Your Mediators Need Data Conversion![*]

Sophie Cluet
INRIA Rocquencourt
Le Chesnay, France
Sophie.Cluet@inria.fr

Claude Delobel
LRI, University of Paris XI
Orsay, France
Claude.Delobel@lri.fr

Jérôme Siméon
INRIA Rocquencourt
Le Chesnay, France
Jerome.Simeon@inria.fr

Katarzyna Smaga
INRIA Rocquencourt
Le Chesnay, France
ks146747@rainbow.mimuw.edu.pl

Abstract

Due to the development of the World Wide Web, the integration of heterogeneous data sources has become a major concern of the database community. Appropriate architectures and query languages have been proposed. Yet, the problem of data conversion which is essential for the development of mediators/wrappers architectures has remained largely unexplored.

In this paper, we present the YAT system for data conversion. This system provides tools for the specification and the implementation of data conversions among heterogeneous data sources. It relies on a middleware model, a declarative language, a customization mechanism and a graphical interface.

The model is based on named trees with ordered and labeled nodes. Like semistructured data models, it is simple enough to facilitate the representation of any data. Its main originality is that it allows to reason at various levels of representation. The YAT conversion language (called YATL) is declarative, rule-based and features enhanced pattern matching facilities and powerful restructuring primitives. It allows to preserve or reconstruct the order of collections. The customization mechanism relies on program instantiations: an existing program may be instantiated into a more specific one, and then easily modified. We also present the architecture, implementation and practical use of the YAT prototype, currently under evaluation within the OPAL[*] project.

1 Introduction

The number of intra/internet applications integrating heterogeneous data is rapidly increasing. Central to these applications is the conversion of data from one format to another. Yet, data conversion is usually done in an *ad hoc* manner, by developing non reusable software. In this paper, we propose a declarative approach to specify data conversion and integration that could form the backbone of a system based on a mediator/wrapper architecture. A prototype validates the approach.

To illustrate the problem, consider the following application. A car dealer company wants to build an intranet application. Among other things, the company stores information about its dealers in a relational system and descriptions of the cars it sells in SGML documents. The goal of the application is to integrate all information in an object database and to provide an HTML interface so that employees can view it on the Web. A global view of the application when running is illustrated on Figure 1. We are concerned with the development of such applications.

Using our system, the application programmer first imports two generic conversion programs providing an ODMG view of relational and SGML data. Using a graphical interface, these programs are combined and customized for this specific application, providing a single unified conversion (noted **(1)** on Figure 1) from both relational and SGML, to ODMG. The application programmer does not have to see the details of the conversions and can concentrate on their integration, through some high level graphical representation. Then, an ODMG to HTML conversion program **(2)** is imported and customized in a similar way. If the application requires a different Web display for one particular car or class of cars, the program can be further customized to provide different behaviors. In this particular case, we assumed that the ODMG data were materialized. It is also possible for it to be virtual. In which case, the conversions to and from the object world are composed to yield a one-step conversion program. Our system uses type checking to verify the coherence of the conversions, and in particular, the coherence of their composition.

Our system, called YAT[1], is designed to be the basis of a mediator/wrapper system. The current version of YAT only allows to materialize the target data representation. Of course, a complementary goal is to be able to query it without fully materializing it. This is the topic of most work on mediator/wrapper architectures (e.g., [1, 2, 3, 4, 5, 6, 7, 8, 9]). We decided to focus our work on facilitating the specification of conversions, since it requires richer modeling and restructuring capabilities than what is considered usually. Efficient querying of the target data representation (without materializing it) as well as the management of updates of both source and target data will be considered in future works.

In short, YAT relies on a data model allowing a rich and

[*]This work is partially supported by the OPAL project (Esprit IV project number 20377) and the AFIRST association (Association Franco-Israélienne pour la Recherche Scientifique et Technique).

[1]For Yet Another Tree-based system.

uniform representation of data and a rule based language.

The YAT Model: In a manner similar to, e.g., [10, 3, 8, 11], we use a simple representation of graphs to capture data coming from heterogeneous sources. In contrast to most previous work (and in the style of [11]), the outgoing edges from a vertex are ordered. This is essential to capture ordered collections that are frequent in data exchange formats, or in documents. The main originality of the model is its ability to capture various levels of representation. A YAT model can be instantiated into another more specific (eventually "ground") model. This novel feature is essential for allowing customization of conversion or integration programs. Also, it is a key component of the type verification that can optionally be used to validate conversion programs. Finally, it is central to allow conversion programs to be combined (in parallel) or composed (sequentially) in a coherent manner.

The YAT Language: YATL is rule-based. It is based primarily on enhanced pattern matching facilities, powerful restructuration primitives and explicit Skolem functions, as in [12, 13, 8], to handle the creation and use of new identifiers. Its main originality is in the management of non-set collections and the existence of a natural graphical representation. For instance, it can preserve the order or reorder a collection and perform grouping in a collection with duplicates (bag). Both these features are needed in most standard formats (e.g., ODMG, HTML, SGML).

The YAT prototype: The prototype was developed in the VERSO group at INRIA. The model and the language described in the present paper are both supported by our system. The system is implemented in Objective CAML [14]. The graphical interface, that is used to specify conversion and integration, is implemented in JAVA. YAT is now being used within the OPAL European Project that is concerned with Intranet tools for the manufacturing industry. An on-line demonstration of the YAT system is also available[2].

There exist several conversion or integration languages (e.g., [15, 16, 17, 11, 18, 13]), as well as many query languages for semistructured data, offering similar capabilities [19, 6, 7]. However, query languages are not well suited to describe data conversions, that require richer restructuring capabilities. Some proposals focusing on database translations like [15, 16, 17, 20] give an answer to this problem but follows a schema-driven approach which misses flexibility. Finally, [18, 13] is the closest related work, but the proposed language cannot address recursive translations and ordered collections. In any case, no approach so far offers the ability to deal in an appropriate manner with collections such as lists or bags, or the original mechanism used in YAT which allows to provide typing with the flexibility required by a semistructured data model.

The paper is organized as follows. In Section 2, we introduce the model and instantiation mechanism. The language is presented through some examples in Section 3. Section 4 shows how one can use instantiation to customize, combine and compose programs. Section 5 presents the YAT system architecture and prototype.

[2]Please consult the YAT system homepage for a description of the system, up-to-date implementation status, and on-line demonstration, at `http://www-rocq.inria.fr/verso/Jerome.Simeon/YAT/`.

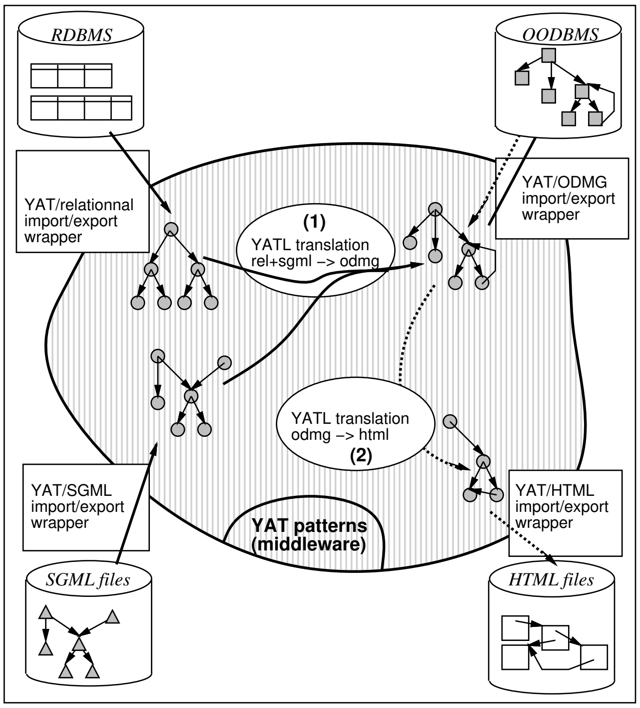

Figure 1: Translation Scenario

2 The YAT data model

Since we are interested in modeling data coming from arbitrary heterogeneous sources, we need a flexible representation model. The current trend in the literature on data integration is to propose graph or tree models (e.g., [19, 10, 3, 11]). The main reason for the popularity of this kind of model is its simplicity and the fact that one can easily map anything into a tree/graph. Thus, and unsurprisingly, we propose yet another tree data model. We represent data in a way that does not differ much from that of [11].

Yet, there is a difference. Whereas, previous work proposed models whose instances were data trees/graphs, we propose a model that can be instantiated (or refined) into another. This property is the main originality of the YAT model, and we will see that it is indeed an essential feature.

Let us illustrate this using the example scenario introduced in Section 1. We concentrate on the representation of ODMG data. A formal definition of the model and the instantiation mechanism can be found in [21]. Figure 2 shows four models — two of which incompletely drawn — that illustrate different levels of data representation. At the top left side of the figure, the Yat model captures any data. On its right, a first instance model of Yat captures ODMG compliant data. The Car Schema model on the bottom left side is an instance of both previous models and represents data that complies to a specific ODMG schema. Finally, the Golf model represents the actual database (we only represented the single object ***c1***) and is an instance of all its predecessors.

A Model consists of a set of patterns and their associated variables domains. We start by defining variables domains.

- We use uppercase letters to denote variables. There exist two kinds of variables: (i) data variables (e.g., L

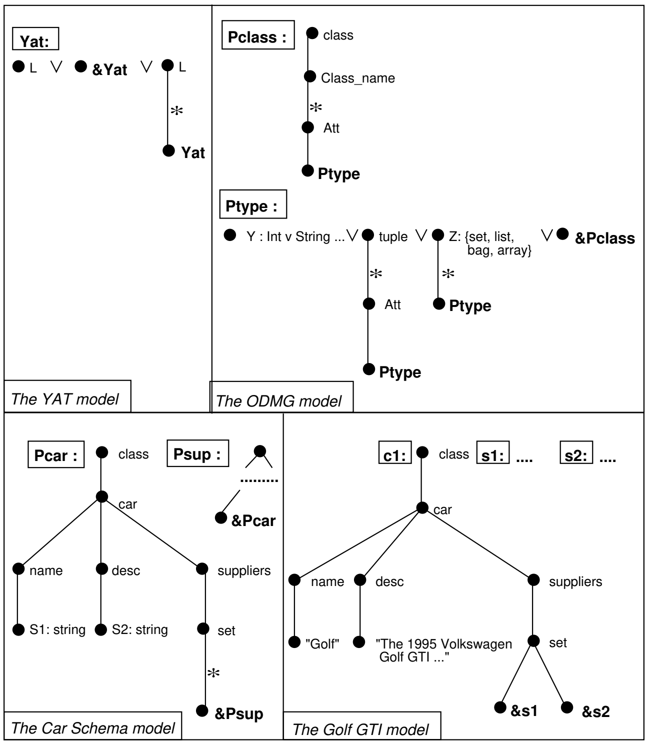

Figure 2: From General to Specific

in YAT, *Class_name* in ODMG, $S1$ in Car Schema) and (ii) pattern variables (e.g., ***Yat***, ***Pclass***, ***Pcar***). We use bold fonts for patterns.

By default, the domain of a data variable is the set of all data constants and variable names. For instance, the domain of the Yat variable L includes constants such as *class*, *car* or "Golf" and variables such as *Class_name* and *Att*. Note that constants can be either symbols (e.g., *class*, *name*) or atomic data (e.g., "Golf", 1995). The default domain can be restricted. For instance, consider the ODMG model and its ***Ptype*** pattern. The domain of Y is defined as the union of *string*, *int*, etc. Thus, a variable whose domain is a string or an actual string like "Golf" are part of its domain.

The domain of a pattern variable is the set of all its instance patterns. In this example, the domain of ***Yat*** includes **c1** but also ***Pcar*** and ***Pclass***.

- A pattern represents a structure. It is identified by a name (that can be constant or variable) and is defined by a *union* of pattern trees. A pattern tree is an ordered tree whose nodes are labeled with data variables or constants. Additionally, leaves may also be labeled with pattern names (e.g., ***Yat***, ***Ptype***) or references to pattern names (e.g., &***Yat***, &***Pclass***, &s**1**). The former denotes dereferencing (i.e., a pattern name label will be instantiated by a pattern tree), and is used to represent deeply recursive tree structures. The latter resembles references to objects in a standard object model and allows sharing and the management of cyclic structures (as in the Car Schema model).

  The edges of a pattern tree are labeled with indicators of occurrences that are used in the instantiation process. In the example, there are two such indicators: "$\star$" which indicates zero or more occurrences and the empty symbol which indicates exactly one occurrence.

  A pattern whose value is not a union (i.e. is defined by a single pattern tree), that does not contain any variable and whose edges are all empty labeled is called a *ground* pattern. E.g., **c1** is a ground pattern but ***Yat*** is not. A ground pattern can only be instantiated by itself. Ground patterns are used to represent real data, like in usual semistructured data models.

Model Instantiation relies on pattern instantiation which itself relies on variable domain inclusion. More precisely, (i) each pattern of the instance model must be an instance of some pattern of the source model and (ii) a variable can be instantiated either by a constant belonging to the variable's domain or by a variable whose domain is a subset.

Let us illustrate pattern instantiation by first considering why ***Pclass*** is an instance of ***Yat***. We have explained variable instantiation. Edges instantiation relies on indicators of occurrence. An empty labeled edge can only be replaced by a similar edge. A "$\star$" labeled edge can be replaced by any ordered sequence of edges, with or without label. Thus, assuming that ***Ptype*** is an instance of ***Yat*** (which is indeed the case) and considering the rightmost argument of the Yat union, we can see that the subtree whose root is *Att* is an instance of ***Yat***. The same can be said of the subtree whose root is *Class_name* (we use pattern dereferencing here). Going up one more step, we see that ***Pclass*** is indeed an instance of ***Yat***.

Now, consider the ***Pclass*** and **c1** patterns. The variable *Class_name* has been instantiated by the *car* symbol. The following "$\star$" edge is instantiated by a sequence of empty labeled edges. Consider the leftmost one. It goes to a node whose label *name* is an instance of the variable *Att*. The next edge is correctly empty labeled and goes to the node labeled "Golf". "Golf" is an instance of ***Ptype*** since it is a string and thus belongs to the domain of the Y variable.

To conclude this presentation of the model, let us point out the fact that there is a clear analogy between the instantiation process and subtyping. Indeed, through instantiation (resp. subtyping), we obtain patterns (resp. types) whose domains are more and more restricted. Therefore, the YAT model combines some features of typed data models, with the flexibility required to manage heterogeneous data. This property of the YAT model can be used for various purposes: to help the programmer write conversion programs or type them (see Section 3), to customize, combine or compose programs (see Section 4).

Finally, we introduce the YAT syntax for patterns with an example. Patterns ***Pcar*** and ***Psup*** from Figure 2 are given below. A labeled node is denoted by its label followed by the list of its sons between brackets($\langle$ ... $\rangle$), edges are denoted by arrows with their corresponding label. If a node has a single son, brackets can be omitted.

Pattern for Car Objects

$$\begin{aligned}\boldsymbol{Pcar} \;:\; class \rightarrow car\langle\; & \rightarrow name \rightarrow S1 : String, \\ & \rightarrow desc \rightarrow S2 : String, \\ & \rightarrow suppliers \rightarrow set \xrightarrow{\star} \&\boldsymbol{Psup} \;\rangle\end{aligned}$$

Pattern for Supplier Objects

$$\boldsymbol{Psup} : class \rightarrow supplier\langle \rightarrow name \rightarrow S1 : String, \rightarrow city \rightarrow S2 : String, \rightarrow zip \rightarrow S3 : String\rangle$$

3 The YAT language

In this section, we present YATL, our language for the specification of data conversion. This language supports a graphical interface. Thus, programmers do not have to write YATL programs, they are generated by the system given a graphical specification.

We introduce YATL basic features, demonstrate the language ability to handle heterogeneity problems between data sources and present advanced YATL features to deal with collections. Then, we address the problem of detecting cycles in a program. Finally, we show how YATL programs can be type checked. A formal definition can be found in [21].

3.1 Basic features

YATL is a rule-based language, each rule describing a part of the data conversion. A rule is composed of a body and a head. The body contains patterns and boolean predicates whose roles are to *filter* the input data, and external functions that *compute* additional data. The head of a rule contains a single pattern which describes how the data which has been filtered in the body must be re-structured.

Let us illustrate this with a first program generating car and supplier objects from a set of SGML brochures. The brochures comply to the following Document Type Definition (i.e. its grammar or DTD) and give information about the cars, such as year of construction, technical description and suppliers. The number element is distinct for each brochure, and the title element stores the name of the car.

Brochures DTD

```
<! DOCTYPE  brochure [
<! ELEMENT  brochure  - -  (number, title,
                            model, desc, spplrs) >
<! ELEMENT  number    - -  (#PCADATA) >
<! ELEMENT  title     - -  (#PCADATA) >
<! ELEMENT  model     - -  (#PCADATA) >
<! ELEMENT  desc      - -  (#PCADATA) >
<! ELEMENT  spplrs    - -  (sup)* >
<! ELEMENT  supplier  - -  (name,address) >
<! ELEMENT  name      - -  (#PCADATA) >
<! ELEMENT  address   - -  (#PCADATA) > ] >
```

The first rule of the program creates the supplier objects. Figure 3 illustrates sample input and output for Rule 1.

Rule 1

$$\begin{array}{lll} \boldsymbol{Psup}(SN) : & & \boldsymbol{Pbr} : \\ class \rightarrow supplier & & brochure\langle \rightarrow number \rightarrow Num, \\ \quad \langle \rightarrow name \rightarrow SN, & \Longleftarrow & \quad \rightarrow title \rightarrow T, \\ \quad \rightarrow city \rightarrow C, & & \quad \rightarrow model \rightarrow Year, \\ \quad \rightarrow zip \rightarrow Z\rangle & & \quad \rightarrow desc \rightarrow D, \\ & & \quad \rightarrow suppls \xrightarrow{\star} supplier \\ & & \quad\quad \langle \rightarrow name \rightarrow SN, \\ & & \quad\quad \rightarrow address \rightarrow Add\rangle\rangle, \\ & & Year > 1975, \\ & & C \texttt{ is } \mathrm{city}(Add), \\ & & Z \texttt{ is } \mathrm{zip}(Add) \end{array}$$

Note that we have not given a domain to the various data variables used in the rule. This means that they are associated to the default domain and can be instantiated by any constant. The body of the rule is on the right-hand side of the $\Longleftarrow$ symbol. It consists of (i) a pattern representing the SGML brochures, (ii) one simple predicate which eliminates oldest cars and (iii) two external functions which extract city and zip code from an input address. The head of the rule consists of a single pattern whose name is parameterized by a variable (SN) representing the name of a supplier. Parameterized pattern names correspond to explicit skolem functions as found in [12, 13, 8]. Intuitively, the above skolem states that one object will be created for each supplier name encountered in the input SGML brochures. Thus, if the same supplier name appears twice in the input, only one object will be created.

This somehow implies that the name of a supplier represents its key. If it is not the case, the result of transformation described by Rule 1 is non deterministic (e.g. if two suppliers share the same name SN and not the same address, then YATL semantics do not specify the value chosen for $\boldsymbol{Psup}(SN)$). Syntactical restrictions can guarantee the determinism of a program. However, this has a cost that the user may not want to pay (e.g., the ability to remove duplicate values). In YAT, we accept potentially non-deterministic programs and alert the user at run time when the same pattern name is associated to two distinct values.

Let us now use the above example to explain how rules are applied on input patterns $\boldsymbol{b1}$, $\boldsymbol{b2}$ from Figure 3. The input of a rule is always a set of ground patterns (i.e., completely instantiated patterns) which is processed in five phases.

1. Each pattern in the input set is matched against the body of the rule thus forming the following set of variable bindings.
   { [$\boldsymbol{Pbr} = \boldsymbol{b1}$; $Num = 1$; $T =$ "Golf"; $\ldots$; $SN =$ "VW center"; $Add =$ "Bd Lenoir ..."]
   [$\boldsymbol{Pbr} = \boldsymbol{b2}$; $Num = 2$; $T =$ "Golf"; $\ldots$; $SN =$ "VW2"; $Add =$ "Bd Leblanc ..."]
   [$\boldsymbol{Pbr} = \boldsymbol{b2}$; $Num = 2$; $T =$ Golf"; $\ldots$; $SN =$ "VW Center"; $Add =$ "Bd Lenoir ..."] }

2. External functions are evaluated to generate new bindings. The first element of the above set then becomes:
   [$\boldsymbol{Pbr} = \boldsymbol{b1}$; $Num = 1$; $T =$ "Golf"; $\ldots$; $Add =$ "Bd Lenoir ..."; $C =$ "Paris"; $Z = 75005$]

   External functions are typed. This means that a type filter is applied on the set of variable bindings before they are evaluated.

3. Predicates are applied to filter the set of variable bindings. In our case, all bindings are kept.

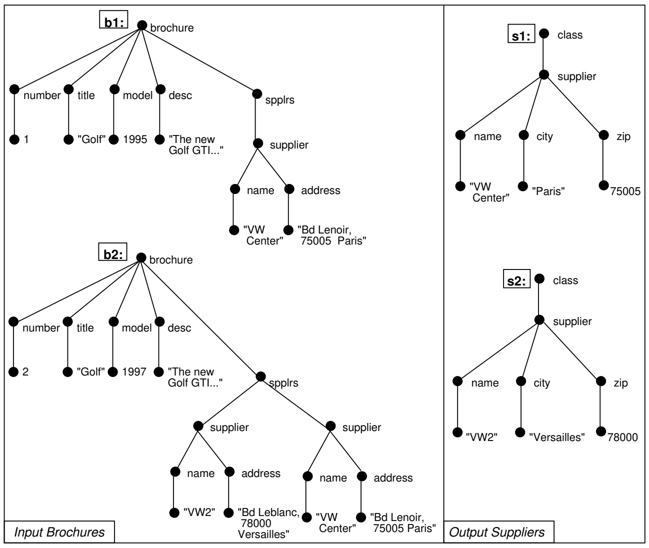

Figure 3: Applying Rule 1 on two SGML brochures

4. Skolem functions are evaluated. In this case, $\boldsymbol{s1}$ is associated to $\boldsymbol{Psup}$("VW center") and $\boldsymbol{s2}$ is associated to $\boldsymbol{Psup}$("VW2"). Skolem functions are not dependent of a given rule but are global to a program. In other words, the association between $\boldsymbol{Psup}$("VW center") and $\boldsymbol{s1}$ can be recalled in another rule of the program.

5. Finally, the set of output patterns is constructed (in a straightforward manner as far as this example is concerned) and each output pattern is associated to its name (see Figure 3).

Note that step 4 and 5 can be mixed since it is not necessary to evaluate all skolem functions before computing the corresponding pattern tree. Let us now consider the second and last rule of our program. It creates one car object for each brochure, each car referencing its set of suppliers.

Rule 2

$$
\begin{array}{lcl}
\boldsymbol{Pcar}(\boldsymbol{Pbr}): & & \boldsymbol{Pbr}: \\
class \rightarrow car & & brochure \\
\quad \langle \rightarrow name \rightarrow T, & & \quad \langle \rightarrow number \rightarrow Num, \\
\quad\quad \rightarrow desc \rightarrow D, & \Longleftarrow & \quad\quad \rightarrow title \rightarrow T, \\
\quad\quad \rightarrow suppliers & & \quad\quad \rightarrow model \rightarrow Year, \\
\quad\quad\quad \rightarrow set \overset{\{\}}{\rightarrow} \boldsymbol{\&Psup}(SN)\rangle & & \quad\quad \rightarrow desc \rightarrow D, \\
 & & \quad\quad \rightarrow suppls \overset{\star}{\rightarrow} supplier \\
 & & \quad\quad\quad \langle \rightarrow name \rightarrow SN, \\
 & & \quad\quad\quad\quad \rightarrow address \rightarrow Add\rangle\rangle
\end{array}
$$

The creation of links from cars to suppliers is handled by the use of the parameterized pattern name $\boldsymbol{Psup}(SN)$. Because skolem functions are global to a program and can be processed independently from their values, Rule 1 and Rule 2 can be applied in any order. So, it is not necessary to use stratification mechanism to preserve YATL declarativity. Note that we use here the **&** symbol to denote a reference to a supplier. In other words, the corresponding leaves in the output pattern tree will point to a supplier pattern tree (e.g., $\boldsymbol{\&s1}$). Omitting the **&** symbol entails a dereferenciation, e.g., $\boldsymbol{s1}$ would be replaced by its associated tree. Remark that it requires that the value associated to $\boldsymbol{s1}$ exists. Therefore, dereferenciation is handled at the end of rules processing.

Note the use of symbol {} in the above rule's head. This annotation is used to create a node (here, the node *set*) with several sons, all distinct and in no specified order. In the above rule, node *set* will aggregate all the references to the suppliers of a given car.

Let us now see how cyclic references are managed in YATL. For this, we redefine the way suppliers are created. Rule $1'$ shows how cars can be associated to their suppliers.

Rule $1'$

$$
\begin{array}{lcl}
 & & \boldsymbol{Pbr}: \\
 & & brochure \\
\boldsymbol{Psup}(SN): & & \quad \langle \rightarrow number \rightarrow Num, \\
class \rightarrow supplier & & \quad\quad \rightarrow title \rightarrow T, \\
\quad \langle \rightarrow name \rightarrow SN, & & \quad\quad \rightarrow model \rightarrow Year, \\
\quad\quad \rightarrow city \rightarrow C, & \Longleftarrow & \quad\quad \rightarrow desc \rightarrow D, \\
\quad\quad \rightarrow zip \rightarrow Z, & & \quad\quad \rightarrow suppls \overset{\star}{\rightarrow} supplier \\
\quad\quad \rightarrow sells & & \quad\quad\quad \langle \rightarrow name \rightarrow SN, \\
\quad\quad\quad \rightarrow set \overset{\{\}}{\rightarrow} \boldsymbol{\&Pcar}(\boldsymbol{Pbr})\rangle & & \quad\quad\quad\quad \rightarrow address \rightarrow Add\rangle\rangle, \\
 & & C \texttt{ is } \mathrm{city}(Add), \\
 & & Z \texttt{ is } \mathrm{zip}(Add)
\end{array}
$$

Remember that the **&** symbol before a pattern name corresponds to a reference. Should we remove the **&** symbol in both Rules $1'$ and 2, we would face a cyclic program. Detection of cyclic programs is addressed in Section 3.4.

3.2 Dealing with heterogeneity

To illustrate YATL's ability to handle heterogeneity between multiple data sources, we propose to write a rule taking its data from two distinct sources: a relational database whose schema is given below and the above SGML documents.

Suppliers Relational Database Schema

suppliers[sid: integer, name: string, city: string,
address: string, tel: string]
cars[cid: integer, broch_num: string]
sales[sid : integer, cid: integer, year: integer, sold: integer]

Rule 3 creates one car object for each distinct car in the relational database if it corresponds to a brochure. Note that the SN variable is used in both body patterns to indicate that the supplier name in the relational database and in the SGML documents should be the same. The external function 'sameaddress' takes care of the heterogeneity between addresses in the database and in the SGML documents.

Rule 3

$$
\begin{array}{lcl}
 & & \boldsymbol{Pbr} : \\
 & & brochure \\
 & & \quad \langle \rightarrow number \rightarrow Num, \\
 & & \quad \rightarrow title \rightarrow T, \\
 & & \quad \rightarrow model \rightarrow Year, \\
\boldsymbol{Pcar}(Cid) : & & \quad \rightarrow desc \rightarrow D, \\
class \rightarrow car & & \quad \rightarrow suppls \stackrel{\star}{\rightarrow} supplier \\
\quad \langle \rightarrow name \rightarrow T, & \Longleftarrow & \qquad \langle \rightarrow name \rightarrow SN, \\
\quad \rightarrow desc \rightarrow D, & & \qquad \rightarrow address \rightarrow Add \rangle \rangle, \\
\quad \rightarrow suppliers & & \boldsymbol{Rsuppliers} : \\
\qquad \rightarrow set \stackrel{\star}{\rightarrow} \boldsymbol{\&Psup}(Sid) \rangle & & suppliers \stackrel{\star}{\rightarrow} row \\
 & & \quad \langle \rightarrow sid \rightarrow Sid, \\
 & & \quad \rightarrow name \rightarrow SN, \\
 & & \quad \rightarrow city \rightarrow C, \\
 & & \quad \rightarrow address \rightarrow Add_2, \\
 & & \quad \rightarrow tel \rightarrow Tel \rangle, \\
 & & \boldsymbol{Rcars} : \\
 & & cars \stackrel{\star}{\rightarrow} row \\
 & & \quad \langle \rightarrow cid \rightarrow Cid \\
 & & \quad \rightarrow broch\_num \rightarrow Num \rangle, \\
 & & \text{sameaddress}(Add, C, Add_2)
\end{array}
$$

This rule shows how one can handle eventual inconsistencies in the data sources using the YAT environment. Note that a similar approach to handle heterogeneity has been followed in [11, 18, 13].

3.3 Dealing with collections

Like other languages for data conversion/integration, YATL handles sets. However, an original feature of the language is that, due to its data model and its active domain semantics (as opposed to fix-point), it is also well adapted to the manipulation of collections such as multi-sets and lists. YATL provides one primitive for grouping and another for ordering. These two simple features are sufficient to deal appropriately with collections. We illustrate this point with two examples.

Rule 4 creates an ODMG list of supplier objects (as created by Rule 1') ordered by their name. For this it uses a primitive that combines both grouping (to remove duplicates) and ordering on a given criterion. This requires that the set of variable bindings be processed (in that case grouped and ordered) before the actual construction of the corresponding edges. In the example, the criterion is a single variable. More variables can be used. Also, one can perform grouping on some criteria and ordering on some others.

Rule 4

$$
\begin{array}{lcl}
 & & \boldsymbol{Pbr} : \\
 & & brochure \\
 & & \quad \langle \rightarrow number \rightarrow Num, \\
\boldsymbol{Pallsups}() : & & \quad \rightarrow title \rightarrow T, \\
list \stackrel{[]_{SN}}{\rightarrow} \boldsymbol{\&Psup}(SN) & \Longleftarrow & \quad \rightarrow model \rightarrow Year, \\
 & & \quad \rightarrow desc \rightarrow D, \\
 & & \quad \rightarrow suppls \stackrel{\star}{\rightarrow} supplier \\
 & & \qquad \langle \rightarrow name \rightarrow SN, \\
 & & \qquad \rightarrow address \rightarrow Add \rangle \rangle
\end{array}
$$

The possibility to handle ordered collections is important for various application domains. Ordered collections exist in many standard data models, and are essential to deal with documents. The above example was rather database oriented. The YATL language is also able to manipulate arrays, which are the subject of increasing interest [22, 23, 24]. This is illustrated by the following rule that transposes any input matrix. Note the use of the index edges I and J to store the information about the original ordering of the sons.

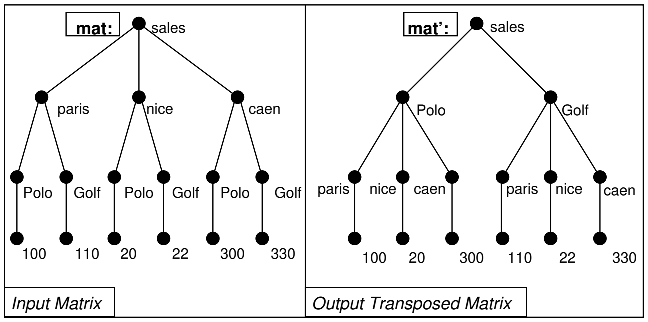

Figure 4: Transposing a 3×2 Matrix

Rule 5

$$
\boldsymbol{New(Id)} : \; Mat \stackrel{[]_J}{\rightarrow} Y \stackrel{[]_I}{\rightarrow} X \rightarrow A \Longleftarrow \boldsymbol{Id} : \; Mat \stackrel{I}{\rightarrow} X \stackrel{J}{\rightarrow} Y \rightarrow A
$$

Figure 4 illustrates the application of Rule 5 on a matrix containing statistics on car sales.

3.4 Detecting cyclic programs

Remember our comment about the program presented in Section 3.1 which consisted of Rules 1' and 2: if we remove the & symbol in both rules we potentially introduce a cycle in the program. Indeed, let us consider a car $\boldsymbol{c1}$ and a supplier $\boldsymbol{s1}$ referencing each other. By removing the & symbol, we break the referencing mechanism and require that the $\boldsymbol{s1}$ tree includes the $\boldsymbol{c1}$ tree which itself should include the $\boldsymbol{c1}$ tree, etc.

The bad news is that the general problem of statically detecting cycles in a YATL program is undecidable (it is similar to finding wrong recursive calls in a programming language). However, we may detect possibly cyclic programs and forbid their use. One simple way to do that is to construct the dependency graph of dereferenced Skolems. If this graph is cyclic, it implies a potential cycle in the program that is thus rejected. If we consider the above example, there exists a cycle between the $\boldsymbol{Psup}$ and $\boldsymbol{Pcar}$ skolem functors.

However, this method has a serious drawback. It rejects conversion programs working by recursion on the input tree, programs commonly used for arbitrarily deep recursive structures. Fortunately, this important class of programs features an interesting characteristic: they usually rely on a skolem functor whose sole parameter is a pattern name and whose recursive invocation is always performed on a subtree of the input. This is further explained in the next section. These recursive programs are called *safe-recursive*. This can be checked syntactically and guarantees the absence of cycles at run time.

To summarize, we reject programs that are not *safe-recursive* and that feature a cycle in the graph of dependencies of the dereferenciated Skolems.

3.5 Typing in YATL

To conclude this section, we briefly explain how typing can be performed in YATL.

A program takes a model as input and returns another model (see Section 2). Input and output models can easily be inferred by considering the program (i) input and output patterns, (ii) predicate/function signatures and (iii) variable domains.

For instance, consider the program consisting of Rule 1 and Rule 2. The input model of the program consists of the single brochure pattern $\boldsymbol{Pbr}$ with restrictions on the type of variables Add and $Year$. The type of Add is given by the signature of functions *city* and *zip*, that of $Year$ by the ">" predicate. The output model consists of two patterns $\boldsymbol{Pcar}$ and $\boldsymbol{Psup}$ and restrictions on variables C, Z.

The couple of input/output models will be called the signature of a conversion program and noted $M_{IN} \mapsto M_{OUT}$. Now, once the signature of a program has been inferred, this information can be used in several ways.

It can be used to check if two programs can safely be composed or combined. This is explained in the next section. It can also be used to check that the input (or output) model of a program is indeed an instance of a more general model. For instance, the user may check that a program generates *car* and *supplier* objects compliant with a given ODMG schema or, more generally, with the ODMG model. Finally, by turning typing on at run time, we may verify that all input data is involved in the conversion process. This is simply done by adding a rule such as Rule Exception to the program. The rule should be applied only when another rule cannot (see the following section to understand how this is possible). It features an empty head. Its body contains one pattern that matches any input data and an external function that invokes an exception.

Rule Exception

$$
.\Longleftarrow \boldsymbol{Yat} : \boldsymbol{P2} : \boldsymbol{Yat},\\
\mathrm{E}\ \mathtt{is}\ \mathrm{exception}()
$$

It is important to understand that typing in YAT is in no way constraining. Programs do not need it to be executed. If a part of the input data does not match the input model of a program, no conversion will be performed on it, but no error will occur.

4 Customizing, combining and composing conversion programs

One of the most interesting feature of a YATL program is its ability to be "instantiated". Program instantiation constitutes the basis of program customization. We show now how it works using an example. We next show how programs can be combined so as to provide both customized and general behaviors. Finally, we explain program composition.

4.1 Customizing programs

No matter the user-friendliness of a language or graphical interface, most programmers do not like to start from scratch. The common behavior consists in fetching bits and pieces of programs and to adapt them. With YAT, the programmer starts with a general program that can be instantiated

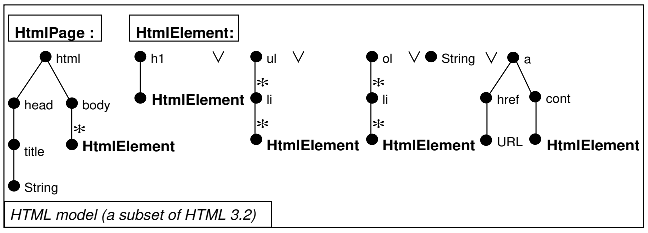

Figure 5: HTML patterns

into one that corresponds to its input data. Customization can follow. For instance, through successive instantiations/customizations, we may find programs generating HTML pages from (i) any input, (ii) ODMG compliant data, (iii) data corresponding to a specific ODMG schema or (iv) an object of that schema. Let us illustrate this with an example.

Below is a program that converts any ODMG data into HTML, thus allowing users of our Intranet application to view data from a Web browser. The specified translation is that implemented by the O_2Web system [25]. An object is converted into an HTML page (Rule Web1), an atomic value into a string (Rule Web2), a collection or a tuple into a list of HTML items (Rules Web3 to Web5) and an object reference to an HTML anchor (Rule Web6). The rules bodies contain ODMG patterns, while rules heads contains HTML patterns as illustrated in Figure 5. Four facts are noteworthy.

1. The program creates a new identifier for each HTML page through the $\boldsymbol{HtmlPage}$ skolem function. It is the HTML wrapper's responsibility to map these pattern identifiers to a real URL when creating the actual HTML pages.
2. The program uses recursive dereferenciation on skolem $\boldsymbol{HtmlElement}$ (in rules Web3, Web4 and Web5), but is *safe-recursive* since the skolem parameter is a pattern name ($\boldsymbol{Ptype}$) and the recursive calls are performed on a subtree ($\boldsymbol{P_2}$).
3. Some edges are labeled with the $\star$ symbol. This label provides implicit grouping as the $\{\}$ symbol but without duplicate elimination.
4. Finally, many variables are typed. For instance, X in Rule Web4 (resp. Web5) can only be instantiated with the *set* or *bag* (resp. *list* or *array*) symbol. $\boldsymbol{P_2}$ in Rules Web3 to Web6 can only be instantiated by a pattern name instance of $\boldsymbol{Ptype}$.

Web1

$$
\begin{array}{l}
\boldsymbol{HtmlPage}(\boldsymbol{Pclass}) :\\
html\langle \rightarrow head \rightarrow title \rightarrow Classname,\\
\quad \rightarrow body\langle \rightarrow h1 \rightarrow Classname,\\
\qquad\quad \rightarrow ul \xrightarrow{\star} li\\
\qquad\qquad \langle \rightarrow Att2,\\
\qquad\qquad\quad \rightarrow \boldsymbol{HtmlElement}(\boldsymbol{Ptype})\rangle\rangle\rangle\\
\qquad \Longleftarrow\\
\qquad\quad \boldsymbol{Pclass} :\\
\qquad\quad class \rightarrow Classname \xrightarrow{\star} Att \rightarrow \boldsymbol{Ptype},\\
\qquad\quad Att2\ \mathtt{is}\ \mathrm{concat}(Att, \text{" : "})
\end{array}
$$

Web2

$$\boldsymbol{HtmlElement}(\boldsymbol{Ptype}) : S \Longleftarrow \boldsymbol{Ptype} : Data,\quad S \texttt{ is } \mathrm{data\_to\_string}(Data)$$

Web3

$$\begin{array}{l}
\boldsymbol{HtmlElement}(\boldsymbol{Ptype}) : \\
ul \xrightarrow{\star} li\langle \rightarrow Att2, \\
\qquad \rightarrow \boldsymbol{HtmlElement}(\boldsymbol{P_2} : \boldsymbol{Ptype})\rangle \\
\qquad \Longleftarrow \\
\qquad\qquad \boldsymbol{Ptype} : \\
\qquad\qquad tuple \xrightarrow{\star} Att \rightarrow \boldsymbol{P_2} : \boldsymbol{Ptype} \\
\qquad\qquad Att2 \texttt{ is } \mathrm{concat}(Att, \text{" : "})
\end{array}$$

Web4

$$\begin{array}{l}
\boldsymbol{HtmlElement}(\boldsymbol{Ptype}) : \\
ul \xrightarrow{\star} li \rightarrow \boldsymbol{HtmlElement}(\boldsymbol{P_2} : \boldsymbol{Ptype})\rangle \\
\quad \Longleftarrow \\
\qquad \boldsymbol{Ptype} : \\
\qquad X : \{set, bag\} \xrightarrow{\star} \boldsymbol{P_2} : \boldsymbol{Ptype}
\end{array}$$

Web5

$$\begin{array}{l}
\boldsymbol{HtmlElement}(\boldsymbol{Ptype}) : \\
ol \xrightarrow{o(I)} li \rightarrow \boldsymbol{HtmlElement}(\boldsymbol{P_2} : \boldsymbol{Ptype})\rangle \\
\quad \Longleftarrow \\
\qquad \boldsymbol{Ptype} : \\
\qquad X : \{list, array\} \xrightarrow{I} \boldsymbol{P_2} : \boldsymbol{Ptype}
\end{array}$$

Web6

$$\begin{array}{l}
\boldsymbol{HtmlElement}(\boldsymbol{Ptype}) : \\
a\langle \rightarrow href \rightarrow \boldsymbol{\&HtmlPage}(\boldsymbol{Pclass}), \\
\quad \rightarrow cont \rightarrow Classname\rangle \\
\quad \Longleftarrow \\
\qquad \boldsymbol{Ptype} : \boldsymbol{\&Pclass}, \\
\qquad \boldsymbol{Pclass} : \\
\qquad class \rightarrow Classname \rightarrow \boldsymbol{P_2} : \boldsymbol{Ptype}
\end{array}$$

Now, let us suppose that we want to display objects of the class car in a different way and the Golf GTI object in yet another way. Using a tool such as O_2Web, this would entail writing appropriate methods from scratch using C++ or some other O_2 supported programming language. With YAT, the user instantiates the general program by giving a more specific pattern. This instantiation process is done automatically, and the resulting new program is equivalent to the previous one, but more specific. This program can then be customized accordingly to the programmer's needs. Let us illustrate this by instantiating the Web program through the $\boldsymbol{Pcar}$ pattern introduced in Section 3 (see Rule 2). Instantiation results in the following Rule.

rule WebCar

$$\begin{array}{l}
\boldsymbol{HtmlPage}(\boldsymbol{Pcar}) : \\
html \\
\;\langle \rightarrow head \rightarrow title \rightarrow car, \\
\quad \rightarrow body\langle \rightarrow h1 \rightarrow car, \\
\qquad\quad \rightarrow ul\langle \rightarrow li\langle \rightarrow \text{"name : "}, \\
\qquad\qquad\qquad\quad \rightarrow T1\rangle, \\
\qquad\qquad\quad \rightarrow li\langle \rightarrow \text{"age : "}, \\
\qquad\qquad\qquad\quad \rightarrow D1\rangle, \\
\qquad\qquad\quad \rightarrow li\langle \rightarrow \text{"suppliers : "}, \\
\qquad\qquad\qquad\quad \rightarrow ul \xrightarrow{\star} li \rightarrow a \\
\qquad\qquad\qquad\quad \langle \rightarrow href \rightarrow \boldsymbol{\&HtmlPage}(\boldsymbol{Psup}), \\
\qquad\qquad\qquad\qquad \rightarrow cont \rightarrow Classname\rangle\rangle\rangle\rangle\rangle \\
\quad \Longleftarrow \\
\qquad \boldsymbol{Pcar} : \\
\qquad class \rightarrow car \\
\qquad\quad \langle \rightarrow name \rightarrow T, \\
\qquad\quad\; \rightarrow desc \rightarrow D, \\
\qquad\quad\; \rightarrow suppliers \rightarrow set \xrightarrow{\star} \boldsymbol{\&Psup}\rangle, \\
\qquad \boldsymbol{Psup} : \\
\qquad class \rightarrow Classname \rightarrow \boldsymbol{P_2} : \boldsymbol{Ptype}, \\
\qquad T1 \texttt{ is } \mathrm{data_to_string}(T), \\
\qquad D1 \texttt{ is } \mathrm{data_to_string}(D)
\end{array}$$

The body of the rule contains two function calls and two patterns: one is the given $\boldsymbol{Pcar}$ pattern, the other one is an incomplete $\boldsymbol{Psup}$ pattern[3] which has been obtained through instantiation of rule Web6. It specifies that pattern name $\boldsymbol{Psup}$ can only be instantiated by an object belonging to some class.

Let us see how Rule WebCar has been derived. Four rules have been used: Rule Web1 on the upper part of the car pattern, Rule Web2 on the two subtrees associated to properties *name* and *desc*, Rule Web4 on the subtree associated to the *supplier* property and Rule Web6 on the leaf containing a reference to $\boldsymbol{Psup}$. Then, output trees have been generated and, due to the dereferenciation, have been appended together to form the head part of the rule. One exception is the pattern that matched the $\boldsymbol{\&Psup}$ leaf. Since it has not been dereferenced, it has been added to the rule body along with all encountered function calls. Remark that two calls to the 'data_to_string' external functions are derived from rule Web2 and therefore, the system must provide appropriate renaming of variables (here to $T1$ and $D1$) to avoid conflicts.

Once derived, the new rule can be rewritten to provide a customized presentation for car objects. For instance, if we do not want to display the supplier of a car, we can rewrite Rule WebCar in the following way:

rule newWebCar

$$\begin{array}{lcl}
\boldsymbol{HtmlPage}(\boldsymbol{Pcar}) : & & \\
html & & \boldsymbol{Pcar} : \\
\;\langle \rightarrow head \rightarrow title \rightarrow car, & & class \rightarrow car \\
\quad \rightarrow body & & \;\langle \rightarrow name \rightarrow T, \\
\quad\; \langle \rightarrow h1 \rightarrow car, & \Longleftarrow & \;\; \rightarrow desc \rightarrow D, \\
\qquad \rightarrow ul\langle \rightarrow li\langle \rightarrow \text{"name : "}, & & \;\; \rightarrow suppliers \\
\qquad\qquad\qquad \rightarrow T1\rangle, & & \;\;\; \rightarrow set \xrightarrow{\star} \boldsymbol{\&Psup}\rangle, \\
\qquad\qquad \rightarrow li\langle \rightarrow \text{"age : "}, & & T1 \texttt{ is } \mathrm{data_to_string}(T), \\
\qquad\qquad\qquad \rightarrow D1\rangle\rangle\rangle\rangle & & D1 \texttt{ is } \mathrm{data_to_string}(D)
\end{array}$$

[3] Since we only give the $\boldsymbol{Pcar}$ pattern for the instantiation, the system does not assume any knowledge of the $\boldsymbol{Psup}$ pattern referenced in $\boldsymbol{Pcar}$.

4.2 Combining programs

Rule WebCar can be considered as the unique rule of a program translating car objects. Obviously, we would like to combine this program with the previous one so as to be able to convert all objects into appropriate HTML pages. However, by doing this without caution we would provide two translations for a given car object.

To solve this problem, YATL interpreter organizes the set of rules of a program hierarchically. Thus, a program now consists of a set of rule hierarchies. For a given input pattern, the more specific rules (leaves in the hierarchy) matching the input are applied first. If matching cannot be obtained, less specific rules in the hierarchy are tried and so on.

Rule hierarchies are built by the YATL interpreter according to possible rule conflicts. A conflict occurs only when: (i) there is a subtype relationship between two rules input models (as for rules WebCar and Web1 in the above example), and (ii) the skolem functions used in these rules are the same (e.g. there is no conflict for rules 1 and 2 defined in Section 3 as they do not code for the same set of output patterns).

Finally, the user may also enforce the rule hierarchy to apply some rules in a particular order, according to the conversion he wants to achieve. For that purpose, YATL allows to enforce a specific hierarchy of rules. Of course, in this case, the declarativity of YATL programs is transgressed. As was noted in the previous section, this ability to organize rules can be used to check programs at run time.

4.3 Composing programs

Let us now explain program composition. Recall the example scenario and the program consisting of Rules 1 and 2 that created cars and suppliers from a given brochure. In order to avoid unnecessary loading into the object database, the user may want to combine this program with the above one that performs ODMG to HTML conversion. This would provide a direct conversion from SGML documents to HTML.

A first solution would be to apply successively the two programs. However, this would result in unnecessary processing, since the system would create intermediate ODMG patterns. Using YATL, it is possible to *compose* these two programs and generate a more efficient program.

Let us explain the principle of the composition mechanism. Taking two conversion programs $prg1 : M_1 \mapsto M_2$ whose input (resp. output) model is M_1 (resp. M_2) and $prg2 : M_2' \mapsto M_3$, the system first check if $prg1$ and $prg2$ are compatible (i.e. if M_2 is an instance of M_2'). If this is the case, the system instantiates $prg2$ with the patterns of M_2. This produces a new translation $prg2' : M_2 \mapsto M_3'$. Then, the final composition is straightforward as syntactically identical patterns appear in the output model of $prg1$ and the input model of $prg2'$.

The signature of the two programs that we want to combine (SGML to ODMG and ODMG to HTML) are compatible. Therefore, the ODMG to HTML translation can be instantiated on $\boldsymbol{Pcar}$ and $\boldsymbol{Psup}$ patterns. It generates a rule WebCar′ (which only differs from Rule WebCar by having additional informations about pattern $\boldsymbol{Psup}$) and another rule WebSup. Finally, the program containing Rule 1 and Rule 2 is composed with this instantiated program. Two new rules result from this composition. The first, from the composition of Rule 1 and Rule WebSup, creates HTML pages for the suppliers contained in our SGML brochures. The second, from the composition of Rule 2 and

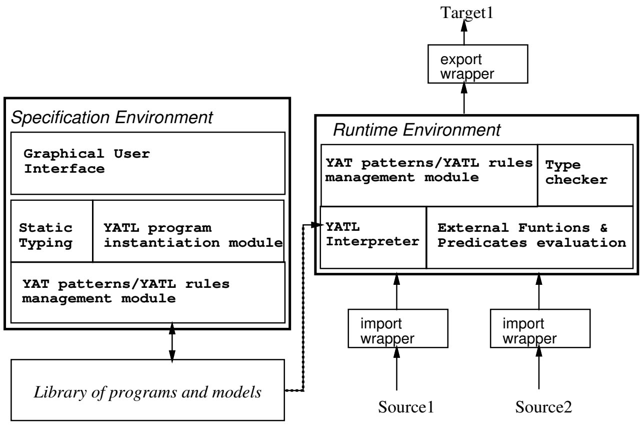

Figure 6: YAT System architecture

Rule WebCar′, generates HTML pages for the car descriptions. It is given below.

Rule (2+Webcar′)

$$
\begin{array}{l}
\boldsymbol{HtmlPage}(\boldsymbol{Pcar}) : \\
html \\
\quad \langle \to head \to title \to car, \\
\quad\quad \to body \langle \to h1 \to car, \\
\quad\quad\quad\quad \to ul \langle \to li \langle \to \text{"name : "}, \\
\quad\quad\quad\quad\quad\quad \to T1 \rangle, \\
\quad\quad\quad\quad\quad \to li \langle \to \text{"age : "}, \\
\quad\quad\quad\quad\quad\quad \to D1 \rangle, \\
\quad\quad\quad\quad\quad \to li \langle \to \text{"suppliers : "}, \\
\quad\quad\quad\quad\quad\quad \to ul \xrightarrow{\star} li \to a \\
\quad\quad\quad\quad\quad\quad\quad \langle \to href \to \boldsymbol{\&HtmlPage}(SN), \\
\quad\quad\quad\quad\quad\quad\quad\quad \to cont \to supplier \rangle\rangle\rangle\rangle\rangle \\
\quad \Longleftarrow \\
\quad\quad\quad \boldsymbol{Pbr} : \\
\quad\quad\quad brochure \langle \to number \to Num, \\
\quad\quad\quad\quad\quad \to title \to T, \\
\quad\quad\quad\quad\quad \to model \to Year, \\
\quad\quad\quad\quad\quad \to desc \to D, \\
\quad\quad\quad\quad\quad \to suppls \xrightarrow{\star} supplier \\
\quad\quad\quad\quad\quad\quad \langle \to name \to SN, \\
\quad\quad\quad\quad\quad\quad\quad \to address \to Add \rangle\rangle, \\
\quad\quad\quad T1 \texttt{ is } \mathrm{data\_to\_string}(T), \\
\quad\quad\quad D1 \texttt{ is } \mathrm{data\_to\_string}(D)
\end{array}
$$

5 Architecture and implementation

In this section, we briefly present the architecture of the YAT System as well as the current status of the prototype.

5.1 Architecture

The YAT system provides a complete environment to design data conversion programs. It does not provide the ability to query or update external sources. Still, we believe that it can serve as the basis for a mediator/wrapper system for data integration and are currently working in that direction.

The heart of the system is composed of: (i) a module to manipulate YAT data and programs, (ii) a module for type checking and type inference that can be called *on demand* and (iii) a YATL interpreter.

Figure 6 describes the current architecture. There are three main parts : (i) the specification environment, (ii) the run-time environment and (iii) a library of programs and formats. Both environments rely on the "YAT model/YATL rules management module" for the internal representation and manipulation of patterns and rules. The specification environment provides (i) a module to statically check/infer the type of a program and (ii) a graphical user interface to specify translations, using (iii) program instantiation to customize existing programs if needed. The execution environment uses wrappers to import (resp. export) data into (resp. from) YAT and an interpreter to perform the translation. The interpreter relies on a separate module to process external functions and predicates and, if required by the user, a type checker. The library allows to save and import programs and models.

The runtime environment can be used independently or be linked to import/export wrappers to generate stand-alone executables (e.g. like LaTeX2HTML). In this case the translation system connects to the import program to retrieve the input data whenever required. If the HTML output wrapper is used, the generated executable can be used as a CGI script.

5.2 Prototype

The YAT system is implemented in the Verso Database Group at INRIA. It is written in Objective CAML [14] and JAVA (for the graphical interface). It is now fully operational. A first stable version has been delivered to our partners of the OPAL project. It provides all the above features as well as some export/import wrappers (HTML, O_2 database and OPAL specific data) and appropriate conversion programs. Figures 7 and 8 show the YATL editor main window with Rule $1'$ and the pattern editor. [26] gives a complete description of this graphical interface.

We are currently enhancing this first version of the prototype with new wrappers, and plan to work on performance issues.

Acknowledgments

The authors would like to thank Serge Abiteboul, Daniel Chan and Anne-Marie Vercoustre for useful comments on earlier versions of the paper.

References

[1] W. Litwin, L. Mark, and N. Roussopoulos, "Interoperability of multiple autonomous databases," *ACM Computing Surveys*, vol. 22, no. 3, pp. 267–293, Sept. 1990.

[2] M. J. Carey, L. M. Haas, P. M. Schwarz, M. Arya, W. F. Cody, R. Fagin, M. Flickner, A. W. Luniewski, W. Niblack, D. Petkovic, J. Thomas, J. H. Williams, and E. L. Wimmers, "Towards heterogeneous multimedia information systems: The garlic approach," in *Research Issues in Data Engineering*, Los Alamitos, California, Mar. 1995, pp. 124–131.

[3] Y. Papakonstantinou, H. Garcia-Molina, and J. Widom, "Object exchange across heterogeneous information sources," in *Proc. of IEEE Int. Conf. on Data Engineering (ICDE)*, Taipei, Taiwan, Mar. 1995, pp. 251–260.

[4] A. Tomasic, L. Raschid, and P. Valduriez, "Scaling heterogeneous databases and the design of disco," in *Proceedings of the 16th International Conference on Distributed Computing Systems*, Hong Kong, May 1996, pp. 449–457.

[5] S. Adali, K. Candan, Y. Papakonstantinou, and V. S. Subrahmanian, "Query caching and optimization in distributed mediator systems," in *Proc. of the ACM SIGMOD Conf. on Management of Data*, Montreal, Canada, June 1996, pp. 137–148.

[6] P. Buneman, S. B. Davidson, G. Hillebrand, and D. Suciu, "A query language and optimization techniques for unstructured data," in *Proc. of the ACM SIGMOD Conf. on Management of Data*, Montreal, Canada, June 1996, pp. 505–516.

[7] S. Abiteboul, D. Quass, J. McHugh, J. Widom, and J. L. Wiener, "The lorel query language for semistructured data," *International Journal on Digital Libraries*, vol. 1, no. 1, pp. 68–88, Apr. 1997.

[8] M. F. Fernandez, D. Florescu, A. Y. Levy, and D. Suciu, "A query language for a web-site management system," *SIGMOD Record*, vol. 26, no. 3, pp. 4–11, Sept. 1997.

[9] R. Goldman and J. Widom, "Data guides: Enabling query formulation and optimization in semistructured databases," in *Proc. International Conference on Very Large Data Bases*, Athens, Greece, Aug. 1997, pp. 436–445.

[10] P. Buneman, S. B. Davidson, and D. Suciu, "Programming constructs for unstructured data," in *Proc. Int. Workshop on Database Programming Languages*, Gubbio, Italy, 1995.

[11] S. Abiteboul, S. Cluet, and T. Milo, "Correspondence and translation for heterogeneous data," in *Proc. Int. Conf. on Database Theory (ICDT)*, Delphi, Greece, Jan. 1997.

[12] M. Kifer, W. Kim, and Y. Sagiv, "Querying object-oriented databases," in *Proc. of the ACM SIGMOD Conf. on Management of Data*, San Diego, California, June 1992, pp. 393–402.

[13] Y. Papakonstantinou, S. Abiteboul, and H. Garcia-Molina, "Object fusion in mediator systems," in *Proc. International Conference on Very Large Data Bases*, Bombay, India, Sept. 1996, pp. 413–424.

[14] X. Leroy, *The Objective Caml system release 1.07*, INRIA, Dec. 1997, Documentation and user's manual. `ftp://ftp.inria.fr/lang/caml-light/`.

[15] U. Dayal and H. Hwang, "View definition and generalisation for database integration in multibase: A system for heterogeneous distributed databases," *IEEE Transactions on Software Engineering*, vol. 10, no. 6, pp. 628–644, Nov. 1984.

[16] S. Abiteboul and R. Hull, "Restructuring hierarchical database objects," *Theoretical Computer Science*, vol. 62, pp. 3–38, 1988.

[17] A. S. Kosky, *Transforming Databases with Recursive Data Structures*, Ph.D. thesis, University of Pennsylvania, 1996.

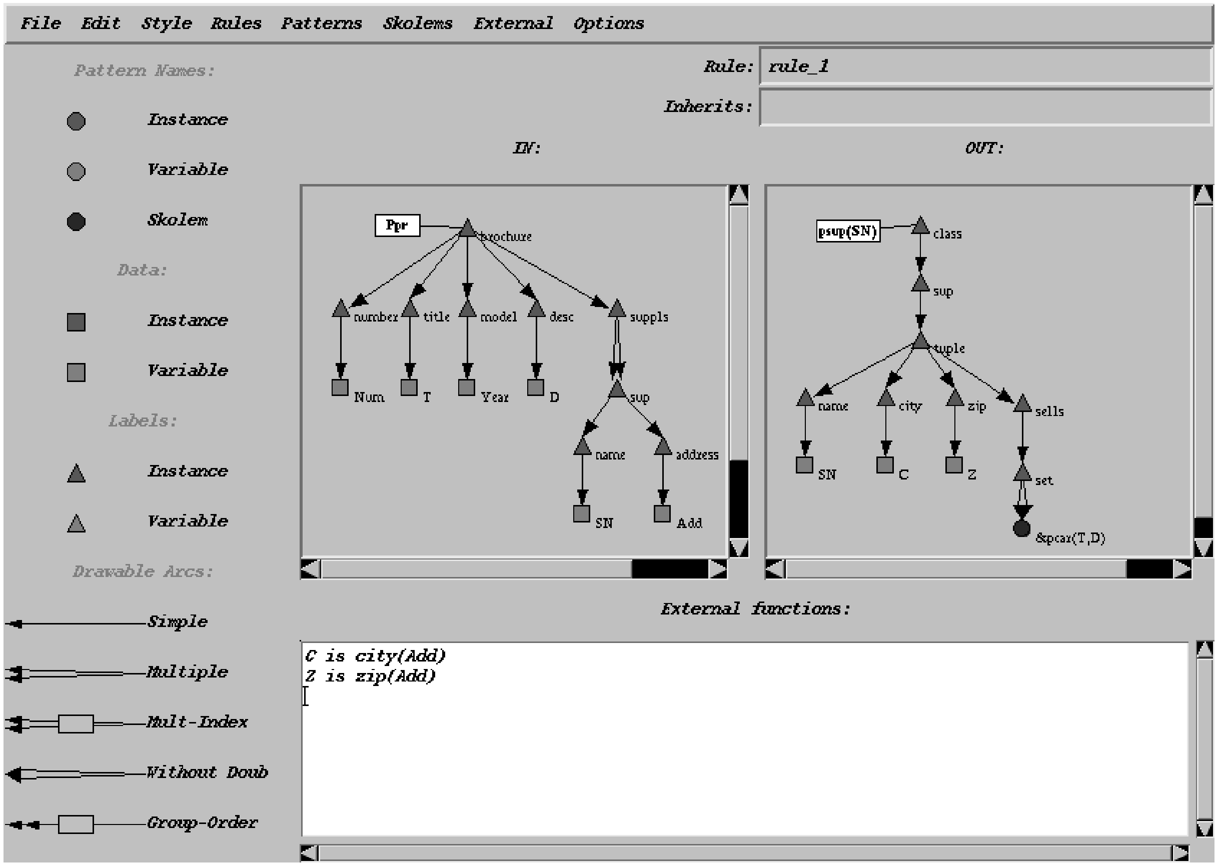

Figure 7: Rule $1'$ using YATL Editor

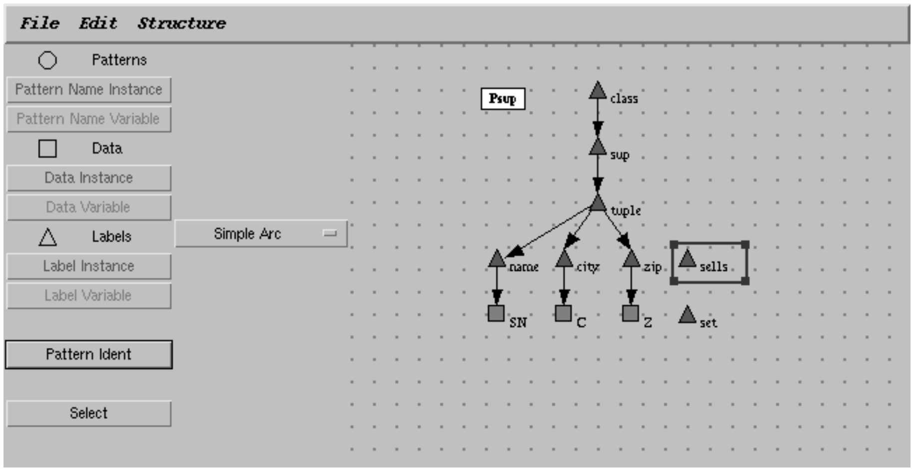

Figure 8: YATL Pattern Editor

[18] Y. Papakonstantinou, H. Garcia-Molina, and J. Ullman, "Medmaker: A mediation system based on declarative specifications," in *Proc. of IEEE Int. Conf. on Data Engineering (ICDE)*, New Orleans, Louisiana, Feb. 1996, pp. 132–141.

[19] S. Abiteboul, "Querying semi-structured data," in *Proc. Int. Conf. on Database Theory (ICDT)*, Delphi, Greece, Jan. 1997, pp. 1–18.

[20] S. B. Davidson and A. S. Kosky, "Wol: A language for database transformations and constraints," in *Proc. of IEEE Int. Conf. on Data Engineering (ICDE)*, Birmingham, UK, Apr. 1997, pp. 55–65.

[21] S. Cluet and J. Siméon, "Data integration based on data conversion and restructuring," Technical report, Verso database group - INRIA, Oct. 1997, `http://www-rocq.inria.fr/verso/Jerome.Simeon/YAT/`.

[22] L. Fegaras and D. Maier, "Towards an effective calculus for object query languages," in *Proc. of the ACM SIGMOD Conf. on Management of Data*, San Jose, California, May 1995, pp. 47–58.

[23] L. Libkin, R. Machlin, and L. Wong, "A query language for multidimensional arrays: Design, implementation, and optimization techniques," in *Proc. of the ACM SIGMOD Conf. on Management of Data*, Montreal, Canada, June 1996, pp. 229–239.

[24] A. P. Marathe and K. Salem, "A language for manipulating arrays," in *Proc. International Conference on Very Large Data Bases*, Athens, Greece, Aug. 1997, pp. 46–55.

[25] O_2 Technology, Versailles, *The O_2Web Reference Manual version 4.6*, Sept. 1996.

[26] K. Smaga, "Interface graphique pour la traduction de données," M.S. thesis, Université Paris VI-CNAM-ENST, Sept. 1997.