A Formal Framework and a Procedural Approach for Data Interchange*

N. Boudjlida and O. Perrin
CRIN - Bâtiment LORIA
Nancy I University - Campus Scientifique
B.P. 239, 54506 Vandoeuvre-lès-Nancy CEDEX – FRANCE

Abstract

Data interchange between tools in software engineering environments or between federated databases requires agreement on data formats and accurate mechanisms to map data from a representation into another one. Static approaches often suggest a common data interchange format with ad hoc data converters. This paper proposes a dynamic approach where only one converter is needed. The approach is based on an abstract representation model to describe the data structures. The model encompasses a set of operators that aim at making two structures compatible. The same set of operators apply to data instances. The fundamentals of the approach are that the data integration process is viewed as a process that builds a subtype relationship among the data of the tools that need data interchange. This paper discusses the data model, the associated operator set and rewriting rules for structure compatibility. It also provides a brief overview of the system that implements the proposed approach.

1 Introduction

Software systems integration [14, 17] covers various facets including the ability to federate different, possibly heterogeneous, systems into a unique one. This is typically the case when one wants to federate some existing database systems [15, 10, 9] and applications that may have been developed under different Database Management Systems and/or different platforms or hosting systems. The main problem to be addressed, besides the *transparency* of the federation, is how to enable an application program to manipulate data that comes from another application, while preserving data integrity and without modifying the application programs. Similarly, in Software Engineering Environments [5, 3], five dimensions of integration are commonly admitted [18] : *platform, presentation, data, control* and *software process* integration dimensions. This paper focuses on the data integration dimension defined as *the capabilities that are offered by an environment to permit to a software tool to act on data or objects that are produced by another tool.* The need for such features seems obvious : as far as possible, enable an environment to *transparently* evolve, i.e. change a tool into another one or incorporate a new tool in the environment at a minimal cost, permit to different tools to interoperate on the same persistent data, etc.

Nevertheless, tools or application programs[1] interoperability must deal with data structures incompatibility. Indeed, the structure of the objects that are produced by a tool may not be conform to the structures that are manipulated by the "interoperable tools". So, the data integration dimension requires the development of mechanisms that enforce data structures conformity and data interchange between the tools. These mechanisms must not only deal with the data structures, but also with data instances to make these migrate from a representation into another one. Furthermore, they must, as far as possible, preserve data semantics and data integrity.

This paper proposes a formal framework, where data structures and semantics can be expressed, and a computational process for data interchange. The data interchange process consists in, first comparing the structures of data that are admitted by the tools who need to interoperate. Then, a knowledge-based interactive process attempts to fill the gap that may exist between the data structures and, finally, data instances are converted from one structure into the other. Some approaches suggest the development of as much pairs of data converters as the number of tools. Other approaches use a common format (e.g. CDIF, EDI) for representing data and appeal to the

---

*In Proceedings of the International Conference on Systems Integration, ICSI'94, pages 476–485, São Paulo, Brazil, August 1994. IEEE Comp. Society Press.

[1] We'll use the term *tool* in the meaning of "piece of software code that act on persistent data".

development of ad hoc data converters (from and to the common format). Our proposal suggests a dynamic approach where at first, the data representations are described, then they are made compatible using a set of operators that implement a set of rewriting rules called *structural rewriting rules*, and finally data are converted from one representation into another by means of a set of operators that implement *contents rewriting rules*. One of the benefits of this approach is that only one "data converter" is needed : the rewriting system. Its underlying concepts and formalisms are detailed herafter.

Section 2 contains a simplified conceptual framework based on abstract data types specification and implementation where the data integration process is argued to be a subtype relationship construction process. Section 3 introduces the structural concepts of our model and the various levels of compatibility. Section 4 presents structural and contents rewriting rules and operators. An overview of the current state of the implementation of the approach is given in section 5. Finally, section 6 discusses the approach with regard to similar ones and presents on-going work.

2 The Conceptual Framework

When dealing with non-persistent data, as in common programming languages, the abstraction mechanisms are useful for program evolution. Indeed, encapsulation and information hiding helps in modifying the implementation with a null (or a minimal) impact on existing programs. But, in presence of persistent data, it is clear that a change in the implementation impacts the existing data. Data integration may be abstracted in terms of data types specification and implementation : an abstract data type specification may be provided either algebraically [6] or using an underlying representation data type (which itself can be an abstract type). For instance, one can specify a *set-type* providing algebraic axioms that hold among the *set operators* and without fixing any implementation structure. Alternatively, one may also specify the *set-type* as a *table* or a *list*: in this situation, the semantics of the *set operators* is expressed using the operators that are available on the representing type (for instance, inserting an element into a set is specified as an insertion into a *table or a list*).

Let us now view a software tool as an operator, and the input and the output data types of a tool as probably complex types. Let T be a tool that takes I as an input type and that produces O as an output type. Let $REP_{T.I}$ and $REP_{T.O}$ be the implementations of the types I and O. Let $T\prime$ be a tool whose input is O (identical to the output type of T), but with a representation $REP_{T\prime.I}$ that is different from $REP_{T.O}$. Making $T\prime$ use data coming from T requires establishing a correspondance between the two representations and transforming data under the representation $REP_{T.O}$ into data under the representation $REP_{T\prime.O}$ (figure 1)[2].

Considering again the set-type example, and assuming persistent sets, the integration process would consist in migrating sets under, let say, table representation (resp. list) into sets under list representation (resp. table).

But under what conditions the process can succeed and can be considered as consistent ? We argue that *the data integration process is a process that attempts to build a subtype relationship between two types of data.* If we consider that a data type D_1 is a subtype of D_2 if and only if every instance of D_1 is an instance of D_2 [4], one can reasonably think that to enable a tool T_1 to act on the objects that are produced by a tool T_2, it is sufficient to "prove" that the output type of T_1 is a subtype of the input type of T_2. This cannot hold when persistent data exists since "T_1 *is a subtype of* T_2" does not mean that existing instances of T_1 can immediately be used as instances of T_2. As a trivial example, the subtype relationship being reflexive[3] "*integer is a subtype of integer*", but 32-bits integers cannot immediately be used as 16-bits integers. So, reasoning at an abstract level [8] is not sufficient, and one must also consider the concrete implementation of the data types.

In the following, we say that a *data type* D_1 *is integratable to a data type* D_2 if there exists :

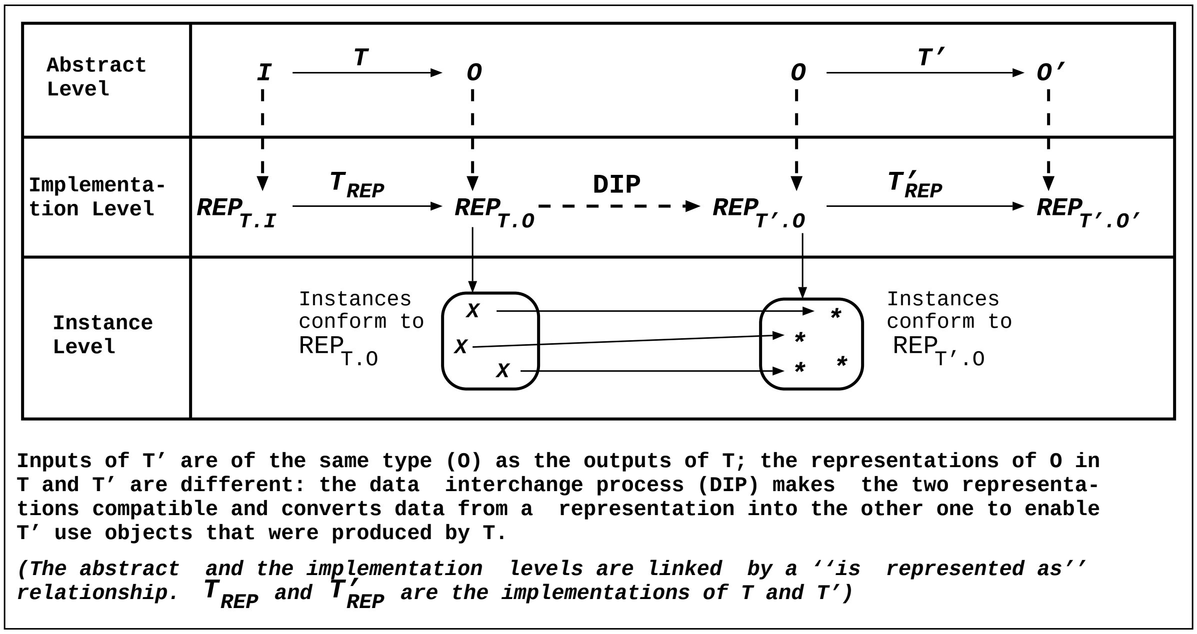
1. a transform S that makes D_1 structurally compatible with D_2 (intuitively S is the process that renders D_1 a "structural" subtype of D_2),
2. and a transform C that makes concrete instances of D_1 conform to the concrete implementation of D_2.

In this framework, the data integration process will consist in the indentification and the implementation of the transforms S and C. This conducted us to define various levels of object structure compatibility together with an ordering among structures (section 3.2.2). These definitions help in deciding wether

---

[2] A similar scenario holds when T is changed into a functionally equivalent tool which uses a representation that is different from the one used by T. This is conceptually similar to the change of the implementation of an operation T that is defined for a data type D.

[3] It is also transitive and anti-symetric: it defines a partial order among the types.



Inputs of T′ are of the same type (O) as the outputs of T; the representations of O in T and T′ are different: the data interchange process (DIP) makes the two representations compatible and converts data from a representation into the other one to enable T′ use objects that were produced by T.

(The abstract and the implementation levels are linked by a ''is represented as'' relationship. T_{REP} and T'_{REP} are the implementations of T and T′)

Figure 1: Data interchange scenario

the structures are integratable or not. Indeed, ordering of data structures is helpful to determine wether a type definition can be rendered a subtype of another one: if two data-structure definitions can be ordered then they are made structurally compatible and the associated instances are converted from the source definition to the target one. Thus, the integration process will proceed as a sequence of transforms that uses a set of operators that apply on the structures as well as on the instances (section 4.3). Essential transforms are renaming of attributes, folding/unfolding of structures, expanding structures, etc. These concepts and the associated mechanisms are detailed hereafter.

3 The Structural Concepts

In this section, we briefly introduce the model we use to deal with persistent data integration (refer to [13] for a more formal definition). The model encompasses classical constructs 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 express the fact that an element in 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" to deduce object definition compatibilities.

3.1 Object Structure Definition

We assume the following sets:

- let $\mathcal{S}$ be an infinite set of symbols. This set is partitioned into disjoint subsets $\mathcal{S}_r$. The subsets contain symbols with identical roles and meanings. 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 with role r.

  For example, the following elements belong to $\mathcal{S}$: *age, position, comment.*

- let $\mathcal{T}$ be the set of atomic types (integer, string, boolean, real). We use usual standard behaviors for these atomic types. Every type has a corresponding domain, namely $\mathcal{D}_{integer}$, $\mathcal{D}_{string}$, $\mathcal{D}_{boolean}$ and $\mathcal{D}_{real}$. Furthermore, we consider that the equality between two types is a *syntactic equality* and we associate the *inclusion semantics* to the subtype relationship.

- 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 $\mathsf{t} \in \mathcal{T}$ (in our approach, null values are non-informative values).

In a *"flat object definition"* every attribute has an atomic type and in a *complex object definition* [16, 2, 1] an attribute can be defined by an object definition.

Complex object are built using the usual *tuple, type-union* and *set constructors.*

The *tuple constructor* is closely related to the record type in PASCAL, and to the aggregation mechanism [16]. It will be denoted as a pair of square brackets ([...]). Further, $[a_1 : r_1, \cdots, a_n : r_n, \cdots, a_m : r_m]$ *is a subtype of* $[a_1 : t_1, \cdots, a_n : t_n]$ if every r_i is a subtype of the corresponding t_i, for $i \in 1 \ldots n$ and $n \leq m$.

The *union of type constructor*, denoted as a pair of angle brackets ($\langle \ldots \rangle$), is like the PASCAL's variant record or the generalization mechanism [16]. It is used so that objects of differents types can be viewed as generically identical. The definition of the *subtype relationship* between two types that are obtained using this construct, is similar to the one provided for tuple types since the unioned types inherit attributes from the constructed super-type.

Finally, the *set constructor*, represented as a pair of curled brackets ($\{\ldots\}$), enables the grouping of objects of a given type in a "container" ([7] contains an extended version of the model, described as the Format Model). The *subtype relationship among sets* is defined as follows: let S_1 be a set of elements of type t_1 and S_2 a set of elements of type t_2. S_1 is a subtype of S_2 if and only if t_1 is a subtype of t_2.

3.2 Role Concept and Object Definition Compatibility

3.2.1 The Role Concept

Intuitively, the *role concept* 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 have commonalities devoted to the same "role": *text* and *description*, *x* and *xpos*, and *y* and *ypos*. The role concept means that the same semantic interpretation holds for a data element in two different definitions (see Example 1).

Furthermore, the definition F_2 is included in the definition F_1 (denoted as $\mathrm{Dom}(F_2) \subseteq \mathrm{Dom}(F_1)$). Consequently, we can find out a transform that makes F_2 compatible with F_1 (and conversely if we authorize some loss of information).

3.2.2 Object Compatibility Levels

We introduce three object compatibility levels: *general, partial and total compatibility* (propositions 1 to 3 below). Proposition 4 defines all the authorized states where two definitions are compatible and will help in identifying the set of operators that will make two object definitions compatible. (In the following definitions, $\ell_r \equiv \jmath_r$ means that ℓ_r and $\jmath_r$ are "semantically equivalent", i.e. the role of ℓ_r in F_1 is the same as the role of $\jmath_r$ in F_2).

Proposition 1 (General compatibility)

Two object definitions F_1 and F_2 are said generally compatible and 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* $\ell_r \equiv \bigcup_x (\jmath_{r_x})$*).*

□

The general compatibility of two object definitions, F_1 and F_2, makes no assumption on the types of their data elements. It enforces the compatibility of the elements that are different in the object definitions. The next proposition supposes that every type of the elements that are common to the object definitions are *syntacticaly equal.*

Proposition 2 (Partial compatibility)

Two object definitions F_1 and F_2 are partially compatible, and denoted $F_1 \simeq F_2$, iff:

- $F_1 \sim F_2$,
- $F_1(\ell_r) = F_2(\jmath_r)$ *(or* $F_1(\ell_r) = \bigcup_x F_2(\jmath_{r_x})$*), i.e.* $\mathtt{t}_{\ell_r} = \mathtt{t}_{\jmath_r}$ *(or* $\mathtt{t}_{\ell_r} = \mathtt{t}_{\jmath_{r_x}}$ *foreach x such that* $\ell_r \equiv \bigcup_x (\jmath_{r_x})$*).*

□

Proposition 3 (Total compatibility)

Two object definitions 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.,* $\mathtt{t}_{\ell_r} = \mathtt{t}_{\jmath_{r'}}$.

□

The total compatibility property is the highest achievable property: there exists a one-to-one correspondance between each element of the definitions.

Finally, the following proposition extends the preceding ones to cover all the compatibility cases (Example 2 shows why it is needed).

Proposition 4

Given two object $F_1 = [\ell_{r_1} : \mathtt{t}_1; \ldots; \ell_{r_n} : \mathtt{t}_n]$ $(n \geq 1)$ *and*

Example 1

Let $F = [\mathit{text}\text{: }\mathsf{string};\mathit{xposition}\text{: }\mathsf{integer};\ \mathit{yposition}\text{: }\mathsf{integer}]$ be an object definition. o_F^1 and o_F^2 are two object instances of F:

$$o_F^1\ \texttt{= [}\mathit{text}\texttt{ = "sample"; }\mathit{xposition}\texttt{= 150; }\mathit{yposition}\texttt{ = 150],}$$

$$o_F^2\ \texttt{= [}\mathit{text}\texttt{ = "longer one"; }\mathit{xposition}\texttt{ = 0; }\mathit{yposition}\texttt{ = 0].}$$

Both the objects are considered as simple objects. An example of a complex object definition is:

$$F' {=} [\mathit{Position}\text{: }[x\text{:}\mathsf{integer};\ y\text{: }\mathsf{integer}];\ \mathit{Text\_block}\text{:}\{\mathit{Line}\text{: }\mathsf{string}\}]$$

An instance of F' (considered as a composite object), denoted $o_{F'}^1$ is defined as:

$$o_{F'}^1\texttt{=[}\mathit{Position}\texttt{=[}x\texttt{=150;}y\texttt{=150];}\mathit{Text\_block}\texttt{=}\{\mathit{Line}\texttt{="A much more";}\mathit{Line}\texttt{="complex example"}\}\texttt{]}$$

One should 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.

Example 2

1) $F_1 = [\mathit{date}\text{: }\mathsf{string}]$ and $F_2 = [\mathit{date}\text{: }\mathsf{string}]$,
2) $F_1 = [\mathit{length}\text{: }\mathsf{string}]$ and $F_2 = [\mathit{length}\text{: }\mathsf{integer}]$,
3) $F_1 = [\mathit{length}\text{: }\mathsf{string};\ \mathit{checked}\text{: }\mathsf{boolean}]$ and $F_2 = [\mathit{length}\text{: }\mathsf{string}]$, (cases 3 and 4)
4) $F_1 = [\mathit{day}\text{: }\mathsf{string};\ \mathit{month}\text{: }\mathsf{string};\ \mathit{year}\text{: }\mathsf{string}]$ and $F_2 = [\mathit{date}\text{: }\mathsf{string}]$, (cases 5 and 6)

$F_2 = [\jmath_{r_1}\text{: }\mathsf{t}'_1;\ \ldots;\jmath_{r_m}\text{: }\mathsf{t}'_m]$ ($m \geq 1$), if $\ell_{r_i} \in Dom(F_1)$ and $\jmath_{r_j} \in Dom(F_2)$, and $\ell_{r_i} \equiv \jmath_{r_j}$, then 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) *$F_1(\ell_{r_i}) = \texttt{undefined}_{\mathsf{Type}}$ (the element does not exist),*
4) *$F_2(\jmath_{r_i}) = \texttt{undefined}_{\mathsf{Type}}$ (the element does not exist),*
5) *$F_1(\ell_{r_i}) = \bigcup_x F_2(\jmath_{r_x})$ (ℓ_{r_i} is a composition of several elements in F_2),*
6) *$F_2(\jmath_{r_i}) = \bigcup_x F_1(\ell_{r_x})$ ($\jmath_{r_i}$ is a composition of several elements in F_1).*

□

3.3 Ordering of Object Definitions

Intuitively, the idea that elements of a definition are ordered in terms of their "goodness of descriptions" is already quoted in [11]. In practice, object definitions ordering will help in deciding wether they are compatible or not.

Let $\mathcal{E}$ be a set, and $\preceq$ an order on $\mathcal{E}$ (we call $(\mathcal{E},\preceq)$ a preordered set). Two elements $x,y \in \mathcal{E}$ can be ordered if there exists $z \in \mathcal{E}$ such that $x \preceq z$ and $y \preceq z$. The structure $(\mathcal{E},\preceq)$ represents the notion of inclusion of a definition into another one: *considering two definitions F_1 and F_2, $F_1 \preceq F_2$ iff F_1 is included in F_2*. (Notice the *"symetry"* between the definitions of the subtype and the ordering relationships).

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

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

□

For example, the structure F_2=[*text*: string; *xpos*: $\mathsf{integer}$; *ypos*: $\mathsf{integer}$ "contains' F_1=[*text*: string] (and F_2 is a subtype of F_1). This intuitive notion is now formalized by a partial order.

Proposition 6

Given F_1=[ℓ_{r_1}:t_1;...;ℓ_{r_n}:t_n] *and* F_2=[ℓ_{r_1}:t_1;...;ℓ_{r_n}:t_n;...;ℓ_{r_m}:t_m]. *F_1 and F_2 satisfy:*

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

□

We deduce, from propositions 4 and 6: considering F_1 = [ℓ_{r_1}: t_1; ...; ℓ_{r_n}: t_n] and F_2 = [$\jmath_{r_1}$: t'_1; ...;$\jmath_{r_n}$:t'_n], $F_1 \preceq F_2$ iff t_n = $\mathtt{undefined}_{\mathsf{Type}}$ or $\ell_{r_n} \equiv \jmath_{r_n}$ or $\ell_{r_n} \equiv \bigcup_x (\jmath_{r_x})$

For example, F_1 = [*text*: string ; *xpos*: $\mathtt{null}_{\mathsf{integer}}$; *ypos*: $\mathtt{null}_{\mathsf{integer}}$] and F_2 = [*text*: string; *xpos*: $\mathsf{integer}$; *ypos*: $\mathsf{integer}$] can be ordered as $F_1 \preceq F_2$. Moreover, F'_1 = [*text*: string] $\preceq F_2$.

4 Rewriting Rules and Operators

To make a tool use objects that have been produced by another tool, it is sometimes necessary to transform the objects of the latter, i.e. to make them fully compatible with the object structures admitted by the former. The data model is provided with a set of atomic operators that make totally compatible two generally or partially compatible object definitions. Thus, the transformation cycle will consist in *(i)* determining the level of compatibility of the two object definitions, *(ii)* making them totally compatible, *(iii)* transforming the object instances.

This section introduces rewriting rules that allow for restructuring both of the data definitions and the data instances. Each rewriting expression is "atomic": that means that a complete transformation process can be represented by a compound rewriting expression. The rules are divided *structural rewriting rules* (SRR) that apply on object definition ([2] quoted some of them) and *content rewriting rules* (CRR) that apply on object instances. From these rules, a set of operators is derived and will be effectively used to render compatible the definitions and the instances.

4.1 Structural Rewriting Rules

In the rule expressions, the symbol → means that the left handside of a rule *"may be rewritten into"* the right handside.

Rule 1 [ℓ_1: t_1; ...; ℓ_{i-1}: t_{i-1}; ℓ_i:[ℓ_k: t_k; ...; ℓ_{k+m}: t_{k+m}]; ℓ_{i+1}: t_{i+1}; ...; ℓ_n: t_n]
→ [ℓ_1: t_1; ...; ℓ_{i-1}: t_{i-1}; ℓ_k: t_k; ...; ℓ_{k+m}: t_{k+m}; ℓ_{i+1}: t_{i+1}; ...; ℓ_n: t_n]
Rule 1 flattens (or unstructures) a nested tuple definition.

Rule 2 ⟨ℓ_1: t_1; ...; ℓ_{i-1}: t_{i-1}; ℓ_i:⟨ℓ_k: t_k; ...; ℓ_{k+m}: t_{k+m}⟩; ℓ_{i+1}: t_{i+1}; ...; ℓ_n: t_n⟩
→ ⟨ℓ_1: t_1; ...; ℓ_{i-1}: t_{i-1}; ℓ_k: t_k; ...;ℓ_{k+m}: t_{k+m}; ℓ_{i+1}: t_{i+1}; ...; ℓ_n: t_n⟩
Intuitively, this rule unstructures an embedded type-union definition.

Rule 3

[ℓ_1: t_1; ...; ℓ_i:⟨ℓ_k: t_k; ...; ℓ_{k+m}: t_{k+m}⟩; ...; ℓ_n: t_n]
→ ⟨[ℓ_1: t_1; ...; ℓ_k: t_k; ...; ℓ_n: t_n]; ...; [ℓ_1: t_1; ...; ℓ_{k+n}: t_{k+n}; ...; ℓ_n: t_n]⟩
The third rule rises the type-union constructor through a definition.

Rule 4

{⟨ℓ_1: t_1; ...; ℓ_n: t_n⟩} → ⟨ {ℓ_1: t_1}; ...; {ℓ_n: t_n} ⟩
This rule rises the type-union constructor through the set constructor.

Rule 5 (Renaming of attributes)

[ℓ_1: t_1; ...; ℓ_k: t_k; ...; ℓ_n: t_n] → [ℓ_1: t_1; ...; $\jmath_k$: t_k; ...; ℓ_n: t_n]

Rule Composition As underlined above, these rules are atomic. The following example shows how one can combine them to make two object definitions compatible. Given the definitions: F = [*BirthDate*: date] and F' = [*Birth*:[*day*: $\mathsf{integer}$; *month*: $\mathsf{integer}$; *year*: $\mathsf{integer}$]], the following sequence of rules applies :

ϕ = rewrite(*date*: date → *date*: $\mathsf{integer}$)

ψ = rewrite(*date*: $\mathsf{integer}$ → *date*:[*day*: $\mathsf{integer}$; *month*: $\mathsf{integer}$; *year*: $\mathsf{integer}$])

ξ = rewrite(*BirthDate* → *Birth*)

and results into:

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

4.2 Content Rewriting Rules

The Content Rewrite Rules (CRR) are introduced to take into account some differences based on types, degrees or scale expression of attributes.

Rule 6 $[\ell_1\colon \mathsf{t}_1; \ldots; \ell_k\colon \mathsf{t}_k; \ldots; \ell_n\colon \mathsf{t}_n] \rightarrow [\ell_1\colon \mathsf{t}_1; \ldots; \ell_k\colon \mathsf{t}'_k; \ldots; \ell_n\colon \mathsf{t}_n]$ where t'_k is different from t_k. Intuitively, this rule states that one can cast the type of an attribute, regarding its instances.

Rule 7 $[\ell_1\colon \mathsf{t}_1; \ldots; \ell_k\colon \mathsf{t}_k; \ldots; \ell_n\colon \mathsf{t}_n] \rightarrow [\ell_1\colon \mathsf{t}_1; \ldots; \ell_k\colon \mathsf{t}'_k; \ldots; \ell_n\colon \mathsf{t}_n; \ell_{n+1}\colon \mathsf{t}_{n+1}]$, i.e. the resulting definition is extended with ℓ_{n+1}.

Rule 8 $[\ell_1\colon \mathsf{t}_1; \ldots; \ell_k\colon \mathsf{t}_k; \ldots; \ell_p\colon \mathsf{t}_p; \ldots; \ell_n\colon \mathsf{t}_n]$
$\rightarrow [\ell_1\colon \mathsf{t}_1; \ldots; \ell_k\colon \mathsf{t}_k; \ldots; \phi(\ell_p)\colon \mathsf{t}_p; \ldots; \ell_n\colon \mathsf{t}_n]$
if ℓ_k and ℓ_p are linked by a calculus relation, named ϕ.

Rule 9 Similarly to rule 8,
$[\ell_1\colon \mathsf{t}_1; \ldots; \ell_k\colon \mathsf{t}_k; \ldots; \ell_n\colon \mathsf{t}_n] \rightarrow [\ell_1\colon \mathsf{t}_1; \ldots; \phi(\ell_k)\colon \mathsf{t}_k; \ldots; \ell_n\colon \mathsf{t}_n]$

Rule 10 $[\ell_1\colon \mathsf{t}_1; \ldots; \ell_{k-2}\colon \mathsf{t}_{k-2}; \ell_{k-1}\colon\mathsf{t}_{k-1}; \ell_k\colon \mathsf{t}_k; \ell_{k+1}\colon \mathsf{t}_{k+1}; \ldots; \ell_n\colon \mathsf{t}_n]$
$\rightarrow [\ell_1\colon \mathsf{t}_1; \ldots; \ell_{k-2}\colon \mathsf{t}_{k-2}; \ell'_k\colon \mathsf{t}'_k; \ell_{k+1}\colon \mathsf{t}_{k+1}; \ldots; \ell_n\colon \mathsf{t}_n]$, where ℓ'_k is a new attribute with an identical role to (ℓ_{k-1},ℓ_k) and t'_k is the type of the new attribute.

4.3 From Rules to Operators

The on-going implementation of the proposal is a procedural approach, i.e. from the above-mentioned set of rules we have derived a set of operators whose composition enables object structures as well as object contents transformation. These operators are presented below.

FOLD and UNFOLD implement the first rule. The rule 1 stipulates that a nested structure can be "flattened" and, conversely, a flat structure can be rewritten into a nested structure.

Example: F = [*Name*: [*LastName*: string; *FirstName*: string]] is UNFOLDed into F' as follows :
F' = UNFOLD(F, *Name*) = [*LastName*: string; *FirstName*: string]

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

Example: Given F = [*Locality*: ⟨*BirthCity*: string; *BirthCountry*: string⟩], UNCOMPOSE(F, *Locality*) = [*Locality*: string]

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

Example: Given F = [*Name*: string; *BirthDate*: date; *Locality*: ⟨*BirthCity*: string; *BirthCountry*: string⟩], DISTRIBUTE(F, *Locality*) results into
[⟨[*Name*: string; *BirthDate*: date; *BirthCity*: string]; [*Name*: string; *BirthDate*: date; *BirthCountry*: string]⟩]

SET and UNSET: 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 F be [*SisterBrother*: {⟨*SisterName*: string; *BrotherName*: string⟩}].
UNSET(F, *SisterBrother*) = [*SisterBrother*: ⟨{*SisterName*: string}; {*BrotherName*: string}⟩]

RENAME: 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 that have the same meaning.
Example: RENAME([*Birth*: date], *Birth*, *BirthDate*) = [*BirthDate*: date]

CAST: The CAST operator modifies the type of an attribute with a role r to make it compatible with another attribute that has 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: CAST([*task*: string; *length*: string; *code*: integer], *length*, integer) = [*task*: string; *length*: integer; *code*: 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: COMPLETE allows a one-to-one correspondence between two definitions. It is a way to augment a definition with additional attributes to obtain two definitions having the same degree, i.e. the same number of attributes. Therefore, with such an operator, it is easy to provide a complete compatibility between two definitions with different degrees. At the object level, the additional attributes are given a null value. In a symetric way, SKIP allows to delete some attributes from a definition. These two operators realize the rule 7.

Example: Considering F = [*task*: string; *length*: integer] and F' = [*task*: string; *length*: integer; *Programmer's_Name*: string], then F' = COMPLETE(F, *Programmer's_Name*, string) and F = SKIP(F', *Programmer's_Name*)

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

FORMAT: 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: Consider the attribute *duration* in F = [*task*: string; *start_date*: date; *duration*: integer]. If we want to define the duration as a pair of attributes *start_date* and *end_date*, then FORMAT(F, *duration*, *end_date*, date, *start_date* + *duration*) delivers [*task*: string; *start_date*: date; *end_date*: date]

SCALE: 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: Consider a duration expressed in hours, and another one expressed in minutes:

F = [*duration_in_hours*: integer] and

F' = [*duration_in_minutes*: integer]

SCALE(F, *duration_in_hours*, *duration_in_minutes*, integer, x * 60) transforms F into F'.

GROUP and UNGROUP: The UNGROUP and GROUP operators are related to the last rule which allows for splitting or composing attributes. The GROUP operator solves the usual case where several attributes are grouped into a unique one in the target definition. Conversely, the UNGROUP solves the opposite problem.

Example: Consider a date attribute represented as a string F = [*start_date*: string]. The following expression transforms it into a 3-uple string:

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 get an object o' = [*start_date_day* = `"1"`; *start_date_month* = `"8"`; *start_date_year* = `"1992"`].

5 The Current Implementation

In the current implementation, similar tools in an environment are grouped into classes (e.g. Compile Class contains compilers, Drawing Class contains drawing toools like Tgif and xfig, etc). A *browser* enables selecting a tool within a class together with the object(s) to be processed by the tool (figure 2). The *object interchange process* is automatically activated if the selected object has not been produced by the selected tool. (e.g. one wants to edit a Tgif figure using xfig).

The *data converter sub-system* attempts then, to determine the compatibility level that holds between the data structures that are manipulated by the selected tool and the ones that are manipulated by the tool that created the object. The *data converter* is implemented as a knowledge-based system that attempts to unify both of the data definitions. The structure unification process being undecidable, the *converter sub-subsytem* may inter-act with the user providing her/him with the elements of information for which the system couldn't decide on its own. The information includes the concerned attributes, their source and target types and their initial values (figure 3).

-bb-error = =

Figure 2: Tool server front-end

-bb-error = =

Figure 3: Attribute definitions conflict

The user's decisions are used to solve the data definition conflicts, i.e. to determine object definition compatibility (as defined in section 3.2.2) and, at the same time, they are inserted into the system's knowledge base. If no conflict remains, the systems creates a copy of the selected object that conforms to the data structure that is admitted by the selected tool (which can, by now, act on it). The object copy is obtained thanks to a sequence of operators (section 4.3) that is generated by the first phase of the *data converter subsystem*.

At the end of the selected tool activation, the resulting object may be transformed back to its original structure. (The current implementation runs on Sun worksations. We envisage to port it on a PCTE-based platform.)

6 Concluding Remarks

This paper introduced a data model to conceptually characterize objects and their definitions, where each object can be transformed by some operators. The method for transforming data can be used both in *a priori* and *a posteriori* integration. First, the model should be compatible with each tool without changing its internal data format. Then, for each data definition, we construct a mapping that translates a definition into another one. Finally, a set of operators allows us to build a translation procedure.

Let's make some comments with regard to 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 or that the appropriate data converter exists. Such a process is advantageous in an *a priori* situation, but has drawbacks in an *a posteriori* case. With our model, each tool keeps its own data format. 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 from 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.

A distinctive characteristic of the model is that each tool continues to manage its data and there exists an "on the fly" translation when another tool wants to access these data. This characteristic provides simplicity and efficiency. The identification and the representation of data elements, and the semantic interpretation are already done, and so, data translation and integration process are quite easy. Moreover, some level of automation is allowed. Our model presents solutions on current problems stated in the introduction to this paper: 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 scales* (FORMAT operator). We provide answers about syntactic constraints, but 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 change and modification processes faster than the adoption of a common data format as suggested by the work around CDIF, for instance.

Finally, the approach suggests a data-oriented view to solve conflicting situations such as homonymy and synonymy, rather than the classical tool-oriented view. Indeed, the "*One data file, various tools to manage these data*" scheme is preferred to the classical "*one tool, one data format*" scheme. To implement this kind of scheme, we introduced a group (or class) concept ([12]), where various tools with the same functionalities are grouped into classes. For each class, there exists a common data semantic which allows for a high automation of the translation process. The new scheme could then, be described as: "*given data, give me a list of authorized tools I could use to manage* *these data with a maximum degree of automation in the interoperability process*".

Current investigations concern the study of the expressive power of the proposed model together with the definition of formal properties of the set of operators. Moreover, the formulation and the preservation of data integrity constraints during the data interchange process are being tackled.

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.

[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] J.A. Goguen, J.W. Thatcher, and E.G. Wagner. An Initial Algebra Approach to the Specification, Correctness and Implementation of Abstract Data Types. *in Current Trends in programming methodology*, 4, 1978.

[7] R. Hull and C.K. Yap. The Format Model: A Theory of Database Organization. *Journal of the ACM*, 31(3):518–537, July 1984.

[8] B. Liskov and J.M. Wing. Specifications and Their Use in Defining Subtypes. In *Proceedings of OOPSLA'93*, pages 16–28, 1993.

[9] A. Motro. Superviews: Virtual Integration of Multiple Databases. *IEEE Transactions on Software Engineering – Vol. SE-13, No. 7*, pages 785–798, July 1987.

[10] S. Navathe, E. Ramez, and J. Larson. Integrating User Views in Database Design. *IEEE Computer*, pages 50–62, January 1986.

[11] A. Ohori. Semantics of Types for Database Objects. *Theorical Computer Science*, 76:53–91, 1990.

[12] 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.

[13] O. Perrin and N. Boudjlida. Towards a Model for Persistent Data Integration. In *CAiSE '93 – The Fifth Conference on Advanced Information Systems Engineering – Lecture Notes in Computer Science 685*, pages 93–117, Paris, France, June 1993.

[14] W. Rossak and P.A. Ng. Some thoughts on Systems Integration: A Conceptual Framework. *Journal of Systems Integration*, 1(1):97–114, July 1991.

[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.

[18] A. Wasserman. Tool Integration in Software Engineering Environments. In *Software Engineering Environments: International Workshop on Environments*, pages 137–149, Berlin, 1990.