SERC-0017
March, 1996
Issue 1

Perspectives on Erlang & TINA

Maurice Castro

Copies of this document may be obtained by contacting:

Director
SERC
723 Swanston St
Carlton, Victoria, 3053

Prepared by the Software Engineering Research Center.

Abstract

This paper assesses the viability of using the functional programming language Erlang to implement TINA blocks. It identifies potential changes required in both Erlang and TINA to acheive systems which are better suited to interfacing. The changes are identified by focusing on the TINA IDL and TINA's handling of data structures used to exchange information, and examining a number of mechanisms to interface Erlang to the data structures currently provided by TINA.

The contents of this technical report have been submitted to the TINA '96 Conference.

1 Introduction

Telecommunications Information Networking Architecture (TINA) provides an architectural basis for telecommunications systems. TINA systems are constructed from units known as building blocks or blocks. This paper assesses the viability of using the functional programming language Erlang to implement TINA blocks and examines the issue from two perspectives represented by the two questions:

- What changes are required to TINA to support Erlang?
- What changes are required to Erlang to support TINA?

The paper focuses on the TINA architecture's handling of data structures used to exchange information, and examines a number of mechanisms to interface Erlang to the data structures currently provided by TINA.

Initially, the paper describes TINA's existing mechanisms for interfacing components of a system, and the gross features of the Erlang language. Subsequent sections identify: techniques which can be used to interface Erlang to TINA without alteration to TINA and the trade-offs required; limitations inherent to the TINA interface and changes recommended to the TINA interface; and an evaluation of changes that could be made to the Erlang language.

2 TINA

TINA [2] provides a framework for services in the telecommunications domain. It addresses the requirements of voice-based, multimedia, and information services over a wide variety of media. In addition, it provides operations and management functions to control the network and the services offered by the network.

The architecture employs distributed computing concepts. Mechanisms for interoperability, software re-use and flexible data placement, and task assignment are used to provide computational support to implement the services.

TINA is being developed by an international consortium known as TINA-C. While most of the information relating to TINA is the proprietary property of the members of that consortium, this document is based on information extracted from publicly released drafts of TINA-C documents.

Only the computational aspects of the TINA architecture are addressed here. Issues relating to the provision of services, network design, management of the network, or task assignment are not covered. It is concerned specifically with the implementation of computational objects and interactions between computational objects.

Computational objects interact with each other through defined interfaces. There are two major classes of interface present in TINA:

Operational interfaces provide defined operations for a server object to offer functions to a client object. Operations have arguments and return results.

Stream interfaces are used for passing structured information such as video or voice bit streams. These interfaces do not support any operations

The operational interfaces are used to communicate data and commands between objects. Connections between stream interfaces are made by communicating with the network.

2.1 Objects, Blocks and Interfaces

A computational object is used to represent data and a set of operations on that data. Access to the data contained within an object is through the use of a computational interface. Interfaces consist of a set of defined operations which may be invoked by other objects. An object may have many interfaces.

Building blocks are formed from a collection of objects (see figure 1). They are the fundamental structuring unit of computation within TINA. Blocks provide contracts which are used by other blocks to construct services. Blocks must contain at least one object which supports contracts for the management of objects within a block. All the objects of a block are required to be co-located. Objects can only be moved when the block that contains them is moved.

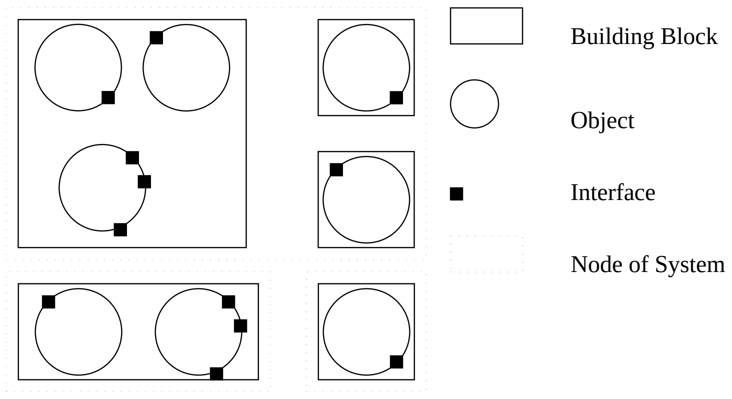

Figure 1: Relationship between Objects, Blocks, and Interfaces

2.2 Data Structures

The data structures used to communicate between TINA blocks are specified using a language known as TINA-ODL (TINA Object Definition Language). TINA-ODL [6] is a superset of the IDL language [8] developed by the Object Management Group (OMG). The features added provide facilities required for the telecommunications domain.

TINA's object description language supports a number of fundamental data types and compound data types. These data structures are used in the formal part of the contracts that describe the form of the input and that of the output of a block.

Table 1 lists the fundamental data types supported by TINA-ODL. Table 2 lists the compound types. Data type definitions are taken from IDL language definition [8].

A significant feature of the description language is that although the syntax allows recursive types to be specified, these types are not permitted by the semantics of the language. Recursive types may only be specified through the use of the *sequence* compound type.

The syntax and semantics of arrays have been borrowed from C: The size of an array is specified by a single parameter, and arrays start at index **0** and end at $size - 1$.

short	- signed short integer $(-2^{15} \dots 2^{15} - 1)$
long	- signed long integer $(-2^{31} \dots 2^{31} - 1)$
unsigned short	- unsigned short integer $(0 \dots 2^{16} - 1)$
unsigned long	- unsigned long integer $(0 \dots 2^{32} - 1)$
float	- single-precision floating point number conforming to IEEE 754
double	- double-precision floating point number conforming to IEEE 754
char	- 8-bit quantity where ISO Latin-1 characters and some graphic and formatting characters are defined. The meanings of other character values are implementation dependent. The characters may be translated in transmission to retain meaning.
octet	- an 8-bit quantity that is guaranteed not to be translated when transmitted.
enum	- enumerated names are mapped to a data type which is capable of representing 2^{32} values. The order of identifiers in the specification determines the relative order of identifiers.
boolean	- a quantity taking only the values **TRUE** and **FALSE**
string	- a series of 8-bit quantities excluding null.
any	- any TINA-ODL object

Table 1: Fundamental data types provided by TINA

struct	- defines a record containing fields of specified types.
sequence	- a list built from elements of a specified type. It is implemented as a one dimensional array with a maximum size determined at compile time and a length determined at run time.
array	- collections of a specified number of instances of a data type. Arrays may be multidimensional.
union	- a collective data structure that can have multiple internal structures. A switch type specifier is used to specify the interpretation of the contents of the union by identifying the structure used.

Table 2: Compound data types provided by TINA

2.3 Exception Handling

The exception mechanism is based on the OMG-IDL notion of exceptions [9, page 7-4]. The exceptions data structure resembles a struct. These structures are used to return information relating to exceptional conditions which may occur during a request.

2.4 Using Objects

To access an interface, an object must acquire an **interface reference**. These are created when an object offering an interface is created, or they can be passed to an object by another object.

TINA supports two classes of interaction between clients and servers using interfaces:

Interrogations require the client to pass zero or more arguments to the server. The invocation is then processed by the server, and any results are returned. Finally the client is informed of the success or failure of the invocation.

Announcements are similar to interrogations except that results are not returned, and the client is not informed of the outcome of the invocation.

Interrogations with multiple terminations [9] are specified as having a single 'normal' termination. Results are returned via *out* and *inout* parameters in the specification. An interface also has a group of 'abnormal' terminations which return results using the exception mechanism.

When an object is created, an interrogation operation is implicitly invoked. This operation is invoked only when the object is instantiated. The operation may accept one or more parameters.

Interfaces may support several operations. An operation on an interface appears as a function-call in a program invoking the operation. The following are specified when an operation is defined:

- the type of the result returned by the operation.
- the name of the operation.
- the parameters of the operation.
- a list of exceptions that can be raised by invoking this operation.

There are three types of parameters available to an operation: *out* - only outputs results, *in* - only accepts input, and *inout* - parameter may be passed both into and back from the operation.

3 Erlang

The Erlang language [1] is a minimalist functional language employing pattern matching for rule selection. This paper uses the language description provided in [1]. Although the description does not reflect the latest version of the language, it contains all the basic concepts of the language and is sufficiently complete for use in the evaluation of the potential interaction of Erlang and TINA.

Two significant features of the language's handling of variables are that it supports only a single assignment to variables, and it requires the scope of variables to be restricted to a clause of the function.

The language has support for concurrent programming and is suitable for soft real time applications.

integers - positive and negative integers must be representable with at least a 24 bit range.

floats - floating point values with at least 24 bits of precision.

atoms - a constant with a name. Atoms may be quoted and hence may include any character. Unquoted atoms must begin with a lower case letter.

PIDs - a process ID; Uniquely identifies a process within a system.

references - an object which is guaranteed to be unique. Comparison for equality is the only operation permitted on this type.

Table 3: Fundamental data types provided by Erlang

tuples - a structure which stores a fixed number of items

lists - a structure which can store a variable number of items

Table 4: Compound data types provided by Erlang

3.1 Data Types

Table 3 lists the fundamental data types provided by Erlang. Table 4 lists the compound data types.

Erlang does not currently have a syntactic mechanism for describing types. The language is dynamically typed. The structuring elements of the language - *tuples* and *lists* - use position within the structure to identify fields rather than names.

3.2 Messages

Erlang has an inbuilt Interprocess Communication Mechanism (IPC). This mechanism allows messages to be sent from one Erlang process to another Erlang process (see figure 2). The Process ID (PID) is used to identify the target process for the *send* operation. Messages may contain any Erlang structure. Message passing mechanism is asynchronous in nature. Sending a message does not block the sender, and messages are stored in a mailbox associated with each process. Messages are retrieved using pattern matching. The first message in order of arrival at the process which matches the pattern is retrieved.

Work is being conducted into program design techniques for Erlang which are based on Communicating Finite State Automata (CFSA). Messages are a critical part of the implementation of CFSAs in Erlang.

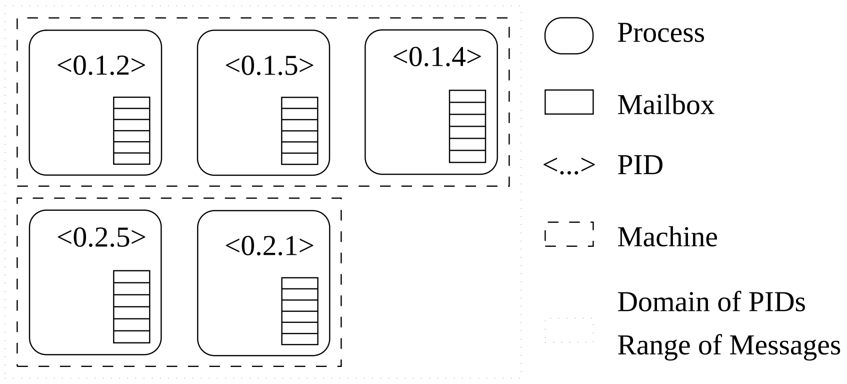

Figure 2: Relationship between Processes, Machines and Messages

4 Compatibility of Erlang & TINA

Erlang and TINA are drawn from two widely differing traditions. TINA's IDL is well suited to C and C++ applications. Both these languages are imperative, statically typed, and have a wide range of fundamental types available. Erlang is drawn from a minimalist functional school. The language is based on a small number of orthogonal types using only two structuring elements.

4.1 Fundamental Data Types

While TINA is rich in types, the collection of fundamental types offered is not orthogonal. For example, the IDL supports 4 types of integers, and 2 types of floating point value. Erlang is more frugal in providing data types. It supports a single type of floating point value and a single type of integer.

Erlang has no provision for the octet data type. Characters are handled as atoms with names of one character in length. Strings are also implemented as atoms.

The *enum* type provided by TINA has no equivalent in Erlang as an ordering is imposed on the values of an *enum*. The value is defined by the order in which the strings are specified in the *enum*. The closest equivalent in Erlang is to use the atom data type. The atom data type has the same semantics as *enum* for equality operations, but as the ordering on atoms is lexicographical, inequalities have different semantics.

There is no boolean type present in Erlang. It may be represented using the atom *true* and a test for equality.

The *any* type provided by TINA is implicit in Erlang as types are determined dynamically and are only significant if an operation is performed on an item unsuited to the operation.

Erlang's PIDs are not explicitly present in TINA's IDL; however, they are implicitly recognized as they would form part of the interface reference.

Erlang's reference type is not supported by IDL.

4.2 Compound Data Types

Erlang currently supports two compound data types - *tuples* and *lists* - while TINA's IDL supports 4 types - *array*, *sequence*, *struct* and *union*. The mismatch in the number of structuring elements and the structuring mechanisms requires Erlang to use its structuring mechanisms to represent more than one of the TINA types.

Furthermore, Erlang programs have to adopt a convention in representing TINA types so they can be distinguished within the source code.

Only the semantics of Erlang's list mechanism correspond well to the sequence mechanism in TINA. None of the other mechanisms in TINA have a direct correspondence to an Erlang mechanism.

Tuples are Erlang's closest match for TINA's *struct* data type. The semantics of the struct data type are based on the use of names to identify fields. This is at variance with Erlang's use of position to identify fields. For example the IDL declaration:

```
struct equiv {
   short a;
   long b;
};
```

is equivalent to

```
struct equiv {
   long b;
   short a;
};
```

and programs written in C and C++ using either of those declarations would be identical. The ordering of a *tuple* is essential in Erlang programs, and changing the order would result in major rewriting of the code. A more dramatic difference would be observed should a new field be added. While a C or C++ program would be unaffected, an Erlang program would need significant modification. This is caused by the pattern matching mechanism used in Erlang: *tuples* can only be matched to *tuples* of the same length.

Unions are naturally represented using *tuples*. To represent *unions* as *tuples* it is necessary to store the *switch* type which identifies the structure being used in the *tuple* to ensure that expressions which have the same structures, but different types, are distinguished. For example, the following IDL description for a point:

```
union Point switch (PointType) {
   case i:
      struct Ipoint {
         short ix;
         short iy;
      };
   case f:
      struct Fpoint {
         float fx;
         short fy;
      };
};
```

could be represented as two types of *tuple*. An example of each class would be:

```
{i, {3, 4}}
{f, {3.0, 4.0}}
```

Erlang is well suited to dealing with this type of structure as it can use pattern matching to select rules based on the contents of the *union*. The following is an example of an Erlang routine to move a point by one unit in the x-direction:

```
upx({i, {X, Y}}) -> {i, {X+1, Y}};
upx({f, {X, Y}}) -> {f, {X+1.0, Y}}.
```

Erlang has no indexed data types. There is no direct equivalent for the TINA *array* data type. However, it can be implemented using a *list* in Erlang. If indexed access to the *list* is required, functions can be written which return the nth element of a *list*.

4.3 Interfaces

The interface mechanisms of Erlang and TINA are poorly matched. TINA identifies function parameters as one of the three types: *in*, *out* and *inout*. Other information is passed using the exception mechanism. Erlang only allows function parameters to be used to pass values into a function. The return value of a function is used to output results.

The IDL mechanism appears to be well suited to implementation in C as pointers can be used for *out* and *inout* parameters to allow values to be modified in situ. The IDL mechanism is incompatible with the Erlang approach. Many of the fine grained parallelism benefits offered by functional languages are dependent on functions having immutable parameters. By ensuring that parameters cannot change, multiple operations in a program clause can be invoked at one time. Furthermore, dependency analysis is simplified, allowing independence between clauses to be easily identified, providing greater opportunities to enhance the parallelism of a program.

Common Lisp has been mapped to OMG's IDL [7]. The solution proposed for passing variables to and from a module implemented as a Lisp function to the caller of the interface are applicable to Erlang. Under Lisp, parameters are passed into the function in the same order as in the IDL declaration, but the *out* parameters are excluded. Return results are passed by using the *multiple-values-bind* macro, with the first value representing the function result, the second parameter, the exception value, and the subsequent values corresponding to each of the *inout* and *out* parameters in the order of declaration in the IDL description.

Either a macro preprocessor or a change in the Erlang language would be required to rewrite the output values and return results to the calling interface.

An alternative method would be to return a tuple from the interface function (map_curpos in the example) which would contain the return value, exception value, and all out or inout parameters in the order of declaration in the IDL declaration. This method has the advantage of avoiding the need for changes to the language.

The Common Lisp implementation also yields a mechanism for naming functions. Under Lisp, the names of entry point functions are derived from the interface name followed by a hyphen then the name of the operation. As the hyphen is the subtraction and negation operator in Erlang, substituting the underscore ('_') is appropriate.

An implementation in Erlang based on the Lisp concepts might appear as follows:

```
// IDL
interface map {
    void    curpos(in long t,
                   out long x,
                   out long y);
};

% Potential Erlang Implementation
map_curpos(T) ->
    {R, X, Y} = find(T),
% ...
    [R, normal, X, Y].
```

4.4 Exceptions

The exception mechanism in TINA is based on the assignment of values to an exception data structure, and returning an exception state. Erlang only allows a single assignment to a variable, and restricts the scope of a variable to a clause. This makes the assignment mechanism in Erlang unsuitable for use in this context. A new mechanism is required to support the semantics of Erlang[1].

Two possible solutions include using records in the process dictionary, and introducing a set of functions which set fields outside the Erlang module to the required values to indicate an exception. If the process dictionary is used to store the value of an exception's parameters, then the value of the exception must be recovered from the process dictionary before the process is terminated. This can be accomplished by calling a function in the Erlang module that examines the appropriate entries in the dictionary. The second solution requires the introduction of a new set of BIFs to the language.

4.5 Erlang Messages

The message mechanism provided by Erlang is only poorly compatible with the interface mechanisms provided by TINA. TINA models all computational communication between blocks using function calls. At a syntactic level, functions can be provided which allow messages to be sent and received from Erlang modules and a binding between Erlang PIDs and instances of a TINA block can be provided. There are two objections to supporting messages between Erlang processes and processes outside the current TINA block. The first is that the extremely low overhead processes required by Erlang would be burdened by the binding mechanism required by TINA. The second objection is that a new mechanism would need to be introduced into TINA. Furthermore, the new mechanism would mix the functional metaphor with the message metaphor in non-Erlang blocks, potentially leading to confusion.

5 Evaluation of TINA IDL

The interface description language employed by TINA is limited by its C and C++ heritage. The choice of data types for the IDL, the method of passing parameters and returning results, and the exception mechanism are all closely modeled on C mechanisms. This allows C and C++ programs to be written which are easily shown to be immune from artifacts of translation and the loss of efficiency caused by translation. However, the non-generic nature of the interface makes implementation of TINA blocks in languages other than C and C++ unwieldy.

The current IDL is not suited to interfacing with conventional languages such as COBOL [4]. A critical problem in accessing COBOL is the observation that PIC statements in COBOL are not accurately reproducible in IDL. For instance PIC 9(11) is not representable in the IDL as it represents a range of values that exceeds the capacity of TINA's long type. Furthermore, this is a practical problem as the integer size provided by TINA's IDL is inadequate as it only allows 42 million dollars worth of cents to be represented. This is inadequate to represent turnover in the accounting modules for many telecommunications functions. These problems are not limited to COBOL, for instance Sybase's money type [10] can represent 9^{14} dollars accurately with $\frac{1}{100}$ of a cent precision.

From the point of view of interfacing to languages other than C and C++ a more generic IDL would be desirable. A single type of integer and a single size

[1]Lisp differs from Erlang in that the *setq* operator, unlike Erlang's assignment, has a global effect and can be used to assign elements to a structure

of floating point value, would be provided. These values would be defined to be the same as the larger of the float and signed integer types provided. This would introduce a conversion cost to C and C++ programs, but, that cost would be minor compared with other elements of the communications mechanism. Although a BCD type might be considered desirable, the current string mechanism can be used to fulfill the function of passing large fixed point values throughout the system.

The current mechanism which uses parameters to pass both data into and out of a function is well suited to languages which allow data to be updated in situ, but it is poorly suited to functional languages. IDL appears to have adopted the view that interfaces are procedures, rather than functions. The IDL also assumes the presence of either pointers or *var* parameters. A more general mechanism would eliminate *inout* and *out* parameters, hence providing only *in* parameters to a function. The current mechanism of returning results from an interface uses the *inout* and *out* parameters present in the parameter list of the function. A preferable mechanism for returning data would be to return a structure (or pointer to a structure depending on the language[2]) from a function. The structure would contain the return state of the module, all the out parameters, and the values of the exception fields. This mechanism entails the cost of generating the structure but has the advantage of being more generally applicable than the current mechanism. The removal of *inout* parameters may result in an increase in storage space requirements but is likely to improve program quality by removing side effects from interface calls.

6 Evaluation of Erlang

Erlang is currently a small language well attuned to its target environment. The language has minimised syntactic sugar and has a small number of fundamental types.

To operate within the TINA environment, Erlang must either introduce an *octet* type, or specify that the names of atoms are not translated by software. An *octet* type would be the preferred solution as it differentiates octets from human readable strings and allows strings to be represented in local character sets.

A perceived difficulty with using Erlang in the context of TINA is the requirement that major modifications must be made to programs when the number of fields of a tuple is either increased or decreased. A possible solution for this problem is the introduction of a structure which allows the programmer to name the elements of the structure[3]. However, this approach is probably inappropriate Erlang. Currently Erlang supports no user defined types[4], and type checking is a simple process conducted when a value is used by a BIF or an operator. The introduction of a user defined type will require significant modifications to the language to allow the declaration of the type. Furthermore, it is simple to arrange for an Erlang program to consist of a public interface (sensitive to changes in structures) and a working component (insensitive to changes in structures). The public interface would translate the passed structures into an internal form to be manipulated by the working component.

[2]Both ANSI C [5, page 225] and C++ [3, page 138] permit structures to be returned

[3]Erlang version 4.3 introduced the concept of records. This feature is currently unofficial and undocumented and it appears to be implemented using a preprocessor.

[4]The type of a tuple may be considered the number of elements present in a tuple. The number of elements in a tuple is far less restrictive than typical of user defined types in other languages

7 Conclusion

At the present time Erlang is well suited for use as a language for application to TINA. The language is simple and avoids constructs likely to encourage errors.

Erlang currently does not have support for the *enum* type provided by TINA. The current semantics of *enum* type place an ordering on the elements of the enumeration. If only equivalence operation is required over *enum*, Erlang can support it. However, if ordering is required, Erlang currently cannot support this data type.

Currently an IDL description can be written for any Erlang program to accurately represent the actions of the program. The converse is untrue.

Erlang currently lacks an *octet* type. To interface with TINA, Erlang must either provide an octet type or treat all strings (atoms) and characters as octets. Octets are guaranteed not to be translated when transmitted.

TINA's current IDL is not well suited to interfacing with languages other than C or C++. By reducing the number of fundamental types, removing *inout* parameters and using the function as the basic model of interaction rather than a procedure, the IDL would be better placed for interfacing with other languages.

References

[1] Joe Armstrong, Robert Virding, and Mike Williams. *Concurrent Programming in Erlang*. Prentice-Hall, 1993.

[2] Martin Chapman and Stefano Montesi. Overall concepts and principles of TINA: Version 1.0, 2 1995. TB_MDC.018_1.0_94 Publicly Released.

[3] Margaret A. Ellis and Bjarne Stroustrup. *The Annotated C++ Reference Manual*. Addison-Wesley, 1990.

[4] Dave Gamble. 95-3-36: Extending IDL for COBOL and other languages, 3 1995. http://www.omg.org/docs/95-3-36.txt, document.

[5] Brian W. Kernighan and Dennis M. Ritchie. *The C Programming Language*. Prentice Hall, second edition, 1988.

[6] B. Kitson, P. Leydekkers, N. Mercouroff, and F. Ruano. TINA object definition language (TINA-ODL) MANUAL: Version 1.3, 6 1995. TR_NM.002_1.3_95 Publicly Released.

[7] Tom Mowbray and K. L. White. 94-3-11: OMG IDL mapping to Common Lisp, 12 1994. http://www.omg.org/docs/94-3-11.ps.

[8] Pramila Mullan. 95-1-17: First ODP IDL standalone draft, 12 1994. http://www.omg.org/docs/95-1-17.ps.

[9] N. Natarajan, F. Dupuy, N. Singer, and Christensen H. Computational modelling concepts: Version 2, 2 1995. TB_A2.HC.012_1.2_94 Publicly Released.

[10] Sybase Inc. *Student Guide*, 1992. updated: 92-4-6.