*Extreme Markup Languages 2002*

**Montréal, Québec**
**August 6-9, 2002**

# An extensible and modular processing model for document trees

José Luis Sierra
*Dpto. Sistemas Informáticos y Programación*

Baltasar Fernández-Manjón
*Dpto. Sistemas Informáticos y Programación*

Alfredo Fernández-Valmayor
*Dpto. Sistemas Informáticos y Programación*

Antonio Navarro
*Dpto. Sistemas Informáticos y Programación*

## *Abstract*

This paper describes a processing model for XML document trees that combines syntax-directed translation ideas with the construction of modular semantic-based interpreters. This model has two key features: *extensibility* (i.e., new functionalities can be incrementally added) and *modularity* (i.e., processing can be done from auto-contained modules). This model introduces four stages in tree processing. In the first stage, a set of *operational links* are established between the element nodes of the document tree. In the second stage, each element node is decorated with a set of *parameters* and a *composition function*. In the third stage, an *evaluation order* is decided from the link relationships, and the composition functions are applied according to this order, thus obtaining the *semantic value* associated with the tree. This semantic value can be another function that will be evaluated in the fourth stage to yield the final result. We describe this conceptual model, present several examples of its use, and outline an object-oriented framework used to implement our approach.

# An extensible and modular processing model for document trees

## *Table of Contents*



# An extensible and modular processing model for document trees

*José Luis Sierra, Baltasar Fernández-Manjón, Alfredo Fernández-Valmayor, and Antonio Navarro*

## § 1 Introduction

The concept of application that underlies generalized markup languages such as SGML [ISO 1986] [Goldfard 1990] and XML [W3C 2000b] [Bradley 2000] is basically linguistic. We consider that the development of an SGML/XML application is equivalent to using SGML/XML to devise a special purpose markup language. As the development of conventional language processors is a well-understood activity (once the language has been adequately established), the development of an interpreter or a compiler for this language is a systematic task that can be engineered with standard techniques in the area [Aho 1986] [Friedman 2001]. This paper tries to address whether the development of processors for XML applications could take advantage of these techniques. In particular, the approach described in this paper is to consider these processors as *interpreters*.

Our interpreters operate on document trees. To some extent, a program that operates on a document using a tree-based API (such as DOM core [W3C 2000a]) has strong similarities with an interpreter that operates on an abstract syntax tree [Friedman 2001]. Awareness of this fact makes it possible to apply some common techniques in the development of interpreters when we develop programs to process XML documents (for instance, to hide representations' details using abstraction levels, or, more important, to use underlying grammatical structure to direct processing). Anyway, and according to D. A. Espinosa [Espinosa 1995], this kind of *syntax-based interpreter* has several shortcomings. The reason is that the analysis of the parsing tree is interleaved with the semantics actions. So, it leads to interpreters that are longer and more difficult to understand than had the aspects been separated. It also leads to inefficiency when the same sub-tree must be re-analyzed (e.g., in the interpretation of a loop). Finally, it leads to *monolithic* interpreters, because the addition of a new feature on the interpreted language can involve global changes in the overall structure of the interpreter.

A more powerful *semantic-based* approach is an alternative to syntax-based interpreters. This approach is based on the denotational definition of the interpreted language [Stoy 1977]. In this way, the interpreter firstly *analyzes* the abstract syntax tree to produce a semantic representation of this tree (a *function*). This function can be subsequently executed. This approach, called *analyze eval*, is described with more detail by Abelson and Sussman [Abelson 1996], and is applied thoroughout this paper. Interpreters organized in this way are more understandable, because syntax and semantics aspects are clearly separated. In addition, because representations of the semantics are explicitly built, this approach is suitable to support *modularity* (i.e., the possibility to develop auto-contained interpretation modules that can be reused without the need to change them). By doing so, more complex interpreters can be built by assembling pre-existing simpler interpreters, with each one adding a given feature to the final language.

In this paper, we propose a processing model for XML document trees that leads to extensible and modular markup interpreters. We think that these two features are essential for the processing of XML-based markup languages. On one hand, because we are dealing with an extensible markup language, the processing model must lead to the construction of *extensible* interpreters (in the sense discussed by J. K. Ousterhout [Ousterhout 1990]). In this way, the resulting interpreter must be amenable to being enlarged with new commands to deal with new markup structures. On the other hand, and more important, the model must lead to *modular* interpreters. Accordingly, new markup language processors must be obtained from the integration of existing ones without modifying them. We think that modularity in the interpretation framework is a needed counterpart to modularity in

the definition of markup languages, that is present in one of the core XML technologies (XML namespaces [W3C 1999b]).

The paper is structured as follows. Section 2 motivates our approach. Section 3 gives a conceptual description of the processing model proposed in this paper. Section 4 outlines an object-oriented framework that supports this model. Section 5 discusses related work. Finally, section 6 summarizes the conclusions and outlines future work.

## § 2 Extensibility, modularity and the processing of document trees

Consider the following markup structures:

```
<!ENTITY % shape "(Rectangle | Union)">
<!ELEMENT Rectangle EMPTY>
<!ATTLIST Rectangle xo NMTOKEN #REQUIRED
                    yo NMTOKEN #REQUIRED
                     w NMTOKEN #REQUIRED
                     h NMTOKEN #REQUIRED>
<!ELEMENT Union (%shape;)+>
```

These structures allow the description of shapes, made with the union of simpler rectangular shapes. The simplest shape is a rectangle that, in turn, is characterized by the coordinates of its left upper square, and by its width and height. Suppose we want to write a DOM-based program to bind these representations into domain-specific ones, in terms of object-oriented classes like those outlined in the class diagram of Figure 1.

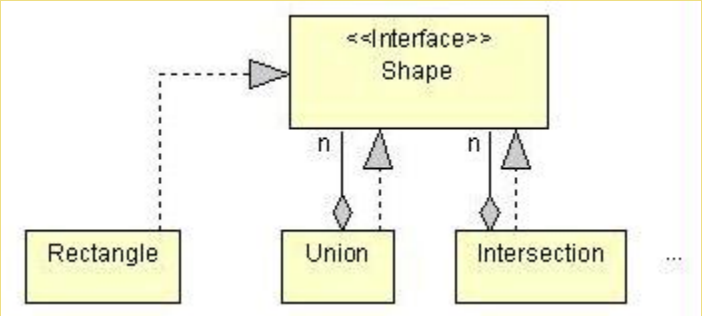


**Figure 1 Classes and interfaces for representing shapes**

A simple organization for this processor is a *functional* one. Here we associate a function with each element type. In addition, we associate dispatching functions with the different content models. The result is outlined in Figure 2. This kind of organization is common both for DOM-based tree processors [Maruyama 1999] and for syntax-based interpreters [Abelson 1996] [Friedman 2001]. Here, the processing of each relevant markup structure is carried out by a function. The contextual information needed for doing the processing is passed as input parameters, while the result of the processing is returned as the function result.

This functional, monolithic, organization presents several shortcomings. One is that the processors obtained are not extensible. To illustrate this, consider an extension to the language with a new operation (e.g., `Intersection`) to produce the final shape:

```
<!ENTITY % shape "(Rectangle | Union | Intersection)">
<!ELEMENT Rectangle EMPTY>
...
<!ELEMENT Intersection (%shape;)+>
```

Here, *the processor must be explicitly modified* to include a new function associated with the new structure (see Figure 3). That way, the functional organization leads to non-extensible processors.

```
(1) Shape process(Document doc)
(2) Shape processShape(Node n)
(3) Shape processRectangle(Node n)
(4) Shape processUnion(Node n)
(5) List processShapes(Node n)
```

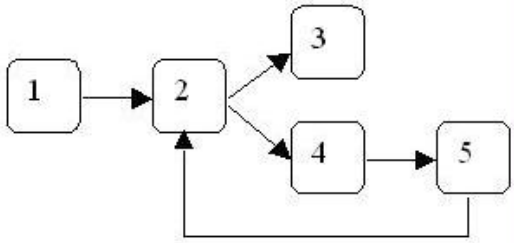


**Figure 2 Functional organization for an XML processor**

Rounded rectangles represent functions.

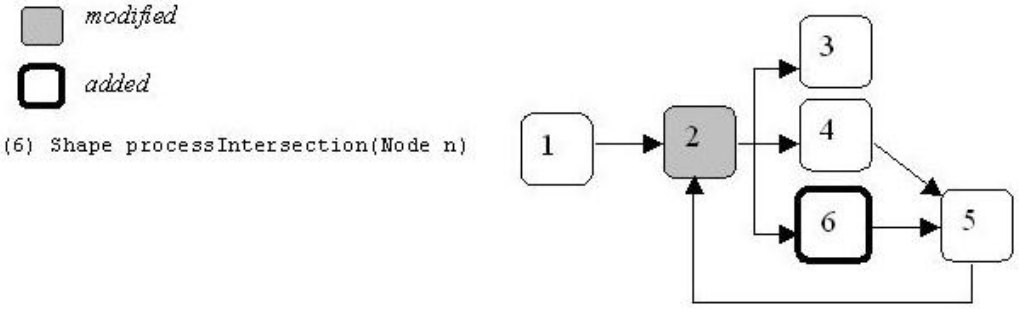


**Figure 3 Addition of a new markup structure**

The inclusion of a new markup structure implies the explicit modification of the functionally-organized processor.

To achieve extensibility, we can use the advantages provided by object-oriented programming. We can encapsulate the functionally-organized processor into a class. Then we can use the implementation inheritance and the dynamic binding to extend the processor for supporting new structures. With each extension, we subclass the processor, override methods it is necessary to change, and add methods to support the new structures. Figure 4 shows application of this strategy to our example.

The problem is that this solution lacks of modularity. For example, consider the language is extended to allow the scaling of the shapes by a factor:

```
<!ENTITY % shape "(Rectangle | Union | Intersection | Scale)">
<!ELEMENT Rectangle EMPTY>
...
<!ELEMENT Scale (%shape;)>
<!ATTLIST Scale f NMTOKEN #REQUIRED>
```

and suppose we do not want to explicitly represent a scaled shape, but we can directly apply the scale factor to the resulting representation. In this way, the representation of:

```
<Scale f="2">
  <Union>
    <Rectangle xo="5" yo="5" w="5" h="4"/>
    <Rectangle xo="10" yo="10" w="1" h="1"/>
  </Union>
</Scale>
```

should be equivalent to the representation of:

```
<Union>
  <Rectangle xo="10" yo="10" w="10" h="8"/>
  <Rectangle xo="20" yo="20" w="2"  h="2"/>
</Union>
```

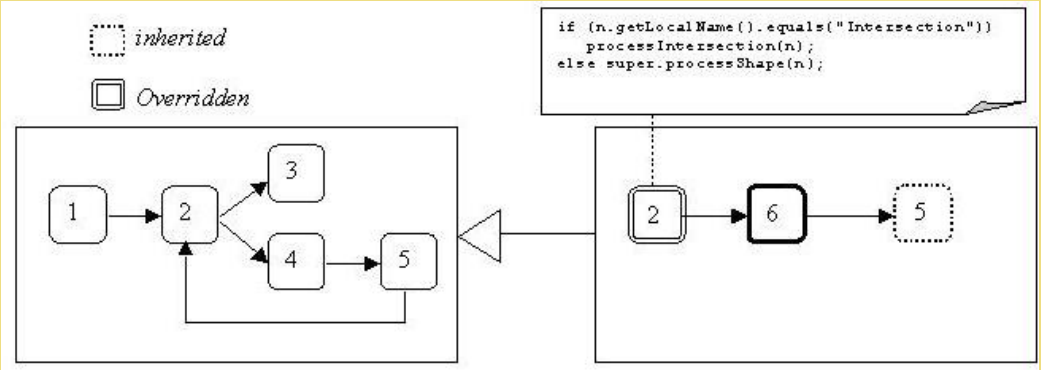


**Figure 4 Object-oriented organization**

An object-oriented organization enables extensibility.

To implement this extension, we need to explicitly pass a scale parameter to the processing methods. This leads us *to overload all the existing methods* — with exception of the root one — ( Figure 5), i.e., we need to rebuild the entire interpreter. Hence, supporting such new structures requires many changes in the processor (i.e., to carry out a *global change*). Since adding a new feature in the supported language can imply this sort of global change, this organization is not modular.

The reason for this lack of modularity is the inclusion of a new input argument in the processing functions, together with the need to propagate this new argument. In the construction of language processors, this kind of contextual information is commonly referred to as *inherited attributes* [Friedman 2001]. A way to avoid breaking the modularity of the processor by the addition of new inherited attributes is to explicitly devise the processor to support this feature (i.e., the addition of new contextual information as required). For instance, this can be done by incorporating an *environment model* in the processor. The problem with this approach is that it compromises the processor with a particular organization (that of the so-called *environment-passing interpreters* [Friedman 2001]), which will be invalid when a different environment model is required.

How can these shortcomings be solved? Previous work realized in the programming language implementation arena suggest possible solutions.

First, syntax-based processors interleave two different aspects that can be easily separated. On one hand, there is the *analysis* of the document tree to extract the relevant information. On the other hand, there is the *semantics* given to the markup structures (i.e., to the element types). By applying an *analyze eval*-like approach, these aspects can be done separately.

Second, the semantic of each element type must be given independently. In addition, we must be able to manage these semantics as first-class objects (e.g., by using closures in a functional language or objects in an object-oriented one). In doing so, a *semantic* approach to the construction of processors is used. Because semantics can be manipulated, it is possible to apply uniform adaptations to them. So, modularity problems can be aborted in a systematic way.

The next section formulates a processing model that systematically contemplates extensibility and modularity in the processing of document trees.

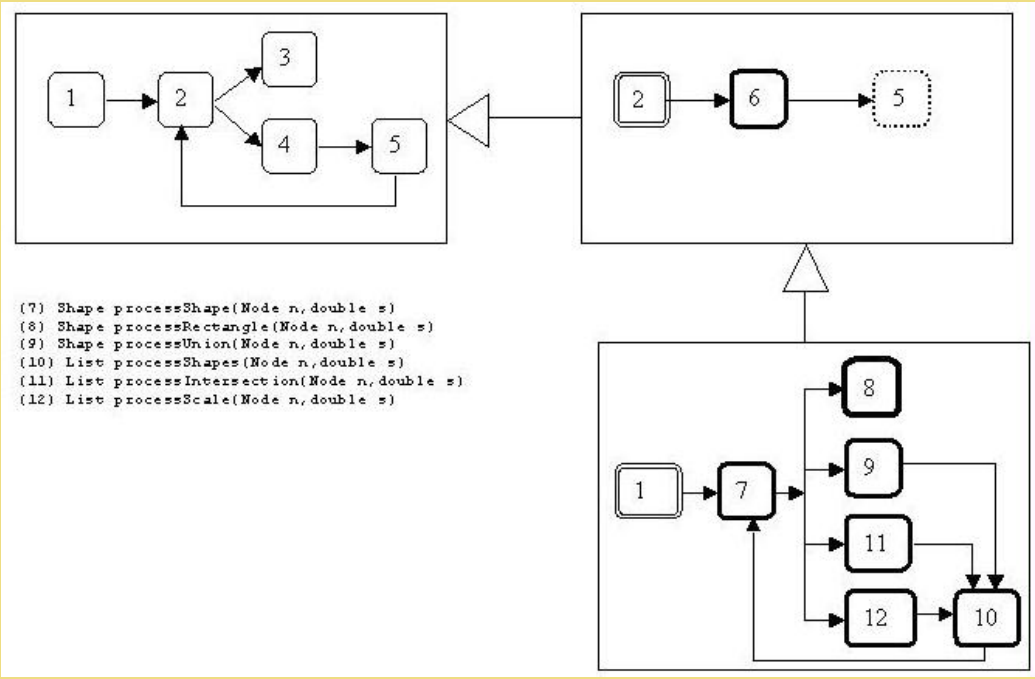


**Figure 5 Consequences of adding a new markup structure**

Introducing a new markup structure can lead, in the object-oriented organization, to the redefinition of the overall interpreter.

## § 3 The processing model

This section describes a processing model for document trees that enables the construction of extensible and modular tree processors. The processing model combines the *analyze eval* approach for the construction of interpreters with some concepts taken from the *syntax-directed translation* processes used in the implementation of programming languages [Aho 1986].

From a conceptual point of view, the model introduces the following stages in a tree's processing:

| | |
|---|---|
| The *dependency hypergraph construction* stage | In this stage, a set of *operational links* is established between the element nodes of the document tree. These links can be one-to-one or one-to-many, and they are explicitly labeled with a *role name*. Indeed, they are represented as labeled-ordered (hyper)arcs in a (hyper)graph made with the element nodes of the document tree. The resulting hypergraph is named a *dependency hypergraph*. |
| The *dependency hypergraph decoration* stage | In this stage, a *composition function* and a set of parameters are assigned with each node in the dependency hypergraph. The composition function is used to obtain the *semantic value* associated with a node from the parameters and the semantic values of those nodes linked to it. |
| The *semantic composition* stage | In this stage, an order suitable for the application of the composition functions is calculated. Then, these composition functions are applied to compose the semantic values associated with each node. The semantic value associated with the document element represents the *tree semantic value*, and, so, the overall tree semantics. |
| The *evaluation* stage | The tree semantic value can be, in turn, another function that must be evaluated on the appropriate parameters to obtain the final result. |

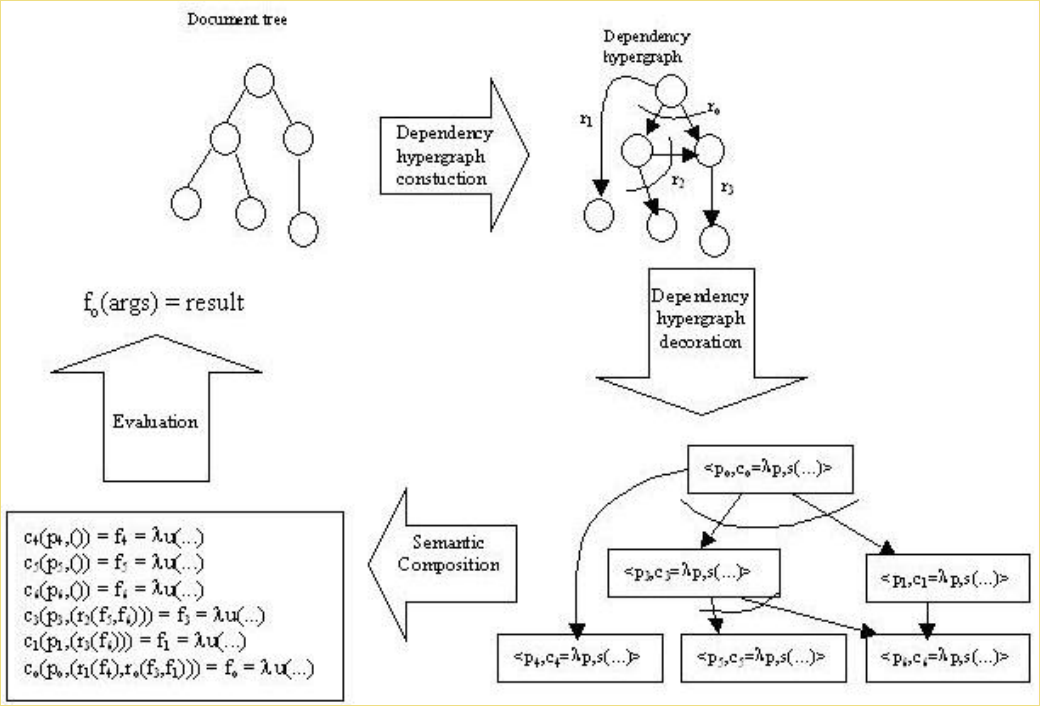


**Figure 6 Sketch of the processing model**

Figure 6 graphically sketches the process. Note that the first three stages correspond with a particular organization of the tree analysis phase in an *analyze eval* approach. This organization explicitly enables us to separate those tasks related to tree manipulation (i.e., linking, parameter extraction) from the semantic-oriented ones (i.e., composition of the semantic assigned to each element type). Note also that the three first stages can be related with a syntax-directed translation process.

## 3.1 *The dependency hypergraph construction stage*

The processing of a document tree begins by making the operational links between the element nodes of the tree explicit. An operational link has an element node as *source* and an ordered sequence of element nodes as *target*. In addition, the link has an associated role name. The processing of element nodes will be specified in terms of the target nodes linked to them. Note that the operational links do not prescribe how the linked nodes must be actually processed. This processing will be added in later stages. Links only make explicit the relevant nodes for the processing of a given one.

Operational links require neither maintaining the structure of the document tree, nor any locality bound. Operational links can be established between remote nodes, separated with arbitrarily large distances in the document tree. In effect, structural relationships (such as those established by the standard XML `ID-IDREF` mechanisms) can be used to establish appropriated operational links.

The operational linking induces a graph structure on the element nodes of the document tree. The nodes of this graph are the element nodes. The arcs are the operational links established between these nodes. Because arcs are, in turn, structured, they actually can be considered as *hyperarcs* (in the sense of the AND/OR graphs described, for instance, by Nilsson [Nilsson 1980]). Consequently, the resulting graph is a *hypergraph*, which we will call *dependency hypergraph*.

Note that there are some similarities between a dependency hypergraph and a *dependency graph* in a syntax-directed translation framework. Indeed, both structures constrain the order of computations over the tree. But there are also some differences. The most relevant is that dependency hypergraphs

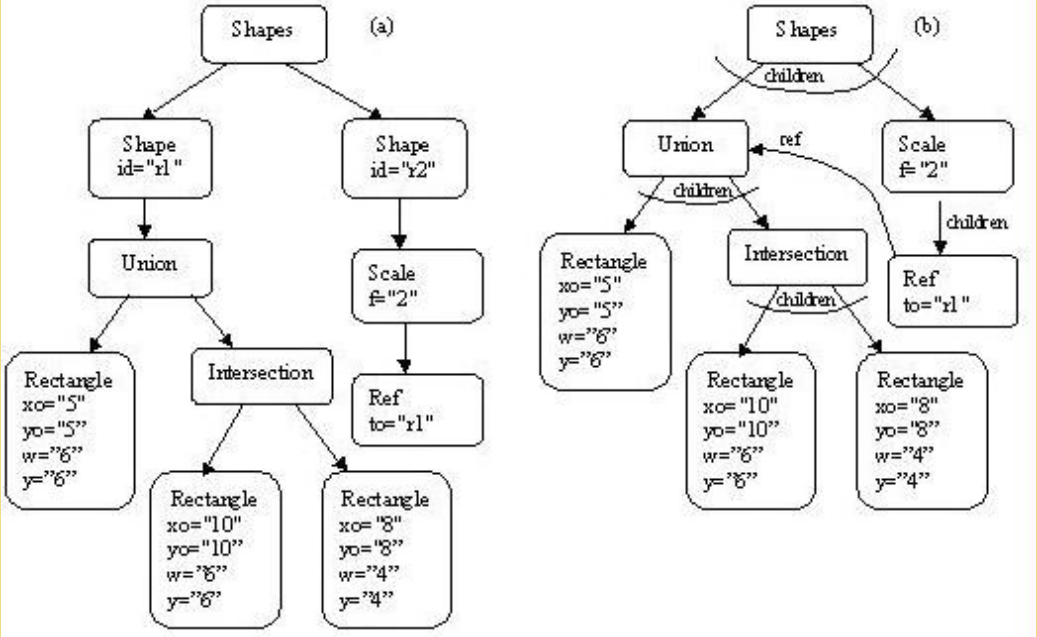


**Figure 7 A document tree (a) and a dependency hypergraph associated with it (b)**

are given *prior* to the establishment of the computations, using the structural information of the document tree. In the case of dependency graphs defined on "attributed" syntax trees, dependencies between attributes are automatically induced from a computation schema given *a priori* (the syntax-directed translation schema). This is consistent with the formulation of a descriptive markup language, where the main goal is to describe the structure of some contents independently of their processing. So, in a markup language, structure is given *a priori*, and operational readings are added *a posteriori*.

To illustrate these ideas, suppose the previous section's language for shape descriptions is augmented with the following declarations:

```
<!ENTITY % shape "(Rectangle | Union | Intersection | Scale |Ref)">
<!ELEMENT Shapes (Shape)+>
<!ELEMENT Shape (%shape;)>
<!ATTLIST Shape id IDREF #REQUIRED>
<!ELEMENT Ref EMPTY>
<!ATTLIST Ref to IDREF #REQUIRED>
```

In this case, by using elements of type `Shape` we can uniquely identify a shape. In addition, shape definitions can be referenced using elements of type `Ref`. Consequently, it seems reasonable to establish operational links between these elements and the referenced shapes. The other operational links are given by the parent/child relationships in the tree. Figure 7 illustrates a document tree conforming this language and the associated dependency hypergraph. Note that elements of type `Shape` are excluded from this graph.

### 3.2 *The dependency hypergraph decoration stage*

Once the dependency hypergraph is available, a set of *parameters* and a *composition function* are assigned to each node in this hypergraph. This leads to a so-called *decoration* of the dependency hypergraph. This decoration is a structurally isomorphic hypergraph, where each node has been replaced with its assignment. From here, the document tree is no longer available, so all the relevant information must have been represented in terms of values assigned to the parameters. However, the

composition function is supposed to work on these parameters and on the semantics values assigned to the linked nodes, in order to obtain a new semantic function.

Composition functions are the primary pieces used in the processing specification. They are independent of the explicit tree manipulation, which isolates them from changes in the superficial structure of the markup language. All the required information is exposed by the parameters and by the semantic values for the linked nodes. Information about these semantic values must be given in a way such that the structure of the operational links is preserved. This allows the composition functions to reference the semantic roles associated with each operational link. Figure 8 illustrates a decoration for the hypergraph in Figure 7. There, *CRect*,*CUnion*, etc., refer to the composition functions assigned to the element nodes (definitions omitted).

### 3.3 *The semantic composition stage*

Once the dependency hypergraph has been appropriately decorated, the composition functions in the nodes must be applied to obtain the tree semantic value. We refer to this process as *semantic composition*. To do semantic composition, an admissible application order for the composition functions must be selected. More precisely, an application order is *admissible* if the semantic value for a node is presented when required. Figure 9 shows two different admissible application orders for the example in Figure 8.

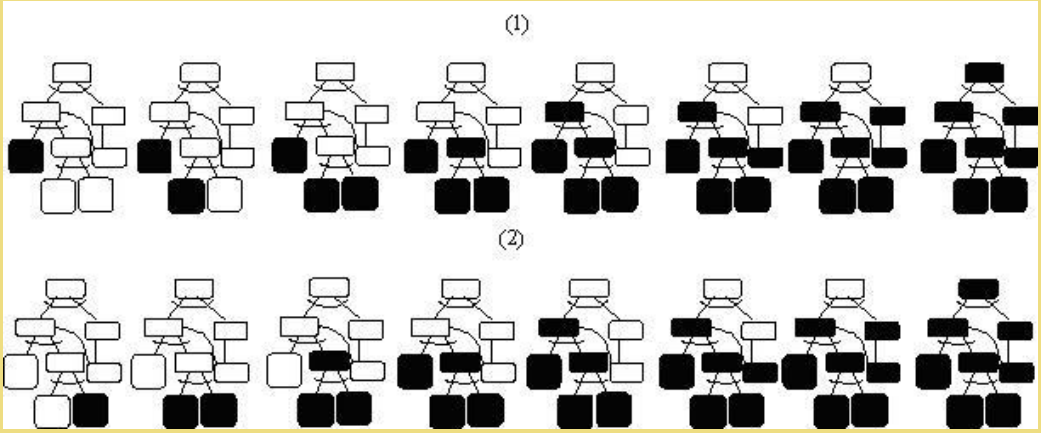


**Figure 9 Two admissible application orders for the example in Figure 8**

The semantic composition stage is similar to evaluating computations in the syntax-directed translation model. When the evaluation order is decided during translation time (i.e., for each particular syntax tree), the attributes of the dependency graph are usually topologically sorted (i.e., an attribute must be preceded by all the attributes used in its computation). This method works for acyclic dependency graphs (definitions yielding cyclic graphs are usually considered to be ill-formed), and can be applied to decide the application order in our semantic composition stage. In this case, because of the order in the operational links, application should be in inverted topological order.

Thus, due to the composition functions' black box nature, an application built according to a topological order, although admissible, is not necessarily the most appropriate. Indeed, it could lead to an *eager* evaluation order for the composition functions. In this case, a *lazy* evaluation strategy is more appropriate. Here, semantic values can be computed under demand. In addition, when a semantic value is computed, it can be cached in order to avoid its recomputation. This is the composition strategy followed in the framework outlined in section 4.

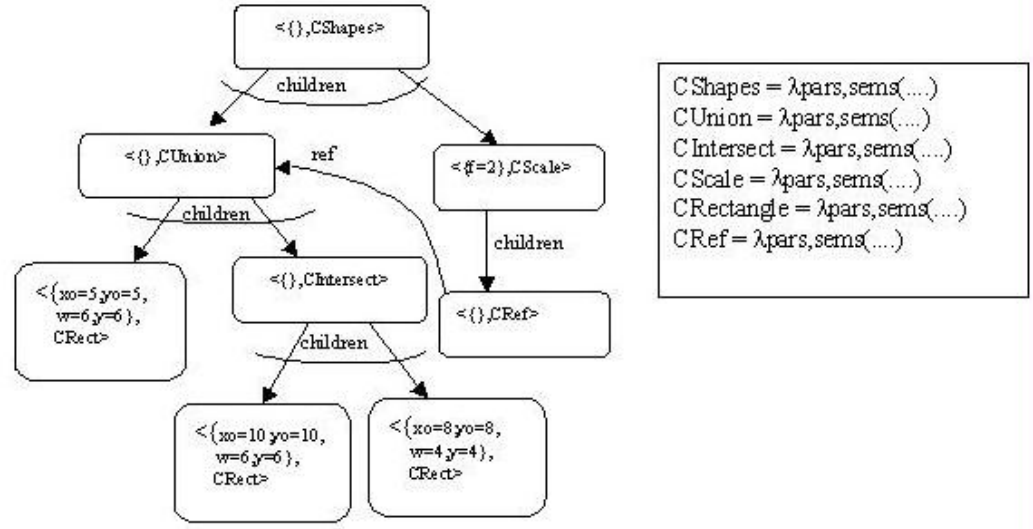


**Figure 8 Illustration of a decoration for the dependency hypergraph in Figure 7**

## 3.4 *The evaluation stage*

The semantic composition stage yields a semantic value associated with the document element. In many cases, this value can be a function that must be evaluated to obtain the processing result. This evaluation stage usually requires preparing the initial parameters for the evaluation, performing the invocation, and re-collecting the result.

## 3.5 *Extensibility and modularity*

The construction of a processor for a markup language using the model described here would define:

i. a procedure `L` to obtain the operational links for the element nodes,

ii. a procedure `P` to assign a set of parameters with each element node in the dependency hypergraph, and

iii. a procedure `C` to assign a composition function to each one of these elements.

These procedures can be independently specified for each element type, and subsequently be combined with a general *processing engine* that uses them consistently with the process model described here. We will refer to triples of the form `<Le,Pe,Ce>` associated with an element type `e` as *markup interpreters* for `e`. In addition to the markup interpreters, the processing engine requires an evaluation procedure *Ev* to obtain the final result. The process is as follows:

- During the dependency hypergraph construction stage, the engine starts by visiting the document element. When the engine visits an element node of type `e`, it recovers the procedure `Le` from the markup interpreter for `e` and uses it to obtain the operational links associated with the node. Then, the engine proceeds to visit the target element nodes of these operational links.
- During the dependency hypergraph decoration stage, the engine assigns the parameters and composition functions with the element nodes in this hypergraph. Here, the engine uses `Pe` and `Ce` for each element node of type `e`.

- During the semantic composition stage, the engine calculates an admissible evaluation order on the hypergraph and proceeds to apply the composition functions in this order.
- Finally, the engine delegates in the evaluation procedure *Ev* the evaluation of the tree semantic value.

Obviously, such behavior for a generic engine is only conceptual. An implementation can be free to carry out an observationally equivalent, yet more efficient, processing.

A particular processor can be now considered as the addition of a set of markup interpreters `<Leo,Peo,Ceo>`...`<Len,Pen,Cen>` to a generic engine *E*. Because new interpreters can be added, the processor can be extended to deal with other markup structures. This leads to *extensible processors*.

For instance, for the shape markup language, we can begin by providing two interpreters `IRectangle= <LRectangle,PRectangle,CRectangle>` and `IUnion=<LUnion,PUnion,CUnion>` to obtain a processor observationally equivalent to that sketched in Figure 2. Then, we can extend this processor with a new interpreter `IIntersection = <LIntersection,PIntersection,CIntersection>` to obtain a processor equivalent to those sketched in Figures 3 and 4.

What happens if the new interpreter is not compatible with the existing ones? The idea is to apply adaptors to the existing interpreters, the new incorporated interpreter, or both. An *adaptor* is a procedure for generating interpreters from interpreters. In addition, because interpreters are structured in different procedures, adaptors can be structured accordingly into procedures for adapting those procedures.

For instance, when the processor induced by the interpreters `{ IRectangle, IUnion, IIntersection }` must be extended with `IScale = < LScale , PScale , CScale >`, some adaptation is required, because the scale factor must now be propagated by the composition functions of `IUnion` and `IIntersection`, and it must be used by `IRectangle`. In this way, if we suppose `CScale` is defined as:

```
CScale(pars,sems) = λscale(
      1. get the scale factor f from pars
      2. get the semantic value s associated with the child element
from sem
      3. apply s to f*scale
 )
```

(i.e., as a function that, for each parameter set and for each semantic assignment, gives a function that accepts a scale parameter and behaves as described), and `CUnion` is defined as:

```
CUnion(pars,sems) = (
      1. get from sems the list of semantics ls for the children
elements.
      2. make a Union with ls.
)
```

we need to transform this function with:

```
C'Union(pars,sems) = λscale (
      1. get from sems the list of semantics ls for the children
elements.
      2. make a new list ls’ evaluating each element in ls with scale
      3. make a Union with ls’.
)
```

This can be done by applying an adaptor in the form:

```
AddScale(c) =
```

```
   λ<pars,sems> (
     λscale (
          1. get from sems the list ls of semantics for the children
elements.
          2. make a new list ls' with the results of evaluating each
element
             in ls with scale.
          3. make a new sems' equal to sems, but with ls' instead of
ls
             as the semantic values for the children elements.
          4. Apply c to <pars,sems'>
          )
 )
```

Thus, `AddScale(CUnion)` = `C'Union`. In addition, this adaptor could also be applied to adapt `CIntersection`. In this way, the same adaptor can be used to adapt semantically-related interpreters in an uniform way.

## § 4 Implementing the processing framework

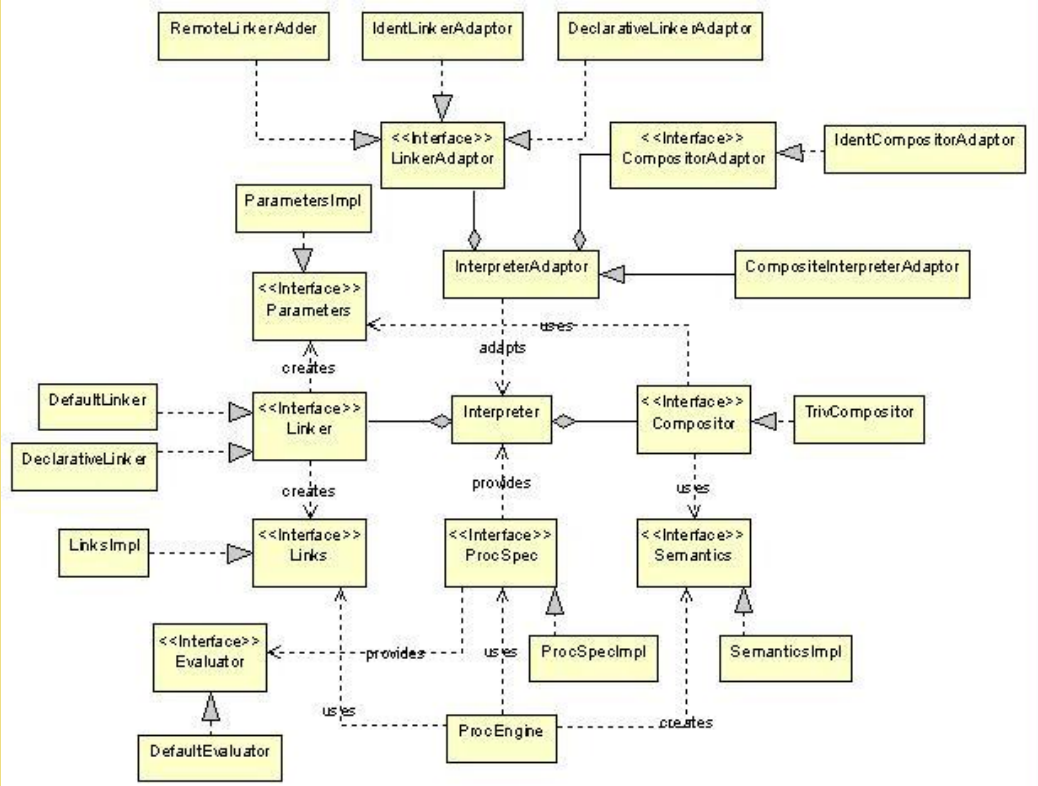


**Figure 10 Class diagram of the main classes and interfaces involved in the framework supporting our processing model**

We have built an experimental object-oriented Java framework that implements the processing model described in this paper. The class diagram in Figure 10 outlines the main classes and interfaces included in this framework. The typical use of this framework for building a document tree processor is as follows:

- To provide suitable implementations of the *Linker* and the *Compositor* interface for the element types that require them. The framework includes two implementations of the *Linker* interface. *DefaultLinker* creates a parameter for each attribute present in the element node, and a child link for the children elements. *DeclarativeLinker* allows a linker to be built from an XML description. In addition, a trivial implementation of *Compositor* (*TrivCompositor*) is included. This implementation can be instantiated with a semantic role name and an index. The instance will use the index on the semantic value associated with the role.
- To provide implementations for the required adaptors. This is done by implementing the *LinkerAdaptor* (for adapting linkers) and *CompositorAdaptor* (for adapting compositors) interfaces. The framework includes three LinkerAdaptors. *IdentLinkerAdaptor* is a trivial identity adaptor (i.e., it returns the received linker unchanged). *RemoteLinkerAdder* adds a new linker to a remote node using an attribute name and a value as key, and *DeclarativeAdaper* allows use of an XML description for defining new links and parameters or overriding existing ones. A trivial identity implementation of *CompositorAdaptor* is also provided (*IdentCompositorAdaptor*).
- To provide suitable implementation of the *Evaluator* interface. This will be used to carry out the evaluation stage.
- To provide suitable implementation of the interface *ProcSpec* (process specification). This will be used to obtain the interpreter associated with each element node in the dependency hypergraph, and the evaluator to be used. This implementation can be also built from an XML description using the *ProcSpecImpl* implementation provided by the framework.
- To instantiate the *ProcEngine* class with a suitable process specification, and to apply this instance to process DOM representations of the document trees. *ProcEngine* implements the processing model described in this paper using a lazy strategy.

### 4.1 *Using XML in the framework instantiation*

The existence of XML languages for describing either linkers and process specifications eases the use of this framework. To describe linkers we use the following markup language:

```
<!ELEMENT LinkSpec (Vars?,Parameters?,Links?)>
<!ELEMENT Vars (Var)+>
<!ELEMENT Var (#PCDATA)>
<!ATTLIST Var name NMTOKEN #REQUIRED>
<!ELEMENT Parameters (Parameter)+>
<!ELEMENT Parameter (#PCDATA)>
<!ATTLIST Parameter name NMTOKEN #REQUIRED>
<!ELEMENT Links (Link)+>
<!ELEMENT Link (#PCDATA)>
<!ATTLIST Link name NMTOKEN #REQUIRED>
```

Here, the contents of either `Vars`, `Parameter`, and `Link` must be XPath expressions [W3C 1999a]. These expressions are evaluated taking the element node associated with the linker as the context node. Each `Var` element allows the definition of a variable that can be used in subsequent XPath expressions. When the expression associated with a parameter is evaluated, the content of the result is taken as the value of the parameter. For operational links, the targets are the element nodes in the result. Note that this language can be used to configure either linkers or linker adaptors. For instance, the following XML fragment describes a linker for the `Ref` element in the shapes markup language:

```
<LinkSpec>
  <Vars>
    <Var name="shapeId">@ref</Var>
  </Vars>
  <Links>
    <Link name="ref">//Shape[@id = $shapeId]/*</Link>
```

```
  </Links>
</LinkSpec>
```

The description of process specifications is performed in terms of the following language:

```
<!ELEMENT ProcSpec (Adaptors?,Interpreters,Namespaces)>
<!ATTLIST ProcSpec evaluator CDATA #IMPLIED>
<!ELEMENT Adaptors (Adaptor|Composition)+>
<!ELEMENT Adaptor ((LinkerAdaptor | LinkSpec)?,CompositorAdaptor?)>
<!ATTLIST Adaptor id ID #REQUIRED>
<!ELEMENT LinkerAdaptor EMPTY>
<!ATTLIST LinkerAdaptor ref CDATA #REQUIRED>
<!ELEMENT CompositorAdaptor EMPTY>
<!ATTLIST CompositorAdaptor ref CDATA #REQUIRED>
<!ELEMENT Composition EMPTY>
<!ATTLIST Composition id ID #REQUIRED
                      a1 IDREF #REQUIRED
                      a2 IDREF #REQUIRED>
<!ELEMENT Interpreters (Interpreter)+>
<!ELEMENT Interpreter ((Linker |LinkSpec)?,Compositor)>
<!ATTLIST Interpreter id ID #REQUIRED
                       adaptor IDREF #IMPLIED>
<!ELEMENT Linker EMPTY>
<!ATTLIST Linker ref CDATA #IMPLIED>
<!ELEMENT Compositor EMPTY>
<!ATTLIST Compositor ref CDATA #REQUIRED>
<!ELEMENT Namespaces (Namespace)+>
<!ELEMENT Namespace (Bind)+>
<!ATTLIST Namespace ns CDATA #REQUIRED
                    default IDREF #IMPLIED>
<!ELEMENT Bind EMPTY>
<!ATTLIST Bind tag CDATA #REQUIRED
               interpreter IDREF #REQUIRED>
```

This language allows declaration of the adaptors and interpreters to be used, and the association of interpreters with element types.

Each required adaptor is introduced using an `Adaptor` element. The adaptor can introduce a linker adaptor and a compositor adaptor. If one of these two adaptors is not present, the corresponding identity adaptor is taken instead. For linker adaptors, an explicit description can be given in terms of the linker description language (here, a *DeclarativeLinkerAdaptor* is used); alternatively, the Java class implementing the adaptor to be used can be referenced (using the `ref` attribute of the `LinkerAdaptor` element). For compositor adaptors, the name of the Java class implementing it must be explicitly referenced. Finally, using a `Composition` element, it is possible to create an interpreter adaptor composing other two adaptors. For the shape markup language, a possible description of the adaptors to be used is:

```
<Adaptors>
    <Adaptor id="addScaleA">
         <CompositorAdaptor ref="AddScaleAdaptor"/>
    </Adaptor>
    <Adaptor id="rectangleA">
         <CompositorAdaptor ref="RectangleAdaptor"/>
    </Adaptor>
  </Adaptors>
```

Each interpreter to be used is introduced by an `Interpreter` element. Here, an adaptor to be applied to the interpreter can be referenced (using the `adaptor` attribute). As with linker adaptors, linkers can be described either in terms of the linker description language, or by referencing an external Java implementation. If a linker is not specified, the default linker is used instead. In turn, the `Compositor` element allows reference (using the `ref` attribute) to the Java class implementing the compositor. For instance, the declaration for the interpreter to be associated with the `Ref` element type can be:

```
<Interpreter id="iref" adaptor="addScaleA">
  <LinkSpec>
    <Vars>
      <Var name="shapeId">@ref</Var>
    </Vars>
    <Links>
      <Link name="ref">//Shape[@id = $shapeId]/*</Link>
    </Links>
  </LinkSpec>
   <Compositor ref="CRef"/>
</Interpreter>
```

Finally, the association between interpreters and element types is done for each namespace, using a `Namespace` element. The value of the `ns` attribute must be the namespace URI, or "none" (to refer element types without an associated namespace). Each association is described in terms of a `Bind` element. For instance, for the shapes markup language we have:

```
<Namespace ns="none">
  <Bind tag="Rectangle" interpreter="irectangle"/>
  <Bind tag="Union" interpreter="iunion"/>
  <Bind tag="Intersection" interpreter="iintersection"/>
  <Bind tag="Scale" interpreter="iscale"/>
  <Bind tag="Shapes" interpreter="ishapes"/>
  <Bind tag="Ref" interpreter="iref"/>
</Namespace>
```

The interpretation framework itself is applied to process these two description languages. The resulting classes, together with other minor ones, are not present in the class diagram shown in Figure 10.

## § 5 Related work

The *analyze eval* approach to the construction of interpreters has been described by Abelson and Sussman [Abelson 1996], where its invention was attributed to the teams of Rees & Adams, and Feeley & Lapalme [Rees 1982] [Feeley 1987]. In "Semantic Lego"[Espinosa 1995], the evaluation part of this approach has been denominated as *semantic-based*.

The construction of modular interpreters is a hot topic in the functional programming community. There, the dominant approach is based on *monads* and *monads transformers* [Steele 1994] [Espinosa 1995] [Liang 1995]. According to this approach, interpreters are written in the so-called *monadic style*. Here, two functions (*bind* and *unit*) are used to enable an explicit representation of the control flow in the interpreter. This two functions, together with a polymorphic type used to represent computations, constitutes a monad (see "Monads and Functional Programming"[Wadler 1993] for more details). When the interpreter is programmed, the monad is left as a parameter. So, by defining an appropriate monad, the interpreter can be tailored to a different context. In these frameworks, monads are usually defined using monad transformers. A monad transformer defines a result monad in terms of a source monad. In addition, the transformer is equipped with a mapping for lifting operations to the newly defined monad. Our approach is, to some extent, similar to the monadic one. However, while monadic interpreters are parametric in a monad, our interpreters are parametric in an evaluator adaptor. Because this behavior is explicitly encoded into the framework, we explicitly avoid compromising ourselves with a particular coding style (e.g., the verbose monadic style). In addition, we avoid the use of second order types that arises when general monads are managed. Nevertheless, our adaptors play the role of monad transformers, although we omit the capability of these transformers to lift operations between monads. In the monadic approach, this feature allows use of coarser grain operations as primitives in the interpreter construction (e.g., *fetch* and *update* to operate on an store). But, because our framework explicitly hides control of the internal implementation of the compositors, this feature is dropped from adaptors. If required, this feature should be simulated by applying transformations to the document trees, before their processing, in order to generate structures with the appropriate granularity.

Our assembler adaptors are similar to the so-called *mixins* in the object-oriented paradigm [Bracha 1990]. The informal description of a mixin is a class that is parametric in its superclass (i.e., which *super* variable can be dynamically changed). "A Mixin-Based Semantic-Based Approach to Reusing Domain-Specific Programming Languages" [Duggan 2000] describes another mixin-based Java framework for the development of modular interpreters. There, mixins represents interpreters that can be assembled together in a mixin chain. Each interpreter maintains its own state and encapsulates inner classes for the construction of abstract syntax trees made from expressions. These expressions can be evaluated into computations (the equivalent to our semantic values). The chaining of mixins leads to an associated chaining of states. To recover the appropriate state in each interpretation step, the framework enforces the writing of computations using a monadic style. However, the framework does not includes an equivalent to monad transformers. Instead, the monadic operations must be explicitly defined in each mixin to recover the appropriate state from the state chain.

There are several works describing the application of syntax-directed translation techniques to markup languages. Kuikka and Pentonnen have described a method to perform document transformation by automatically deriving a *translation schema* from a source to a result document grammar [Kuikka 1993]. In "SIMON: A Grammar-based Transformation System for Structured Documents" [Feng 1993], document transformations are specified using *higher order attribute grammars* [Vogt 1989]. F. Neven [Neven 1999] has shown how to apply the attribute grammar formalism to extended BNF grammars (and, hence, to document grammars) oriented to be used as document query mechanisms. In "Adding Semantics to XML" [Psaila 1999], a method to associate semantic functions with documents (following the attribute grammar paradigm) is described. While these approaches perceive the syntax-directed translation engine as the *processor*, our approach conceives this engine as a way to *generate the actual processor* by assembling smaller processors, according to the *analyze eval* spirit.

Note that the *analyze eval* approach adopted here can also be identified in the processing model underlying publication approaches such as XSL [W3C 2001] (and its predecessor DSSSL [ISO 1996]). In effect, in XSL, documents are transformed into a markup language of formatting objects. The resulting specification must be subsequently evaluated to generate the presentation. The first phase corresponds with an analysis phase, while the second one is a particular type of evaluation phase.

The work described here relies on our previous work on the DTC [structured Documents, document Transformations and software Components] approach [Sierra 2000a] [Sierra 2000b] [Sierra 2001]. DTC is an approach to the development of applications that are described using domain-specific languages [van Deursen 2000]. We use XML to define domain-specific languages describing the contents managed by the application. Then, we transform these contents to languages supported by pre-existing software components. Hence, software components are interpreters for markup languages that must be combined to obtain more complex interpreters able to give operational support to the desired application domain. We have applied this approach both to the domain of route searching in transport networks [Sierra 2000a] and the domain of educational hypermedia applications [Navarro 2000]. Work described here substantially improves the previously reported research. In other studies [Sierra 2000a] [Sierra 2000b], we adopted a syntactic-based approach and focused on component composition instead of language composition. In this way, components might be reorganized for different applications in the same application domain. Later, we adopted a semantic-based approach and focused on the composition of languages [Sierra 2001]. Components were automatically assembled by interpreting documents trees. Components carried out both the analysis and the evaluation phase. In addition, we did not include any special adaptation mechanism in our framework. Adoption of the *analyze eval* approach described in this paper raises the true linguistic nature of DTC and alleviates many of its previously encountered shortcomings.

## § 6 Conclusions and future work

Extensibility and modularity are two key factors in the successful operationalization of XML applications. Just as XML applications can combine pre-existing vocabularies from different namespaces, the processors for the resulting XML applications should naturally arise from combining the processors associated with the pre-existing aplications. This would require extensibility also on the operational level. Regardless, extensibility alone is not sufficient. In effect, modularity is also required in the processors to allow new extensions to be added without needing to modify previous ones.

As the definition of modular language processors is not straightforward, we present a processing model that encourages the use of these kinds of processors. This model combines ideas from the construction of modular, semantic-based, interpreters, with syntax-directed translation techniques. Because, according to this model, the structure of the resulting processors is explicitly represented (in terms of composition functions associated with element nodes), extensibility and modularity is made possible. In effect, we can build processors by extending a generic engine with new markup interpreters associated with element types, and we can use adaptors as required to integrate these interpreters.

We also show how the processing model can be built using object-oriented technologies. We have built an experimental object-oriented framework to support our model. The use of high-level descriptions (given by XML documents) eases the instantiation of this framework, so we can focus on the essential aspects of the processors' development for document trees (i.e., linkers, compositors, and adaptors), instead of focusing on lower-level details concerning the internal work of the framework.

The next steps in our work will continue experimentation with our processing model and the associated application framework in order to refine them, especially the aspects concerning adaptation. For this purpose, we want to develop a library of general purpose adaptors, together with mechanisms to ease their definition and application.

## Acknowledgements

This work has been partially funded by CICYT [the Spanish Commission of Science and Technology] through the projects TIC2000-0737-C03-01 and TIC2001-1462.

Also, we would like to thank Tonya R. Gaylord (of Mulberry Technologies, Inc.) for careful editing of this paper.

## Bibliography

**[Abelson 1996]** Abelson, H., Sussman, G. J.: "Structure and Interpretation of Computers Programs. Second Edition". Mc Graw Hill. 1996

**[Aho 1986]** Aho, A., Sethi, R., Ullman, J. D.: "Compilers: Principles, Techniques and Tools". Adisson-Wesley. 1986

**[Bracha 1990]** Bracha, G.: "Mixin-based Inheritance".ACM SIGPLAN Notices. 25(10). 1990

**[Bradley 2000]** Bradley, N.: "The XML Companion. Second Edition". Addisson-Wesley. 2000

**[Duggan 2000]** Duggan, D.: "A Mixin-Based Semantic-Based Approach to Reusing Domain-Specific Programming Languages". 2000

**[Espinosa 1995]** Espinosa, D. A.: "Semantic Lego". PhD. Dissertation. Columbia University. 1995

**[Feeley 1987]** Feeley, M., Lapalme, G.: "Using Clousures for Code Generation".Journal of Computer Languages 12(1). 1987

**[Feng 1993]** Feng, A.,Wakayama, A.: "SIMON: A Grammar-based Transformation System for Structured Documents".Electronic Publishing. 6(4). 1993

**[Friedman 2001]** Friedman, D. P., Wand, M., Haynes, C. T.: "Essentials of Programming Languages (Second Edition)". MIT Press/McGraw-Hill. 2001

**[Goldfard 1990]** Goldfard, C. F.: "The SGML Handbook". Oxford University Press. 1990

**[ISO 1986]** ISO. "Standard Generalized Markup Language (SGML). ISO/IEC IS 8879". International Standards Organization. 1986

**[ISO 1996]** ISO. "Document Style Semantics and Specification Language (ISO/IEC 10179)". International Standards Organization. 1996

**[Kuikka 1993]** Kuikka, E., Pentonnen, M.: "Transformation of Structured Documents with the Use of Grammars".Electronic Publishing. 6(4). 1993

**[Liang 1995]** Liang, S., Hudak, P., Jones, M.: "Monad Transformers and Modular Interpreters". 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. 1995

**[Maruyama 1999]** Maruyama, H., Tamura, K., Uramoto, N.: "XML and Java: Developing Web Applications". Addison-Wesley. 1999

**[Navarro 2000]** Navarro, A., Sierra, J. L., Fernández-Manjón, B., Fernández-Valmayor, A.: "XML-Based Integration of Hypermedia Desing and Component-Based Techniques in the Production of Educational Applications". In "Ortega, M.,Bravo, J. (eds).Computers and Education in the 21st Century". Kluwer. 2000.

**[Neven 1999]** Neven, F.: "Extensions of Attribute Grammars for Structured Document Queries". 7th International Workshop on Database Programming Languages. Kinloch Rannoch - Scotland. December 1 - 3. 1999

**[Nilsson 1980]** Nilsson, N. J.: "Principles of Artificial Intelligence". Tioga Publishing Company, Palo Alto, CA. 1980

**[Ousterhout 1990]** Ousterhout, J. K. "TCL: An Embeddable Command Language". Proceedings of the USENIX Association Winter Conference. 1990

**[Psaila 1999]** Psaila, G., Crespi-Reghizzi, S.: "Adding Semantics to XML". Second Workshop on Attribute Grammars and their Applications. WAGA'99. Amsterdam. The Netherlands. March 26. 1999

**[Rees 1982]** Rees, J., Adams, N. I. I.: "T: A dialect or LISP or, lambda: The Ultimate Software Tool". Conference Record of the 1982 ACM Symposium on LISP and Functional Programming. 1982.

**[Sierra 2000a]** Sierra, J. L., Fernández-Manjón, B., Fernández-Valmayor, A., Navarro, A.: "Integration of Markup Languages, Document Transformations and Software Components in the Development of Applications: the DTC Approach". International Conference on Software ICS 2000. 16th IFIP World Computer Congress. Beijing - China. August 21-25. 2000.

**[Sierra 2000b]** Sierra, J. L., Fernández-Manjón, B., Fernández-Valmayor, A., Navarro, A.: "Developing Applications with XML Documents, Document Transformations and Software Components". 10th International Conference on Computing and Information ICCI-2000. Kuwait. November 18-21. 2000.

**[Sierra 2001]** Sierra, J. L., Fernández-Valmayor, A., Fernández-Manjón, B., Navarro, A.: "Operationalizing Application Descriptions with DTC: Building Applications with Generalized Markup Technologies". 13th International Conference on Software Engineering & Knowledge Engineering SEKE'01. Buenos Aires. Argentina. June 13-15. 2001.

**[Steele 1994]** Steele, G.: "Building Interpreters by Composing Monads". 21st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. Portland, Oregon. Janury 17-21. 1994

**[Stoy 1977]** Stoy, J. E.: "Denotational Semantics: The Scott-Strachey Approach to Programming Language Theory". The MIT Press. 1977

**[van Deursen 2000]** van Deursen, A., Klint, P.,Visser, J.: "Domain-Specific Languages: An Annotated Bibliography".ACM SIGPLAN Notices. 35(6). 2000.

**[Vogt 1989]** Vogt, H. H., Swierstra, S. D., Kuiper, M. F. "Higher-Order Attribute Grammars". Proceedings of the ACM SIGPLAN'89 Conference on Programming Language Design and Implementation. 1989

**[W3C 1999a]** W3C. "XML Path Language (XPath) Version 1.0 (W3C Recommendation)". World Wide Web Consortium. 1999

**[W3C 1999b]** W3C. "Namespaces in XML (W3C Recommendation)". World Wide Web Consortium. 1999

**[W3C 2000a]** W3C. "Document Object Model Level 2 (W3C Recommendation)". World Wide Web Consortium. 2000a

**[W3C 2000b]** W3C. "Extensible Markup Language (XML) Second Edition (W3C Recommendation)". World Wide Web Consortium. 2000b

**[W3C 2001]** W3C. "Extensible Stylesheet Language (XSL) (W3C Recommendation)". World Wide Web Consortium. 2001

**[Wadler 1993]** Wadler, P. "Monads and Functional Programming". Proceedings of the 1992 Marktoberdorf International Summer School in Program Design Calculi. Springer Verlarg. 1993

## The Authors

**José Luis Sierra**
*Dpto. Sistemas Informáticos y Programación, Universidad Complutense*
Madrid
Spain
jlsierra@sip.ucm.es

Mr. José Luis Sierra is a Computer Science assistant professor at UCM. He is preparing a Ph.D. dissertation on the use of domain-specific markup languages in the development of applications.

**Baltasar Fernández-Manjón**
*Dpto. Sistemas Informáticos y Programación, Universidad Complutense*
Madrid
Spain
balta@sip.ucm.es

Dr. Baltasar Fernández-Manjón is a Computer Science professor at UCM. His research interests lie in the educational uses of computers, markup languages, and user modelling.

**Alfredo Fernández-Valmayor**
*Dpto. Sistemas Informáticos y Programación, Universidad Complutense*
Madrid
Spain
alfredo@sip.ucm.es

Dr. Alfredo Fernández-Valmayor is a Computer Science professor at UCM, and his research interests include markup languages and its application to hypermedia and educational systems construction.

**Antonio Navarro**
*Dpto. Sistemas Informáticos y Programación, Universidad Complutense*
Madrid
Spain
anavarro@sip.ucm.es

Dr. Antonio Navarro is a Computer Science assistant professor at UCM. His research interests include markup languages, hypermedia, and software engineering.

**Extreme Markup Languages 2002**
Montréal, Québec, August 6-9, 2002
*This paper was formatted from XML source via XSL*
*Mulberry Technologies, Inc., August 2002*