Visualizing Persistent Objects using Higher-Order Functions in SGML

Florian Matthes Axel Wienberg

AB 4-022 Software Systems
Technical University Hamburg-Harburg
D-21071 Hamburg, Germany
{f.matthes,ax.wienberg}@tu-harburg.de

Abstract

We describe a novel approach for the visualization of persistent objects based on an orthogonal extension of SGML by variables, conditionals, function abstraction and function application. This enables a seamless integration with persistent languages and makes this approach particularly well-suited for the presentation of bulk data structures on various media.

After a definition of the syntax, evaluation semantics and type rules of this model, we present the implementation and use of the Structured Tycoon Markup Language (STML), a specific example of the model applied to the polymorphic, strongly-typed persistent object-oriented programming language Tycoon-2.

1 Motivation and Overview

An information system comprises database, application, and presentation services which tend to be realized using distinct technologies (database systems, programming languages, GUI toolkits) and which can be distributed over the network based on a client/server architecture as shown in the example of Figure 1.

Significant progress has been made in the last decade to overcome the impedance mismatch [Copeland and Maier 1984] between the database and the application services through the development of object-oriented databases [Cattell 1994], database programming languages [Schmidt and Matthes 1994] and persistent programming systems [Atkinson and Morrison 1995].

Moreover, virtually all database products today are equipped with graphical GUI design tools, report generators and mail merge tools to bridge the other gap, namely between presentation and application services. These tools help to visualize and pretty-print database objects on diverse windowing systems and output media like text files, postscript printers, e-mail messages or Word documents.

In this paper, we report on an alternative approach to simplify the presentation of complex persistent objects which is based on SGML as an open, platform-independent and highly popular standard to describe the content and the layout of text documents. The rationale of our work is to make use of the rich toolbox of SGML-based browsers, parsers, editors, etc. by a (fully orthogonal and compatible) extension of SGML with a minimum set of dynamic concepts necessary to generate SGML documents based on the content of persistent objects in relational databases, text-retrieval engines, directory services, file systems, etc. (compare Figure 1).

Given a particular application language A (e.g., Persistent C++), and a particular structured presentation language P (e.g., HTML), defined through its SGML document type definition, our approach can be summarized as follows:

1. A document type definition P_A is created by extending P by SGML elements for variables, conditionals, function abstraction and function application based on expressions of A. The

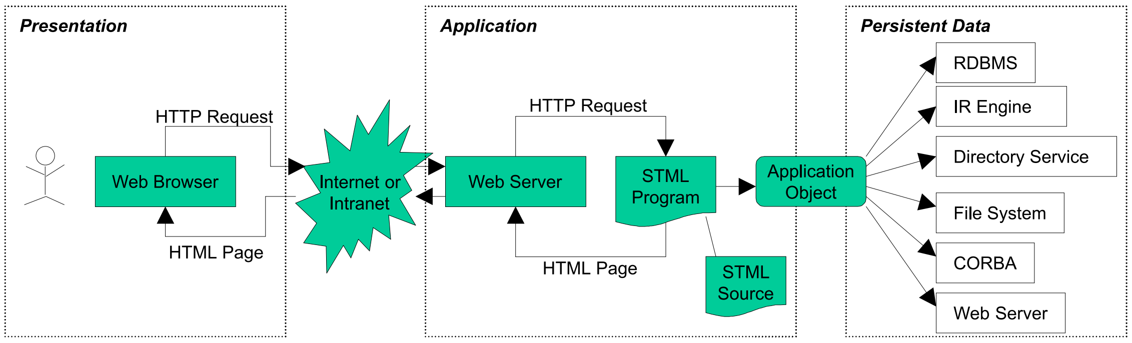

Figure 1: SGML-based visualization of persistent objects over the Internet

document type definition P_A is used by standard SGML tools to assist application programmers in building and maintaining P_A documents which define presentation services with embedded (bi-directional) bindings to typed application and/or database objects.

2. A P_A processor is implemented that checks P_A documents for type-correctness and translates them into executable code of A. In particular, this processor ensures that the generated code will at run-time produce documents which conform to P.

3. At run-time, a small library implements the type-safe binding from the generated code written in A to application and to database objects.

After a brief introduction to SGML in Section 2, we describe how to extend SGML by first-class functions (Section 3). Section 4 gives some insight into the expressiveness of the model by presenting a particular implementation of this model for the persistent, object-oriented programming language Tycoon-2 (A) and for HTML (P). As of September 1997, there are several commercial Web sites of newspapers in Germany which utilize the resulting language P_A (called STML, the Structured Tycoon Markup Language) on a daily basis to provide customer-oriented information services on the Internet via HTML front-ends [HOX 1997]. The paper ends with a comparison of our approach with related academic and commercial models for the generation of structured documents based on database contents (Section 6).

2 Understanding SGML

This section explains the basic SGML concepts necessary to understand how structured persistent objects and higher-order functions fit into SGML's document model.

The Structured General Markup Language (SGML) [ISO8879 1986; Goldfarb 1991; van Herwijnen 1994] is an international standard for content- as well as layout-based structuring and annotation of text documents. It has found widespread acceptance in areas such as electronic publishing and digital libraries, with its most popular application being the HTML language(s) used in the World Wide Web.

```
<!DOCTYPE HTML PUBLIC "-//W3C/DTD HTML 3.2 Final//EN">
<HTML> <HEAD> <TITLE>Sample SGML document</TITLE>
<BODY> ... <IMG SRC="sgml.gif"> ... </HTML>
```

This sample document illustrates that every SGML document consists of a reference to its underlying document type definition (DTD, in this case HTML release 3.2), and a document instance (in this case a title and a body with a nested reference to an embedded image).

A document instance relates to its DTD like a database instance relates to a database schema, filling the structure (nested types) defined in the DTD with actual values. The document text is annotated with start tags and end tags, denoting the range of an element. Tags are enclosed in angle brackets, with an initial slash indicating an end tag. Start tags can contain additional attributes applying to the occurrence of the element, specified as a sequence of *name=value* pairs (e.g., *SRC="sgml.gif"*). The document instance can contain entity references that are substituted with previously defined, potentially long strings by the SGML parser. References to so-called external entities allow the inclusion of other documents.

A specific DTD defines the elements and their attributes that can be used to structure the document instances, as shown in the following excerpt from the HTML DTD:

```
<!ELEMENT HTML O O (HEAD, BODY)>
<!ELEMENT HEAD O O (TITLE & ISINDEX? & BASE?)  +(META|LINK)>
<!ELEMENT TITLE - - (#PCDATA)*>
...
<!ELEMENT IMG - O EMPTY>
<!ATTLIST IMG
      SRC CDATA #REQUIRED
      ALT CDATA #IMPLIED
      ...>
...
```

Every element has a name (*HTML, HEAD, TITLE, IMG,* ...) and a content model, declaring the set and order of elements that may appear inside. This defines a context-free grammar for the logical structure of the document instances. For example, the element *HTML* consists of a *HEAD* followed by a *BODY*.

Since SGML is intended to be written by hand, it contains a number of abbreviation (minimization) features which are also specified as part of the DTD. Opening and closing tags may be declared optional by using the letter *O* instead of the dash (–) in an element definition. The first and second letter refer to the opening and closing tag, respectively. Omitted tags are inferred by the SGML parser based on the content model. When an attribute takes only an enumerated set of values, the attribute name is inferred from the value, and may be omitted. These two features and the context-sensitive whitespace handling preclude a correct parsing of a document instance without knowledge of its underlying DTD.[1]

The interpretation and presentation of an SGML document lies outside the scope of the SGML standard. Documents may be typeset as books or manuals, published as hypertext, or the markup may be used for content-based text retrieval.

Several authors (see, e.g., [Abiteboul 1997] for further references) have already explored a bi-directional mapping between (semi-) structured persistent objects and linear SGML text documents based on the following (rough) correspondence table:

SGML Concept	**Data Modeling / Typing Concept**
document type definition	database schema / type definition
document instance	complex object / typed value
sequence of (attributed) elements	record type definition
repetition of elements	list type definition
choice between elements	union type definition
optional element / attribute	type with null value
predefined element	domain / base type
nesting of elements	nested type definition
recusion in DTD	recursive type definition
external element	reference type definition

Our contribution complements their work by providing programmers with a structured, type-safe model to define both, the (rather static) overall layout and repeating elements of the documents

[1]For this reason, the XML standard currently being developed by the W3 consortium does not support minimization. XML is a restricted variant of SGML that may become the "next HTML". As a basis for our model, it would be just as suitable as SGML.

to be generated and also the (possibly deeply nested) bindings to queries and expressions which extract persistent data from the database and embed them into the result document.

3 Extending SGML with First-Class Functions

In our model, an SGML fragment is viewed as an expression that returns a value. All static document elements (described using tags of the non-extended DTD) are treated as literals, that is, they return themselves. Dynamic document elements may contain variables, conditionals, function abstractions and function applications which interact smoothly with the corresponding concepts of the application language. As a consequence, all visual elements of an application can be defined in SGML documents, cleanly separating presentation and application concerns.

In order to simplify the presentation, our examples make use of STML (Structured Tycoon Markup Language), the extension of SGML with expressions of the application language Tycoon-2. However, as described in Section 4, any higher-order or object-oriented language (Lisp, Standard ML, Java, C++, OQL) could be used instead of Tycoon-2.

We first illustrate the use of STML by examples before we define its syntax, evaluation semantics and type rules.

3.1 Examples of STML Code

In STML, a function abstraction allows a programmer to turn an arbitrary STML fragment into a function by enclosing it with the <*fun*> tag. The (optional) *param* attribute of that tag describes the list of typed formal parameters of the function:[2]

```
<define name=formatEmployee>
  <fun param='lastName :Sgml, sex :Sgml, age :Sgml'>
    <P><if true='sex="m"'>Mr.<else>Mrs.</if>
    <ref name=lastName> <I>(<ref name=age>)</I>
  </fun>
</define>
```

A <*define*> tag introduces a named variable which is bound to the value enclosed by the tag. This value can be an SGML fragment or a (higher-order) STML function. The scope of the variable declaration is the text following the tag within the enclosing sequence (see Section 3.3). A <*ref*> tag gives access to the actual value of a variable in the current lexical scope.

In the example above, the identifier *formatEmployee* is bound to a function for formatting the information about an employee. The gaps in this template are explicitly named and declared; the remaining SGML is checked for well-formedness independent of the actual arguments (for example, the correct balancing of the <I> tag is verified by the parser).

By applying the function, the template can be instantiated repeatedly. This can be done within an STML document using the <*apply*> tag, with the potential benefit of centralizing layout decisions.

```
<P>Our employees:
<apply name=formatEmployee><arg>Maier<arg>f<arg>39</apply>
<apply name=formatEmployee><arg>Smith<arg>m<arg>36</apply>
```

This example produces the following output:

```
<P>Our employees:
<P>Mrs. Maier <I>(39)</I>
<P>Mr. Smith <I>(36)</I>
```

[2]In this example, all arguments are declared to be (long) strings of type *Sgml*. A more realistic example would utilize a single parameter of the type *Employee* declared as a class in Tycoon-2.

More interestingly, the STML function can be applied to persistent data by passing the function as an argument to a method (*forEachEmployee*) of an application object (*project*):

```
<P>Our employees:
<send receiver=project selector=forEachEmployee>
  <arg><ref name=formatEmployee>
</send>
```

Please note the separation between application logic (how iterations are implemented) and presentation details (how persons should be visualized).

Since STML functions and SGML values are treated uniformly, statically nested and anonymous functions can be defined as illustrated in the following example which traverses a nested collection. The variable *p* of the outer function is bound iteratively to each element of the collection object *projects*. The variable *p* is then used as the receiver for a nested iterator method to enumerate all employees of that project.

```
<send receiver='application.projects' selector=forEach>
  <arg><fun param='p :Project'>
   <H1><A NAME=''p.name''>Project <eval>out.writeString(p.name)</eval></A></H1>
    <UL>
      <send receiver=p selector=forEachEmployee>
         <arg><fun param='e :Employee'>
           <LI> ...
         </fun>
      </send>
    </UL></fun>
</send>
```

The *p* argument passed to the STML function is in fact an object of the application programming language. Though the STML text cannot manipulate it directly, the object can be passed in a type-safe way to application objects. The <*eval*> tag encloses source code of the application language which has access to all (typed) variables in its lexical scope.

As indicated by these examples, operations on bulk structures are expressed by iterators of the target language using higher-order functions. These iterators work uniformly over multiple bulk structures (sets, lists, bags) and exceed the expressive power of relational query languages (compare [Matthes and Schmidt 1991; Fegaras 1994; Breazu-Tannen *et al.* 1991; Gawecki and Matthes 1996b]).

3.2 Syntax of STML

Syntactically, STML is defined by augmenting a given "client" DTD (e.g., HTML 3.2) with the element definitions shown in Figure 2. The definition of the element *STML* in the last line of Figure 2 *includes* the additional elements *define*, *ref*, ..., and *eval* into the original content model with the root element named *%client.base*. Inclusion is an idiosyncratic SGML feature which allows the included elements to appear floating anywhere inside the element for which the inclusion was defined. In this case, the scope of the extension is the entire document instance.

The syntax of the additional STML elements is expressed entirely in SGML. For example, the *apply* element is defined to consists of a list of *arg* elements.

As a consequence, STML documents conform to a clearly defined DTD and can be created, edited and maintained with standard SGML tools.

The DTD contains references to the entities *%stml.ident*, *%stml.param*, *%stml.type*, and *%stml.expression*, denoting identifiers, signatures, types and expressions of the application programming language, respectively. In SGML, they are viewed as uninterpreted strings. If these strings consist of alphanumeric characters only, enclosing quotation marks can be omitted. Syntax and type errors in these strings are detected by the compiler of the target language as soon as an STML document is compiled.

```
<!ENTITY % stml.ident "CDATA">
<!ENTITY % stml.param "CDATA">
<!ENTITY % stml.type "CDATA">
<!ENTITY % stml.expression "CDATA">

<!ELEMENT define - - %client.any>
<!ATTLIST define                                    – variable declaration –
        name      %stml.ident              #REQUIRED – name of this variable –
        type      %stml.type   'Sgml'      #IMPLIED  – type of this variable –>
<!ELEMENT ref - O EMPTY>
<!ATTLIST ref
        name      %stml.ident              #REQUIRED – name of referenced variable –>
<!ELEMENT assign - O %client.any>
<!ATTLIST assign
        name      %stml.ident              #REQUIRED – name of assigned variable –>
<!ELEMENT fun - O %client.any>
<!ATTLIST fun
        param     %stml.param              #IMPLIED  – parameter list –
        type      %stml.type               #IMPLIED  – result type –>
<!ELEMENT apply - - (arg*)>
<!ATTLIST apply
        name      %stml.ident              #REQUIRED – name of function –>
<!ELEMENT arg - O (%client.any)>
<!ELEMENT if - - (%client.any, elseif*, else?)>
<!ATTLIST if
        true      %stml.expression         #REQUIRED – predicate –>
<!ELEMENT elseif - O %client.any>
<!ATTLIST elseif
        true      %stml.expression         #REQUIRED – predicate –>
<!ELEMENT else - O %client.any>
<!ELEMENT send - - (arg*)>
<!ATTLIST send
        selector  %stml.ident              #REQUIRED – method selector –
        receiver  %stml.expression 'self'            – receiver expression –>
<!ELEMENT eval - - %stml.expression>
<!ATTLIST eval
        type      %stml.type   'Sgml'      #IMPLIED  – result type –>

<!ELEMENT STML O O  (%client.base)
        +(define | ref | assign | fun | apply | if | send | eval)>
```

Figure 2: STML element definitions

3.3 Evaluation Semantics of STML

There are three kinds of values that can be returned by an STML expression:

- a well-formed SGML text,
- a function mapping STML values to an STML value, with possible side effects on the store, and
- an arbitrary value of the application programming language.

Text that contains no markup evaluates to itself. Client DTD elements evaluate to the concatenation of their opening tag, the evaluation result of all subexpressions, and their closing tag. As a result, SGML text that contains no STML elements is self-evaluating.

STML expressions in a *%client.any* sequence and in an argument list are evaluated in a strict left-to-right order. Evaluation takes place in a dynamic environment mapping variable names to store locations. If at least one expression in a sequence may return non-white SGML text, all results are concatenated, and the sequence too returns SGML text. Otherwise, any whitespace in the sequence is discarded, and the result of the last subexpression is the result of the sequence. These kinds of sequence are called *concatenation* and *value* sequences, respectively.

An STML variable can be defined and initialized using the `<define>` tag, extending the current environment, and can be referenced using the `<ref>` tag. The variable can be updated using `<assign>`. Variables are typed and scoped statically. Because of higher-order functions, variables have potentially unlimited lifetime.

The `<fun>` element is the abstraction operator, also known as lambda. Evaluation returns a function value capturing the function's body and the current environment. Functions are truly first-class: They can be assigned to a variable, passed to another function or be returned form the current function. Functions are applied with the `<apply>` element. Actual arguments have to match the formal parameters' types.

There is no way to decompose an SGML text value or to test complex conditions using STML constructs only. These applications fall into the domain of the application programming language, and will be explained in Section 4.

3.4 Typing Rules for STML

The type algebra of STML is based on the type *Sgml*, the type identifiers inherited from the application language and on function types mapping a list of argument types to a result type. The following EBNF grammar gives the syntax of type expressions and parameter lists denoted by the *%stml.type* and *%stml.param* entities in Figure 2.

Type ::= BaseType | 'Fun' '(' Parameters ')' ':' Type ;
BaseType ::= 'Sgml' | ApplicationLanguageType ;
ApplicationLanguageType ::= name ;
Parameters ::= [Parameter { ',' Parameter }] ;
Parameter ::= name ':' Type ;

The type **Sgml** denotes well-formed SGML text. By well-formed we mean syntactically valid markup, a proper balancing of opening and closing tags, and tag and attribute names corresponding to those defined in the client DTD. We have not attempted to enforce conformance to the client DTD's element structure in computed SGML text, although with a more elaborate type system, this would be possible, too.

Arbitrary nesting of function types is possible. There is no mechanism to give names to STML types, so type checking is purely based on structure. Application language types are treated as abstract types: An application language type only matches another application language type of the same name.

In addition to the types used for variables, STML expressions can have the types *Bottom*, for the null value returned by `<assign>` and `<define>` elements, and *Whitespace*. These types are used internally by the typechecker for determining sequence types. The following example should illustrate the problem of whitespace handling in SGML and the distinction between value and concatenation sequences:

```
<define name=v1>   <ref name=v>   </define>
```

The variable *v1* is bound to a value based on that of the variable *v*. To the parser, the sequence contained in the `<define>` element consists of character data (the initial whitespace), followed by a `<ref>` element, followed by further character data (further whitespace). How should this be interpreted by the compiler? If *v* is a variable containing SGML text, the programmer wants to concatenate whitespace on both sides, and so the sequence is a concatenation sequence. However, if *v* is an STML function or an application language value, the sequence is a value sequence, and the whitespace should be discarded. In order to statically determine the correct behavior, the STML compiler has to track *v*'s type.

4 Integration with an Application Programming Language

This section describes how the services of an application programming language can be utilized seamlessly in STML documents. This is achieved by mapping STML types, variables and values to the corresponding concepts of the application programming language, thus enabling bi-directional data flow, and by allowing application code to be embedded into STML documents.

This tight integration is realized by a compilation from STML into application language code. However, as should be clear from the previous section, we have taken care to define the syntax, type rules and evaluation semantics of STML independent of the specific application language model.

In the following examples we will use the language Tycoon-2 as a concrete example for an application language. As the name implies, Tycoon-2 is the successor of the Tycoon system [Matthes 1997] developed at the University of Hamburg. An earlier version of Tycoon-2, called TooL, is described in [Gawecki and Matthes 1996b]. Tycoon-2 is a polymorphic, strongly-typed object-oriented programming language that integrates external services and data sources, and provides the glue between user-oriented interface and communication services on the one hand side, and computation and storage-oriented services such as full-text and relational databases on the other hand.

4.1 Using Application Language Expressions in STML Tags

As can be seen from the DTD in Figure 2, the `<if>`, `<elseif>`, `<send>`, and `<eval>` tags all contain application code (*%stml.expression*). Evaluating these tags involves evaluating the application language expression. In the case of the `<if>` and `<elseif>` tags, the expression evaluates to a boolean, determining which branch of the conditional is to be taken.

```
<if true='project.employees.size = 0'>
  No employees.<P>
<else>
  <!– list employees... –>
</if>
```

The `<eval>` tag allows execution of arbitrary application code. This code may produce SGML output as a side-effect. Free identifiers in the application code, like *project* in the condition of the `<if>` tag above, can refer to variables defined in STML, for example, defined as a function parameter:

```
<define name='projectHeader'><fun param='project :Project'>
  <H1>Project <eval> out.writeString(project.name) </eval></H1>
</define>
```

The `<send>` tag serves to pass arbitrary STML values to a method of the application language.[3] The receiver is denoted by an application language expression, but all arguments are STML expressions, possibly STML functions.

```
<send receiver='project' selector='discriminate'>
<arg><fun param='project :InternalProject'>
  ...<!- project is an internal project ->
<arg><fun param='project :ExternalProject'>
  ...<!- project is an external project ->
</send>
```

4.2 Accessing STML Values from Application Code

In the previous examples, the variables referenced from application code always contained application values. But functions and SGML text defined in STML can be referenced as well.

```
<define name='separator'><P><HR></define>
<eval>
  application.projects.forEach(fun(p :Project) {
    projectHeader[p]
    p.printDescription(out)
    out.writeString(separator)
  })
</eval>
```

This example, using Tycoon-2, iterates over all projects defined in the application. For each project, it calls the STML *projectHeader* function defined in an earlier example, calls a Tycoon-2 method for the actual project description, and prints the contents of the *separator* variable after each project.

Note how the types and values of the *projectHeader* and *separator* variables are mapped: The **Sgml** type of *separator* becomes *String* on the Tycoon-2 side, allowing it to be passed to *writeString*. The variable *projectHeader* has type **Fun**(*project :Project*):**Sgml**, a function from the application type *Project* to **Sgml**. On the Tycoon-2 side, this becomes a function type, too, and the argument type is naturally mapped to the Tycoon-2 type *Project*. However, the result type **Sgml** turns into *Void*: For efficiency reasons, STML functions returning SGML text are implemented in a side-effecting way, printing on the current output rather than returning a string value. As a consequence, the Tycoon-2 code does not have to explicitly concatenate the output it produces, but can use the implicit (and more efficient) concatenation of the output stream. The type mapping rules are summarized in Figure 3.

STML Type T	Generated Tycoon-2 Type $\rho(T)$
Sgml	*String*(containing well-formed SGML)
ApplicationType	*ApplicationType*
Fun$(i_1 : T_1,...,i_n : T_n)$:**Sgml**	**Fun**$(\rho(T_1),...,\rho(T_n))$ *Void* (output via side-effect)
Fun$(i_1 : T_1,...,i_n : T_n)$:$T$ $(T \neq$ **Sgml**$)$	**Fun**$(\rho(T_1),...,\rho(T_n))\rho(T)$ (no output)

Figure 3: Mapping from STML Types to Tycoon-2 Types

[3]Our implementation is based on an object-oriented application language. In a procedural or functional language, the `<send>` tag could be renamed `<call>`.

4.3 Passing Application Values to STML

There are three ways of introducing application values into STML: As parameters of STML functions called back from the application, through the `<eval>` tag, or through an implementation-dependant initial environment defined for a document.

The initial environment contains at least the current output (*out*) and the application object (*application*), which serves as the global entry point for persistent, application-wide data. The environment can also contain information about the particular request that initiated evaluation of this document, which in turn may include information about the person initiating the request. The concrete implementation is described in Section 5.

The statements inside an `<eval>` tag can include binding statements, introducing new application variables. These are not only visible inside the `<eval>` tag, but also in the following STML code up to the end of the current sequence. The example uses the Tycoon-2 binding operator *::=*.

```
<eval> application.sortProjects project ::= application.projects[0] </eval>
<if true='project.isNotNil'>
  <apply name='projectHeader'><arg><ref name=project></apply>
</if>
```

Since in this example the type of the *project* variable is deliberately left open by the programmer (to be inferred later by the application language's type checker) the STML type checker can only assume that the variable has *some* type. Technically speaking, the variable is assigned a type unification variable, which will at least detect inconsistent use. Exact type checking is deferred to the application language's type checker.

Another way of passing data out of the `<eval>` tag is through assignment. STML variables are mutable, and this property is reflected in application code.

```
<define name='str'>some string</define>
<define name='f'><fun param='x :Sgml'><ref name=x></define>
<eval>
  str := "another string"
  f := fun(s :String):Void { out.writeString(s) }
</eval>
```

This example assigns a Tycoon-2 string value to an STML variable of type **Sgml**, and a Tycoon-2 first-class function value to an STML variable of the corresponding type. Types are mapped in the way described in Section 4.2.

Note that the new function assigned to *f* is semantically equivalent to the one defined in STML. Note also that the Tycoon-2 String type is less strict than the **Sgml** type, allowing strings that are not well-formed SGML.

Call-back parameters are the most common way for accessing application data in STML. An STML function is passed to the application, and is applied there like an ordinary function defined in the application language. Transparent to the application program, the function arguments enter the STML document, with their types being mapped as described above, for example:

```
formatEmployees(f :Fun(String, String, String)Void)
{ employees.forEach(fun(e:Employee) {
    f[e.firstName, e.sex.asString, e.age.asString]
  })
}
```

Assuming the above definition of the method *formatEmployees* in class *Project*, application data can be formatted in STML like this:

```
<send receiver='project' selector='formatEmployees'>
  <arg><ref name=formatEmployee></send>
```

By virtue of the rich bulk type libraries of Tycoon-2 (inherited from its predecessor TooL [Gawecki and Matthes 1996a; Gawecki and Matthes 1996b]), the iterator *employees.forEach* could be an iteration over a persistent program data structure, a selection query, or an iteration over the results of an SQL query submitted to a SQL server as exemplified by the following Tycoon-2 code fragments:

```
employees :Array(Employee)    (* persistent data structure *)

employees: Reader(Employee)   (* computed iteration *) {
  application.allEmployees.select(fun(e:Employee):Bool {
    e.project = self
  })
}

employees: Reader(Employee)   (* SQL query *) {
  cursor ::= sqlConnection.directQuery(
    "select * from Employees where projectId="+self.id),
  FunReader.new(fun() {   ;; call this function for successive elements:
    cursor.next.if(   ;; more employees?
      true: { rowToEmployee(cursor) }   ;; yes ⇒ create object for this row
      false: { nil })})   ;; no ⇒ end of iteration
}
```

5 Using STML on the WWW

One of the communication services supported by Tycoon-2 is the Hypertext Transfer Protocol (HTTP) of the World Wide Web. Tycoon-2 offers an extensible Web Server Framework based on polymorphic resource objects, similar to the W3 consortium's Jigsaw server. By connecting HTTP resource objects to external resources, it implements a *three-tier architecture* as shown in Figure 1. The web browser on the *client* tier connects to the integrated Tycoon-2 web and *application* server representing the second tier, which operates on *data* from the third tier.

The Web Server Framework can be extended by subclassing resource classes using the full power of the Tycoon-2 system to answer a HTTP request, without the overhead traditionally associated with the Common Gateway Interface (CGI). All external system resources are accessed and all computations are performed in statically typed, compiled Tycoon-2 code.

In this context, STML is used for dynamically generating HTML pages based on persistent application objects (e.g., an inventory database), possibly depending on session-specific information (e.g., contents of a virtual shopping basket) and on information associated with an individual client request, (e.g., address of a customer).

The resource class *StmlResource* is defined and is registered with the web server under the file extension *.stml*. Whenever the web server encounters a request for a file name with this extension, it creates an *StmlResource* object. HTTP requests for the resource are then delegated to the resource object. The *.stml* document itself is assumed to contain STML source text (i.e. SGML enriched with higher-order functions), and the HTML page returned by the server is the result of evaluating the STML code (c.f. Figure 1).

This dynamic and personalized document generation involves the following steps:

1. The *.stml* file's timestamp is checked. If the file has not been modified since the last access by the web server, execution proceeds directly with step 6. This implies that the compilation result obtained in steps 2 through 5 is cached persistently.

2. The STML source is parsed and verified by a standard SGML parser. Errors such as invalid markup, mismatched tags, and missing or unknown attribute names or values are reported.

3. The abstract STML tree produced by the SGML parser is type checked as described in Section 3.4.

4. If type checking succeeds, the STML code is compiled into a Tycoon-2 function. Adjacent output is concatenated into block write messages whenever possible, giving performance equal to or exceeding that of static HTML (file) resources. This phase produces an object structure representing a Tycoon-2 abstract syntax tree. We do not produce source text in order to avoid unnecessary dependency on the surface syntax and to avoid costly unparsing and re-parsing by the Tycoon-2 compiler. There is an analogy to STML here: In both cases, we prefer dealing with structure rather than flat text.

5. The Tycoon-2 abstract syntax tree produced in the previous step is compiled to an executable Tycoon-2 function using the reflective facilities offered by the Tycoon-2 system. This includes an additional type-checking phase on the Tycoon-2 side. The function is stored in the resource object.

6. Every time the resource is accessed, an *StmlProcessor* object is allocated which handles the current request. Its slots and methods provide the initial environment for the STML document (see subsection 4.3), containing at least a reference to the HTTP request object, a reference to the application object, and the current HTML output object. The request object includes information about the client issueing the HTTP request, in the form of the HTTP basic authentication scheme (identification by name and password) and id numbers attached as query arguments or cookies.

7. The function created in step 5 is called to produce the actual output. If a compilation error occurred, the error messages are returned instead of the STML page's results. This can only happen during development, since a page that has been successfully compiled once will not have to be compiled again.

6 Related Work

Support for textual presentation with specialized report languages dates back to the 1970s. Over the years, the presentation media evolved, from printers, terminals to graphical workstations and to Hypertext documents published via the World Wide Web.

Dynamically generated HTML has been with the World Wide Web from the beginning. The simplest solutions for generating dynamic output consists of CGI scripts fully responsible for composing a page, usually consisting of a sequence of print statements. The document is embedded in the program, so the author of a dynamic page has to be a programmer. There is no mechanical aid such as an editor that would simplify the creation of these pages, and there is no guarantee that their output is actually HTML. Moreover these scripts tend to be unreliable and slow because they are written in dynamically typed, purely interpreted languages with virtually no support for efficient database access.

As a reaction to these shortcomings, specialized markup languages allow the inclusion of dynamic elements inside documents. This way, the document layout can be designed and modified later without affecting the dynamic parts, and without a need to understand them. Examples are WWW servers that recognize certain special character sequences in HTML documents, and replace them with the current time, the modification date, the contents of some other file, or the output of some shell command. The NCSA server [NCSA 1997] expects these server-side includes in HTML comments, thus allowing page construction with a standard HTML editor. This serves some basic needs, but offers no way to produce conditional or repeating document parts.

A number of database/web integration tools offer specialized markup for the presentation of relational tables, including a restricted form of iteration. These tools are similar in functionality to traditional report languages, binding text fragments to certain events in the output like page breaks or group changes. However, the document structure assumed by these constructs is too rigid to cope with varying media types, or to represent complex data structures.

Most dynamic HTML extensions do not conform to any document standard, precluding the use of a standard editor. For example, the Roxen server [Roxen 1997] uses a huge number of ad hoc tags, one tag for every function offered by the server. There is no DTD available, so the page creator has to resort to manual construction. Other HTML preprocessing tools (e.g. w3msql by Hughes Technologies [MSQL 1997]) use fantasy tags like `<<...>>`, `[...]` or `<!...>` that have no meaning or a different meaning in SGML. Moreover, the dynamically generated parts are usually produced using print statements.

Some major systems like ColdFusion by Allaire [CoFu 1997], HAHTSite by HAHT [HAHT 1997], NetDynamics [NetD 1997], or WebObjects by NeXT, Inc. [WebO 1997] work around this problem by supplying their own WYSIWYG editor. However, this requires updating the development system when the user wants to use new HTML extensions.

Another framework for dynamic HTML is the LiveWire product by Netscape Communications Corporation, offering server-side JavaScript [LiWi 1997]. This offers the flexibility of a full (though dynamically typed) programming language, with access to server-side persistent data. LiveWire introduces just one new tag, the `<SERVER>` tag for including LiveWire statements. This minor extension poses no problems for computer supported editing. Still, dynamically created HTML fragments have to be issued via print statements.

7 Summary and Future Work

STML is defined as an orthogonal DTD extension, which can be added easily to the latest HTML DTD. Any SGML editor can be used for page creation and maintenance. STML tags obey and preserve the document structure. Dynamic parts are created from templates passed to the application in the form of first-class, lexically-scoped STML functions. STML separates the language level (naming and binding) from the application level (the actual operation to be performed), allowing greater flexibility when adding new functionality. Last but not least, STML offers a seamless, type-safe integration with persistent data and external services.

Structured document description languages in the spirit of SGML can also be used to describe interactive elements (buttons, input fields, scrollbars, drop targets) embedded in hierarchical documents. We expect that the extension of SGML with higher-order functions will also simplify the binding from these elements to event handlers in the application code.

Acknowledgments

This paper is a summary of work carried out at Higher-Order Information and Communication Systems GmbH, Hamburg. Andreas Gawecki turned this idea into a project, and supplied the framework for implementing STML on the Web. STML was improved by the comments and remarks of its first users, Hubertus Koehler and Ingo Richtsmeier.

References

Abiteboul 1997: Abiteboul, S. Querying Semi-Structured Data. In Afrati, F. and Kolaitis, P., editors, *Proceedings ICDT '97, 6th International Conference, Delphi, Greece*, volume 1186 of *Lecture Notes in Computer Science*, pages 1–18. Springer-Verlag, 1997.

Atkinson and Morrison 1995: Atkinson, M. and Morrison, R. Orthogonally Persistent Object Systems. *VLDB Journal*, 4(3), 1995.

Breazu-Tannen et al. *1991:* Breazu-Tannen, V., Buneman, P., and Naqvi, S. Structural Recursion as a Query Language. In *Database Programming Languages: Bulk Types and Persistent Data.* Morgan Kaufmann Publishers, September 1991.

Cattell 1994: Cattell, R.G.G., editor. *Object Data Management.* Addison-Wesley Publishing Company, second edition, 1994.

CoFu 1997: Cold Fusion 3.0. http://www.allaire.com, 1997. Allaire Corp., One Alewife Center, Cambridge, MA 02140, USA.

Copeland and Maier 1984: Copeland, G. and Maier, D. Making Smalltalk a database system. In *Proceedings of the ACM-SIGMOD International Conference on Management of Data, Boston, Massachusetts*, pages 316–325, June 1984.

Fegaras 1994: Fegaras, L. Efficient Optimization of Iterative Queries. In Beeri, C., Ohori, A., and Shasha, D.E., editors, *Database Programming Languages, New York City, 1993*, Workshops in Computing, pages 200–225, 1994.

Gawecki and Matthes 1996a: Gawecki, A. and Matthes, F. Exploiting Persistent Intermediate Code Representations in Open Database Environments. In *Proceedings of the Fifth Conference on Extending Database Technology, EDBT'96*, volume 1057 of *Lecture Notes in Computer Science*, Avignon, France, March 1996. Springer-Verlag.

Gawecki and Matthes 1996b: Gawecki, A. and Matthes, F. Integrating Subtyping, Matching and Type Quantification: A Practical Perspective. In *Proceedings of the 10th European Conference on Object-Oriented Programming, ECOOP'96*, pages 26–47, Linz, Austria, July 1996. Springer-Verlag.

Goldfarb 1991: Goldfarb, C. *The SGML Handbook.* Clarendon Press, Oxford, 1991.

HAHT 1997: HAHTsite 3.0. http://www.haht.com, 1997. HAHT Software, Inc., 4200 Six Forks Road, Raleigh, NC 27609, USA.

HOX 1997: Higher-Order Informations- und Kommunikationssysteme GmbH. http://www.higher-order.de, 1997. Burchardstraße 19, D-21071 Hamburg, Germany.

ISO8879 1986: ISO 8879, Geneva. *Information Processing – Standard Generalized Markup Language*, 1986.

LiWi 1997: Netscape LiveWire. http://www.netscape.com, 1997. Netscape Communications Corporation, California, USA.

Matthes and Schmidt 1991: Matthes, F. and Schmidt, J.W. Bulk Types: Built-In or Add-On? In *Database Programming Languages: Bulk Types and Persistent Data.* Morgan Kaufmann Publishers, September 1991.

Matthes 1997: Matthes, F. Higher-Order Persistent Polymorphic Programming in Tycoon. In Atkinson, M.P., editor, *Fully Integrated Data Environments.* Springer-Verlag (to appear), 1997.

MSQL 1997: W3-msql. http://Hughes.com.au, 1997. Hughes Technologies, Inc., PO Box 432, Main Beach, Gold Coast Qld 4217, Australia.

NCSA 1997: The NCSA HTTPd. http://www.ncsa.uiuc.edu, 1997. The National Center for Supercomputing Applications, University of Illinois at Urbana-Champaign.

NetD 1997: NetDynamics 3.1. http://www.w3spider.com, 1997. NetDynamics, Inc., 185 Constitution Dr., Menlo Park, CA 94025, USA.

Roxen 1997: Roxen WWW Server. http://www.roxen.com, 1997. Roxen InformationsVävarna AB, Skolgatan 10, S-582 35 Linköping, Sweden.

Schmidt and Matthes 1994: Schmidt, J.W. and Matthes, F. The DBPL Project: Advances in Modular Database Programming. *Information Systems*, 19(2):121–140, 1994.

van Herwijnen 1994: Herwijnen, E. van. *Practical SGML.* Kluwer Academic, 2nd edition, 1994.

WebO 1997: WebObjects 3.0. http://software.apple.com/webobjects, 1997. 1 Infinite Loop, Cupertino, CA 95014, USA.