Towards a Model for Persistent Data Integration

Olivier Perrin and Nacer Boudjlida

Centre de Recherche en Informatique de Nancy (CRIN-CNRS)
University of Nancy I
Campus Scientifique — B.P. 239
54506 Vandoeuvre-lès-Nancy — FRANCE
email: operrin@loria.fr, nacer@loria.fr

Abstract. Tool integration in software development environments is a major problem, on one hand, for users of these environments, and on the other hand, for tool builders and suppliers. In this paper, we focus on persistent data integration, which is one of the two main points of tool integration. The purpose of this work is to provide a formal data model that includes most of the semantics of the data manipulated to answer problems raised by persistent data integration. To achieve this goal, we introduce a classical data model, and provide a set of operators which are given for the object transformation. We also provide an example describing the transformation process.

Keywords: data integration, integrated software-engineering environments, federated database management systems, meta-data, data heterogenity, PCTE, CDIF.

Introduction

Tool and data integration is a major problem for both users and suppliers of CASE tools. The information held by a tool is seldom compatible with the information that is required by another tool. Moreover, users want to control tools of an environment and need to be able to move information between these tools. The main reason is that users want to get efficient integrated systems.

The integration problem can be approached from two standpoints: the first is the control dimension, which means giving the ability for several tools to communicate. The second is the data dimension, which means enabling a tool to access data from another one. Data integration is relevant only when the tools deal with common data. In [17], there exists a summary of common questions to be answered in order to achieve data integration: "How work must be done to make the data used by one tool useful for the other ?", "How much data managed by a tool is duplicated in or can be derived from data managed by the other ?", "How well do two tools cooperate to maintain the semantic constraints on the data they process ?", "How much work must be done to make the data

generated by one tool usable by the other ?", "How well does a tool communicate changes it makes to the values of nonpersistent, common data so that other tools it is cooperating with may synchronize their values for the data ?". Data integration includes integration of persistent and nonpersistent data. We believe that nonpersistent data integration is relevant to control integration instead of data integration, so in this paper, we concentrate on persistent data integration. We focus on some interest in integrated software-engineering environments (ISEE)([5, 3]) and information systems environments, during both design and operating phases. Our work is also relevant to federated database management systems (FDBMS)([15, 12, 11]) and we examine both *a priori* and *a posteriori* views. Many problems about data exchange in SEE, or cooperating databases in FDBMS concern heterogeneity, i.e. differences in data semantics. Detecting semantic heterogeneity is a crucial problem. The main problems consists in having enough semantic and information to interpret data in a consistent way, and identifying and then solving semantic heterogeneity, such as differences in the definitions (meanings) of two data elements, or differences in the formats of the data elements (values, precision,...).

In [15], there is a list of unsolved problems in FDBMS. An important point deals with the identification and representation of all semantics useful in various FDMBS. This also exists in ISEE, where a various number of platforms and various tools use different formats. Another point deals with the automation, as completely as possible, of a process that transforms data to make a tool use data of another one. This gives rise to questions concerning semantic integrity constraints, serializability, concurrency control and management.

Persistent data integration is characterized by five points:

(*i*) the data being managed,
(*ii*) the representation and the naming of the data elements,
(*iii*) the semantic interpretation,
(*iv*) the syntactic and semantic constraints,
(*v*) the implementation.

The data represents data that each environment's tool can access. It is the "work area" we deal with. The implementation represents file or object structures, concurrency control mechanisms and global environment integrity. Representation and semantic interpretation could be split into four different parts, which can be summarized as follows:

- identical pieces of information are represented by different symbols, which makes up the **synomyms** problem,
- different pieces of information are represented by the same symbol, which makes up the **homonyms** problem,
- different aggregation or scale levels,
- loss of information.

Then, syntactic and semantic constraints are grouped into constraints that are used to manage and process data.

The purpose of this work is to provide a formal data model that includes most of the semantics of the data used by one tool to enable another tool to "understand" these data. The formal model uses the notion of meta-level, commonly used in recent databases or repositories descriptions, and proposes to group attributes which describe the same data. Different levels of compatibility between two definitions of a data description are defined. The model has commonalities with object oriented data models. Its description is split into two parts: the first section is devoted to the introduction of definitions and propositions about compatibility of two data objects. The second section describes the operators of the model, that are given for the object transformation. A third section will show an example to illustrate the mechanisms. Then, we conclude by providing some problems not yet solved by the model.

1 The Data Model

In this section, we describe the model we propose to use to deal with persistent data integration. The model encompasses classical concepts to describe "flat" structures as well as composite structures. It also encompasses an object set concept that enables grouping objects with "similar" descriptions into a set. Finally, a role concept is introduced to include the fact that an element of two different objects has the same meaning. These concepts enable us to define three kinds of object compatibility. Furthermore, a notion of hierarchy of object definitions is introduced. It enables ordering of object definitions on the "goodness of their descriptions" and deducing object definition compatibilities. A feature of object models is their richer semantics. This is important to get enough semantics to provide as much automation as possible. We believe that such a model allows for two tools to have information about *syntax* and *semantic* of shared data. It is the basics of our method to provide a tool with a mechanism to understand and exchange data of another tool of the same class (i.e., tools which share some functionalities and data).

1.1 Objects

We assume the following sets:

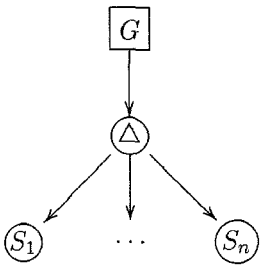
- let $\mathcal{S}$ be an infinite set of symbols. This set is partitioned into disjoint subsets $\mathcal{S}_r$. These subsets contain symbols with identical role and meaning. Let $\mathcal{S}_r \subset \mathcal{S}$ be such a subset where all the symbols belong to the role r. A data element $\ell_r \in \mathcal{S}_r$ is a data element of r role.
  For example, the following elements belong to $\mathcal{S}$: *age, position, comment.*
- let $\mathcal{T}$ be the atomic type set (for example, integer, string, boolean, real). We use usual standard behaviors for these atomic types. We consider that the equality between two types is a syntactic equality.
- let $\mathcal{V}$ be the set of values belonging to $\mathcal{T}$, denoted $\{\mathcal{V}_\mathsf{t} \mid \mathsf{t} \in \mathcal{T}\}$. We assign a specific value to each type $\mathsf{t} \in \mathcal{T}$, the null value, denoted $\mathtt{null}_\mathsf{t}$ which

describes the null value of type t $\in \mathcal{T}$ (in our approach, null values are non-informative values) . Each type has a corresponding domain, namely $\mathcal{D}_{integer}$, $\mathcal{D}_{string}$, $\mathcal{D}_{boolean}$, and $\mathcal{D}_{real}$.

The definitions we give below are very similar to the definitions which are admitted in an object-oriented data model or a the relational data model with nested relations ([10, 1]). Two concepts are commonly used for object definition. The first one is a "flat" structure where all the attributes are defined by an atomic type. The second one introduces composite structures where an attribute can be defined by an object definition. To introduce complex object definition, we classicaly use three type constructors given in [16] and [2]. These are commonly accepted as the tuple constructor, the union constructor, and the set constructor.

Tuple constructor is closely related to the record structure in PASCAL, and to the aggregation described in [16]. This constructor allows to construct more complicated types, and object definitions. The syntactic representation for the tuple constructor is given by a pair of square brackets ([...]). Following is a short example of such a complex object definition. A complex object definition could be [*Name*: string; *Address*:[*Number*: integer; *Street*: string; *City*: string]]. An instance of this object definition would be [*Name* = `"Martin"`; *Address* =[*number* = 9; *Street* = `"Harbor Drive"`; *City* = `"San Diego"`].

The union of type constructors is closely related to PASCAL's record variant structure, and to generalization of [16]. It is used so that objects of differents types can be view as generically the same. The syntactic representation for the union of a type constructor is given by a pair of angle brackets ($\langle \ldots \rangle$). We now give an example: an object definition could be [*Address*: $\langle$*Street Address*: [*Number*: integer; *Street*: string]; *PO-BOX*: string$\rangle$], and an instance would be [*Address* = $\langle$*PO-BOX* = `"400"`$\rangle$]. It is important to notice that we impose two constraints to this constructor. The first one is called the disjoint property and means that the intersection of all subtypes of a given master type is empty. The second one, called the total property, means that the union of all subtypes of a given master type recovers this type. Formally speaking, we define the disjoint and total concepts as follows. Let's consider the following diagram:



where $\triangle$ represents the union of types. The disjoint property means that $S_i \cap S_j = \emptyset$ for i $\neq$ j. The total property means that $\bigcup_{i=1}^{n} S_i = G$.

The third constructor is a set constructor, called the collection constructor. It allows to regroup a set of objects belonging to a given type. The syntactic representation of this constructor is given by a pair of curled brackets ($\{\ldots\}$). In

the following example, we describe the semantic corresponding to this constructor. Suppose we have a text block formed by a set of lines. We write [*Text_Block*: {*Line*: string}]. An instance of this object definition would be [*Text_Block* = {"This is a", "simple example", "of the set constructor"}].

In [9], we could find an extended version of the model, described as the Format model, which subsumes the relational model and some parts of the hierarchical model. It also provides a convenient way to represent types and objects by trees and subtrees.

Definition 1 *An object definition F, either atomic object definition or complex object definition, denoted* $F = [\ell_{r_1}:t_1;\ldots;\ell_{r_n}:t_n]$ $(n \geq 1)$, *is recursively constructed as follows:*

- $\ell_{r_i} \in \mathcal{S}_{r_i}$, $(1 \leq i \leq n)$,
- $t_i \in \mathcal{T}$ (t_i *is an atomic type) or*
  t_i *is a complex type (constructed by recursive application of the aggregation, generalization or collection constructors),* $(1 \leq i \leq n)$.

In the rest of the text, we will call a simple object an object defined by an atomic definition and a composite object an object defined by a complex definition. Example 1 shows two simple objects.

Definition 2 *An object, defined by F and denoted* o_F, *is given by* $[\ell_{r_1}=v_1;\ldots;\ell_{r_n}=v_n]$ $(n \geq 1)$ *where:*

- $\ell_{r_i} \in \mathcal{S}_{r_i}$, $(1 \leq i \leq n)$,
- $v_i \in \mathcal{V}_{t_i}$ *or* $v_i = \mathtt{null}_{t_i}$, $(1 \leq i \leq n)$.

1.2 Object set

In this section, we introduce the object set concept. Such a set contains two components which are a unique object definition, and an object set where objects are consistent with the set definition. Each object of an object set correspond to the given definition.

Definition 3 *An object set, denoted* D_F, *is defined as:*

$D_F = \{F, o_1, \ldots, o_n\}$ *where every* o_i *is an object defined by F* $(i \leq n$, *F unique).*

The object set represents data to be managed. It is an ensemble of instances refering to the definition of the object set. This definition takes advantage of the separation between the definition of the objects and the entities themselves. This approach is widely used in object oriented databases and recent database models, where a meta-level is introduced to give more information about data. It represents a dictionary of object descriptions. For our purpose, it provides two distinct levels with distinct goals: the first level gives semantic of data, while the second concerns object's instances.

Example 1 Let F = [*text*: string;*xposition*: integer; *yposition*: integer] be an object definition. o_1 and o_2 are two objects defined by F:

o_1 = [*text* = "sample"; *xposition*= 150; *yposition* = 150],
o_2 = [*text* = "longer one"; *xposition* = 0; *yposition* = 0].

These two objects are considered as simple objects. A more complex object definition could be given by:

F'=[*Position*: [*x*:integer; *y*: integer]; *Text_block*:{*Line*: string}]

An instance of F' (considered as a composite object), denoted o_3 is defined as:

o_3=[*Position*=[*x*=150;*y*=150]; *Text_block*={*Line*="A much more";*Line*="complex example"}]

This way to describe objects is very clear and self explanatory. However, it becomes complicated and too cumbersome when an object contains a lot of elements. So we suggest to shorten the object writing as follows:

o_3=[[150;150];{"A much more";"complex example"}]

Nevertheless, we notice the importance of the description and the meaning of each symbol of an object o. It is important to know that the first value 150 refers to the x position, and value [150,150] refers to *Position*. So we must keep in mind that semantic is related to a symbol defined in an object. There are two issues: first, to keep the symbol's semantic in the object itself, second, to provide the F definition outside the object o, and for all objects refering to the definition. This is used for the object set concept.

1.3 Domain and degree

We will define a way to get all the symbols existing in a given object definition. Next, we will describe the way to access object's values.

Definition 4 *We assume an object definition F. We write* **Dom**(F) *the set of all the symbols of the definition F, and we recursively define* **Dom**(F) *as:*

- *if* $F = [\ell_{r_1}:t_1]$, *then* $\mathbf{Dom}(F) = \{\ell_{r_1}\}$,
- *if* $F = [\ell_{r_1}:t_1;\ldots;\ell_{r_n}:t_n]$,
  or $F = \{\ell_{r_1}:t_1;\ldots;\ell_{r_n}:t_n\}$,
  or $F = \langle\ell_{r_1}:t_1;\ldots;\ell_{r_n}:t_n\rangle$, *then* $\mathbf{Dom}(F) = \{\ell_{r_i} \in F,\ 1 \leq i \leq n\}$

We notice that **Dom**(F) *and* **Dom**(o_F) *contain the same elements.*

We now assume an instance of F given by the object o_F. Then we write $o_F(\ell_r)$ the corresponding value of ℓ_r.

Example 2 If we consider the object o_1 of the first example, we have:
- $\mathrm{Dom}(o_1) = \{text,\ xposition,\ yposition\}$,
- $o_1(text) =$ `"sample"`,
- $o_1(xposition) =$ `150` and $o_1(yposition) =$ `150`.

For the o_3 object, we write:

- $\mathrm{Dom}(o_3) = \{Position,\ x,\ y,\ Text\_block,\ Line\}$,
- $o_3(Position) =$ `[150;150]`,
- $o_3(Position(x)) =$ `150`,
- $o_3(Text\_block) =$ {`"A much more"`,`"complex example"`}.

Notice that it is not possible to access a particular element of a set.

Definition 5 *Given o_F an object defined by F, we call* **degree** *of o_F, denoted* $\mathtt{deg}(o_F)$ *or #Dom(F), the cardinal of o_F, i.e. the cardinal of domain* $\mathbf{Dom}(F)$.

Example 3
- $\mathtt{deg}(o_1) = 3$.
- $\mathtt{deg}(F') = 5$.

1.4 Roles

We have seen the importance of the semantical information of each part of an object. We describe in this section another concept: the role. Intuitively, it specifies that an element has the same meaning in two different definitions. For instance, consider two object definitions $F_1 =$ [*color*: string; *x*: integer; *y*: integer; *text*: string] and $F_2 =$ [*description*: string; *xpos*: integer; *ypos*: integer]. The two definitions share some parts devoted to the same "role": *text* and *description*, *x* and *xpos*, and *y* and *ypos*. The role concept means that same semantic interpretation is related to a data element in two different definitions. Furthermore, the definition of F_2 is included in the definition of F_1 (what could be written $\mathrm{Dom}(F_2) \subseteq \mathrm{Dom}(F_1)$). Consequently, a transformation can make F_2 compatible with F_1 (and conversely if we authorize some loss of information). There exists a way to establishe this kind of correspondence among data element's roles.

The data model we propose relies on the role concept. The symbol set $\mathcal{S}$ is divided into subsets as shown in the definition part. For instance, we regroup in the $\mathcal{S}_r$ subset all the symbols of the role r in the set $\mathcal{S}$. We distinguish between:

- **the role concept**: it consists of a set of data elements that share some properties (basically at the semantic level, i.e., identical meaning),
- **a data type**: it consists of a set of data elements with operations associated with,

- **a value**: value of a data element of role r and of type t.

There are at least two distinct type definitions:

- types as sets [4],
- types as algebras [7, 8].

The first definition has a historical origin, while the second appeared around 1975. Definition of role is represented by the notion of "types as sets", which serves to classify data elements whereas our notion of type describes the representation of a data element and operators among them. The notion of role seems to approach the notion of abstract data types where some semantic is related to the types. Roles are classes of data sets, while types are complex and include operators.

The next definition proposes to represent the concept in the object model, using the object's definition. We first introduce the binary predicate symbol $\overset{sem}{=}$ which can be interpreted as follows: $\ell_r \overset{sem}{=} \jmath_{r'}$ holds iff r is semantically identical to r', which stands for semantic equality between two elements of the set $\mathcal{S}$.

Definition 6 *Let F_1 and F_2 be two object definitions. We suppose given $\ell_r \in Dom(F_1)$, and $\jmath_{r'} \in Dom(F_2)$. We say that ℓ_r et $\jmath_{r'}$ are compatible, denoted $\ell_r \equiv \jmath_{r'}$, if and only if:*

$$r \overset{sem}{=} r'$$

Compatibility is not restricted to a one-to-one correspondence between two elements (or symbols) of two definitions. We can extend the binary relation $\equiv$ by providing a new context. For instance, such a representation can exist:

$$\ell_r \equiv \bigcup_x (\jmath_{r_x}), \text{ iff } r \overset{sem}{=} r' \text{ with } r' = \bigcup(r_i, \ldots, r_j) \ (i \leq x \leq j)$$

With such a representation, we can introduce the composition of symbols. There is no constraint on types, the only conditions we retain are the description and the role. Exemaple 4 illustrates definition 6.

Definition 7 *Consider $F = [\ell_{r_1} : \mathsf{t}_1; \ldots; \ell_{r_n} : \mathsf{t}_n]$ $(n \geq 1)$ and an associated object $o_F = [\ell_{r_1} = v_1; \ldots; \ell_{r_n} = v_n]$, we define ℓ_{r_m} $(m > n)$ as follows:*

- $\ell_{r_m} \equiv \mathtt{null}_{\mathcal{N}_{r_m}}$, $(m > n)$,
- $F(\ell_{r_m}) = \mathtt{undefined}_{\mathsf{Type}}$,
- $o(\ell_{r_m}) = \mathtt{null}_{F(\ell_{r_m})}$.

Various $\mathtt{null}_{\mathcal{N}_{r_m}}$, $\mathtt{undefined}_{\mathsf{Type}}$ or $\mathtt{null}_{F(\ell_m)}$ represents an undefined role, an undefined type or an undefined value (e.g., `null` value). There are undefined values which could be used to inform or complete a definition. Definition 9 allows to extend a given definition to match another one, and to provide a way to get a one-to-one correspondence between each symbol of the two definitions. In such a situation, we allow for the completion and the extension of a definition F of an object o_F, only with null values. For example, $F_1 =$ [*color*: string; *x*: integer; *y*: integer; *text*: string] and $F_2 =$ [*description*: string; *xpos*: integer; *ypos*: integer] match if we rewrite F_2 into:

Example 4 Let us analyze the objects o_1 and o_2 used in example 1. The following properties hold:
- *text* $\equiv$ *description*,
- $x \equiv$ *xpos*,
- $y \equiv$ *ypos*.

Consider another example, where F_1 = [*Name*:string;*FirstName*:string] and F_2 = [*FullName*:string], then the following property holds:

$$FullName \equiv (Name,\ FirstName)$$

Then, if we apply the definition 8 to F_1 = [*BirthDate*:date] and F_2 = [*BirthDate*:string], we get:

$$BirthDate_{F_1} \equiv BirthDate_{F_2}$$

F_2 = [*color*: $\mathtt{null}_{\mathsf{string}}$; *description*: string; *xpos*: integer; *ypos*: integer]

Concerning compatibility, the next four properties are used to describe multiple levels of compatibility between two object definitions. We introduce three levels: general, partial and total compatibility (propositions 1 to 3 below). Proposition 4 defines all the authorized states when two definitions are compatible. This property helps us later obtain all the operators given by the model.

In order to define general compatibility, we do not make any assumption on the types of the data elements of the two definitions F_1 and F_2.

Proposition 1 (General compatibility of definitions) *Given two object definitions F_1 and F_2, F_1 and F_2 are generally compatible, denoted $F_1 \sim F_2$, iff:*

- $Dom(F_1) \cap Dom(F_2) \neq \emptyset$,
- $\forall \ell_r \in Dom(F_1) \cap Dom(F_2)$, $\exists \jmath_r \in Dom(F_2)$ *(respectively* $\bigcup_x (\jmath_{r_x})$*) such that* $\ell_r \equiv \jmath_r$ *(respectively* $\ell_r \equiv \bigcup_x (\jmath_{r_x})$*).*

□

In the next proposition, we suppose that every type of common elements are syntacticaly equal, and we introduce the partial compatibility of two given object definitions. Therefore, we proceed to define equality between two types. We write that t = t' iff t is syntactically equal to t'. We use the general compatibility proposition to enforce the compatibility of various elements of the definitions.

Proposition 2 (Partial compatibility of definitions) *Given two object definitions F_1 and F_2, F_1 and F_2 are partially compatible, denoted $F_1 \simeq F_2$, iff:*

- $F_1 \sim F_2$,

- $F_1(\ell_r) = F_2(\jmath_r)$ *(or* $F_1(\ell_r) = \bigcup_x F_2(\jmath_{r_x})$*), what can be written* $\mathsf{t}_{\ell_r} = \mathsf{t}_{\jmath_r}$ *(or* $\mathsf{t}_{\ell_r} = \mathsf{t}_{\jmath_{r_x}}$ *foreach* x *such that* $\ell_r \equiv \bigcup_x (\jmath_{r_x})$*).*

□

Then, we characterize the total compatibility property, which is the best achievable property about compatibility of two object definitions. There exists a one-to-one correspondence between each element of the definitions.

Proposition 3 (Total compatibility of definitions)
If we consider two object definitions F_1 *and* F_2*,* F_1 *and* F_2 *are totally compatible, denoted* $F_1 \approx F_2$*, iff:*

- $Dom(F_1) = Dom(F_2)$,
- $\forall \ell_r \in Dom(F_1)$ *and* $\jmath_{r'} \in Dom(F_2)$*, we have* $\ell_r \equiv \jmath_{r'}$ *and* $F_1(\ell_r) = F_2(\jmath_{r'})$*, i.e.,* $\mathsf{t}_{\ell_r} = \mathsf{t}_{\jmath_{r'}}$.

□

Our goal is to provide mechanisms to get a complete compatibility between two object definitions F_1 and F_2, in order to transform objects in regard to their definitions and their compatibility. Our method to build such a transformation proceeds step by step, to finally obtain a complete compatibility between two definitions that represent the same data. In order to make an inventory of all existing cases, we introduce proposition 4. We also provide an example which handle all the existing cases described in the proposition.

Proposition 4
Given two object definitions respectively defined as $F_1 = [\ell_{r_1} : \mathsf{t}_1; \ldots; \ell_{r_n} : \mathsf{t}_n]$ $(n \geq 1)$ *and* $F_2 = [\jmath_{r_1} : \mathsf{t}'_1; \ldots; \jmath_{r_m} : \mathsf{t}'_m]$ $(m \geq 1)$*. If* $\ell_{r_i} \in Dom(F_1)$ *and* $\jmath_{r_j} \in Dom(F_2)$*, and* $\ell_{r_i} \equiv \jmath_{r_j}$*, we can say that one of these statements is satisfied:*

1) $F_1(\ell_{r_i}) = F_2(\jmath_{r_j})$ *(types are syntactically equals),*
2) $F_1(\ell_{r_i}) \neq F_2(\jmath_{r_j})$ *(types are not equals),*
3) $F_1(\ell_{r_i}) = \mathtt{undefined}_{\mathsf{Type}}$ *(the element does not exist),*
4) $F_2(\jmath_{r_i}) = \mathtt{undefined}_{\mathsf{Type}}$ *(the element does not exist),*
5) $F_1(\ell_{r_i}) = \bigcup_x F_2(\jmath_{r_x})$ *(composition of several elements),*
6) $F_2(\jmath_{r_i}) = \bigcup_x F_1(\ell_{r_x})$ *(composition of several elements).*

□

Example 5
1) F_1 = [*date*: string] and F_2 = [*date*: string],
2) F_1 = [*length*: string] and F_2 = [*length*: integer],
3) F_1 = [*length*: string; *checked*: boolean] and F_2 = [*length*: string], (cases 3 and 4)
4) F_1 = [*day*: string; *month*: string; *year*: string] and F_2 = [*date*: string], (cases 5 and 6)

1.5 Hierarchy

Intuitively, the idea that elements of a definition are ordered in terms of their "goodness of descriptions" is already quoted in [13]. But what does it mean? We think that two definitions could be ordered, and therefore, we can deduce when a definition can be compatible with another one.

Let $\mathcal{E}$ be a set, and $\preceq$ an order on $\mathcal{E}$ (we call $(\mathcal{E},\preceq)$ a preordered set). Two elements $x,y \in \mathcal{E}$ are consistent if there exists $z \in \mathcal{E}$ such that $x \preceq z$ and $y \preceq z$. We use such a structure $(\mathcal{E}, \preceq)$ to represent the notion of inclusion of a definition in another one. Consider two definitions F_1 and F_2. We say that $F_1 \preceq F_2$ iff F_1 is included in F_2.

Proposition 5 (Ordering on flat definition)
The information ordering $\preceq$ on flat definition types is the simplest relation satisfying:

$$[\ell_{r_1}:\mathsf{t}_1;\ldots;\ell_{r_n}:\mathsf{t}_n] \preceq [\ell_{r_1}:\mathsf{t}_1;\ldots;\ell_{r_n}:\mathsf{t}_n;\ldots;\ell_{r_m}:\mathsf{t}_m].$$

□

If we consider F_1=[*text*: string] and F_2=[*text*: string; *xpos*: integer; *ypos*: integer], we say that the structure represented by F_2 "contains" the structure represented by F_1. This intuitive notion is now formalized by a partial order.

Proposition 6
Given F_1=$[\ell_{r_1}:\mathsf{t}_1;\ldots;\ell_{r_n}:\mathsf{t}_n]$ and F_2=$[\ell_{r_1}:\mathsf{t}_1;\ldots;\ell_{r_n}:\mathsf{t}_n;\ldots;\ell_{r_m}:\mathsf{t}_m]$. F_1 and F_2 satisfy:

$$\begin{cases} F_1 \preceq F_2 \\ \\ F_1 \simeq F_2 \end{cases}$$

□

Then, we can deduce from definition 9 and proposition 6 that, considering $F_1 = [\ell_{r_1}: \mathsf{t}_1; \ldots;\ell_{r_n}: \mathsf{t}_n]$ and $F_2 = [\jmath_{r_1}: \mathsf{t}'_1; \ldots;\jmath_{r_n}:\mathsf{t}'_n]$, we have:

$$\begin{array}{c} F_1 \preceq F_2 \text{ iff } \mathsf{t}_n = \mathtt{undefined}_{\mathsf{Type}} \text{ or} \\ \ell_{r_n} \equiv \jmath_{r_n} \text{ or} \\ \ell_{r_n} \equiv \bigcup_x(\jmath_{r_x}) \end{array}$$

We now have an order for every object definition, and can interpret this ordering of "goodness of description". For example, F_1 = [*text*: string ; *xpos*: $\mathtt{null}_{\mathsf{integer}}$; *ypos*: $\mathtt{null}_{\mathsf{integer}}$] and F_2 = [*text*: string; *xpos*: integer; *ypos*: integer] can be ordered as follows:

$$F_1 \preceq F_2$$

Moreover, we have:

$$F'_1 = [\textit{text}\text{: string}] \preceq F_2$$

2 Operators

In the previous section, we described the concepts of the data model. However, we believe a data model is not sufficient to allow for data exchange and data sharing. It only enables object description. To make a tool use the objects of another tool, it is sometimes necessary to transform the object of the latter, i.e. to make the objects fully compatible. In this framework, the data model is provided with a set of atomic operators that are used to make totally compatible two generally or partially compatible object definitions. Thus, the transformation cycle consists in:

- analyzing the level of compatibility of the two object definitions,
- making totally compatible the two object definitions,
- transforming the object instances.

We introduce in this section rewritten expressions of definitions. It allows for the restructuration both of the definitions and the instances. Each rewritten expression is an "atomic" rewritten expression. It means that a complete transformation can be represented by a set of rewritten expressions, and that each step of a transformation is represented by such an expression. This view allows us to provide clear and simple rewritten rules, with no huge complexity. It also makes the study of properties attached to expressions easier, for instance composition support.

We first present rewritten rules that define accepted transformations upon definitions. These rules are divided into two parts: structural rewritten rules and content rewritten rules. Then, we introduce operators derived from these rules.

2.1 Structural Rewritten Rules

We discuss here rewritten expressions acting on structures. We have to take into account that it should be possible to transform instances. So we establish a difference between "structural" rewritten expressions and "content" rewritten expressions.

We first present what we call Structural Rewrite Rules (SRR). At this level, we only show how such a rule acts. We do not provide any information on how it is possible to effectively transform structures and instances. Some of these SRR are already quoted in [2].

Rule 1 Given the following definition:

$[\ell_1\colon \mathsf{t}_1;\ \ldots;\ \ell_{i-1}\colon \mathsf{t}_{i-1};\ \ell_i\colon[\ell_k\colon \mathsf{t}_k;\ \ldots;\ \ell_{k+m}\colon \mathsf{t}_{k+m}];\ \ell_{i+1}\colon \mathsf{t}_{i+1};\ \ldots;\ \ell_n\colon \mathsf{t}_n]$

It could be rewritten as:

$[\ell_1\colon \mathsf{t}_1;\ \ldots;\ \ell_{i-1}\colon \mathsf{t}_{i-1};\ \ell_k\colon \mathsf{t}_k;\ \ldots;\ \ell_{k+m}\colon \mathsf{t}_{k+m};\ \ell_{i+1}\colon \mathsf{t}_{i+1};\ \ldots;\ \ell_n\colon \mathsf{t}_n]$

Rule 2 Given the following definition:

$[\langle\ell_1\colon \mathsf{t}_1;\ \ldots;\ \ell_{i-1}\colon \mathsf{t}_{i-1};\ \ell_i\colon\langle\ell_k\colon \mathsf{t}_k;\ \ldots;\ \ell_{k+m}\colon \mathsf{t}_{k+m}\rangle;\ \ell_{i+1}\colon \mathsf{t}_{i+1};\ \ldots;\ \ell_n\colon \mathsf{t}_n\rangle]$

It could be rewritten as:

$[\langle\ell_1\colon \mathsf{t}_1;\ \ldots;\ \ell_{i-1}\colon \mathsf{t}_{i-1};\ \ell_k\colon \mathsf{t}_k;\ \ldots;\ell_{k+m}\colon \mathsf{t}_{k+m};\ \ell_{i+1}\colon \mathsf{t}_{i+1};\ \ldots;\ \ell_n\colon \mathsf{t}_n\rangle]$

Rule 3 Consider the following definition:
$[\ell_1\colon \mathsf{t}_1; \ldots; \ell_i\colon\langle\ell_k\colon \mathsf{t}_k; \ldots; \ell_{k+m}\colon \mathsf{t}_{k+m}\rangle; \ldots; \ell_n\colon \mathsf{t}_n]$
It could be rewritten as:
$[(\![\ell_1\colon \mathsf{t}_1; \ldots; \ell_k\colon \mathsf{t}_k; \ldots; \ell_n\colon \mathsf{t}_n]; \ldots; [\ell_1\colon \mathsf{t}_1; \ldots; \ell_{k+n}\colon \mathsf{t}_{k+n}; \ldots; \ell_n\colon \mathsf{t}_n])\!)]$

Rule 4 Consider the definition:
$[\{\langle\ell_1\colon \mathsf{t}_1; \ldots; \ell_n\colon \mathsf{t}_n\ \rangle\}]$
It could be rewritten as:
$[[\{\ell_1\colon \mathsf{t}_1\ \}; \ldots; \{\ell_n\colon \mathsf{t}_n\}]]$

Rule 5 Consider the definition:
$[\ell_1\colon \mathsf{t}_1; \ldots; \ell_k\colon \mathsf{t}_k; \ldots; \ell_n\colon \mathsf{t}_n]$
It could be rewritten as:
$[\ell_1\colon \mathsf{t}_1; \ldots; \jmath_k\colon \mathsf{t}_k; \ldots; \ell_n\colon \mathsf{t}_n]$

Properties As underlined above, these rules are atomic. The following example shows how we can combine some of these rules. Suppose the two following definitions: $F = [\mathit{BirthDate}\colon \mathsf{date}]$ and $F' = [\mathit{Birth}\colon[\mathit{day}\colon \mathsf{integer}; \mathit{month}\colon \mathsf{integer}; \mathit{year}\colon \mathsf{integer}]]$. We apply the following rules:

$\phi = \text{rewrite}(\mathit{date}\colon \mathsf{date} \rightarrow \mathit{date}\colon \mathsf{integer})$
$\psi = \text{rewrite}(\mathit{date}\colon \mathsf{integer} \rightarrow \mathit{date}\colon[\mathit{day}\colon \mathsf{integer}; \mathit{month}\colon \mathsf{integer}; \mathit{year}\colon \mathsf{integer}])$
$\xi = \text{rewrite}(\mathit{BirthDate} \rightarrow \mathit{Birth})$

and deduce:
$F' = \xi(\psi(\phi(F)))$ and $F = \xi^{-1}(\psi^{-1}(\phi^{-1}(F')))$

2.2 Content Rewritten Rules

We now present some rewritten rules, called Content Rewrite Rules (CRR), that are necessary to take into account some differences based on types, degrees or scale expression of attributes.

Rule 6 Consider the following definition:
$[\ell_1\colon \mathsf{t}_1; \ldots; \ell_k\colon \mathsf{t}_k; \ldots; \ell_n\colon \mathsf{t}_n]$
It could be rewritten as:
$[\ell_1\colon \mathsf{t}_1; \ldots; \ell_k\colon \mathsf{t}'_k; \ldots; \ell_n\colon \mathsf{t}_n]$

Rule 7 Consider the following definition:
$[\ell_1\colon \mathsf{t}_1; \ldots; \ell_k\colon \mathsf{t}_k; \ldots; \ell_n\colon \mathsf{t}_n]$
It could be replaced by:
$[\ell_1\colon \mathsf{t}_1; \ldots; \ell_k\colon \mathsf{t}'_k; \ldots; \ell_n\colon \mathsf{t}_n; \ell_{n+1}\colon \mathsf{t}_{n+1}]$

Rule 8 Consider the following definition:

$[\ell_1\colon \mathsf{t}_1; \ldots; \ell_k\colon \mathsf{t}_k; \ldots; \ell_{k+1}\colon \mathsf{t}_{k+1}; \ldots; \ell_n\colon \mathsf{t}_n]$

If ℓ_k and ℓ_{k+1} are linked by a calculus relation, named ϕ, then we could rewrite the definition as:

$[\ell_1\colon \mathsf{t}_1; \ldots; \ell_k\colon \mathsf{t}_k; \ldots; \phi(\ell_k)\colon \mathsf{t}_{k+1}; \ldots; \ell_n\colon \mathsf{t}_n]$

Rule 9 Assume the following definition:

$[\ell_1\colon \mathsf{t}_1; \ldots; \ell_k\colon \mathsf{t}_k; \ldots; \ell_n\colon \mathsf{t}_n]$

It could be rewritten as:

$[\ell_1\colon \mathsf{t}_1; \ldots; \phi(\ell_k)\colon \mathsf{t}_k; \ldots; \ell_n\colon \mathsf{t}_n]$

Rule 10 Assume the following definition:

$[\ell_1\colon \mathsf{t}_1; \ldots; \ell_{k-2}\colon \mathsf{t}_{k-2}; \ell_{k-1}\colon\mathsf{t}_{k-1}; \ell_k\colon \mathsf{t}_k; \ell_{k+1}\colon \mathsf{t}_{k+1}; \ldots; \ell_n\colon \mathsf{t}_n]$

It could be rewritten as:

$[\ell_1\colon \mathsf{t}_1; \ldots; \ell_{k-2}\colon \mathsf{t}_{k-2}; \ell'_k\colon \mathsf{t}'_k; \ell_{k+1}\colon \mathsf{t}_{k+1}; \ldots; \ell_n\colon \mathsf{t}_n]$

2.3 The operators

We introduce in this section ten operators derived from the rules previously defined. For each operator, we present the rule it corresponds to, and a brief overview.

FOLD and UNFOLD operators

These two operators implement the first rule. The rule 1 says that if there exists a nested construction, then this construction can be "flattened". Conversely, a flat structure can be rewrite to become a nested structure. So these two operators act upon structures.

Example: A typical example of use of such operators is based upon names. Let's assume $F = [\mathit{Name}\colon [\mathit{LastName}\colon \mathsf{string}; \mathit{FirstName}\colon \mathsf{string}]]$. Then, we say that the UNFOLD operator transforms F into F' as:

$$F' = \text{UNFOLD}(F,\ \mathit{Name}) = [\mathit{LastName}\colon \mathsf{string}; \mathit{FirstName}\colon \mathsf{string}]$$

COMPOSE and UNCOMPOSE operators

Rule 2 says that it is possible to "uncompose" the union of attributes already enclosed into a union. Conversely, it says that it is possible to compose several attributes that can be view generically the same in an union. This is the goal of the COMPOSE and UNCOMPOSE operators.

Example: Given $F = [\mathit{Locality}\colon \langle \mathit{BirthCity}\colon \mathsf{string}; \mathit{BirthCountry}\colon \mathsf{string}\rangle]$, the UNCOMPOSE operator transforms F into F' as:

$$F' = \text{UNCOMPOSE}(F,\ \mathit{Locality}) = [\mathit{Locality}\colon \mathsf{string}]$$

DISTRIBUTE and UNDISTRIBUTE operators

The third rule says that it is possible to "rise" the union constructor through a definition. The application of this rule gives the DISTRIBUTE and UNDISTRIBUTE operators. Each attribute of a union is "distribute" through the given definition.

Example: Given F = [*Name*: string; *BirthDate*: date; *Locality*: ⟨*BirthCity*: string; *BirthCountry*: string⟩], the definition can be transformed into a definition named F' defined as:

$$F' = \text{DISTRIBUTE}(F, \textit{Locality}) = [\langle[\textit{Name}\text{: string; }\textit{BirthDate}\text{: date; }\textit{BirthCity}\text{: string}]; [\textit{Name}\text{: string; }\textit{BirthDate}\text{: date; }\textit{BirthCountry}\text{: string}]\rangle]$$

SET and UNSET operators

The fourth rule takes into account sets of union of attributes. It separates the union into several sets. Therefore, SET and UNSET operators rise the union constructor through the collection constructor.

Example: Let's still consider the definition F given by F = [*SisterBrother*: {⟨*SisterName*: string; *BrotherName*: string⟩}]. Then the UNSET operator is decribed as:

$$F' = \text{UNSET}(F, \textit{SisterBrother}) = [\textit{SisterBrother}\text{:}[\{\textit{SisterName}\text{: string; }\textit{BrotherName}\text{: string}\}]]$$

RENAME operator

This operator implements the last structural rule, rule 5. It allows to rename an attribute of a definition, in order to make attributes names more meaningful or identical to other attributes having the same meaning.

Example: A definition F = [*Birth*: date] could be replaced by the new definition F':

$$F' = \text{RENAME}(F, \textit{Birth}, \textit{BirthDate}) = [\textit{BirthDate}\text{: date}]$$

CAST operator

The CAST operator modifies an attribute's type of role r to make it compatible with another attribute of the same role (Rule 6). It means that this operator could transform (i.e. cast) the type t into a type t', with all the modifications that may be induced on the objects defined by the given definition.

Example: Given F = [*task*: string; *length*: string; *code*: integer]. The CAST operator transforms this definition into F' such as:

$$F' = \text{CAST}(F, \textit{length}, \text{integer}) = [\textit{task}\text{: string; }\textit{length}\text{: integer; }\textit{code}\text{: integer}]$$

At the object level, an object o = [*task*= "Design"; *length*= "6"; *code*= 10] is converted into o' such that o' = [*task*= "Design"; *length*= 6; *code*= 10].

COMPLETE and SKIP operators

COMPLETE allows a one-to-one correspondence between two definitions. It is a way to add some attributes to a definition in order to have two definitions with the same degree. Therefore, with such an operator, it is easy to provide a complete compatibility between two definitions with different degrees. At the

object level, null values of the attribute's type are given to instances introduced in an object. SKIP allows to cut some attributes from a definition. These two operators realize the rule 7.

Example: If we suppose two definitions F and F', respectively $F = [task$: string; *length*: integer] and $F' = [task$: string; *length*: integer; *Programmer's_Name*: string]. Then, COMPLETE is defined as:

$$F' = \text{COMPLETE}(F,\ \textit{Programmer's\_Name},\ \text{string}) = [\textit{task}\text{: string};\ \textit{length}\text{: integer};\ \textit{Programmer's\_Name}\text{: string}] \text{ and } F = \text{SKIP}(F',\ \textit{Programmer's\_Name})$$

At the object level, using the COMPLETE operator, we have, if o=[*task* = "Design"; *length* = 10], then o' = [*task* = "Design"; *length* = 10; *Programmer's_Name* = ""].

FORMAT operator

The FORMAT operator includes differences on aggregation levels. It implements the rule 8. It is an easy way to transform the relation between two connected attributes and allows for a representation transformation of a piece of information.

Example: We introduce a definition on time to spend for a given task. Consider $F = [task$: string; *start_date*: date; *duration*: integer]. If we want a new definition which gives the duration of the given task with two attributes: *start_date* and *end_date*, we write:

$$F' = \text{FORMAT}(F,\ \textit{duration},\ \textit{end\_date},\ \text{date},\ \textit{start\_date} + \textit{duration}) = [\textit{task}\text{: string};\ \textit{start\_date}\text{: date};\ \textit{end\_date}\text{: date}]$$

SCALE operator

This operator modifies the format of an attribute, e.g., a numerical value unit. Therefore, it solves the problem which can arise when two attributes are expressed in differents units. It is the application operator of rule 9. This operator can lead to a modification of the type of the considered attribute.

Example: On the time example described above, we now assume a duration expressed in hours, and another expressed in minutes. We have the following definition F: $F = [$*duration_in_hours*: integer]. We can apply the SCALE operator to transform this attribute in: $F' = [$*duration_in_minutes*: integer], and write:

$$F' = \text{SCALE}(F,\ \textit{duration\_in\_hours},\ \textit{duration\_in\_minutes},\ \text{integer},\ \text{x} * 60) = [\textit{duration\_in\_minutes}\text{: integer}]$$

GROUP and UNGROUP operators

The UNGROUP and GROUP operators are related to the last rule. It allows for splitting or composition of one or several attributes. The COMPOSE operator solves the usual case when several attributes are grouped into a unique one in the target definition. Conversely, the UNGROUP solves the opposite problem.

Example: Consider a trivia example. We introduce a date attribute saved as a string on one hand, and as three separate strings on the other hand. So, we have

the definition F given by F = [*start_date*: string]. We would like to obtain such a definition: F' = [*start_date_day*: string; *start_date_month*: string; *start_date_year*: string]. We write:

F' = UNGROUP(F,*start_date*,(*start_date_day*,*start_date_month*,*start_date_year*))= [*start_date_day*: string; *start_date_month*: string; *start_date_year*: string]

At the object level, considering an object o = [*start_date*= "1/8/1992"], and using the UNGROUP operator, we have a new object o' = [*start_date_day* = "1"; *start_date_month* = "8"; *start_date_year* = "1992"].

3 An example

In this section, we illustrate on a short example the problems of data exchange between two tools. We first describe the context, and then give an example on the exchange of two figures that contain text data.

We assume that two tools named respectively A and B need to share some data. Both tools manage their own data, and we assume that one of them needs data from the other one. Assume that A needs to access B's data and respectively, B needs to access A's data. The process which enable the transformation of the data can be split into 3 main steps:

(*i*) get both A's and B's data definitions, as explained in Section 1 of the model. These definitions include the semantic of the data and will help in transforming the data,

(*ii*) point out the differences between definitions, by analyzing roles and types. As a result, we are able to describe differences between the two data's definitions (e.g. semantic differences, removed attributes, added attributes, type of an attribute),

(*iii*) make the definitions compatible in order to transform data. There are two sub-steps:

- compatibility of the definitions,
- application on the data.

After this general description of the process, we are going to describe each step in details. Before, we assume the following elements: the tool A stores its object text data as:

```
text('black',165,0,2,0,5,1,2,0,0,162,58,0,0,24,5,0,0,0,
                    0,["Sample figure"]).
```

While the second tool, B, stores its data as:

```
4 0 16 24 0 -1 0 0.000 2 24 189 164 30 Sample figure
```

```
text(                    -> TEXT object
'black',                 -> color
165,                     -> x draw origin
0,                       -> y draw origin
2,                       -> font number
0,                       -> font style (0,1,2 or 3)
5,                       -> font size
1,                       -> number of lines
2,                       -> text justify (0,1, or 2)
0,                       -> text rotate
0,                       -> pen pattern
162,                     -> line length
58,                      -> line width
0,                       -> object identifier
0,                       -> font DPI
24,                      -> ascendant
5,                       -> descendant
0,                       -> fill pattern
0,                       -> text vertical spacing
0,                       -> rotation
0,                       -> locked
["Sample figure"]).-> text
```

(a)

```
[object      : string;
 color       : string : '#';
 x           : integer;
 y           : integer;
 fontID      : integer;
 style       : integer : $(0,1,2,3);
 fontSize    : integer;
 linesNumber : integer;
 justify     : integer : $(0,1,2);
 textRot     : integer;
 textPattern : integer;
 lineLength  : integer;
 lineWidth   : integer;
 objectID    : integer;
 fontDPI     : integer;
 ascent      : integer;
 descent     : integer;
 fillPattern : integer;
 vertSpacing : integer;
 rotation    : integer;
 locked      : integer;
 text        : string : ["#"];
]
```

(b)

Fig. 1. TEXT object attributes – Tool A

Figures 1(a) and 2(a) give values on the left side of the figure, and A's and B's data semantics on the right side. This is not the definition of the data, but only the semantic attached to each value. Figures 1(b) and 2(b) provide for each attribute the role name and its respective type. For instance, in figure 1(b), we notice the attribute x of type integer which corresponds to the **x draw origin** of the text. With figures 1(b) and 2(b), we retrieve object definitions introduced in the model in the first section. At this point, we assume that we get two object definitions, called F_A and F_B.

The second step concerns the matching of the definitions in order to provide a set of differences between them. To achieve this step, we use the hierarchy property proposed in section 1.4. It gives an order of inclusion between the two definitions. By analyzing them, we get the following:

- **common attributes**: *object, justify, fontID, fontSize, textPattern, color, style, lineWidth, lineLength, x, y, text, rotation, fontDPI,*
- **attributes in F_A missing in F_B**: *linesNumber, objectID, ascent, descent, fillPattern, vertSpacing, locked,*
- **attributes in F_B missing in F_A**: none.

Then, the third and last step concerns the definitions and objects transformations. To make the definitions fully compatible, we have to apply the following sequence of operators (we now assume that tool B wants to access A's data so the transformation will translate A's data in B's format).

```
4               -> object type (TEXT)
0               -> text justify (0, 1 ou 2)
16              -> font
24              -> font size
0               -> pen pattern
-1              -> color
0               -> fontDPI
0.000           -> angle
2               -> text style (1, 2, 4, 8 ou 16)
24              -> width
189             -> length
164             -> x draw origin
30              -> y draw origin
Sample figure   -> text
```

(a)

```
[object      : integer;
 justify     : integer : $(0,1,2);
 fontID      : integer;
 fontSize    : integer;
 textPattern : integer;
 color       : integer;
 fontDPI     : integer;
 rotation    : real;
 style       : integer : $(1,2,4,8,16);
 lineWidth   : integer;
 lineLength  : integer;
 x           : integer;
 y           : integer;
 text        : string]
```

(b)

Fig. 2. TEXT object attributes – Tool B

First of all, we take into account common attributes. By ordering the definition F_A and skipping unused attributes (those which are not in the common set of attributes described above), and obtain a new definition called F'_A defined by:

$$F'_A = [object\text{:string}; justify\text{:integer}; fontID\text{:integer}; fontSize\text{:integer}; textPattern\text{:integer}; color\text{:string}; fontDPI\text{:integer}; rotation\text{:integer}; style\text{:integer}; lineWidth\text{:integer}; lineLength\text{:integer}; x\text{:integer}; y\text{:integer}; text\text{:string}]$$

Next, we are going to make the two definitions totally compatible (as explained in Proposition 3). To achieve this goal, we will apply the following sequence of operators:

- **Object attribute transformation**:
  $SCALE_{F'_A \to F_B}(F'_A, object, object, \text{integer}, \mathcal{F}) = [object\text{: integer}; justify\text{: integer}; fontID\text{: integer}; fontSize\text{: integer}; textPattern\text{: integer}; color\text{: string}; fontDPI\text{: integer}; rotation\text{: integer}; style\text{: integer}; lineWidth\text{: integer}; lineLength\text{: integer}; x\text{: integer}; y\text{: integer}; text\text{: string}]$
  where $\mathcal{F}$ is defined by:
  $\mathcal{F}$: string_value $\leftrightarrow$ integer_value,
  and the following translation table:

Tool A – String value	Tool B – Integer value
'oval'	1
'poly'	2
'polygon'	3
'text'	4
'arc'	5

  At the object level, we obtain the new object:
  $o = [object= 4; justify= 2; fontID= 2; fontSize= 5; textPattern= 1; color=$

'black'; *fontDPI*= 0; *rotation*= 0; *style*= 0; *lineWidth*= 58; *lineLength*= 162; *x*= 165; *y*= 0; *text*= 'Sample figure']

- **FontID attribute transformation**:
  $SCALE_{F'_A \to F_B}(F'_A$, *fontID*, *fontID*, integer, $\mathcal{G}$) = [*object*: integer; *justify*: integer; *fontID*: integer;*fontSize*: integer; *textPattern*: integer; *color*: string; *fontDPI*: integer; *rotation*: integer; *style*: integer; *lineWidth*: integer; *lineLength*: integer; *x*: integer; *y*: integer; *text*: string]
  where $\mathcal{G}$ is defined as follows:
  $\mathcal{G}$: integer_value $\to$ integer_value * 8.
  At the object level, we obtain:
  o = [*object*= 4; *justify*= 2; *fontID*= 16;*fontSize*= 5; *textPattern*= 1; *color*= 'black'; *fontDPI*= 0; *rotation*= 0; *style*= 0; *lineWidth*= 58; *lineLength*= 162; *x*= 165; *y*= 0; *text*= 'Sample figure']
- **FontSize attribute transformation**:
  $SCALE_{F'_A \to F_B}(F'_A$,*fontSize*, *fontSize*, integer, $\mathcal{H}$) = [*object*: integer; *justify* :integer; *fontID*: integer;*fontSize*: integer; *textPattern*: integer; *color*: string; *fontDPI*: integer; *rotation*: integer; *style*: integer; *lineWidth*: integer; *lineLength*: integer; *x*: integer; *y*: integer; *text*: string]
  where $\mathcal{H}$ is defined by:
  $\mathcal{H}$: integer_value $\leftrightarrow$ integer_value,
  and the following translation table:

Tool A – Integer value	Tool B – Integer value
0	8
1	10
2	12
3	14
4	18
5	20
6	24

  At the object level, we have:
  o = [*object*= 4; *justify*= 2; *fontID*= 16;*fontSize*= 20; *textPattern*= 1; *color*= 'black'; *fontDPI*= 0; *rotation*= 0; *style*= 0; *lineWidth*= 58; *lineLength*= 162; *x*= 165; *y*= 0; *text*= 'Sample figure']
- **Color attribute transformation**:
  $SCALE_{F'_A \to F_B}(F'_A$, *color*, *color*, integer, $\mathcal{I}$) = [*object*: integer; *justify*: integer; *fontID*: integer;*fontSize*: integer; *textPattern*: integer; *color*: integer; *fontDPI*: integer; *rotation*: integer; *style*: integer;*lineWidth*: integer; *lineLength*: integer; *x*: integer; *y*: integer; *text*: string]
  where $\mathcal{I}$ is defined by:
  $\mathcal{I}$: string_value $\leftrightarrow$ integer_value,
  and the following translation table:

Tool A – String value	Tool B – Integer value
'black'	-1
...	...

At the object level, we obtain the new object:
o = [*object*= 4; *justify*= 2; *fontID*= 16;*fontSize*= 20; *textPattern*= 1; *color*= -1; *fontDPI*= 0; *rotation*= 0; *style*= 0; *lineWidth*= 58; *lineLength*= 162; x= 165; y= 0; *text*= 'Sample figure']

- **Rotation attribute type cast**:
  $CAST_{F'_A \rightarrow F_B}(F'_A$, *rotation*, real) = [*object*: integer; *justify*: integer; *fontID*: integer;*fontSize*: integer; *textPattern*: integer; *color*: integer; *fontDPI*: integer; *rotation*: real; *style*: integer; *lineWidth*: integer; *lineLength*: integer; x: integer; y: integer; *text*: string]
  At the object level, we have:
  o = [*object*= 4; *justify*= 2; *fontID*= 16;*fontSize*= 20; *textPattern*= 1; *color*= -1; *fontDPI*= 0; *rotation*= 0.000; *style*= 0; *lineWidth*= 58; *lineLength*= 162; x= 165; y= 0; *text*= 'Sample figure']
- **Style attribute transformation**:
  $SCALE_{F'_A \rightarrow F_B}(F'_A$, *style*, *style*, integer, $\mathcal{J}$) = [*object*: integer; *justify*: integer; *fontID*: integer;*fontSize*: integer; *textPattern*: integer; *color*: string; *fontDPI*: integer; *rotation*: integer; *style*: integer; *lineWidth*: integer; *lineLength*: integer; x: integer; y: integer; *text*: string]
  where $\mathcal{J}$ is defined as follows:
  $\mathcal{J}$: $\mathcal{J}_0 = 1$
  $\mathcal{J}_i = \mathcal{J}_{i-1} * 2$
  At the object level, we have:
  o = [*object*= 4; *justify*= 2; *fontID*= 16;*fontSize*= 20; *textPattern*= 1; *color*= -1; *fontDPI*= 0; *rotation*= 0.000; *style*= 1; *lineWidth*= 58; *lineLength*= 162; x= 165; y= 0; *text*= 'Sample figure']

This sequence of operators gives a new object compatible with B's format. By now, the o object can be accessed by tool B. Several comments can be made regarding this example:

- first of all, we only show the A to B data translation. There is no problem to do the opposite translation, but it takes longer. We give here some of the operators that should be use to do this. The COMPLETE operator should be used to make the B's definition totally compatible with A's. Some of the SCALE operators we give in the above example are conserved in the translation. In fact, only functions not using translation tables have to be changed, while bijectives functions (those with translation tables) can be preserved.
- in figures 1(b) and 2(b), there is some syntactical information. For instance, we can notice the *color* attribute describes as: *color*: integer: `'#'`. This definition indicates to the parser that the *color* attribute, known as a string attribute, is between two simple quotes. With such a method, we skip unnecessary information.
  We have also introduced a way to describe default values of an attribute. This was done with the $ character. Between the parenthesis, we give default values of the attribute.

finally, we would like to emphasize that there exists a important step that comes before all those described in this section. It concerns the need of having a unique notation of attributes of the same role. Our method supposes this unicity. A way to have this unicity is to take CDIF ([6]) semantic about attribute of text in a figure. Of course, definitions are rather poor against tool definitions, but it is quite normal because it gives only significant attributes. We do not want to take the entire CDIF format, for the reasons explained in the following section, but only the semantic related to attributes. This allows to unify attribute's roles, and for instance to call the *style* attribute in a common manner (e.g. *FONTSTYLE*). The TEXT object definition in CDIF is presented in figure 3.

```
pic_tb              (UNBOUNDEDTEXT cdif_body pt inst_num
                                   symbol_name font_style)

cdif_body           (BODY cdif_text_body cdif_graph_body)

pt                  (PT integer integer)

inst_num            (INSTANCENUMBER integer)

symbol_name         (SYMBOLNAME string)

font_style          (FONTSTYLE string)

cdif_text_body      (TEXTBODY ({string}))

cdif_graph_body     (GRAPHBODY)

string              "{legal_char}"

legal_char          abcdefghijklmnopqrstuvwxyz
                    ABCDEFGHIJKLMNOPQRSTUVWXYZ
                    0123456789
                    !@#$%^&*()-=_+`[]{};:'\|/?.,><

integer             0 to (2**16)-1
```

Fig. 3. TEXT object attributes – CDIF

4 Comments

We have introduced a data model to conceptually characterize objects and their definitions, where each object can be transformed by some operators. Our method for transforming data elements can be used both in *a priori* and *a posteriori* views. First, the model should be compatible with each tool without changing its internal data format. Then, for each data element definition, we construct a mapping that translates a definition into another for every situation. Finally, a set of operators allow us to build a translation procedure.

Let's make a few comments in regard of other methods. We believe that a common data format (like CDIF) does not solve all the problems. Such a solution implies that each tool changes its internal data format to manage a commonly accepted format. Such a process is advantageous in an *a priori* situation, but has drawbacks in an *a posteriori* case (for instance, if the tool vendor does not want to change the tool format). So, integration could be difficult in such cases. With our model, each tool keeps its own data format and the translation is vendor independent. We believe this is well suited for an *a posteriori* situation and for moving a set of tools to a fine integrated system. Another strong point is the ability to control data without creating and managing a new common format. The format definition step is often very long, because of getting each participant an agreement on each part of the data format. The consensus step may be long, and is only *a priori* oriented. As emphasized before, our model allows for working in both *a posteriori* and *a priori* situations. So, it is useful to integrate old tools with new tools or to provide a way to change a tool in order to allow environment's evolution. These two ways of implementing data integration clearly complement each other, as they cover different steps of data integration problem solving.

We have seen in our model how each tool continues to manage its data and how there exists an "on the fly" translation when another tool wants to access these data. This characteristic provides simplicity and efficiency (data irredundancy). The identification and the representation of data elements, and the semantic interpretation are already made, and so, data translation and integration process are quite easy. Moreover, some level of automation is allowed. Our model presents solutions on current problems described before: the *synonyms* and the *homonyms* problems are dealt with (definition of each object, compatibility levels, GROUP and UNGROUP operators) but not entirely solved, we also suggest a solution to the *loss of information* problem (COMPLETE operator), and then take into account the *different level of scale* problem (FORMAT operator). We provide answers about syntactic constraints, but all the syntactic and semantic constraints part needs further research. Then, the approach offers a preferred extensibility and evolution path. It allows continued operation of existing applications to remain unchanged. Moreover, we feel that our approach makes changes and modification processes faster[1] than the adoption of a common data format as suggested by the work around CDIF, for instance.

An essential part of future research devoted to several unsolved problems, will be as described below. There exists at least two kinds of improvements: first, we distinguish model improvements (quoted with $\diamond$), and then we present implementation improvements (quoted with $\bullet$):

- $\diamond$ *the "semantic constraints" problem.* If a tool has to manage data produced by another tool, it must respect semantic constraints related by the owner tool. For instance, when a tool A considers a constraint saying that the value

[1] More work has to be done on the common data format and on changing tools internal data format on the other hand.

of an attribute x is smaller than the value of an attribute y, another tool B which has to manage A's data should ensure this constraint. So we must provide in our model such a property to enforce semantic constraints,

- ◇ *the "world representation" problem.* We need to investigate all the possible cases of usual semantic that can occur. In particular, we have to account for specific cases. A starting point to solve this problem could be to get evolutions of CDIF format to provide classes of common attribute sets,
- *the "getting representation" problem.* Each tool must provide its data format in a common manner. It is important for the definitions to be described with the same model, so it can be useful to provide a frame to enforce tools to use the same symbol semantic description. Analyzis and comparison steps will be easier if we can use a complete and a common semantic description,
- *the data integrity problem.* We must provide a consistency implementation mechanism to ensure the global consistency criteria. One of the main problem is to track multiple data updates, in various format, and to provide a mechanism which respects the semantic constraints of the proprietary tool. Our approach currently does not support concurrent access to data by various tools. A transaction mechanism may be needed. But, for the moment, we consider that these problems are implementation dependent and we do not deal with them in the model.

As said before, this model needs further improvements. Rather than the classical tool oriented view, we think that a data oriented view will be a great challenge to solve some conflicting situations such as homonymy and synonymy problems. "*One data file, various tools able to manage these data*" scheme is preferred to the classical "*one tool, one data format*" scheme. To provide this control, we will introduce a group (or class) concept ([14]), where various tools with the same functionalities are regrouped into classes. Each tool in a class is closely related to other tools in the same class. For each class, there exists a common data semantic which allows for a high automation of the translation process. The new scheme can be describe as: "*given data, give me a list of authorized tools I could use to manage these data with maximum automation*". Our main effort is currently to find an efficient way to solve some of these problems (in adding new pieces of information in the model itself, i.e. *constraints*).

From a practical point of view, there exists an ongoing UNIX[2] implementation, which will be implemented on a PCTE [5] platform.

References

1. S. Abiteboul, P.C. Fischer, and H.J. Schek Eds. *Nested Relations and Complex Objects in Databases*, volume 361 of *Lecture Notes in Computer Science*. Springer-verlag, 1989.
2. S. Abiteboul and R. Hull. Restructuring Hierarchical Database Objects. *Theorical Computer Science*, 62:3–38, 1988.

[2] UNIX is a registered trademark by AT&T.

3. N. Boudjlida and H. Basson. Integration Mechanisms in ALF, a Process Model-Based Project Support. In IEEE Computer Society Press, editor, *Proceedings of the 2nd International Conference on System Integration, "Managing Large-Scale Integration in the 1990s"*, Morristown, NJ, June 1992.
4. L. Cardelli and P. Wegner. On Understanding types, Data Abstraction, and Polymorphism. *ACM Computing Surveys*, 17(4):471–522, December 1985.
5. ECMA. A Reference Model for Frameworks of Computer-assisted Software Engineering Environments. Technical report, ECMA TR/55 (2nd edition), December 1991.
6. EIA. CDIF Organization and procedure manual. EIA/PN-2329, January 1990.
7. J. V. Guttag. *The Specification and Application to Programming of Abstract Data Types*. PhD thesis, University of Toronto – Computer Science Departement, 1975. Report CSRG-59.
8. J. V. Guttag. Notes on type abstraction. *IEEE Transaction on Software Engineering*, 6(1):13–23, January 1980.
9. R. Hull and C.K. Yap. The Format Model: A Theory of Database Organization. *Journal of the ACM*, 31(3):518–537, July 1984.
10. C. Lécluse and P. Richard. The O_2 Data Model. Technical report, Altair 39-89, October 1989.
11. A. Motro. Superviews: Virtual Integration of Multiple Databases. *IEEE Transactions on Software Engineering – Vol. SE-13, No. 7*, pages 785–798, July 1987.
12. S. Navathe, E. Ramez, and J. Larson. Integrating User Views in Database Design. *IEEE Computer*, pages 50–62, January 1986.
13. A. Ohori. Semantics of Types for Database Objects. *Theorical Computer Science*, 76:53–91, 1990.
14. O. Perrin and N. Boudjlida. Tool Integration in Integrated Software-Engineering Environments: data dimension. In *Fifth International Conference on Software Engineering and its applications*, pages 95–105, Toulouse, December 1992.
15. A. P. Sheth and J. A. Larson. Federated Database Systems for Managing Distributed, Heterogeneous, and Autonomous Databases. *ACM Computing Surveys*, 22(3):183–236, September 1990.
16. J.M. Smith and D.C.P. Smith. Database abstractions: Aggregation and generalization. *ACM Transactions on Database Systems*, 2(2):105–133, June 1977.
17. I. Thomas and B. A. Nejmeh. Definitions of Tool Integration for Environments. *IEEE Software*, pages 29–35, March 1992.