Autonomous Applications – Towards a Better Data Integration Model

András Benczúr[1], Zsolt Hernáth[1], and Zoltán Porkoláb[2]

[1] Eötvös Loránd University, Faculty of Informatics
Dept. of Information Systems
Pázmány Péter sétány 1/C H-1117 Budapest, Hungary
abenczur@ludens.elte.hu hernath@ullman.inf.elte.hu
[2] Eötvös Loránd University, Faculty of Informatics
Dept. of Programming Languages and Compilers
Pázmány Péter sétány 1/C H-1117 Budapest, Hungary
gsd@elte.hu

Abstract. One of the most important and critical part of integrating already existing standalone applications is to design and implement a common data model and the corresponding data access layer which makes both data sources and processed results being shared and accessible over the applications in question. In case of even well-architected applications or application systems, establishing a common data model and the layer that gives access to data costs relatively large human and computer development resources. The problem of integration may be investigated from several aspects. The esence of these aproaches are the same: trying to achieve run-time environment independent applications' logic. One aspect is OMG's Model Driven Architecture Frame Work [5]. The primary goals of OMG's MDA are portability, interoperability and resuability through architectural concernes of specifying Application's logic, their operational environments, and technical aspects of their implementation details, and mappings between them. This paper views the same problem but with focus on different structural apearances of applications' data, mappings between them, and possible integration of such data models. We call applications autonomous, if they are independent of their all time run-time data access environment. Concerning applications' autonomy, our base idea is that the most natural media that is able to carry information on structural apperance of data and mappings between them are data themselves, using Document Data Model also presented here.

1 Introduction

Applied computer science and informatics by now plays a central role in our everyday life. The possibility of getting scientific, technical, business and quite general everyday on-line information accessing the World Wide Web database, together with having personal computers and Internet access possibilities at reasonable prices has changed our everyday life. In professional life, beyond information's access, processing and integrating information coming from different fields and from heterogeneous data sources is more important, so that, information processing needs integrated application systems, rather than standalone applications. Information industry vendors provide for various integrated application and information processing systems, some of them offer much less, than what could be many times achieved by bringing together already existing and efficiently used standalone applications over the same or different technical fields. There are lots of already existing and still efficiently used standalone applications, and some of them are reasonable to be brought together with others.

Figure 1 below shows two standalone applications with heterogeneous data sources, Figure 2 and 3 present two versions of an integrated system consisting of the standalone applications on Figure 1. As it is seen, the integrated system on Figure 2 does not implement a unified common shared data model. The only thing that happened is that a common I/O layer is established in order to make bidirectional mapping, and even cross mapping between data source models and application data models. In the practice the integration middleware implements an interface to the *union* of the applications' domains. The integrated system on Figure 3 provides a unified common application and data source model, and a common unified I/O layer as well, for the integrated application system.

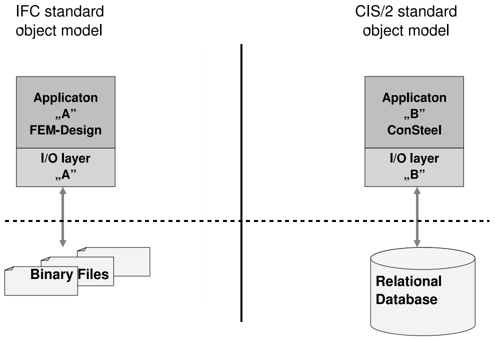

Fig. 1. Standalone applications

A significant part of the necessary knowledge to perform such conversions are data mappings that is only partly application-specific, yet some of those (e.g. retrieval and storage of particular pieces of data) - even in well-architected applications as well - is sometimes hard coded. In contrast of all above, we call applications autonomous, if they are independent of their all-time run-time data access environment. The idea of the notion of applications' autonomy has in fact been set as a generalization of an XML document based common data model - called Document Data Model - as one of our proposals for a real industrial need of integrating the two standalone CAD applications shown in Figure 1.

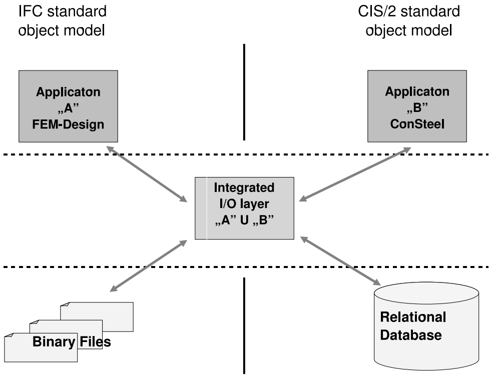
IFC standard object model
CIS/2 standard object model
Applicaton „A" FEM-Design
Applicaton „B" ConSteel
Integrated I/O layer „A" U „B"
Binary Files
Relational Database

Fig. 2. Traditional solution 1. – a kind of gluing together.

The two standalone applications has been designed and partly implemented in heterogeneous run-time data environment, and even, they established different application object models according to different standards: CIS/2 and IFC standard object model [1–3]. Though the two standards are in relation to each other, they have been defined for different goals. Instead of establishing a common object and e.g. a relational storage model and its middleware for the integrated application system, our proposal was a DOM (Document Object Model) based middleware (object cache), and an XML document based mapping between CIS2 and IFC as well as the corresponding storage environments [4, 6]. To support the implementation of the DOM based middleware, a non-full EXPRESS parser [1] has been implemented that, from EXPRESS schemata descriptions, generates C++ class skeletons and necessary type definitions.

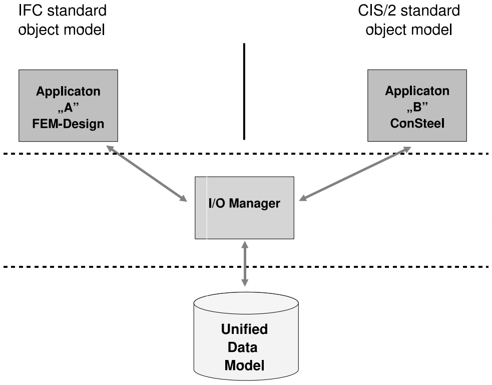
IFC standard object model
CIS/2 standard object model
Applicaton „A" FEM-Design
Applicaton „B" ConSteel
I/O Manager
Unified Data Model

Fig. 3. Traditional solution 2. – another kind of gluing together.

Autonomous applications' data access from and to their all-time run-time data environment is modeled by applying mappings between abstract domains, the semantics of those is defined by real run-time domains and data access engine. Concerning applications' autonomy, our base idea is that the most natural, and probably the most competent media that is able to carry information on semantics of abstract domains and mappings between them are data themselves.

Any application in general can be considered as sequences of optional data retrieval from its run-time environment, processing data, and optionally displaying or storing/restoring processed data. We would like to outline, that from the general perspective above the word data is used in a quite general sense, it may refer to values of particular domains including numbers, letters, pictures, events, processes and also processors. According to the above, an application may formally be viewed as a 3-tupple $A(D, E, \alpha)$, where D is a strictly application-specific non-persistent data model - a set of containers of data of atomic and/or constructed and also constrained types - called application cache memory, E is called run-time data environment that in general implements persistent application data, and $\alpha : D \times E \mapsto E$ is a mapping, called application logic. In the above model α implements all necessary non-application-specific knowledge about accessing data from and to E. To purge away all non-application-specific knowledge concerning data access from the application logic, we develop a finer and more specific model that basically concerns mappings between two particular domains: one that models industrial, scientific or business technologies - called *schema*, and another one that models a set of data carrier means - which together is called a *medium* - some of them preserves data, others makes data visible in some particular form. In our model, mappings between the schemata and media are also considered as an abstract domain called *access environment*.

2 Schema, Medium and Abstract Access Environment

Assume, there is a finite set $\mathcal{T}_a$ of named (i.e. uniquely identified) atomic (i.e. unstructured) types, a finite set $\mathcal{C}_\mathcal{S} = \{$*device, file, record, field*$\}$ of what are called *container sorts*, and a finite set of rules by means of which complex data types and container sorts can be composed. Assume further on, that for each atomic data type, there is a value *null*, and that atomic data types are ordered sets, that is, two instances of the same atomic type are comparable. For any atomic data type, an instance of value *null* indicates that no particular value is instantiated. Whenever we speak about *data types*, we always think of finite subsets of infinite sets of particular values, and whenever we speak about *data* or *container sorts*, we think always of some structuring of complex values or containers. For instance, if *int* $\in \mathcal{T}_a$, *int* is a finite subset of the set $Z = \{0, +1, -1, +2, -2, \ldots\}$ of whole numbers. Staying still by this example, *int* is an unstructured or *raw* type. Opposite to constructed types, there is no way to refer to a "part" of an instance of a *raw* type. For example, there is no way to refer to a value of 23 as part above 100 of the value of 123. In the case of constructed types particular parts of data instances can be referred to according to the *sorts* of such types.

2.1 Schema

Data types are to define applications' internal data model, also referred to as application domain. For our would-be model only a minimal data type system – just for illustration – is developed as follows:

(a) members of $\mathcal{T}_a$ are data types;

(b) if for each $j \in [1, m]$ for some natural number m, x_j are distinct (attribute) names and t_j are (named) data types, then $t < K_P \Rightarrow [x_1 : t_1, \ldots, x_m : t_m] >$ is a data type named t, and called *entity*. K_P is a (possible empty) subset of the set $\{x_1, \ldots, x_m\}$, keeping this order, and is called *primary key constraint*. Sometimes we use the notation $t.y$ to refer to an attribute named y of a type named t;

(c) if t is a data type, then $v\{t\}$ is a data type named v, and called *bag*. An instance of type $\{t\}$ is a bag of instances of type t;

(d) given the data types $t_1, \ldots, t_m$, $u[t_1, \ldots, t_m]$ is a data type named u, and called *union*. An instance of type u is an instance of any one of types $t_1, \ldots, t_m$.

(e) given a finite set of named types $\mathcal{T}$, a finite set of *referential constraints* $\mathcal{C}$, a pair $S(\mathcal{T}, \mathcal{C})$ is a data type named S, and called schema. A *referential constraint* is of form $\phi \Leftarrow \{\psi_1, \ldots, \psi r\}$, where ϕ is a *parent term*, and ψ_j for each $j \in [1, r]$ is a *child term*. Given $t < K_{p_t} \Leftarrow [x_1 : t_1, \ldots, x_{m_t} : t_{m_t}] >$, a *parent term* for type t is K_{p_t} itself of the form $\{t.x_{i_1}, \ldots, t.x_{i_{n_t}}\}$. A *child term* for the *parent term* above, and for type $u[K_{p_u} \Leftarrow [y_1 : u_1, \ldots, y_{m_u} : u_{m_u}] >$ is of form $u[\varrho_{m_{u_1}} y_{m_{u_1}}, ..., \varrho_{m_{u_s}} y_{m_{u_s}}]$, where $\{y_{m_{u_1}}, ..., y_{m_{u_s}}\} \subseteq \{y_1, \ldots, y_{m_u}\}$, $s \leq n_t$, and $\varrho_{m_{u_l}}$ for each $l \in [1, s]$ stands for one of the binary relations *equality (=), element of* ($\in$), *subset* ($\subseteq$) *and superset* ($\supseteq$), depending on the types $t_{m_{t_l}}$ and $u_{m_{u_l}}$. A child term is called a *foreign constraint* if it refers to each attribute of the primary key constraint of the parent term.

(f) if t is a data type $r[*t]$ is a data type, named as r, called $reference\ to\ t$.

Based on the comparability of instances of atomic data types, we assume that instances of the same constructed type are also comparable, also in the case of bags. According to the latter, if T is a data type, U and V are instances of type $\{T\}$, and i_t is an instance of type T, then expressions like $i_t \in U$, $U \subseteq V$ are computable comparisons.

2.2 Medium

For *container sorts*, a rather strict hierarchy with *device* on the top is available. Possible constructions of *container sort* hierarchies are as follows:

(a) a *named device* is a particular container sort that may be either *raw* (unstructured) or sorted as a finite set of *named files*. A device is homogeneous concerning accessing its data;

(b) a *named file* is a finite set of possible differently structured (or *sorted*) *named records*. Specifying the structure of a file, XML-like regular expressions are to be used;

(c) a *named record* is a tuple of *named fields*. Describing a record is similar to that of an entity type, but instead of named types named container sorts are to be used. Just like an *entity* type, a *record* may also specify a *primary key constraint* in the same form as there, but the roles of set of attributes are played by a set of *named fields* here;

(d) a *named field* may be either *raw* or of sort of a named *container sort*s but *field*, or a sort of *reference to* any of *named container sort*s but itself. There is a difference between a *container sort* or a *reference to* that: an instance of a *reference to* a named *container sort* refers to a uniquely identified instance of that *container sort*;

(e) if $s_1, \ldots, s_l$ are named container sorts, $s_u[s_1, \ldots, s_l]$ is a container sort named s_u, and called *union*. An instance of sort s_u, is an instance of any one sort of sorts $s_1, \ldots, s_l$;

(f) given a finite set of raw or sorted named devices $\mathcal{D}$, a finite set of referential constraints $\mathcal{C}$, a pair $M(\mathcal{D}, \mathcal{C})$ is a container sort named M, and called medium. Referential constraints here have the same form as that of schemata's, but the role of types and the role of attributes are now played by *named records* and *named fields*, respectively;

(g) if s is a named container sort, $r_s[*s]$ is a container sort named r_s, and called *reference to s*.

2.3 Abstract Access Environment

Given a schema $S(\mathcal{T}_\mathcal{S}, \mathcal{C}_\mathcal{S})$, and a medium $M(\mathcal{D}_\mathcal{M}, \mathcal{C}_\mathcal{M})$, suppose, $\mathcal{T}_\mathcal{S} = \mathcal{T}_\mathcal{P} \cup \mathcal{T}_\mathcal{C}$ with $T_P \cap T_C = \emptyset$ is held. A total mapping $\sigma : S \mapsto 2^M$ is a serialization of schema S over the medium M, if the following rules are satisfied.

- The inverse mapping $\sigma^{-1} : 2^M \mapsto S$ is a partial mapping such that $\sigma^{-1}(F_M)$ for some $F_M \subset 2^M$ is defined, iff there is $T \subseteq S$ such that $F_M = \cup_{t \in T} \sigma(t)$, and $\sigma \circ \sigma^{-1}$ on each $T \subseteq S$ is the identity.
- Let $\Pi \subset 2^{\mathcal{T}_\mathcal{P}}$ denote a partition over $\mathcal{T}_\mathcal{P}$ (i.e. $\cup \Pi = \mathcal{T}_\mathcal{P}$ and elements of Π are pairwise disjoint), then for each $T \in \Pi$, let $\sigma(T) = D$ be such that,
  (i) $D \in \mathcal{D}_\mathcal{M}$ is a *raw device*;
  (ii) σ defines and preserves a particular ordering over Π, and also over each $T \in \Pi$;
  (iii) for each distinct pair $U, V \in \Pi$ $\sigma(U) \neq \sigma(V)$ holds.

- Each $t \in T_C$ is mapped over *named fields* of M under particular *sort*- and *constraint* restrictions listed below:
  (i) distinct elements of $T_a \cap T_C$ are mapped to distinct raw fields: a field that an element of T_a is already mapped to becomes of that type;

(ii) a *bag* type is mapped to a field of sort file, or a sort *reference to* a file;

(iii) a *reference to* a type is mapped to a field of sort *reference to* an adequate *sort*. For *set* types $files$ are adequate $sorts$. For a type $\{t\} \in \Pi$, the *devecie* that t is mapped to is an adequate $sort$. For *entity* types, adequate $sorts$ are $records$ or $fileds$;

(iv) for *entity* types, attributes of type different from an *entity* type are mapped according to (i)-(iii). Attributes of some *entity* type are mapped as if it were a *reference to* an *entity* type, or to a field of sort *reference to* a *record*;

(v) σ preserves type integrity: assume, the image of the attributes of type $t < K_{p_t} \Leftarrow [x_1 : t_1, \ldots, x_m : t_m] >$ with $K_{p_t} = \{x_{i_1}, \ldots, x_{i_n}\}$ by σ are distributed over the $records$ $r_1, \ldots, r_s$ with primary key constraints $K_{p_{r_1}}, \ldots, K_{p_{r_s}}$. Assume furthermore that for each $k \in [1, s]$, there exist a $c_{M_k} \in \mathcal{C}_{\mathcal{M}}$ of form $\phi_{r_k} \Leftarrow \{\psi_{k_1}, \ldots, \psi_{k_p}\}$, and consider the foreign key constraints, $\phi_{r_k} \Leftarrow \{\psi_{r_j} : j \in_{j \neq k} [1, s]\}$ for each $k \in [1, s]$, where ϕ_{r_k} refers to r_k as parent, and ψ_{r_j} for each $j \in_{j \neq k} [1, s]$ refers to r_j as a child, suppose, C_k denotes the set $\{\psi_{r_j} | j \in_{j \neq k} [1, s]\}$. Then, on the one hand, $\cup_{l=1}^{s} K_{p_{i_l}} \subseteq \{\sigma(t.x_{i_1}), \ldots, \sigma(t.x_{i_n})\}$, and on the other hand, $C_K \subseteq \{\psi_{k_1}, \ldots, \psi_{k_p}\}$ shall hold.

(vi) σ preserves also schema's integrity: keeping denotations of (v), assume in addition, that images of types $u_1, ..., u_r$ in foreign key constraint $\{t.x_{i1}, ..., t.x_{i_n}\} \Leftarrow \{u_1[\varrho_{u_{1_1}} y_{u_{1_1}}, ..., \varrho_{u_{1_n}} y_{u_{1_n}}], ..., u_r[\varrho_{u_{r_1}} y_{u_{r_1}}, ..., \varrho_{u_{r_1}} y_{u_{r_n}}]\}$ are spread over records, $R_{u_{1_1}}, ..., R_{u_{1_{q_1}}}, \ldots, R_{u_{r_1}}, ..., R_{u_{r_{q_r}}}$, and consider the foreign key constraints $\phi_{r_k} \Leftarrow \{\psi^k_{r_{u_{m_j}}} : m \in [1, r], j \in [1, q_m]\}$ for each $k \in [1, s]$, where ϕ_{r_k}, just like in (v), refers to r_k as parent, and $\psi^k_{r_{u_{m_j}}}$ refers to $R_{u_{m_j}}$ as child. Then for the foreign key constraint c_{M_k} for each $k \in [1, s]$, $\{\psi^k_{r_{u_{m_j}}} : m \in [1, r], j \in [1, q_m]\} \subseteq \{\psi_{k_1}, \ldots, \psi_{k_p}\}$ shall hold.

Given a *schema* $S(\mathcal{T}_{\mathcal{S}}, \mathcal{C}_{\mathcal{S}})$ and a *medium* $M(\mathcal{D}_{\mathcal{M}}, \mathcal{C}_{\mathcal{M}})$, S is said *serializable* over M, if there is some serialization σ of schema S over medium M. If so, a 3-tuple $\varepsilon(S, M, \sigma)$ is called an *abstract access environment* for S and M.

3 Instantiation, Data Exchange, Autonomy

In the following, from autonomy's point of view, the base abstract domains are schema, medium, and access environment. Applications' data environment are modeled as instances of domains above.

3.1 Instantiation

An instantiation of a schema S is a swarm (i.e. a bag or set) of instances of types of S. A schema instantiation is allowed to produce either replicas of the same instance of the same type or no instance at all of particular types. We'd like to outline, we don't concern here with whether or not a schema instance is consistent, i.e. no primary or foreign key constraint is violated. It is the responsibility the application that creates instances and operates on them.

An instantiation of a medium M is to carry, i.e., temporarily store, preserve, and display data. A medium instance is a finite set of uniquely identified instances of *named devices*, each of which may either be raw, or sorted. A sorted named device is a finite set of instances of named files, each of them uniquely identified on the device. Unlike schema instantiations, a medium instance only for container sorts record and field, is allowed to have replicas of instances (i.e. that carries replicas of type instances). Instances of records are tuples of instances of fields.

Recall, that we want application data to carry all necessary non-application-specific information, so that from now on, we assume that instances of types and also of container sorts carries all necessary information on their types and sorts. In addition, when we consider instances of schemata and media, they are assumed to carry schemata's and media's integrity constraints. Given a schema S, a medium M, and an abstract access environment $\varepsilon(S, M, \sigma)$ for S and M, an instance over S over an instance over M can easily be serialized induced by σ: unless primary key constraints are violated, each atomic piece of data (data of atomic types) of the instance over S has to be mapped to a new instance of the container sort (i.e. a container of that sort) the type of the atomic data by σ is mapped to. The mapping induced by σ is denoted by $\sigma_\diamond$, and a medium instance that carries serialized data of a schema instance is called the total serialization of that schema instance. Similarly to the above, the inverse mapping σ^{-1} also induce a partial mapping – denoted by $\sigma_\diamond^{-1}$ – which maps particular subsets of serialized data of an instance over a medium to subsets of an instance over a schema. For given instances I_S and I_M over S and M, a 3-tuple $\Delta(I_S, I_M, \sigma_\diamond)$ is considered as an instantiation of $\varepsilon(S, M, \sigma)$, and called abstract access method between instances I_S and I_M. Since instances I_S and I_M carries all necessary information on S and M, mappings $\sigma_\diamond$ and $\sigma_\diamond^{-1}$ are given for the abstract access method $\Delta(I_S, I_M, \sigma_\diamond)$ between I_S and I_M.

3.2 Data Exchange

Assume schema S is serializable over medium M, and let $\varepsilon(S, M, \sigma)$ be an abstract access environment for S and M. Let $\mathcal{I}_\mathcal{S}$ denote the (infinite) set of all possible instances over S, and $\mathcal{I}_{\mathcal{M}_{\mathcal{I}_\mathcal{S}}}$ denote the set of total serializations of all elements of $\mathcal{I}_\mathcal{S}$ i.e. the set $\{\sigma_\diamond(I_S) : I_S \in \mathcal{I}_\mathcal{S}\}$. A $K_S \in \mathcal{I}_\mathcal{S}$ is called a *key expression* over S, if it is a set. An element of a key expression over S is called *key term* over S. A key term is considered as an instance pattern, and determines a (possible empty) subset or sub-bag of any particular $I_S \in \mathcal{I}_\mathcal{S}$ based on some type-dependent pattern matching (e.g. in the case of atomic types a type-dependent pattern matching is checking equality of values of the same type), by that null values are not considered. Since serializations map integrity constraints of schemata onto integrity constraints of media, the image of a key expression over S by $\sigma_\diamond$ is a key expression over $\mathcal{I}_{\mathcal{M}_{\mathcal{I}_\mathcal{S}}}$. Given particular instances $I_S \in \mathcal{I}_\mathcal{S}$, $I_M \in \mathcal{I}_{\mathcal{M}_{\mathcal{I}_\mathcal{S}}}$ and a key expressions $K_S \in \mathcal{I}_\mathcal{S}$, $K_M \in \mathcal{I}_{\mathcal{M}_{\mathcal{I}_\mathcal{S}}}$, let $I_S(K_S) \subseteq I_S$ and $I_M(K_M) \subseteq I_M$ denote the subsets that match K_S and K_M, respectively, which themselves are also particular instances over S and M at the same time. Let $K_S \in \mathcal{I}_\mathcal{S}$ be a key expression; the abstract access method $\Delta(I_S, I_M, \sigma_\diamond)$ between I_S and I_M together with key expression K_S determines particular instances $I_S - I_S(K) \cup \sigma_\diamond^{-1}(I_M(\sigma_\diamond(K_S))) \in \mathcal{I}_\mathcal{S}$, and $I_M - I_M(\sigma_\diamond(K_S)) \cup \sigma_\diamond(I_S(K_S)) \in \mathcal{I}_{\mathcal{M}_{\mathcal{I}_\mathcal{S}}}$, called a partial input data access from I_M into I_S, and partial output data access from I_S onto I_M, respectively. In the above formulas, only K_S is out of scope of $\Delta(I_S, I_M, \sigma_\diamond)$, so one may associate a tuple $(K_S, \Delta(I_S, I_M, \sigma_\diamond))$ with the set $I_S - I_S(K_S) \cup \sigma_\diamond^{-1}(I_M(\sigma_\diamond(K_S)))$, and a tuple $(\Delta(I_S, I_M, \sigma_\diamond), K_S)$ with the set $I_M - I_M(\sigma_\diamond(K_S)) \cup \sigma_\diamond(I_S(K_S))$.

3.3 Autonomy

Given a schema $S(\mathcal{T}_\mathcal{S}, \mathcal{C}_\mathcal{S})$, a medium $M(\mathcal{D}_\mathcal{M}, \mathcal{C}_\mathcal{M})$, assume S is serializable over M and $\varepsilon(S, M, \sigma)$ is an *abstract access environment* for S and M. Suppose, $\mathcal{I}_\mathcal{S}$, denote the set of all possible instances over schema S, and $\mathcal{I}_{\mathcal{M}_{\mathcal{I}_\mathcal{S}}}$ be the set of the total serializations of all elements of $\mathcal{I}_\mathcal{S}$. Let $I_{\varepsilon(S,M,\sigma)}$ denote all abstract access method instances over $\varepsilon(S, M, \sigma)$. Let $\mathcal{I}_\mathcal{S}^{\{\}} \subset \mathcal{I}_\mathcal{S}$ denote the set of possible key expression over S. Let a_c, d_c, $\delta_{(a_c,d_c,\sigma_\diamond)}$ denote variables of types $*S$, $*M$ and $*\varepsilon(S, M, \sigma)$ called *application cache*, *data carrier* and *access interface*, respectively. For particular instances $I_S \in \mathcal{I}_\mathcal{S}$ and $I_M \in \mathcal{I}_{\mathcal{M}_{\mathcal{I}_\mathcal{S}}}$ let us assume, that $*a_c = I_S$ and $*d_c = I_M$ hold , and let us furthermore define $\mathcal{A}_\mathcal{D}$ as $\mathcal{A}_\mathcal{D} = (I_{\varepsilon(S,M,\sigma)} \cup \{\emptyset\}) \times \mathcal{I}_\mathcal{S}^{\{\}} \cup \mathcal{I}_\mathcal{S}^{\{\}} \times (I_{\varepsilon(S,M,\sigma)} \cup \{\emptyset\})$. An autonomous application can now be formalized as $A(a_c, d_c, \delta_{(a_c,d_c,\sigma_\diamond)}, \alpha)$, where $\alpha : \mathcal{A}_\mathcal{D} \mapsto \mathcal{I}_\mathcal{S} \cup \mathcal{I}_{\mathcal{M}_{\mathcal{I}_\mathcal{S}}} \cup I_{\varepsilon(S,M,\sigma)}$ a mapping, called application logic, such that

(i) for each $a \in \mathcal{A}_\mathcal{D}$ of any of the forms $(I_S, \emptyset)$ or $(\emptyset, I_S)$, $\alpha(a)$ is a mapping over $\mathcal{I}_\mathcal{S}$, called computation that effects also on contents of variables a_c and so do on variable $\delta_{(a_c,d_c,\sigma_\diamond)}$.

(ii) for each $i \in \mathcal{A}_\mathcal{D}$ of form $(K_S, \Delta(I_S, I_M, \sigma_\diamond))$, $\alpha(i)$ is an partial input from I_M into I_S that performs the assignment $a_c = I_S - I_S(K_S) \cup \sigma_\diamond^{-1}(I_M(\sigma_\diamond(K_S)))$, causing change also in the content of variable $\delta_{(a_c,d_c,\sigma_\diamond)}$.

(iii) last, for each o of form $(\Delta_{(I_S,I_M,\sigma_\diamond)}, K_S)$, $\alpha(o)$ is a partial output from I_S onto I_M, that performs the assignment $d_c = I_M - I_M(\sigma_\diamond(K_S)) \cup \sigma_\diamond(I_S(K_S))$, which changes the content of variable $\delta_{(a_c,d_c,\sigma_\diamond)}$.

4 Real Data Environment and Computer Aided Autonomy

Using variables application cache, data carrier, and access interface is not simple formalism, which helps us to establish a formalization of autonomous application presented above. Application cache is a main memory area in process space which may actually be accessed through a pointer. For data carrier the situation is the same – using any programming languages you have means to open files, to connect to a database, which is really a pointer to some particularly structured data instance for an application. Considering access interface that is to hold an instance of an abstract access environment may seem a bit more artificial, just like data (instances of types) that carry all necessary knowledge about their types and integrity constraints. Yet one should realize, that such instances are mappings between finite sets induced by a mapping between a finite set of types, and a finite set of container sorts, which can be implemented as data. The data model we need is introduced in the next subsection and called

4.1 Generalized Document Data Model

The data model introduced here, is a generalization of Document Data Model (DDM) that has been introduced for one of our proposals for a common data model for applications on Figure 1. DDM is a data representation in which one can naturally code information and structuring knowledge (also out of scope of any application logic), and so it is one possible model favouring our need in achieving autonomous applications. Document Data Model is a semi-structured data model, where information about structuring and typing data is separated from the real content of data.

Data in DDM are represented in forms of ordered pairs (R, S). R and S are called raw data and schema, respectively. Both of those are simple byte streams, but S, if given, is always an XML document or document fragment [4]. The above organization of semi-structured data has two advantages: the same raw data can be structured in several way, so that there might be a number of pairs where only the schema components are different, each of them providing for adequate structuring and occasionally additional semantics the all-time processing phase needs, so that they can be separately stored with storing only one instance of the raw data; the other is that the data representation above enables to define structured data types and primitive abstract operations on data.

For instance, data types and type templates in DDM are represented by pairs like T(∅, S), where S is non-empty [6]. To instantiate such types or templates one should provide the raw data component for such pairs. If the raw data component itself is a kind of schema, e.g. an XML general entity, a real type is instantiated.

Pairs like V(R, ∅) where R is non-empty are typeless (i.e. void) data. Typeless data, if meaningful, can be converted to any type by providing for a schema component. One has to realize, that the schema components are not only for structuring the raw data. The schema components may describe or identify also methods characteristic for the structured type represented. Changing the schema components can be considered as a cast operation.

Last, pairs D(R, S), neither R nor S is empty, constitute typed data instances.

A step forward from DDM leads us to the Generalized Document Data Model, where the schema component of a piece of data specifies all non-application-specific knowledge of its flow through an application, including data retrieval and storage, and in addition there is a global schema interpreter available. Using GDDM, knowledge about data processing is carried by data themselves, and the knowledge is interpreted by the G schema interpreter. To get and interpret such knowledge, the knowledge carrier media has to be accessed.

4.2 Implementation Considerations

In the abstract model of autonomous application introduced in the previous subsection the knowledge in question is implemented by the abstract access environment. Since an abstract access environment is a serialization - i.e. a structural mapping and type serialization - it is easily implemented by an XML document. One difficulty, what is a kind of discomforts – much rather than crux, to implement such an XML document by hand. Instantiating an abstract access environment cause formally also no difficulties by using variables application cache and data carrier, provided if they already contain instances of application domain and medium. Instantiating application domain is always natural part of any applications, but instantiating a medium needs to communicate some data carrier server including data storage- and/or display engine, e.g. some RDBMS, WEB browser. The latter gives semantics among others to the induced serializations $\sigma_\diamond$, and its inverse mapping, and provides also mean for the evaluation of a subset of a medium instance using a key expression.

To implement the model presented before is nothing elese as binding our abstract data environment to a real one. By such a binding an autonomous application is integrated into a real run-time environment. When we speak on binding, we think of an early binding, that is abstract operations invoked by the application are linking to particular operation libraries provided by the all-time run-time environment. Our idea is that run-time environments that supply integrating autonomous applications provide for a DLLs that implements instantiations of real data carrier media's that are available there, and also DOM or SAX based XML document processing middleware that implements operations $\sigma_\diamond$ and $\sigma_\diamond^{-1}$ between real data carrier instances and application domains. Integrating an autonomous application into such run-time environments needs to rebuild the DLLs in order to extend it's application domain knowledge. Type conflicts within a middleware caused by collision of type names of different applications can be avoided by using XML namespaces implementing abstract access environments.

4.3 Computer Aided Autonomy

Instead of following a today's fashionable and usual way, i.e. establishing a GDDM based integrated autonomous application environment with providing GDDM oriented program development tools, we belive that traditional programming languages and their development environment extended by computer aided implementation of a desired GDDM representation of data environment will be much more popular, and will result much more satisfaction for the community of professional application developers. The computer aided autonomy we propose is a computer generated GDDM representation of applications' schemata, media and abstract access environment. If there is a language on which abstract domains and serializations between them can be defined, and that can easily be embedded into any host languages, we have a language (the host language that embeds the domain definition language) that is suitable to develop autonomous application. We would like to outline, that such a language is not for extending host languages' domain definition capabilities, but much rather facilitating them to define abstract data environments. Preprocessing embedded statements of a program will generate a pure host language text and the GDDM representation of the abstract data environment (see Figure 4).

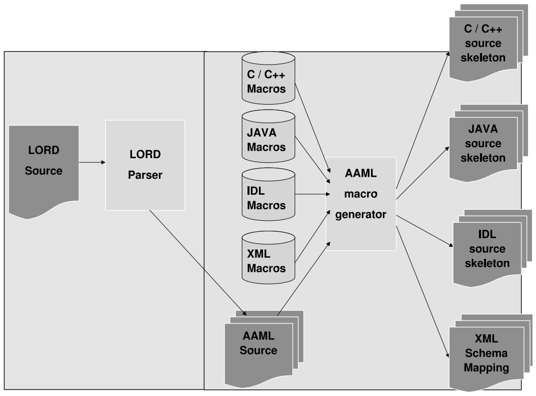

Fig. 4. Environment for integrating autonomous applications.

We are in progress to define a language, called LORD (Lay-Out Relationship and Domain definition language), that consists of two components. The one is developed to define schemata, media, and serializations of schemata over media, the other one is a macro language which provides way to implement context-dependent macro definitions, context-dependent and context-keeping macro expansion and what are called meta-macro definitions, the expansion of those are macro or meta-macro definitions.

5 Syntactical considerations

LORD has been designing for natural embedding into arbitrary host languages – or with other words – a statement including embedded symbols should not look strange against the host language's style. One way to reach that, LORD language constructions are considered as invocations of *macros* defined in some intelligent macro language which provide means for defining meta-macros (the expansion of those results in macro definitions) and *context-dependent macro expansions*. The above featured macro language presented here is AAML (*Autonomous Applications' Macro Language*).

5.1 AAML

In AAML, any host language statement that embeds AAML macro invocations (an embedding statement may contain more than one macro invocations) may generate single or more pure host language statements, depending on the corresponding macro definitions. The recursive process of such text generations is controlled by *Complementary Expansion Method* (CEM). CEM is based on *invocation patterns* and *invocation-contexts*. Invocation patterns control formal parameter declarations and their value settings for macro definitions, and invocation-contexts control statement completion and context-dependent text generation.

Macro definitions may obtain complete, and also incomplete statements. Expanding a macro invocation, complete statements are expanded in the usual way – all the macro invocations within the complete (embedding) statement, in the order of invocations from left to right are recursively expanded. Incomplete statements shall, however, be first completed and the completion is then in the usual way to be expanded.

5.2 Other approaches

There have been many other approaches to make programming languages either syntactically extendible or integrating domain specific concepts into the host language.

Extendible Syntax [9] was introduced for incremental syntax definition by extending core language. Syntax extensions were placed into the host language between special syntactical delimiters. The implementation is based on LL parsing. Because the class of LL languages is not closed under union and concatenation, the syntax definition sometime uncomfortable.

The *Java Syntactic Extender* (JSE) [10] is a macro system. The expressiveness of the syntax that can be introduced is limited. As in most macro systems, a macro identifier is required in invocations and the JSE parser is not extensible. Opposite to the above, AAML does not restricts the syntax of macro names.

Bravenboer and Visser gives a detailed discussion in [8] on syntactical extension of host languages respect to domain specific extensions, advertising the METABORG method. The METABORG is a method providing concrete syntax mainly for domain abstractions to application programmers. In contrast, our goal, concerning LORD, is not to extent the host language rather than integrate and assimilate domain-specific concerns that otherwise could be formulated in the language. This way our approach is more make map a certain domain into many host languages.

6 Conclusions

Application integration needs to design and implement a common data model and the corresponding data access layer. Information and software industry hoped finding real solutions by establishing data standards for particular groups of application fields, like IFC, CIS/2. Such standards mostly concern *data exchange format* and are highly domain specific. Another approach are e.g. *data warehouses*, that implements a common data access layer over heterogeneously structured data sources for applications.

Integration of applications of possibly different domains and/or following different standards are not generally solved and still costs a large amount of human and computer resources.

Autonomous applications using *Document Data Model* can easily follow structural differences of data, and, in addition, structural information of data is carried by themselves. This integration model is also capable to express collaboration protocols between applications processing data of possibly different domains or using different standards. Due to comprehensive XML technology and generative programming philosophy (like that of LORD) computer aided integration in our model may hope better chance.

References

1. Industrial automation systems and integration - Product data representation and exchange - Part 11: Description methods: The EXPRESS language reference manual Reference Number ISO 10303-11:1994, ISO Switzerland (1994)
2. Industrial automation systems and integration - Product data representation and exchange - Part 22: Implementation methods: Standard Data Access Interface specification, Reference Number ISO/DIS 10303-22. ISO, Switzerland (1993)
3. CIMsteel Integration Standards Release 2 (Second Edition) http://www.cis2.org/
4. Extensible Markup Language (XML) 1.0 (Third Edition) http://www.w3.org/TR/2003/PER-xml-20031030
5. Mukerji, J., Miller, J.: Overview and guide to OMG's architecture http://www.omg.org/docs/omg/03-06-01.pdf
6. Abiteboul, S., Buneman, P., Suciu, D.: Data on the WEB – From Relations to Semistructured Data and XML, W3C Proposed Edited Recommendation 2003, San Francisco (2000)
7. Hernath, Z., Vinceller, Z.: Generalized Document Data Model for Integrating Autonomous Applications In Proceedings of International Conference of Applied Informatics (ICAI'6), Eger, Hungary (2003)
8. Bravenboer, M., Visser, E.: Concrete Syntax for Abstract Objects: Domain-Specific Language Embedding In Proceedings of Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'04), Vancouver, Canada (2004) 365–383
9. Cardelli, L., Matthes, F., Abadi, M.: Extensible Syntax with Lexical Scoping SRS Research Report 121, DEC Systems Research Center (1994)
10. Bachrach, J., Playford, K.: The Java Syntactic Extender. In Proceedings of Object-Oriented Programming, Languages, Systems, and Applications (OOPSLA'01), Tampa, Florida (2001) 31–42