Component Communication and Data Adaptation

Welf Löwe Markus L. Noga

University of Karlsruhe

Program Structures Group

Adenauerring 20a, D-76131 Karlsruhe, Germany

{loewe|noga}@ipd.info.uni-karlsruhe.de

November 7, 2001

Abstract

Predefined components often require adaption to interoperate, especially an adaption of the data they exchange. This paper presents an XML-based approach to the data adaption problem. It covers serialization of objects to XML, their transformation with XSLT, as well as deserialization, i.e., parsing and reconstruction of the transformed objects. We statically analyze classes to generate specific document types and allow static preprocessing. Compared to normal interpreted XML processing, this approach eliminates introspection overhead, thus accelerating the components' communication.

1 Introduction

Component systems are usually built in two stages. First, preexisting components are bought or extracted from legacy systems, while missing components are designed and implemented. In the second stage, components are assembled to a system. Systems integration is personally, physically, and temporarily separated from component design. Hence, mismatches between components are the rule, not the exception. This makes adaption, especially adaption of the data exchanged by components, an integral part of component-based systems design.

Classic middleware architectures for components include CORBA [18], (D)COM [16] and Enterprise JavaBeans [21]. They power highly scalable distributed systems whose processing performance is crucial. Thus, they employ highly efficient binary encodings to marshal data structures and method calls in a distributed system. Unfortunately, these dense encodings are not easily amenable to adaption. A different approach is necessary.

The XML – Extensible Markup Language [24] – has emerged as the standard in general purpose, platform-independent data exchange. XML documents carry logical markup: they are tagged according to their content structure. Document Type Definitions (DTDs) and XML Schemas [25, 26] provide type-checking for XML documents. Documents are easily accessible to standard transformation techniques, e.g., XSLT – Extensible Stylesheet Language Transformation [27] – processors. This makes XML a good candidate to connect components and host adaptions.

Instead of binary channels, we connect components with XML streams. Object structures to be communicated are serialized to XML by the sender, transported via some channel and rebuilt from the stream by the receiving component. This way, data adaption may be effected with widely available XSLT processors.

Most available XML parsers and XSLT transformation processors are interpreters: they read arbitrary well-formed XML, read and analyze the definition of the required structure and validate the document against it. Along with many others, the parsers provided by the Apache project [2], James Clark [8] and IBM [1] fall into this category. Most XSLT processors are interpreters as well, e.g., xt [8] and saxon [13].

In a component context, the data definitions (DTDs or Schemas) for individual components are available at compile time. Respectively, adaption transformations (XSLT scripts) must be provided at deployment time for the system to work. These data allow considerable optimizations in parsers [17] and transformers [19, 9]. They reduce the considerable performance penalty of XML versus binary encodings, but reaping these benefits requires new tools.

As components and deployment contexts may change rapidly in software evolution, manual implementation is not an option. We take a compiler-based approach and analyze component sources with the RECODER system [15] to generate DTDs and Schemas. In contrast to generic solutions, this approach allows for application-specific DTDs and Schemas which carry full type information. Specialized serializers and deserializers are automatically generated and subsequently woven into the components with RECODER. To optimize the entire processing pipeline, we also employ XSLT script compilers [19, 9].

The remainder of this paper is organized as follows: we first discuss related work in 1.1 below. In section 2, we introduce the basic model for communication between components and discuss the problems to solve. Section 3 specifies the processing pipeline to serialize a data structure, to transform it, and rebuild the new data structure. Section 4 shows possible optimizations. Finally, section 5 summarizes our results and outlines directions for future work.

1.1 Related work

In [7], Chen et. al use Enterprise JavaBeans runtime introspection to serialize bean state to XML. Their approach is interpretative, although they formulate generation of specific serializers as a possible future enhancement. They discuss typing and DTD generation, but do not mention Schemas. Their application context is web page generation, which does not require deserialization. Consequently, this topic is not discussed.

The Castor project [6] provides a generic framework for Java data bindings. Among other targets, Castor supports relational databases and XML. The project employs generator techniques to create specific serializers and deserializers, but does not create specific typed DTDs and Schemas for Java classes.

The emerging web service standards SOAP and WSDL [22, 23] complement our approach. SOAP defines a communication envelope for XML data exchange, and may thus carry messages originating from our infrastructure. WSDL is a web services definition language, which widely peruses XML Schema and provides very little in the way of extensions. E.g., typed references to services are a missing concept. WSDL may be used to encapsulate schema definitions generated by our process when calling via SOAP.

Classic component IDLs, e.g. for [18], include typed references to components or services. They support both call-by-value and call-by reference, using generators to supply stubs and skeletons for remote invocations. For the time being, call-by reference and typed references are outside our scope. However, our infrastructure is ready for future extension with references, as the RECODER [15] framework is sufficiently powerful for the required analyses and stub/skeleton generation.

2 Basic Model

We first we define a model for components. This model builds on the existing object oriented type systems, which is summarized next. In the last subsection, we model the problem of communicating data structures and subsequently consider the impact of adaptations and transformations.

2.1 Component and Type Model

For the purpose of this paper, we define components to be software artifacts with typed input and output ports. This definition focuses on computational components, but is sufficiently general to cover all other variants. Input ports are connected to output ports via communication channels called connectors. The notion of ports and connectors are known from architecture systems [20, 5].

While some connectors may be as complex as most components, and thus require the same amount of consideration in design, we assume most connectors simple point-to-point data paths. Figure 1 sketches this basic component model.

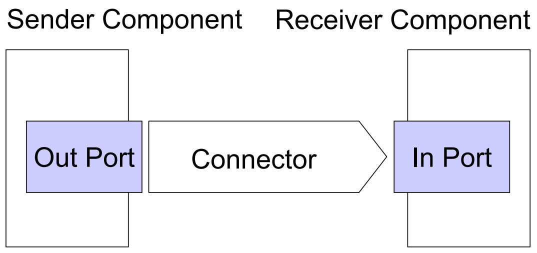

Figure 1: Basic Component Model

In general, ports are implemented by sequences of basic constructs like method calls, RPCs, RMIs, input output routines etc. provided by the implementation language or the component system. In contrast to those basic features, a port abstract from implementation details and defines points in a component that provides data to its environment and requires data from its environment, respectively. A connector is defined by an out-port and and in-port sharing the same connector identity.

We refer to [3] for automatic component adaptation by changing port implementations and to [12] for extracting ports from legacy code.

Technically, we assume the programs contain send and receive calls to objects of type `Port` naming the connector identity in the first parameter. The types to send and to receive are the second parameter and the return type, respectively.

Most type systems in modern programming languages are roughly equivalent. They distinguish the types in three dimensions:

- value types (where the identity and the value of an object is the same) vs. reference types (where identity and value of objects can be distinguished),
- concrete types (with implementation of methods) vs. abstract types (interfaces),
- language defined types (where the implementation of methods is not accessible) vs. user defined types (where we assume the implementation of methods available).

Users define types with classes. A class defines two types, namely the class type with the shared (or static) attributes and the object type capturing the attributes of the single instance objects of that class.

There is an inheritance relation between types. We have

- subtype inheritance of types (the subtype declares to implement the supertype interface and
- implementation inheritance (the subtype gets the supertype's method implementations, conflicts are resolved, however.).

For the examples and implementations in the present paper, we use the Java type system, which is briefly defined in the appendix, cf. appendix A.

2.2 Communicating data structures

All data structures reside in an address space associated with their producer. If a prospective consumer resides in the same space, it may be granted direct access via a local handle. In general, prospective users may reside in different address spaces, outside the scope of local handles.

There are two main approaches to distribution: value and reference semantics. Using value semantics, the producer passes a copy of the data structure to the consumer. As data structures may consist of structured objects with mutual references, this requires deep copying. This makes for large initial transmissions, but all subsequent consumer operations act on the local copy.

Using reference semantics, the producer passes a globally valid handle to the consumer and exposes a remote access interface. While handles are usually of fixed and small size, all subsequent consumer operations must traverse the network to act in the producer's address space.

Value and reference semantics are not equivalent, but designers are recommended not to exploit the difference. Clean design simplifies porting between local systems and different distributed architectures. This includes porting between value and reference semantics. Depending on access profiles, communication overhead and latency, either of them may be optimal. [14] discusses general data structure transformations to optimize w.r.t. an access profile.

Adaptations and transformations complicate the architectural choice. Consider the issue of efficiency again. Using value semantics, the data structure is copied and transformed exactly once.

With reference semantics, the remote access interface must provide access to the transformed data structure. The straightforward approach triggers a complete transformation atomically for every access to the data structure. This preserves reference semantics, but leads to huge inefficiencies for repeated accesses. In general, there are no partial transformations to alleviate this problem.

A modification of this approach caches the transformed data structure in the producers' address space. To preserve reference semantics, cache consistency must be maintained.

However, in the present paper we focus on values that are to transmit rather than on references for remote access.

3 Architecture

We first present our XML-based architecture for connectors. Then, we discuss the generator architecture to automate connector generation. Proceeding to details, we then defines our XML representation of data objects. We discuss the design decisions made. Finally, we briefly outline important properties of the generated code.

3.1 Connector Architecture

For two components to communicate, an output port must be linked to an input port via a connector. Often, this connector will be an entirely passive construct. In the general case, however, it may perform data adaptations. Until now, we have abstracted from connectors' internal structure. We now propose the connector architecture shown in Figure 2.

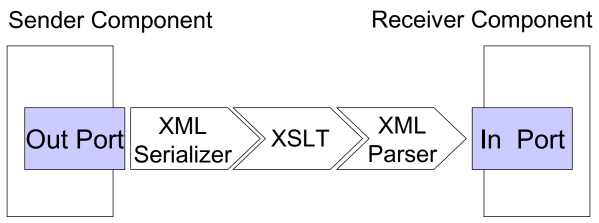

Figure 2: Adapted communication between components.

A connector consists of the following subparts:

1. A basic transport channel capable of transmitting byte streams, e.g., a UNIX pipe, a TCP/IP socket or an HTTP connection.

2. A serializer that traverses the internal data structures of the producing component and captures their state in an XML document written to the channel. In general, state may extend over an entire object graph — individual objects are just a simplification.

3. An optional XSLT processor that reads, filters and transforms the generated XML document to make it appropriate for the consuming component. It may be operate either at the producing or the consuming end of the channel.

4. A deserializer parses the incoming XML document and reconstructs the object graph or a transformed version thereof in the consumer's address space.

In our component model, both input and output ports are typed. This translates into grammatical restrictions on the possible outputs of a serializer and the legal inputs for a deserializer. We obtain tight bounds on the admissible data exchange language by generating specific DTDs from the respective input and output port types.

3.2 Generating Connectors

The availability of formal specifications for both port types and exchange formats greatly simplifies the implementation of connectors. Writing them by hand is time consumptive and prone to errors. Using specifications, we can generate them automatically.

Our *aXMLerate* project [4] provides a toolkit to generate parsers from DTD and XML Schema definitions. It is introduced in [10], which additionally demonstrates the performance benefits of generated over traditional XML processing.

The optional transformation step is defined by an XSLT script. Again, we use an *aXMLerate* generator to compile the script [19] instead of using a standard interpreting XSLT processor.

Figure 3 shows how connectors are generated in a preprocessing phase.

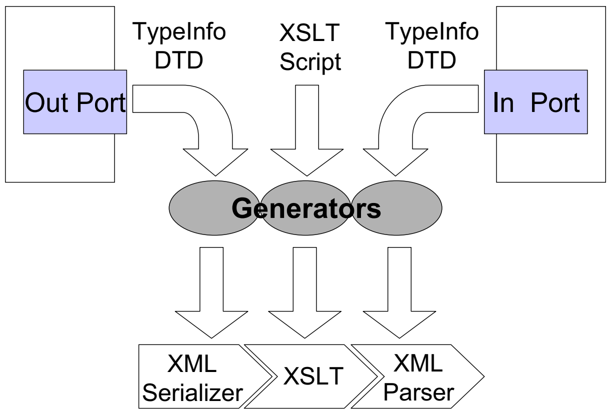

Figure 3: Generating a connector

For deployment, the generated subparts of the channel are closely integrated with the consuming and producing components. We use RECODER to weave the appropriated method calls and implementations into the components, replacing the abstract ports. Figure 4 shows the resulting program.

3.3 XML Representation

The XML representation must encode all information in an object graph. This includes the value of variables and references between objects, but also the type information required to resolve polymorphic

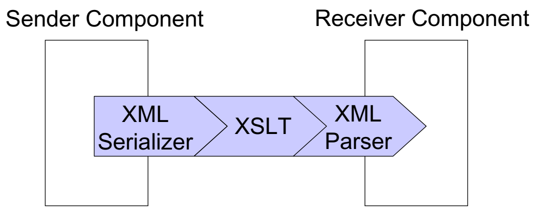

Figure 4: Connector woven into the components

calls. We first deal with value and reference types. Then, we discuss how to preserve polymorphism due to subtype inheritance and the representation of implementation inheritance. Finally, we consider the problem of language and externally defined types.

A variable of a value type may only be assigned identityless values of exactly the same type. There are two kinds of value types, primitive and complex ones. Primitive value types contain a single elementary value. We map every one of them directly to an XML element, e.g. a Java `int` is mapped to:

```
<!ELEMENT int #PCDATA>
```

Complex value types are recursively defined as records, unions or arrays of value types. Their semantics equal primitive value types in that instances do not have identity other than their contents. We note that the order of definition imposes an order relation on record fields regardless of their names. Unions are trivially ordered, as exactly one of their variants is active. Finally, array entries are trivially ordered by their indices.

Therefore, we can represent records, unions and arrays as XML elements with sequential, alternative or iterated content models, respectively. To facilitate reconstruction, the array element requires a length attribute whose value must equal its number of children. Unfortunately, this constraint cannot be expressed in a DTD. As Java does not support complex value types, we give an example in C notation:

```
typedef struct { int x; B y; } A;
typedef union  { int x; B y; } B;
typedef B[]                    C;
```

are mapped to

```
<!ELEMENT A (int, B)>
<!ELEMENT B (int| B)>
<!ELEMENT C (B)*    >
<!ATTLIST C length CDATA #REQUIRED>
```

Now we consider reference types. Here, the static type of a variable need not correspond exactly to the dynamic type of an object assigned to it at runtime. In polymorphism, it may instead be assigned objects of an arbitrary subtype. We thus need to differentiate between the data layout of an object of known type on one hand and a reference to an object of known supertype on the other.

We postpone the problem of polymorphism for now by postulating an entity defining a DTD content model fragment for every reference type, e.g., for every class `X`, there is some entity definition

```
<!ENTITY classX "...">
```

With this postulate, data layout becomes simple. Using these entities, we define elements representing concrete reference types exactly as for record, union and array value types above. In Java, there are only record and array reference types. Here is an example for classes (that is, records):

```
class X { int a; X  b; }
```

to

```
<!ELEMENT classX (int, &classX;)>
```

where `&classX;` is a reference to the entity postulated above.

In an open world, arbitrarily many subtypes may occur in polymorphism. As DTDs are finite, we assume that all data types are known at deployment time. In this closed world, the set of types admissible at run time is the transitive subtype closure over a variable's static type, minus the abstract types therein.

Armed with this set, it is now possible to solve the subtyping problem and define the postulated entities. The necessary content model fragment must admit exactly one of the elements representing the data layout of every set member. As we serialize object graphs to XML trees, this is not sufficient for notational reasons. To break cycles, we add one more alternative to the content model: a reference to a previously serialized element. Thus, if `X` had exactly two additional concrete subclasses `Y` and `Z`, we would map it to this entity:

```
<!ENTITY classX "(ref | classX | classY | classZ)">
```

Notationally, forward references are encoded by recursively inlining the target type. This recursion terminates for both value types and backward references. The element to encode backward references, `ref` , is defined like a primitive type:

```
<!ELEMENT ref #PCDATA>
```

By convention, serialized objects are implicitly enumerated sequentially, starting with `1`. References are simply indices in decimal notation, with `0` representing the special value `null`.

Since we only consider data and not service transmission, the handling of polymorphism completes the discussion of abstract types and subtype inheritance. Implementation inheritance is captured by inlining the data layout of the supertypes. This must obeys the conflict resolution mechanisms of the language. In Java, e.g., there are no conflicts as a class inherits implementations from exactly one supertype.

Types defined by the language report and in standard or external libraries do not provide an implementation in source code. For those types, mappings and serializer/deserializer routines must be defined by hand, as explained for the primitive Java value types above. For all user defined types, the mapping is automatically constructed by the above definitions and the serializer/deserializer routines are generated.

By construction, every legal object graph is mapped to an XML document valid under the generated DTD. The inverse does not hold in three cases.

First, a DTD cannot adequately restrict primitive values. An integer must be encoded as `#PCDATA`, so arbitrary sequences of letters will appear valid according to the DTD. Unless a more powerful document type system like XML Schema is used in future versions, there is no way around this.

Second, there is no guarantee that a reference actually points to an object of the correct type, or indeed an object at all. While the use of `ID` and `IDREF` attributes solves the dangling reference problem, these global keys and references cannot uphold subtype relations. They should also be deprecated due to poor nesting behavior. We chose the implicit enumeration scheme because it allows for one-pass processing: all references are guaranteed to point backwards. The scheme also reduces document sizes compared to explicit encoding.

Third, the DTD cannot verify that an array's length attribute does indeed equal its number of children. There is no way around this.

Several other design decisions were involved in devising this mapping scheme. We only discuss the three most salient ones here.

Our mapping does not explicitly state field names, using types instead. We took this stance because DTDs do not separate element names and element types. They are separate in XML Schema. Currently, we point to the bijective map between field names and sequence positions.

Modern programming practice disdains macros. Entities are macros, so their use should be deprecated. However, current alternatives are even less appealing. It is possible to encode them as elements, which would keep DTD sizes roughly the same, but double the number of elements in every document without encoding any additional information. As DTDs are generated, we could also expand entities in the generator. This would leave document sizes unchanged, but vastly increase DTD sizes and reduce their readability. When moving to XML Schema, named content model fragments provide an elegant solution to this problem.

The class names used in 3.3 are of cause simplified examples. In practice, we use a configurable mangling mechanism to map names and escape illegal characters like Java package separators and array brackets. If readability is completely peripheral, it may be desirable to use still terser encodings. A Namespace approach may be another interesting route, but DTDs do not support them explicitly.

3.4 Generated Code

The generated serializer code traverses an object graph once in depth-first order and prints XML in document order, which is depth-first. It maintains a hash table of serialized objects for the active session to detect back edges in almost $O(1)$.

The generated deserializer code parses an XML document once in document order and reconstructs the object graph in depth-first order. It maintains an array of deserialized objects for the active session to reconstruct back edges in $O(1)$. A second pass to resolve references is not required.

Generated XSL transformations may traverse the input document in an arbitrary fashion, although depth-first is an important special case. They may cache intermediate results of any size, but the output document is written in depth-first order.

Due to programming language access restrictions, e.g. `private` and `protected` member variables in Java, the data access sections of serializers and deserializers must be woven into the classes they act on. To properly deal with `null` references, public access to the serializers cannot occur via class members. We chose to group all static access methods for a package in a separate serializer class. Encoding and decoding of backward references and primitive types is handled by DTD-invariant serializer stream classes.

4 Optimizations

The generation of components specialized to actual communication and adaptation task per se improves the performance of communication. However, we can achieve additional speed-ups by considering the connector subparts as a whole.

4.1 Skipping Explicit Intermediate Structures

There are quite a view intermediate structures constructed during communication and adaptation. First, the serializer produces an XML encoding of the object structure to communicate. Second, the transformator parses the XML document and constructs an internal DOM representation of it. Third, it serializes the transformed XML encoding. Forth, the deserializer parses the latter XML document builds an internal DOM representation which is traversed to call the respective object constructors in the receiver component context. Figure 5 sketches the process and the produced data structures.

The first optimization is to execute the transformer and deserializer in the receiver component process. Then we can skip the explicit XML structure encoding the transformed object graph. Together with this structure, the writing of XML documents in the XSLT transformer is obsolete. Measurements show that this writer consumes 30% of the whole transformation [19]. Moreover, the parser in the deserializer can be omitted as well.

Recall a property of the transformator and the deserializer: however the transformation access the input XML structure, it produces the output in document (depth-first) order. The deserializer also visits the transformed XML document in document order to call the appropriate constructors.

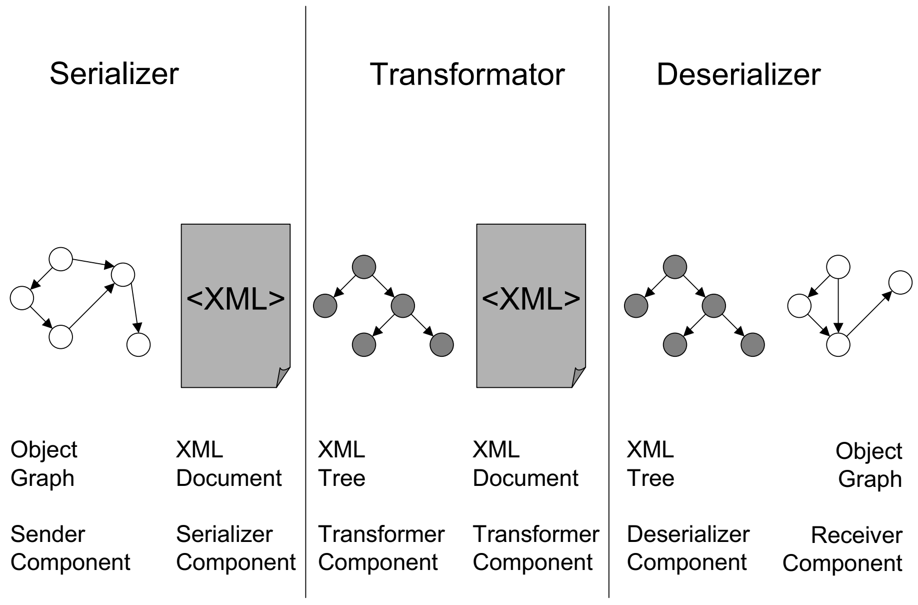

Figure 5: Meta structures (displayed in grey) in the connector implementation.

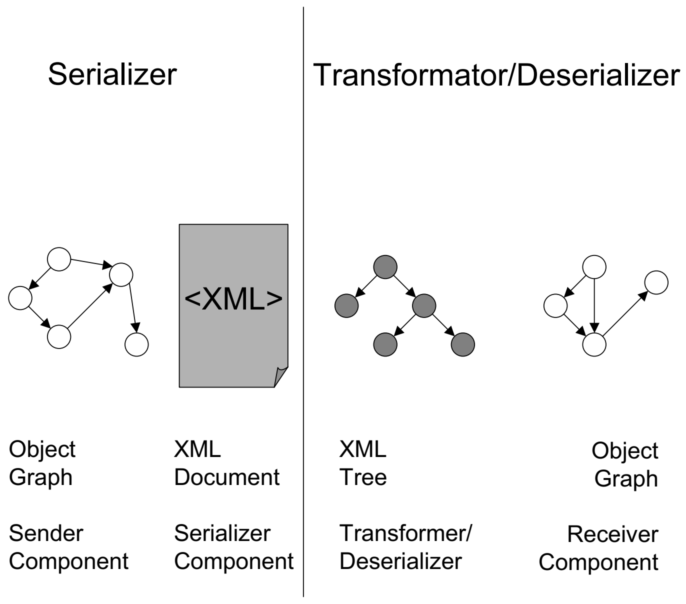

Figure 6: Meta structures that remain if the communication and adaptation processes are merged with the receiver process.

Hence, we can also omit the DOM structure for the output document. Instead of constructing an XML element in the transformation, we immediately call the appropriate constructor for the output structure. Figure 6 shows the remaining structures in the process.

Note, that above approach remains applicable also for XSLT 1.1, where snippets of the output can be assigned to variables and used later to construct the actual output document. Only for those snippets, we construct a DOM structure.

The second optimization applies if the sender and receiver component run in the same process. In this special case we have additional opportunities: there is no need for the serializer to generate a textual XML encoding of the source object structure. Instead, the serializer can construct its DOM representation directly. As before, the XML writing in the serializer and the XML parsing in the transformator disappear. Figure 7 shows the only remaining meta-structure in the process.

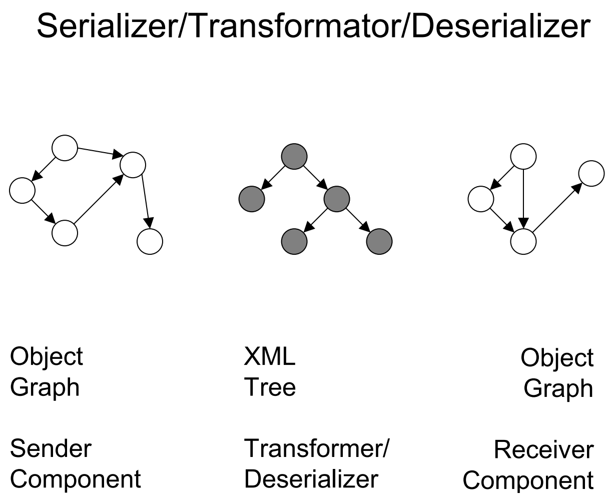

Figure 7: Meta structure that remains if the communication and adaptation processes are merged with the sender and receiver processes.

4.2 Optimizing Serializer and Transformer

In addition to the general optimization above, we optimize the communication by preprocessing the DTD and the XSLT scripts. If static analysis detects parts of the document type that are provably never visited by the transformation then there is no need to generate an XML representation for these elements, however. Filtering transformation extremely profit from this optimization.

5 Conclusion

As components are usually integrated into environments they are not customized for, the communicated data is to adapt to make the components interact correctly. We defined an XML based architecture for component connection with adaptation as an integral part.

The connectors are deployed automatically from specifications. We use type information from the components on the provided and required data in order to serialize and deserialize the XML representation of the data. XSLT specifications define the necessary transformations.

In contrast to standard XML processing we exploit the pre-agreed and specialized document types encoding special data type objects. This approach supports the design of transformations XSLT design tools benefit from precise input and output document types. Moreover, it improves the software process for the design of connectors in general introducing a notion of static type safety. Last but not least they facilitate to optimize serializer, deserializer and transformator.

The next step is the extension of the architecture and the deployment process to references in order to get full marshalling opportunities.

Furthermore, not all optimizations are implemented in our system. Especially the static analysis of the XSLT scripts does not exploit the abstract interpretation of the XSLT scripts on the DTD.

Future work will also focus on our optimization techniques at hand. If we could establish a lazy construction of the XML document encoding the sender object structure, we could go beyond the static analysis of the XSLT scripts. Whenever a XML element is visited by the XSLT processing for the first time, we generate this element from the object structure.

References

[1] *XML Parser for Java.* IBM AlphaWorks, `http://alphaworks.ibm.com/aw.nsf/techmain/xml4j`, 2001.

[2] *Xerces Java Parser.* Apache XML Project, `http://xml.apache.org/xerces-j/`.

[3] U. Aßmann, T. Genßler, and H. Bär. Meta-programming Greybox Connectors. In Richard Mitchell, Jean Marc Jézéquel, Jan Bosch, Bertrand Meyer, Alan Cameron Wills, and Mark Woodman, editors, *Proceedings of the 33th TOOLS (Europe) conference*, pages 300–311, 2000.

[4] *aXMLerate Project.* B2B Group, University of Karlsruhe, `http://i44pc29.info.uni-karlsruhe.de/B2Bweb/`.

[5] Len Bass, Paul Clement, and Rick Kazman. *Software Architecture in Practice.* Addison Wesley, 1998.

[6] *The Castor Project.* ExoLab Group, `http://www.castor.org/`, 2001.

[7] Li Chen, Elke Rundensteiner, Afshan Ally, Rice Chen, and Weidong Kou. Active page generation via customizing xml for data beans in e-commerce applications. *LNCS*, 2040:79–97, 2001.

[8] James Clark. *XT.* `http://www.jclark.com/xml/xt/`, 1999.

[9] Johannes Dieterich. *Generierung von Graphersetzern als XML-Transformatoren.* Universtität Karlsruhe, IPD Goos, Jun 2001.

[10] T. Gaul, W. Löwe, and M. Noga. Foundations of Fast Communication via XML. *Annals of Software Engineering — Special Volume on Object-Oriented Web-Based Software Engineering*, 2002. in revision.

[11] James Gosling, Bill Joy, and Guy Steele. *The Java™ Language Specification.* Addison-Wesley, 2 edition, 1996.

[12] Dirk Heuzeroth, Welf Löwe, Andreas Ludwig, and Uwe Aßmann. Aspect-oriented configuration and adaptation of component communication. In Jan Bosch, editor, *Third International Conference on Generative and Component-Based Software Engineering, GCSE*, page 58 ff. Springer, LNCS 2186, 2001.

[13] Michael Kay. *The SAXON XSLT Processor.* `http://saxon.sf.net/`, 2001.

[14] W. Löwe, R. Neumann, M. Trapp, and W. Zimmermann. Robust dynamic exchange of implementation aspects. In *TOOLS-Europe — Technology of Object-Oriented Programming*, 1999.

[15] Andreas Ludwig.*RECODER Homepage.* `http://recoder.sf.net/`, 2001.

[16] *Component Object Model.* Microsoft, `http://www.microsoft.com/com/`.

[17] Markus Noga. *Erzeugung validierender Zerteiler aus XML Schemata.* Universtität Karlsruhe, IPD Goos, `http://www.noga.de/markus/XMLSchema/Diplomarbeit.pdf`, Oct 2000.

[18] *Corba 2.4.2 Specification.* OMG, `http://www.omg.org/technology/documents/formal/corbaiiop.htm`.

[19] Tobias Schmitt-Lechner. *Entwicklung eines XSLT–Übersetzers.* Universtität Karlsruhe, IPD Goos, May 2001.

[20] M. Shaw and D. Graham. *Software Architecture in Practice – Perspectives on an Emerging Discipline.* 1996.

[21] *Enterprise JavaBeans 1.1 Specification.* SUN Microsystems, `http://java.sun.com/products/ejb/docs.html`.

[22] *Simple Object Access Protocol (SOAP) 1.1.* W3C Note 08 May 2000, `http://www.w3.org/TR/2000/NOTE-SOAP-20000508`, 2001.

[23] *Web Services Description Language (WSDL) 1.1.* W3C Note 15 March 2001, `http://www.w3.org/TR/2001/NOTE-wsdl-20010315`, 2001.

[24] *Extensible Markup Language (XML) 1.0.* W3C Recommandation, `http://www.w3.org/TR/1998/REC-xml-19980210`, 1998.

[25] *XML Schema Part 1: Structures.* W3C Recommendation 2 May 2001, `http://www.w3.org/TR/2001/REC-xmlschema-1-20010502`, 2001.

[26] *XML Schema Part 2: Datatypes.* W3C Recommendation 2 May 2001, `http://www.w3.org/TR/2001/REC-xmlschema-2-220010502`, 2001.

[27] *XSL Transformations (XSLT).* W3C Recommandation, `http://www.w3.org/TR/xslt`, 1999.

A Sample Type System

Most type systems in modern programming languages are roughly equivalent. For our example in this paper and for our running system, we chose the Java type system for its platform-independence. Any other object-oriented type system may be substituted instead.

The Java type system is defined in the Java language report [11]. It defines three kinds of types: primitive types, class types and array types.

The language report defines eight primitive types: `byte`, `short`, `int`, `long`, `float`, `double`, `char` and `boolean`. Users may not define additional ones. Primitive types are value types. A variable of a primitive type is a container for a single value of the type's value set. Values have no identity — two fives are always the same. I.e., assigning an integer five to a long variable simply assigns the value, five. There is no notion of subtypes, although some primitive types' value sets may contain others'.

There are two kinds of class types: interfaces and classes. The language report defines several class of them, most prominently `java.lang.Object`, but in contrast to primitive types, users may create both new interfaces and classes. A class consists of a set of method signatures S, a map $M : S \to I$ from signatures to implementations and a set of fields$F$, i.e., named and typed variables. Interfaces have neither implementations nor fields.

A class $C = (S, M, F)$ inherits from exactly one parent class $C' = (S', M', F')$ and arbitrarily many interfaces $C'', C''', \ldots$ (possibly none). Inheritance is implemented by union of the method signature sets $S, S', \ldots$ and union of the field sets F and F'. Where the new class does not define an implementation mapping $M(s)$ for a signature $s \in S$, the parent's mapping $M'(s)$ applies.

A class is either concrete or abstract. Interfaces are always abstract. Only concrete classes may be instantiated, i.e., objects of this class may be created at runtime. Each such object has an innate identity — it can be distinguished from another object of the same type containing the same values.

A variable of a class type contains a reference to an object of any subtype of the class type, or `null` . The subtype relation is simply the transitive closure of the inheritance relation.

Array types may be defined by the user. Every array type is characterized by its base type, which may be any arbitrary type. Arrays are always concrete and may be instantiated like classes. An array instance has innate identity and consists of a fixed-length sequence of containers of the base type. A variable of an array type can be assigned a reference to an instance of any subtype of the array type, or `null`. In Java, an array type is a subtype of another array type if this holds for their base types.

All predefined class types are subtypes of `java.lang.Object`. So are all arrays[1]. This makes all user-defined types subtypes of java.lang.Object. Therefore, the class and array types form a lattice with top element `java.lang.Object` and bottom element `null`.

[1]Object arrays are the only source of dimension mismatch: `Object[] x=new Object[y][]` is legal because all array instances are objects.

B Example

Consider the following Java class definitions. Although stripped of methods and constructors, they use most features of the type system: abstract classes, concrete classes, interfaces, inheritance and arrays.

```
abstract class A              { int       i; A a; }
interface B                   { }
class C extends A             { float     f; B b; }
class X extends C implements B { X         x; }
class Y extends X             { boolean[] b; }
class Z           implements B { X         x; }
```

Figure 8 shows an object graph built from these classes. We wish to pass it between component ports. The generator analyzes all classes and produces a DTD, serializers and deserializers.

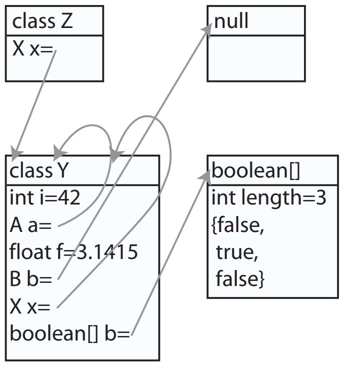

Figure 8: A sample object graph

Here, we reproduce the generated DTD without its invariant header. The effects of computing the transitive subtype closure are clearly visible in each class' entity definitions.

```
<!ENTITY  classY "(ref | classY)">
<!ELEMENT classY  (int, &classA;, float, &classB;,
                  &classX;, &arrayOfboolean;)>
<!ENTITY  classA "(ref | classY | classX | classC)">
<!--      class A is abstract -->
<!ENTITY  classX "(ref | classY | classX)">
<!ELEMENT classX  (int, &classA;, float, &classB;, &classX;)>
<!ENTITY  classC "(ref | classY | classX | classC)">
<!ELEMENT classC  (int, &classA;, float, &classB;)>
<!ENTITY  classZ "(ref | classZ)">
<!ELEMENT classZ  (&classX;)>
<!ENTITY  classB "(ref | classY | classX | classZ)">
<!--      class B is abstract -->
<!ENTITY  arrayOfboolean "( ref | arrayOfboolean )">
<!ELEMENT arrayOfboolean  ( boolean )*>
<!ATTLIST arrayOfboolean length CDATA #REQUIRED>
```

The generated `XMLSerializer` class contains one static serializer for every class, array and interface. It handles backward references and nulls, then uses virtual dispatches to serialize the appropriate runtime type. E.g., for `Y`:

```
public static void
serializeClassY(mlnoga.util.XMLSerializerStream s, Y o) {
  if(s.serializeReference(o))
    return;
   o.serializeXML(s);
}
```

Two additional methods are woven into `Y` itself. The first is invoked by the static serializer above. It serializes the runtime data type and invokes the second to serialize the data layout. Prior to serializing its own fields with the appropriate static serializers, this method invokes the superclass data layout serializer for `X`.

```
public void serializeXML(mlnoga.util.XMLSerializerStream s) {
    s.openingTag("<classY>");
    serializeBodyClassY(s);
    s.closingTag("</classY>");
}

protected final void
serializeBodyClassY(mlnoga.util.XMLSerializerStream s) {
    serializeBodyClassX(s);
    XMLSerializer.serializeArrayOfBoolean(s, this.b);
}
```

The following XML fragment is the serialization of `Z` in figure 8 performed by the generated code.

```
<classZ>
  <classY>
    <int>42</int>
    <ref>2</ref>
    <float>3.1415</float>
    <ref>0</ref>
    <ref>2</ref>
    <arrayOfboolean length="3">
      <boolean>false</boolean>
      <boolean>true</boolean>
      <boolean>false</boolean>
    </arrayOfboolean>
  </classY>
</classZ>
```