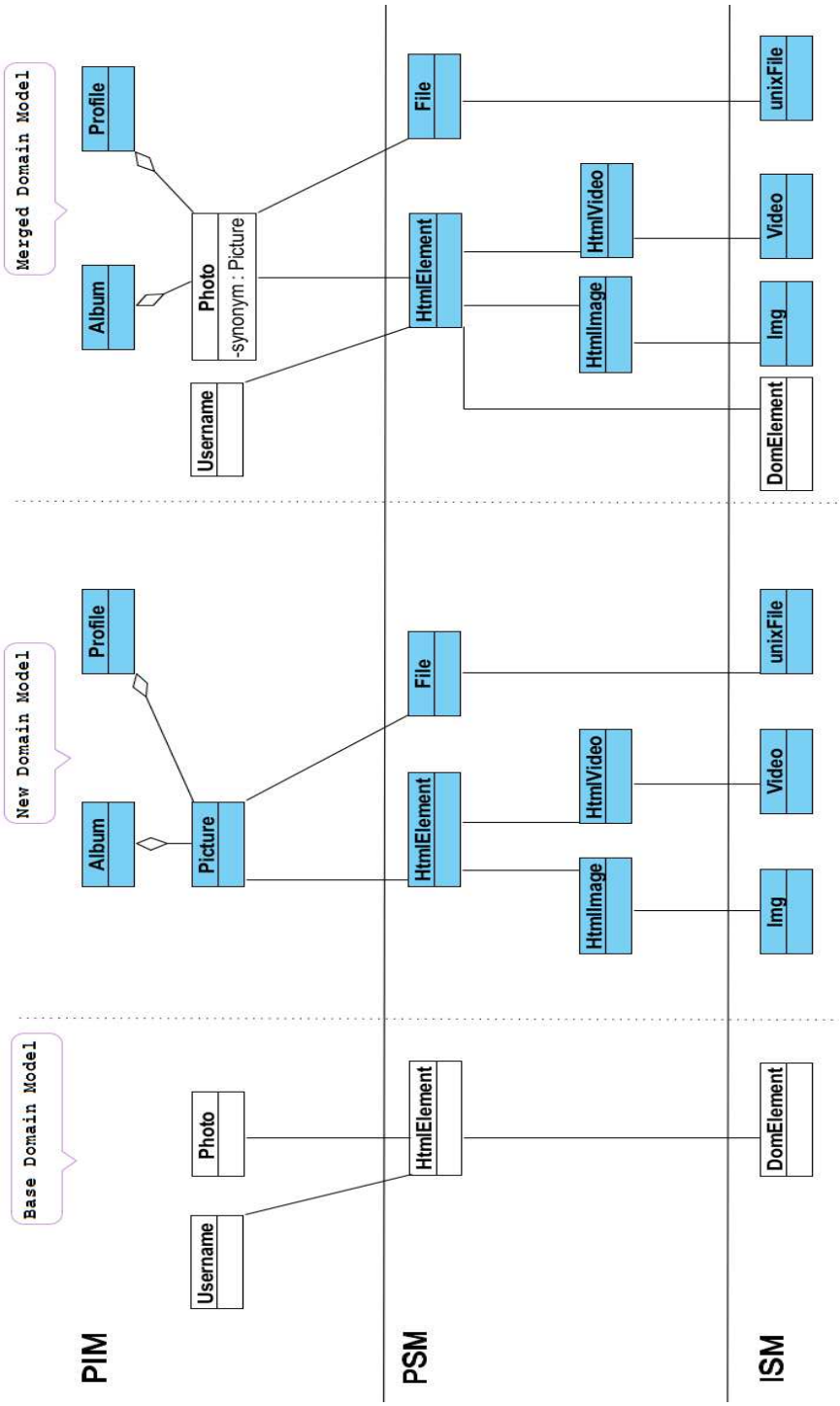

Fig. 5. Data and Containers merge

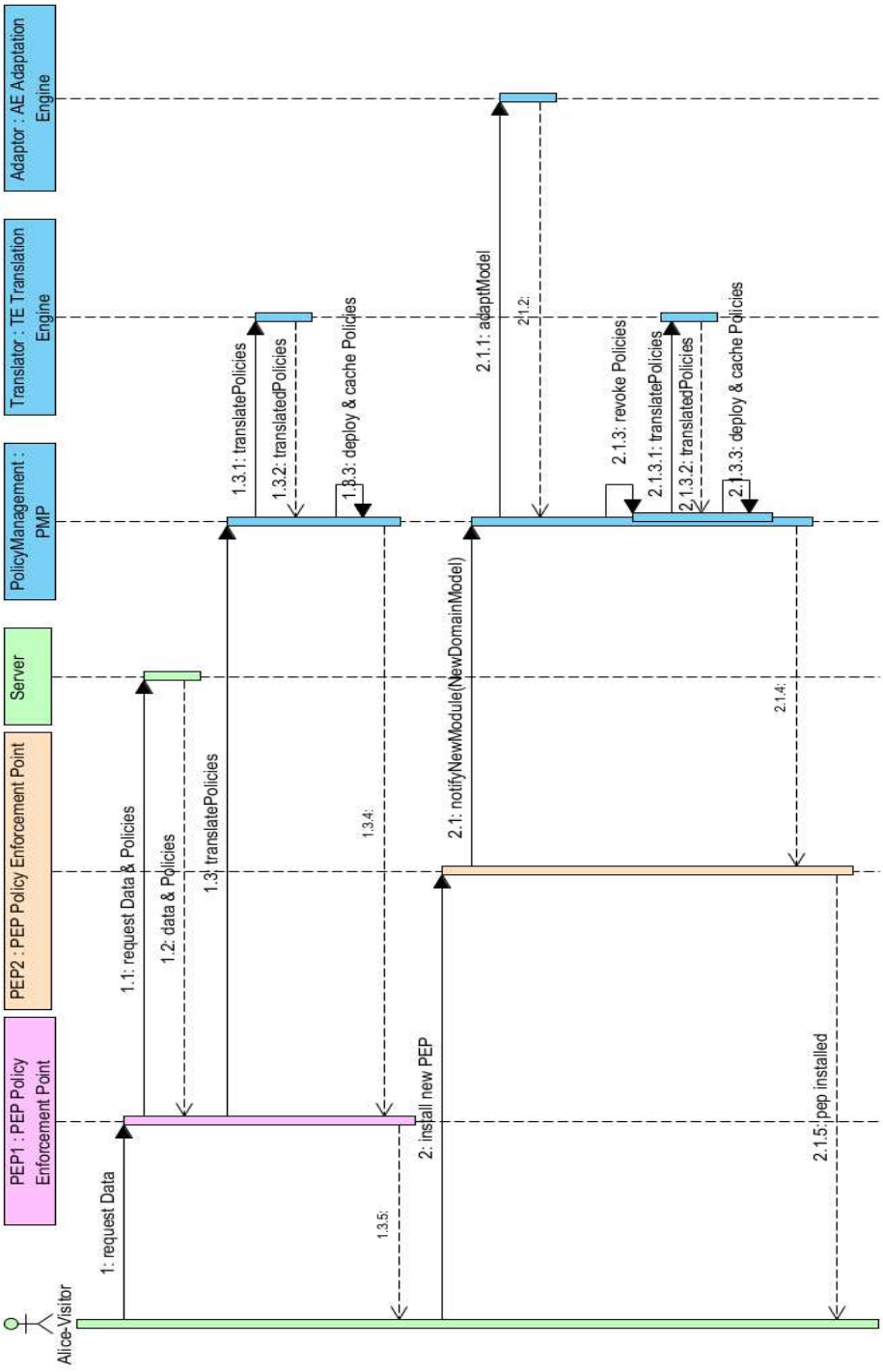

Fig. 6. Deployment and adaptation sequence diagram