Getting Erlang to talk to the outside world

Joe Armstrong
Swedish Institute of Computer Science
Box 1263
SE-164 29 Kista, Sweden
joe@sics.se

ABSTRACT

How should Erlang talk to the outside world? - this question becomes interesting if we want to build distributed applications where Erlang is one of a number of communicating components.

We assume these components interact by exchanging messages - at this level of abstraction, details of programming language, operating system and host architecture are irrelevant. What is important is the ease with which we can construct such systems, and the precision with which we can isolate faulty components in such a system. Also of importance is the efficiency (both in terms of CPU and bandwidth requirements) with which we can send and receive messages in the system.

One widely adopted solution to this problem involves the XML family of standards (XML, XML-schemas, SOAP and WSDL) - we argue that this is inefficient and overly complex and propose basing our system on a simpler binary scheme called UBF (Universal Binary Format). The UBF scheme has the expressive power of the XML set of standards - but is considerably simpler.

UBF has been prototyped in Erlang - the entire scheme (equivalent in semantic power to the XML series of standards) was implemented in a mere 1100 lines of Erlang. UBF encoding of terms is also shown to be more space efficient than the existing "Erlang term format". For example, UBF encoded parse trees of Erlang programs are on average about 60% of the size of the equivalent ETS format encoding which is used in the open source Erlang distribution.

Categories and Subject Descriptors

C.2.2 [**Computer Communications**]: Network Protocols; D.1 [**Software**]: Programming Techniques; D.1 [**Software**]: Programming Languages

1. INTRODUCTION

We are interested in building reliable distributed systems out of asynchronously communicating components. We assume that the components are written in different programming languages, run on different operating systems and operate anywhere in the network. For example, some components may be written in Erlang, others in Java, others in C; the components might run on Unix, or Windows or Solaris.

We ask the questions "How should such systems interact?" and "Can we create a convenient language-neutral transport layer to allow such applications to be easily constructed?"

Suppose further that we have several different components and that they collaborate to solve some problem - each individual component has been tested and is assumed to work, and yet the system as a whole does not work. Which component is in error?

There are a number of conventional methods for solving parts of this problem, for example, we could use an interface description language (like Sun XDR [14] or ASN.1 [9]) or we could use a more complex framework like Corba [13]. All these methods have associated problems - many of these methods are supposedly language neutral but in practice are heavily biased to languages like C or C++ and to 32 bit word length processor architectures. The more complex frameworks (like Corba) are difficult to implement and are inappropriate for simple applications. Proprietary solutions for component interaction (like Microsoft's COM and DCOM) are not considered, since they are both complex and, more damagingly, not available on all platforms.

Out of this mess a single universal panacea has emerged - XML. The XML series of standards, notably XML[3], XML-schemas[6], [15] with SOAP[12], [7], [8] and WSDL[4] has emerged as the universal solution to this problem.

The XML solution involves three layers:

- *A transport layer* - XML provides a simple transport layer. XML encoded terms can be used to encode complex structured objects.
- *A type system* - XML schemas provides a type schema for describing the type of the content of a particular

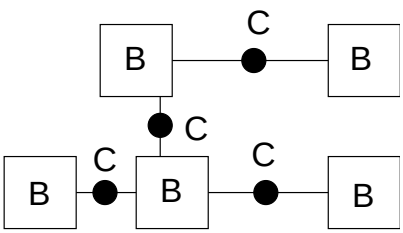

Figure 1: Black boxes and Contract Checkers

ACM SIGPLAN Erlang Workshop '02 Pittsburg, PA USA

XML tag.

- *A protocol description language* - SOAP defines how simple remote procedure calls can be encoded as XML terms. More complex interactions between components can be described using the Web Service Description Language (WSDL).

The above architectural layering is desirable, in so much that it separates transport of data (XML), the types of the data (XML-schemas) and the semantics of interactions between different components in the network (SOAP and WDSL).

Unfortunately, while the architecture is essentially correct, the details leave much to be desired. The individual components suffer from a number of significant problems.

We argue in the next section of the paper that XML is overly complex and overly verbose. Following this section we propose a simpler and more efficient but equally expressive binary format, which could be used as a complement to XML.

Our proposed schema has been implemented fully in Erlang and partially in Java and C - we present some preliminary results in the final section of the paper.

Our type system has an expressive power similar to that of the expressive power of XML-schemas, though we believe our scheme to be much simpler. Our contract language has many similarities to WSDL but again we believe it to be simpler and more expressive.

Our architecture also has many similarities to the `.NET` architecture, though we believe our architecture to be simpler and more powerful.

The remainder of the paper describes the system in detail gives some performance figures and describes our initial experience with the system.

2. PROBLEMS WITH XML

2.1 Complexity

XML, XML-schemas, SOAP and WDSL are a complex set of inter-related standards. A full implementation of the above standards requires many tens of thousands of lines of code and the implementation of a number of minor standards (like XML-name-spaces and XML-path) etc.

The XML standard itself has a grammar of 89 productions and requires many pages of explanatory text - entire text books have been written just to explain the (simple) standard. Having implement three XML parsers in Erlang I am in the position to say that XML is decidedly not simple to implement - amazingly, most of the complexity occurs in the implementation of a number of features which the vast majority of programmers will never use (these are antediluvian hang-backs to SGML).

The original design of XML had a notion of structure (described by a regular grammar) but no notion of type. Structure was described using DTDs (Data Type Descriptions) - but the DTD's did not themselves have an XML syntax. This was viewed by some as a disadvantage - XML-schemas came to the rescue - using XML-schemas XML structures could be described in XML itself, and a rich set of types was added.

What was been described by the XML DTD[5]

```
<!ELEMENT ROOT (A?,B+,C*)>
```

became in XML-schemas:

```
<element name="ROOT">
    <complexType content="elementOnly">
        <choice>
           <element ref="t:A">
           <sequence>
               <element ref="t:B">
               <element ref="t:C">
           </sequence>
        </choice>
    </complexType>
</element>
```

The notation for saying that the content of a tag should be of a particular type is equally verbose.

XML-schemas has 19 built-in (or primitive) types and three type constructors

The net result of this is that, if you want to express types you have to use XML-schemas. Unfortunately, the verbosity of the specification makes the schemas difficult to read.

In retrospect, a much simpler alternative would have been to extend XML with a small number of primitive data types.

For example, XML has a construct like:

```
<!ELEMENT xxx (#PCDATA)>
```

it would have been easy to extend this with expressions like:

```
<!ELEMENT xxx (#INTEGER32)>
```

Meaning that xxx is a 32 bit integer.

Such an extension would have provided a succinct and readable alternative to XML schemas.

2.2 Verbosity

XML encodings are *incredibly* verbose. The designers of XML excuse themselves with the words: "Terseness in XML markup is of minimal importance."[3] Unfortunately, the very verbosity of XML makes efficient parsing impossible, since at the very least the parser must examine every single input character. This property limits the usefulness of XML as a transport format for mobile devices with limited bandwidth.

Interestingly, one of the most common XML applications designed for such devices, namely WAP, uses an ad hoc method [11] to compress XML WAP programs, providing striking evidence that raw XML is inappropriate as a universal format for low-bandwidth devices.

Another strange property of XML is that binary data must be encoded prior to transmission. For example, a JPEG image must first be base64 encoded. Base64 encoding processes data in 24 bit groups, replacing each 3 byte sequence on input with a 4 byte sequence on output, lines are limited to 76 characters and only printable characters are transmitted.

This is all very strange and highly inefficient (especially considering that SOAP uses TCP/IP for data transport and TCP/IP itself is designed for efficient transport of binary data) - the bit about 76 characters probably has something to do with punched cards, and the restriction to printable characters has something to do with transmission systems that may only pass seven bits of a byte in a transparent manner.

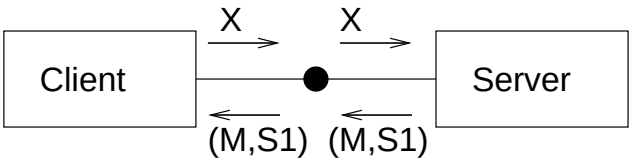

Figure 2: Client/Server with Contract Checker

Unfortunately the weird quoting rules of SGML apply to XML - you might naively think that binary data could be transmitted "as is" - unfortunately you can't just quote binary data in XML - if the binary data just happened to contain a valid XML end tag then chaos would ensue. Most programming language have quoting conventions which allow an arbitrary sequence of characters to be quoted, XML does not; thus, for example, any data can be placed within a CDATA block except data containing the string `]]>` - this fact severely limits the usefulness of CDATA section, making it impossible to (say) quote an arbitrary XML program - since it itself might contain a CDATA section. One wonders why such a convention was adopted.

3. OUR ARCHITECTURE

Our architecture is shown in Figures 1 and 2. Figure 1 shows a number of communicating components. The components are assumed to be black boxes, at any particular time a component might behave as a client or as a server. Between the components we place an entity which we call a *contract checker* (shown in the diagram as a black blob), the contact checker checks the legality of the flow of messages between the components.

We assume the contract checker starts in state `S` (see Figure 2); the client sends a message `X` intended for the server, the contract checker checks that `X` is of the correct type and that it is expected in state `S`, if so it is sent to the server. The server responds with a *Message* $\times$ *State* tuple `{M, S1}` the contract checker checks that this message is an expected response in the current state, if so `{M, S1}` is sent to the client and the state of the contract checker updated to `S1`.[1]

The contract checker is parameterised with a contract that specifies the ordering and types of the allowed message sequences between the client and the server.

The contract is is written using a simple non-deterministic finite state machine and a simple type language.

The contract is modeled as a set of four tuples of the form:

$$\{\mathcal{S}_{in}, \mathcal{T}_{in}, \mathcal{T}_{out}, \mathcal{S}_{out}\}$$

This means that if the server is in state $\mathcal{S}_{in}$ and it receives a message of type $\mathcal{T}_{in}$ then it may possibly respond with a message of type $\mathcal{T}_{out}$ and change its state to $\mathcal{S}_{out}$.

The contract checker assumes that the start state of the server is `start` this is assigned to some state variable `S`.

If the client sends the server a message `X` the contract checker checks that there are some rules in the contract where $\mathbf{S} = \mathcal{S}_{in}$ and *typeof*$(\mathbf{X}) = \mathcal{T}_{in}$ - if there are any such rules then the client is said to follow the contract and the message `X` can be safely sent to the server. If no such rules match, then the client is said to have broken the contract and both client and server are informed about this.

If the client has sent a valid message then the set of expected output responses of the server in pruned to a set of two tuples

$$\{\mathcal{T}_{out}, \mathcal{S}_{out}\}$$

being the allowed set of *Type* $\times$ *State* tuples that the server can respond with.

The server must respond with a `{Msg, State}` tuple - the contract manager checks if there is a tuple in the response set where $\mathbf{State} = \mathcal{S}_{out}$ and *typeof*$(\mathbf{Msg}) = \mathcal{T}_{out}$

If there is such a tuple then the response is accepted and `Msg` is sent back to the client and the global value of the state `S` is updated to `State`.

Note that the operation of the client/server in completely transparent in normal operation. In the case where both the client and server follow the contract no changes are made to the messages passed between the client and server - the only possible difference between client/server interaction using a contract checker and not using a contract checker is a a slight timing difference.

4. UBF - A UNIVERSAL BINARY FORMAT

Contracts are written in a langauge we call UBF. UBF has two components:

- **UBF(A)** is a data transport format, it is roughly equivalent to well-formed XML.
- **UBF(B)** is a programming language for describing types in UBF(A) and protocols between clients and servers. UBF(B) is roughly equivalent to verified XML, XML-schemas, SOAP and WDSL.

While the XML series of languages had the goal of having a human readable format the UBF languages take the opposite view and provide a "machine friendly" format.

UBF is designed to be easy to implement. As a proof of concept - UBF drivers for Erlang, Oz, Java and TCL can be found at the authors web site [1]. Implementors are welcome to add new languages.

UBF is designed to be "language neutral" - UBF(A) defines a language neutral binary format for transporting data across a network. UBF(B) is a type system for describing client/server interactions which use UBF(A).

5. UBF(A) - A BINARY TRANSPORT FORMAT

UBF(A) is a transport format, it is designed to be easy to parse and to be easy to manipulate with a text editor. UBF(A) is based on a byte encoded virtual machine, 26 byte codes are reserved. Instead of allocating the bye codes from 0 we use the printable character codes to make the format easy to read and edit.

Simplicity is the goal, so we define a minimal set of primitive types (four, compared with XML-schemas which has 19) and two types of "glue" for building complex types from more simple types.

[1]Note that this is unlike the convention RPC mechanism, where a server responds with a message in response to a particular query, and the next state of the server (if it is statefull) is *implied* by the protocol.

5.1 Primitive types

UBF(A) has four primitive types. When a primitive tag is recognized it is pushed onto the "recognition stack" in our decoder. The primitive types are:

Integers - integers are written as sequences of bytes described by the regular expression `[-][0-9]+`. That is, an optional minus (to denote a negative integer) followed by a sequence of at least one digits. No restrictions are made as to the precision of the integer, precision issues are be dealt with in UBF(B).

Strings - strings are written enclosed in double quotes, thus:

```
"...."
```

Within a string two quoting conventions are observed, `"` must be written `\"` and `\` must be written `\\` - no other quotings are allowed (this is so we can write a double quote *within* a string).

Binary Data - binary data is encoded, thus:

```
Int ~....~
```

First an integer, representing the length of the binary data is encoded, this is followed by a tilde, the data itself which must be exactly the length given in the integer and than a closing tilde. The closing tilde has no significance and is retained for readability. White space can be added between the integer length and the data for readability.

Constants - constants are encoded as strings, only using a single quote instead of a double quote.

Constants are commonly found in symbolic languages like Lisp, Prolog or Erlang. In C they would be represented by hashed strings. The essential property of an constant is that two constants can be compared for equality in constant time.

In addition any item can be followed by a *semantic tag* this is written `‘...‘`. This tag has no meaning in UBF(A) but might have a meaning in UBF(B). For example:

```
12456 ~....~ ‘jpg‘
```

Represents 12456 bytes of raw data with the semantic tag "jpg". UBF(A) does not know what "jpg" means - this is passed on to UBF(B) which might know what it means - finally the end application is expected to know what to do with an object of type "jpg", it might for example know that this represents an image. UBF(A) will just encode the tag, UBF(B) will type check the tag, and the application should be able to understand the tag.

5.2 Compound types

Having defined our four simple type we define two type of "glue" for making compound objects.

Structs - structures are written:

```
{ Obj1 Obj2 ... Objn }
```

The byte codes for `{` and `}` are used to delimit a structure. `Obj1..Objn` are arbitrary UBF(A) objects. The decoder and encoder must map UBF(A) objects onto an appropriate representation in the application programming language (for example structs in C, arrays in Java, tuples in Erlang etc.).

Structs are used to represent *Fixed numbers of objects*

Lists -lists are used to represent *variable numbers of objects.* They are written with the syntax:

```
# ObjN & ObjN-1 & ... & Obj2 & Obj1
```

This represents a list of objects - the first object in the list is `Obj1` the second `Obj2` etc.- Note that the objects are presented in reverse order. Lisp programmers will recognize `#` as an operator that pushes NIL (or end of list) onto the recognition stack and `&` as an operator that takes the top two items on the recognition stack and replaces them by a list cell.

Finally we need to know when an object has finished. The operator `$` signifies *end of object.* When `$` is encountered there should be only one item on the recognition stack.

5.3 White space

For convenience blank, carriage return, line feed, tab and comma are treated as white space. Comments can be included in UBF(A) with the syntax `%...%` the usual quoting convention applies.

5.4 Caching optimizations

So far we have used exactly 26 control, characters, namely:

```
%"~’‘{}#&\s\n\t\r,-01234567890
```

This leaves us with 230 unallocated byte codes. These are used as follows: The byte code sequence

```
>C
```

Where `C` is not one of the reserved byte codes or `>` means store the top of the recognition stack in the register `reg[C]` and pop the recognition stack.

Subsequent reuse of the single character `C` means "push `reg[C]` onto the recognition stack"

6. PROGRAMMING BY CONTRACT

Central to UBF is the idea of a *contract.* The contract regulates the set of legal conversations that can take place between a client and a server.

A software component (the contract checker) is placed between a client and server and it checks that all interactions between the client and server are legal.

The contract is written using types - the contract says (in a formal language) something like:

> "If I am in state S and you send me a message of type T1 then I will reply with a message type T2 and move to state S1, or, I will reply with a message of type T3 and move to state S2 ... etc."

The contract checker dynamically checks that both sides involved in a transaction obey the contract. Our contracts are expressing in a language we call UBF(B).

UBF(B) has:

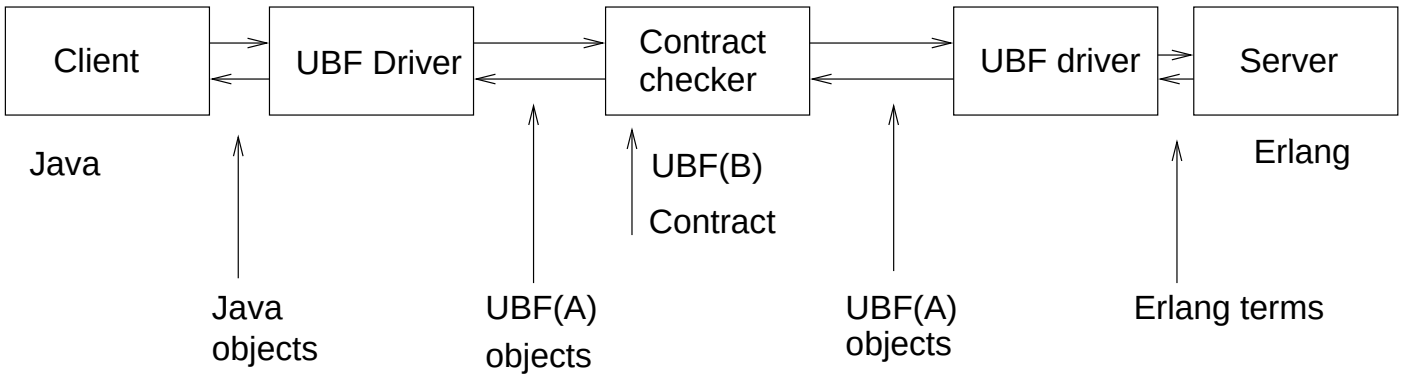

Figure 3: Client server with a contract checker.

A type system - for describing the types of UBF(A) objects.

A protocol description language - for describing client-server interaction in terms of a non-deterministic finite state machine.

An LALR(1) grammar for UBF can be found in appendix A.

6.1 The type system

The type system used here to describe the type of UFB(A) encoded objects is a simplified version of the type system used to describe Erlang terms[2]. The notation:

- `int()` Means a UBF(A) integer.
- `string()` Means a UBF(A) string.
- `constant()` Means a UBF(A) constant.
- `bin()` Means a UBF(A) binary data item.
- `X()` Means an Object of type `X`

UBF(A) literals are written as follows:

- `"..."` - denotes a UBF(A) string.
- `[a-z][a-zA-Z0-9_]*` - denotes a UBF(A) constant.
- `[-][0-9]+` - denotes a UBF(A) integer.

Complex types are defined recursively:

`{T1, T2, ..., Tn}` Is the *tuple type* if `T1` .. `Tn` are types. We say that `{X1, X2, ..., Xn}` is of type `{T1, T2, ..., Tn}` if `X1` is of type `T1`, `X2` is of type `T2`, ... and `Xn` is of type `Tn`.

$[T]$ Is the *list type* if `T` is a type. We say that `# Xn & Xn-1 & ... X2 & X1` is of type `[T]` if all `Xi` are of type `T`.

$T1|T2$ Is the *alternation type* if `T1` and `T2` are types. We say that `X` is of type `T1 | T2` if `X` is of type `T1` or if `X` is of type `T2`.

6.2 New types

New types are introduced in UBF(B) with the notation:

```
+TYPES X() = Type1; Type2; ...
```

Where `Type1`, `Type2`, ... are simple types, literal types or complex types.

Examples of types are:

```
+TYPES
  person()    = {person,
                 firstname(),
                 lastname(),
                 sex(),
                 age()};
  firstname() = string();
  lastname()  = string();
  age()       = int();
  sex()       = male | female;
  people()    = [person()].
```

This type schema defines a number of different types. For example, it is easily seen that:

```
'person' >p
# {p "jim" "smith" 'male' 10} &
  {p "susan" "jones" 'female' 14} & $
```

Is of type `people()`.

Note that unlike XML UBF(A) encoded terms do not contain any tag information. To make this clearer, suppose we had made an XML data structure to represent the same information, this might be something like:

```
<people>
  <person>
    <firstname>jim</firstname>
    <lastname>smith</lastname>
    <sex>male</sex>
    <age>10</age>
  </person>
  <person>
    <firstname>susan</firstname>
    <lastname>jones</lastname>
    <sex>female</sex>
    <age>14</age>
  </person>
</people>
```

The XML data structure contains a large number of redundant tags - whereas our representation omits all the tags The sizes of the first representation is 65 bytes and the second 215 (ignoring white space which is redundant) - we might thus expect that parsing the UBF expression would be at least three times as fast as parsing the XML expression.

Note that UBF(B) type is a *language independent type schema*. It defines the types of messages after encoding, and is thus universally applicable to any programming language which produces UBF encoded data.

Language independent type schemas are the basis of *Contracts* between clients are servers.

6.3 The Contract Language

We start with a simple example:

```
+NAME("file_server").
+VSN("ubf1.0").

+TYPES
info()         = info;
description()  = description;
services()     = services;
contract()     = contract;

file()         = string();
ls()           = ls;
files()        = {files, [file()]};
getFile()      = {get, file()};
noSuchFile()   = noSuchFile.

+STATE start
       ls()      => files()     & start;
       getFile() => binary()    & start
                 | noSuchFile() & stop.

+ANYSTATE
        info()        => string();
        description() => string();
        contract()    => term().
```

The program starts with a sequence of type definitions (these follow the `TYPES` keyword) - these define the types of the message that are visible across the interface to the component.

Here, for example we see type `getFile()` is defined as `{get, file()}` where `file()` is of type `string()`.

Given this definition it can easily be seen that the UBF(A) sequence of characters `{'get' "image.jpg"}$` belongs to this type.

Reading further (in the `STATE` part of the program) we see the rule:

```
+STATE start
     ls()      => files()     & start;
     getFile() => bin()       & start
               | noSuchFile() & stop.
```

In English, this rule means:

> If the system is in the state `start` and if it receives a message of type `ls()` then respond with a message of type `files()` and move into the `start` state, otherwise, if a message of type `getFile()` is received then either respond with a message of type `bin()` and move to the state `start`, or respond with a message of type `noSuchFile()` and move to the state `stop`.

To continue with our example, we requested a file named `image.jpg` the valid responses are of type `bin()` or `noSuchFile()` which corresponds to UBF(A) encoded sequences like `NNN~ ... ~$` or `'noSuchFile'$`.

Note that it might not always be possible for a component to distinguish between two different state transitions on the basis of the response alone. Consider the following fragment of a contract:

```
+TYPES running() = string();
       error()   = string().

+STATE running
       request() => ok() & running;
                 |  error() & stopping.
```

If we knew a component was in the state `running` and we sent it a message of type `request()` then we would expect it to respond with one of the types `ok()` or `error()` - unfortunately these types are indistinguishable, since both are represented as strings in UBF(A). For this reason we require that the server responds with a `State X Message` pair, not just a message. The server *explicitly* reveals its next state to the contract checker.

7. IMPLEMENTATION DETAILS

The entire UBF system has been prototyped in Erlang. The entire system is about 1100 lines of commented Erlang code.

- UBF encoding/decoding 391 lines.
- Contract parsing 270 lines.
- Contract checker and type checker - 301 lines.
- Run-time infrastructure and support libraries - 130 lines.

This compares favorably with the complexity of an XML implementation - as an example an incomplete implementation of XML which I wrote two years ago has 2765 lines of Erlang code. This should be compared with the 391 lines of code in the UBF implementation.

8. PERFORMANCE

So far, the system has been implemented entirely in Erlang and no thought given to embedding the UBF encoding/decoding software and the type checking software into the Erlang run-time system.

The only measure of performance we give here concerns the packing density of UBF encoded Erlang terms.

As a simple check we compared the size of the encoding of the parse tree of a number of Erlang modules, with the size of the a binary produced by evaluating the expression:

```
size(term_to_binary(epp:parse_file(F, [], [])))
```

The algorithm used to serialize the term representing the parse tree was a simple two-pass optimizer which cached the most frequently used constants which occurred in the program.

Based on a sample of 24 files we observed than on average the UBF(A) encoded data was 59 % of the size of the corresponding data when encoded in the Erlang external term format. In applications where bandwidth is expensive and communication relatively slow (for example, communication using mobile devices and GPRS) such a reduction in data volume would have considerable benefit.

9. FUTURE WORK

Our system of contracts uses only a very simple type system, it is easy to envisage extensions to allow more complex types and extensions to describe non-functional properties of the system.

The non-functional properties of the system are of particular interest, an example of these might be to add simple timing constraints, allowing rules such as:

```
+STATE S1
   T1 => T2 & S2 before Time1
      |  T3 & S3 after  Time2
...
```

meaning that if a component is in state `S1` and receives a message of type `T1` then it might respond with a message of type `T2` and change to state `S2` within `Time1` or, respond with a message of type `T3` and change state to a state `S3` after a time `Time2`.

Stricter contracts with timing constraints could be very useful in designing real-time systems of interacting components.

Other extensions could be imagined which would allow us to define contracts like subroutines - so that one contract you use a sub-contract to perform a specific task.

10. RUNNING THE SYSTEM

Since our system essentially exchanges characters, we can use telnet to observe a session and test the behaviour of the system. Here is an examples of commands issued in a telnet session where the client is talking directly to the file server specified by the `file_server` contract given above:

```
'info'$
{"I am a mini file server",'start'}$
```

Recall the the system starts in the state `start` the contract says that the `info` command can be sent in any state. The response should be a string, and the new state (in this case `start` since the state is not changed by an `ANYSTATE` rule).

The application returns a two tuple, containing a descriptive string and the new state. This is converted by the application driver to the UBF tuple `{"I am ... ", 'start'}$`.

```
'ls'$
{{'files',
  #
  "ubf.erl"&
  "client.erl"&
  "Makefile"& ...}
 'start'}$
```

Here the client sends a message of type `ls()` - the server responds with tuple `{{'files',#..., 'start'}$` message. This first element in the tuple is of type `files()`.

Finally we ask the system to describe itself:

```
'contract'$
{'contract',
 {{'name',"file_server"},
  {'info',"I am a mini file server"},
  {'description',"

Commands:

    'ls'$  List files
    {'get' File} => Length ~ ... ~
                 | noSuchFile
"},
  {'services',#},
  {'states',
   #{'start',
     #{'input',{'tuple',#{'prim','file'}&
        {'constant','get'}&},
     #{'output',{'constant','noSuchFile'},'stop'}&
      {'output',{'prim','binary'},'start'}&}&
     {'input',{'constant','ls'},
     #{'output',
        {'tuple',
          #{'list',{'prim','string'}}&
        {'constant','files'}&},'start'}&}&}&},
  {'types',
        #{'file',{'prim','string'}}&}}}$
```

The system responds to a message of type `info()` with a parse tree representing the contract itself.

In our contract itself we used the generic type `term()` to describe the contract. The contract itself is a well typed term in UBF, but a discussion of the abstracted form of the contract itself is not relevant to this paper.

The example is given to illustrate the *introspective* power of the system. Not only can we run the system, we can also ask the system to describe itself. We believe this to be a desirable property of a distributed component in a system of communicating components.

11. A LARGER CONTRACT

Our previous examples showed the basic syntax of a contract. We finish with a more complex example. The contract below describes an IRC[10] like protocol.

```
+NAME("irc").

+VSN("ubf1.0").


+TYPES

info()         = info;
description()  = description;
contract()     = contract;

bool()         = true | false;
nick()         = string();
oldnick()      = string();
newnick()      = string();
```

```
group()        = string();
logon()        = logon;
proceed()      = {ok, nick()}
listGroups()   = groups;
groups()       = [group()];
joinGroup()    = {join, group()}
leaveGroup()   = {leave, group()};
ok()           = ok;
changeNick()   = {nick, nick()}
%%               send a message to a group";
msg()          = {msg, group(), string()}
msgEvent()     = {msg, nick(), group(), string()};
joinEvent()    = {joins, nick(), group()};
leaveEvent()   =  {leaves, nick(), group()};
changeNameEvent() = {changesName,
                     oldnick(),newnick(), group()}.

%% I am assigned an initial (random) nick

+STATE start logon() => proceed() & active.

+STATE active

   listGroups() => groups() & active;
   joinGroup()  => ok() & active;
   leaveGroup() => ok() & active;
   changeNick() => bool() & active;
%%                 false if not in group
   msg()        => bool() & active;

   EVENT => msgEvent();        % Message from group
   EVENT => joinEvent();       % Nick joins group
   EVENT => leaveEvent();      % Nick leaves group
   EVENT => changeNameEvent(). % Nick changes name

+ANYSTATE
        info()        => string();
        description() => string();
        contract()    => term().
```

This example introduces a new keyword `EVENT`. The syntax:

```
+STATE S1
   ...
   EVENT => T2;
   ...
```

means that the server can spontaneously send a message of type `T2` to the client. Normally, messages are send to the client in reponse to requests, `EVENT` is used for asynchronous single messages from the server to the client. Since the server cannot be sure that the client has received such a message no change of state in the server is allowed.

12. EXPERIENCE

The initial version of UBF was completed in about three weeks of intensive programming - the system design changed was re-designed implemented and re-implemented several times.

Once the basic infrastructure was running a simple interface to Oz was implemented - and following this an interface to Java. The Oz and Java implementation only concerned UBF(A) and not the contract language or checker.

The first non-toy application (IRC) was implemented to test the system on a non-trivial example. I started by writing the contract and then made an Erlang client and server which followed the contract.

Interestingly the contract checker proved extremely helpful in developing the IRC system - I often develop systems by writing a client and server in the same time frame, shifting attention between the client and server as necessary. Using the contract checker proved helpful in rapidly identifying which of the two components was in error in the event of an error. Also, since the intermediate communication has a fairly readable ASCII subset I was able to test the server by typing simple text queries in a telnet session - in this way I was able to immediately test the server (and the interaction between the client and server) using telnet, rather than my Erlang code (which at some stages was only partially complete).

Interestingly the contract checker often complained about contract violations that I did not believe, so I erroneously assumed that the code for checking the contracts was incorrect. Almost invariably the contract checker was right and I was wrong. I think we have a tendency to believe what we had expected to see - and not that which was actually present - the contract checker had no such biases.

Concentration on the contact itself caused an interesting psychological shift of perspective and forced me to think about the system in meta-level terms considering the client and server as only stupid black box which did what they were told. Trying to write the contracts in a clear manner was also an exercise which resulted in a clearer understanding of the problem by forcing me to think in terms of what messages are sent between the client and server - and nothing else.

The contract proved also a valuable and easy to understand specification of the problem. Having implemented an Erlang client and server and a graphic based Erlang client we decided to add a Java client.

The Java client was developed independently by Luke Gorrie using only the UBF specification and the irc contract. When it came to testing the contract checker could provide extremely precise error diagnostics - of the form:

> I was in state $\mathcal{S}$ and I expected you to send me a message of type $\mathcal{T}$ but you sent me the message `M` which is wrong.

Armed with such precise diagnostics it was easy to debug the Java program. Needless to say when the Java client talked to the Erlang server the system worked first time. Testing both the Java client and the Erlang server could be done independently using only a modified form of the contract checker and the contract concerned.

Having developed the system we have a hight degree of confidence in it's correctness - and if it should fail we'll immediately know which component is broken.

13. ACKNOWLEDGMENTS

I would like to thank Seif Haridi, Per Brand, Thomas Arts, and Luke Gorrie for their helpful discussions - a particular thanks goes to Luke for implementing the Java client

APPENDIX

A. UBF GRAMMAR

```
form -> '+' 'NAME '(' string ')' dot.
form -> '+' 'VSN' '(' string ')'  dot.
form -> '+' 'TYPES' types dot.
form -> '+' 'STATE' atom transitions dot.
form -> '+' 'ANYSTATE'anyrules dot.

types -> typeDef ';' types.
types -> typeDef.

typeDef -> atom '(' ')' '=' type annotation.

annotation -> string.
annotation -> '$empty'.

type ->  primType '|' type.
type ->  primType.

primType -> 'int'      '(' ')'.
primType -> 'string'   '(' ')'.
primType -> 'constant' '(' ')'.
primType -> 'bin'      '(' ')'.
primType -> atom       '(' ')'.
primType -> '{' typeSeq '}'.
primType -> '[' type ']'.
primType -> atom.
primType -> integer.
primType -> integer '.' '.' integer.
primType -> string.

typeSeq -> type.
typeSeq -> type ',' typeSeq.

typeRef -> atom '(' ')'.

transitions -> transition ';' transitions.
transitions -> transition.

transition -> typeRef '=>' outputs.
transition -> 'EVENT' '=>' typeRef.

outputs -> responseAndState '|' outputs.
outputs -> responseAndState.

responseAndState -> typeRef '&' atom.

anyrules -> anyrule ';' anyrules.
anyrules -> anyrule.

anyrule -> typeRef '=>' typeRef.

strings -> string ',' strings.
strings -> string.
strings -> '$empty'.
```

B. REFERENCES

[1] J. L. Armstrong. Ubf - home page, 2002.

[2] J. L. Armstrong and T. Arts. A practical type system for erlang - erlang user conferense, 2002.

[3] T. Bray, J. Paoli, C. M. Sperberg-McQueen, and E. M. (Eds). Extensible markup language (xml) 1.0 (second edition), october 2000, http://www.w3.org/tr/2000/rec-xml-20001006, 2000.

[4] E. Christensen, F. Curbera, G. Meredith, and S. Weerawarana. Web services description language (wsdl) 1.1, march 2001, http://www.w3.org/tr/2001/note-wsdl-20010315/, 2001.

[5] D. Connolly, B. Bos, Y. Koike, and M. Holstege. `http://www.w3.org/2000/04/schema_hack/`, 2000.

[6] D. C. F. (Ed). Xml schema part 0: Primer, may 2002. http://www.w3.org/tr/2001/rec-xmlschema-0-20010502/, 2002.

[7] M. Gudgin, M. Hadley, J.-J. Moreau, and H. F. Nielsen. Soap version 1.2 part 1: Messaging framework, december 2001, http://www.w3.org/tr/2001/wd-soap12-part1-20011217, 2001.

[8] M. Gudgin, M. Hadley, J.-J. Moreau, and H. F. Nielsen. Soap version 1.2 part 2: Adjuncts, december 2001, http://www.w3.org/tr/2001/wd-soap12-part2-20011217., 2001.

[9] ISO/IEC. Osi networking and system aspects - abstract syntax notation one (asn.1). ITU-T Rec. X.680 — ISO/IEC 8824-11, ISO/IEC, 1997.

[10] D. R. J. Oikarinen. RFC 1459: Internet relay chat protocol, May 1993.

[11] B. Martin and B. J. (Eds). Wap binary xml content format, june 1999, http://www.w3.org/tr/wbxml, 1999.

[12] E. Nilo Mitra. Soap version 1.2 part 0: Primer, december 2001, http://www.w3.org/tr/2001/wd-soap12-part0-20011217, 2001.

[13] OMG. *Common Object Request Broker Architecture (CORBA)—v2.6.1 Manual*. The Object Management Group, Needham, U.S.A, 2002.

[14] R. Srinivasan. RFC 1832: XDR: External data representation standard, Aug. 1995.

[15] H. S. Thompson, D. Beech, M. Maloney, and N. M. (Eds). Xml schema part 1: Structures. w3c recommendation, may 2001. http://www.w3.org/tr/2001/rec-xmlschema-1-20010502/, 2001.