Towards an Event Modeling Language

Maurice Castro
maurice@serc.rmit.edu.au

Michael Dwyer
dwyer@serc.rmit.edu.au

Geoff Wong
geoff@serc.rmit.edu.au

Software Engineering Research Centre
Level 3, 110 Victoria St
Carlton, VIC, 3053, Australia

Abstract

Object-oriented programming owes part of its popularity to Booch's notation [2], Rumbaugh's OMT [7], and UML [8]. These notations allow object oriented designs to be expressed graphically and furthermore have enabled the development of case tools for object oriented languages.

The lack of a suitable high level graphical notation has been identified as one of the factors discouraging the uptake of functional programming and in particular the language Erlang [1].

This paper represents a first step in designing a graphical modeling language for functional programming that encourages sound programming practices. The initial target language is Erlang but it is hoped that the notation can be extended to other functional languages.

1 Introduction

Graphical notations such as Booch's notation [2], Rumbaugh's Object Modeling Technique (OMT) [7], and the Unified Modeling Language (UML) [8] have provided a strong impetus to the Object-oriented (OO) programming paradigm.

The absence of a high level graphical representation for functional programming has been identified as one of the factors inhibiting the uptake of the Erlang [1] programming language.

Graphical design notations allow designers to specify programs at a high level in a manner more closely related to the programming paradigm rather than the target language of the project. The notation should aid communications throughout the product life-cycle, but in particular they should aid communications between designers and implementors by highlighting design issues and leaving programmers free to handle the detail of the implementation. Finally these notations are instrumental in the construction of case tools by providing a high level notation which can be translated into programming language constructs.

This paper describes the first steps towards a graphical notation for the Erlang programming language.

In the following sections the Erlang programming language is summarised and its significant features identified, existing techniques are discussed, the objectives of the notation are discussed, the notation is introduced, and several examples of the notation are given.

2 Erlang

The Erlang language [1] is a syntactically impoverished functional language employing pattern matching for rule selection. The language has support for concurrent programming and is suitable for soft real time applications.

2.1 Functions & Modules

Erlang functions return a single result which may be assigned to a variable or used as an argument to another function call. Functions are constructed from clauses which are constructed from function calls and the operators of the language. Functions are arranged into modules which are the unit of code loading. Modules are also the unit of code distribution in a distributed system. It should be noted that no storage is associated with a module and that modules are only collections of functions.

2.2 Types

The language is dynamically typed. Variables are untyped hence any data element may be bound to any variable.

The language supports 7 simple data types: integers, floats, atoms, Process IDs (PID), references, binaries and ports. Atoms are a constant name with a value equal to their own name. PIDs uniquely identify a process. References are values which are guaranteed to be unique. Binary data objects are used to encapsulate sequences of bytes. Ports are used to communicate with the external environment.

The compound data types of the language - tuples and lists - use position within the structure to identify fields rather than names. The arity or number of elements in a tuple is fixed at compile time. The number of entries in a list is determined at runtime and operators are provided which allow elements to be added to or removed from a list.

Type errors result when either bindings are attempted between incompatible structures or a built-in function[1] or operator is applied to an inappropriate type.

2.3 Process Model

Processes are the unit of execution. A process consists of a stack, an associative store known as a process dictionary, a message queue, and a thread of execution through a piece of code.

2.4 Messages

A message may contain any data item and is addressed to exactly one process by naming either a PID or a symbol which translates to a PID (registered name). The following properties of a message are guaranteed:

[1] Built-in Functions (BIF) are defined as the set of functions provided by the interpretor in an interpreted Erlang system

- to be sent immediately
- to be received without errors if received at all

However, messages are not guaranteed to be delivered.

Erlang also defines the concept of a node. Processes within the same node see the same set of translations between registered names and PIDs. In practice a node is an instantiation of the Erlang interpreter. Processes may be started on a local node or a remote node.

2.5 Dominant Language Features

For the purposes of design, processes and messages are the dominant features of the Erlang programming language. Erlang programs can be modeled as a collection of communicating finite state automata.

Processes may be represented as a finite state automata which jump from internal state to internal state depending on a set of rules encoded as functions. Messages represent the set of events which allow processes to change from state to state.

2.6 Dominant Design Features

Processes are relatively inexpensive and easy to use in Erlang, differentiating Erlang from many other languages and forcing processes to be considered as a major design element.

Generic servers are relatively simple to write and their functionality is distinguished by passing in the module and function name of each function to be called when the server receives a particular event.

3 Existing Notations

Isaksson and Sturesson [6] identify a number of existing design notations employed by Erlang programmers. The notations included State Transition Diagrams (STDs) and Data Flow Diagrams (DFDs). This section briefly discusses these techniques and identifies the need for another notation to describe the design of an Erlang program.

3.1 State Transition Diagrams

The State Transition Diagram and its related tabular form represent a process or system as a collection of states connected by a set of transitions. Associated with each transition is a pre-condition and an action – the action may be null. In the diagrammatic form states are usually represented by circles, transitions are represented by arcs and the arcs are annotated with the pre-condition and the action.

When pre-conditions correspond to the presence of a message, STDs are well suited to representing the action of a single process and are typically easily translated into Erlang code. It should be noted that STDs do not provide a one-to-one correspondence with Erlang as Erlang's receive function employs pattern matching and hence may place an ordering on the received messages. STDs have no mechanism to represent this ordering.

STDs do not offer any assistance in defining the interactions between processes.

3.2 Data Flow Diagrams

Data Flow Diagrams represent the flow of data through a system and the transformations applied to the data at each stage. External entities are represented by boxes, transformations are represented by circles, data items are indicated by arrows, and data stores are represented by a pair of parallel lines.

The mapping between DFDs and Erlang is ambiguous. Specifically, the transformations in a DFD can be modeled as either Erlang processes or function calls within a process, and data flows can be either modeled as the arguments of a function call or a message passed between Erlang processes.

3.3 Deficiencies

Neither STDs nor DFDs allow processes to be shown explicitly with their interfaces. This deficiency prevents easy interfacing to an existing process structure.

The notation proposed in this paper allows processes and their public interfaces to be described. The proposed notation is not intended to replace either of the existing notations completely instead it provides a way of explicitly describing the relationships between processes. STDs may be used in conjunction with the proposed notation to describe the procedural aspects of the operation of a process. DFDs may remain a useful first step in specifying the system. However, it is envisaged that DFDs will have reduced usefulness as the design progresses and the proposed notation is employed.

4 Graphical Design Language Features

This section identifies forces which have influenced the notation presented.

The language was required to be:

- Draw-able by hand without the need for special templates.

  As a considerable amount of design work involves a group of people around a white board and one on one communication between a designer and an implementer may need to be carried out at varied locations where only a pencil and paper is available, it was considered essential that the language be usable without special apparatus.

- Machine translatable into skeleton files which can be used by programmers as a starting point for writing programs implementing the design.

  This criteria allows CASE tools to be constructed.

- Useful for both forward and reverse engineering a design.

  Although the language was introduced to aid in the design stages of a project, it was considered desirable that the language should be usable to draw a representation of an implemented system. This extends the usefulness of the language into the maintenance phases of projects and into iterative development environments. By making the language useful in the maintenance phase the chance of retaining a correct and useful design is increased, this design can be used as a starting point in later design cycles.

The ability to perform both forward and reverse engineering allows the CASE tool to 'drive' the development process by allowing the tool to be used at each stage of the life cycle.

- Recursive in nature.

  It is rare that a complete design for a complex system can be written down immediately and completely. The language was required to allow stepwise refinement of the design. By making language elements recursive a design can be drawn and then refined by redrawing an element at a higher level of detail. This property can be used in two ways: The top level diagram can be re-drafted to contain the refined element, or sub-diagrams can be constructed providing greater detail than the top level diagram.

Graphical languages need to balance the amount of information displayed in written form against overcrowding the diagram. Furthermore, the number and types of elements must be restricted to make the diagrams sufficiently simple to take in at a glance.

Where possible familiar notations from other languages should be used. Conversely, notations which resemble those used in other languages must have an equivalent meaning in the new notation to prevent confusion.

The evolution of OMT [7] and Grady Booch's method [2] into UML [8] suggested that rectangular boxes with square corners were the only acceptable geometric form for a widely used graphical notation.

The ease of writing generic server processes which are distinguished only by the set of functions that they call in reaction to a stimulus makes it essential that the notation support a way of representing a process started with specified values for formal parameters. Of particular interest are module and function names used at the instantiation of the server.

5 Notation

The event model is abstracted into instantiation, processes and interactions. Instantiation is the act of bringing a process into existence. Processes are entities which are in a state and may change state as a result of an interaction. Interactions are events. Typically an interaction is a message or messages sent between processes.

The notation supports recursion. Processes may specify either processes in an implementation, a collection of processes in an implementation, or a collection of processes defined in another event model diagram.

Sources of events outside the model may be implemented as dummy processes.

5.1 Processes

A process (see figure 1) is represented by a box with the following compartments:

- Name compartment: the left side of this compartment contains either the name of the process or an 'X' to indicate that the process is known only by its system allocated PID. The right side may contain an optional (though recommended) unique name which is used to refer to the process within the diagram.

- Startup compartment: This compartment is divided into 2 sections with the second section optional. A single line is used to separate the two sections if the second section is present. The first section is called the parameter section and it contains: the module name, function name and arguments used to start the process. The second section is called the instantiation section and it contains a list of relations. Each relation consists of a formal parameter named in the parameter section, an equals sign, and the value assigned to the parameter when the process is started. Formal parameters maybe unmentioned in which case the designer is not specifying the initial value at this time.

- Process life compartment: A set of symbols indicating the lifetime of a process. An infinity symbol ('∞', the symbol 'inf' may be used where a symbol font is not available) indicates that the process is intended to live indefinitely. If the process is intended to live for a fixed period of time a tau ('τ', the symbol 't' may be used if a symbol font is not available) symbol is used. If an estimate of the time or an upper bound on the time is known then an equation using tau may be used. If the process is intended to live to perform a number of jobs and then cease, the letter 'n' is used to denote the number of major tasks the process is to perform.

- Interface compartment: A set of interface boxes, 1 for each public interface provided.

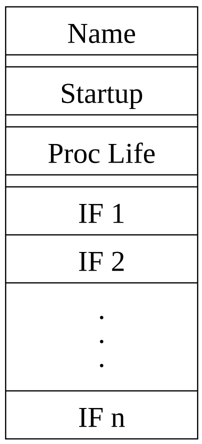

Figure 1: A process

Erlang supports both a direct message based interface and a message based interface that has been encapsulated in functions. An interface box consists of a unique name for the interface typically signifying a purpose for the interface, and either a description of a set of message structures or a set of module names and function names of the interface functions.

An interface consists of a set of message types or functions that logically belong together to achieve some common task.

A dummy process has no interface compartments and the startup and interface compartments are empty. This type of process represents a source of events from originating from outside the model.

5.2 Instantiation

A line with a double arrowhead (see figure 2) is used to indicate the creation of a process. The parent of the process appears at the blunt end of the arrow. The sharp end of the arrow points towards the child. If the arrow points to a solid dot then multiple processes may be created. A number near the dot indicates the number of processes created. Associated with each line is a piece of text which describes the reason for creating the process.

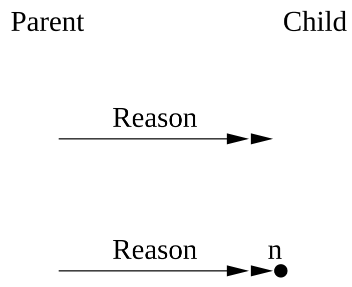

Figure 2: Instantiation

5.3 Interactions

Interactions (see figure 3) are represented as lines with single arrowheads. Interaction lines run from process boxes to interface boxes. It is illegal for a line to connect two interface boxes. If an encapsulated interface is used and a function returns a result caused by receiving a message then a double barb should be used. The double barb indicates the flow of information in two directions.

Associated with each interaction line is a reason for the interaction.

6 Design Technique

The first step in designing an Erlang program is to determine the major concurrent activities of the system. Each of these activities is then defined as a process on the diagram. After the processes are drawn, interactions between the processes are indicated with lines interconnecting processes. Interface boxes are added to the design to provide the detail of the interactions interface. This top level diagram may be stepwise refined by defining processes which compose the processes provided by the top level diagram.

A number of useful 'patterns' are available to the designer:

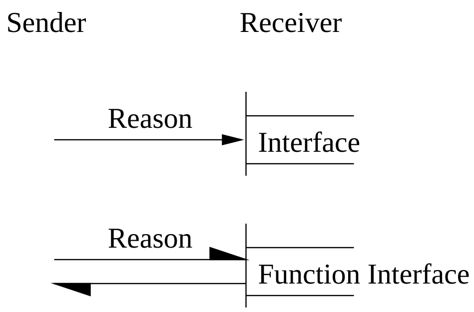

Figure 3: Interactions

- Worker processes - As processes are cheap in Erlang designers may wish to start a worker process to complete some task
- Immortal servers - Most systems will have some co-ordinating process, typically this process will be immortal as this simplifies the problem of contacting the controlling process

A number of decisions are deferred to the implementation:

- The size and borders of the process name space
- The placement of processes on nodes

Description of the program logic can be provided by associating a finite state automata, a message sequence chart, or a textual description with each process.

Although the notation supports both encapsulated and non-encapsulated interfaces. Designers are encouraged to use encapsulated interfaces in nearly all cases. Non-encapsulated interfaces will probably only be used for exposed system interfaces such as gs [4] or Internet [5] libraries.

Designers are encouraged to abstract common process patterns into generic servers which call functions by name in response to a stimulus. This approach minimises maintenance problems by abstracting common code.

7 Examples

This section develops a simple example and a more complex example of the application of the notation. The section closes by commenting on the design process in relation to protocol stacks.

7.1 Simple Example

WWW servers are easily constructed in Erlang. The notation is used to describe a minimalist WWW server [3]. The server consists of a process that listens for incoming connections and then spawns a new process to handle the request contained in the connection. Figure 4 illustrates the approach.

In addition to the graphical representation of the process a short description of the logic of the *server* and *reqhd* processes is required. The *erts* process

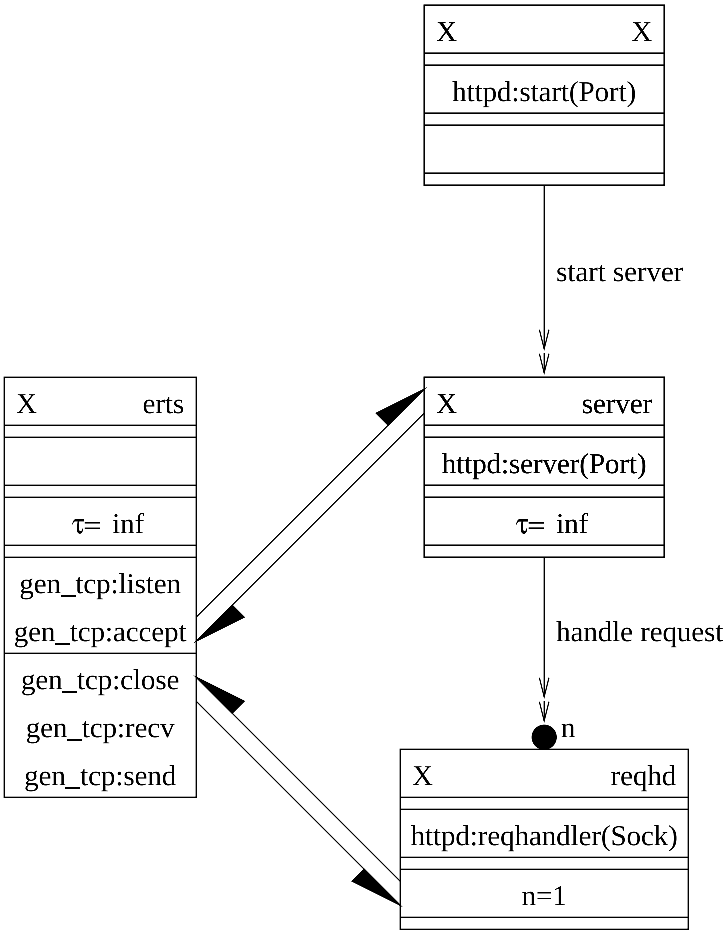

Figure 4: A simple WWW server

represents facilities provided by the Erlang runtime system that are used by the WWW server. The *I* process is used to represent any process that calls the *httpd:start* function and hence starts the server.

7.2 More Complex Example

The second example is an abridged representation of an authenticating WWW proxy server. Many IP networks employ a firewall to control what data flows into and out of their network. A firewall is a computer with at least 2 network interfaces. It prevents traffic from passing from one interface to another until a condition allowing the transfer has been met. One example of such a rule would be that traffic would be prevented from flowing until after the user has successfully identified themselves. One method for implementing an authentication for WWW page access on a firewall is to create a proxy server on a random port on the firewall after the user has successfully authenticated themselves.

This example illustrates the use of a generic server as the basic WWW server is abstracted and reused. Figure 5 shows the major processes of the system. The server code is reused in both the *auth* and *authprxy* processes. The two processes differ only that they are instantiated with the names of different functions associated with the action of getting a new page.

7.3 Protocol Stacks

Protocol stacks are typically a special design case in that they can be implemented either in a single process or in several processes. The design approach presented here is relevant to protocol stacks. As these diagrams do not seek to show the procedural aspects of program flow, a protocol stack implemented in process may not even appear on these diagrams. The authors do not view this as a problem as they consider that the procedural aspects of design are well covered using either a narrative description or STDs, however, current notations neglect processes and their dynamic nature.

8 Iterative Application

The proposed method is recursive and can be applied iteratively. This section will show how the recursion property can be exploited to perform stepwise refinement on a design.

The more complex example (see section 7.2) provided a starting point for an implementation. Further study of the problem indicated that it was simpler and more desirable to separate the roles of authentication and proxying. Furthermore, it was found that it was necessary for the child process which handles an individual connection to be able to shut down the parent WWW server process. This facility would be used if a user failed to authenticate themselves correctly.

Figure 6 shows the separation of the authentication and proxy roles. Note that this could also have been represented in a subdiagram providing greater detail to the combined *authprxy* process (see figure 7). The subdiagram shown includes the interaction with *erts* although this is optional.

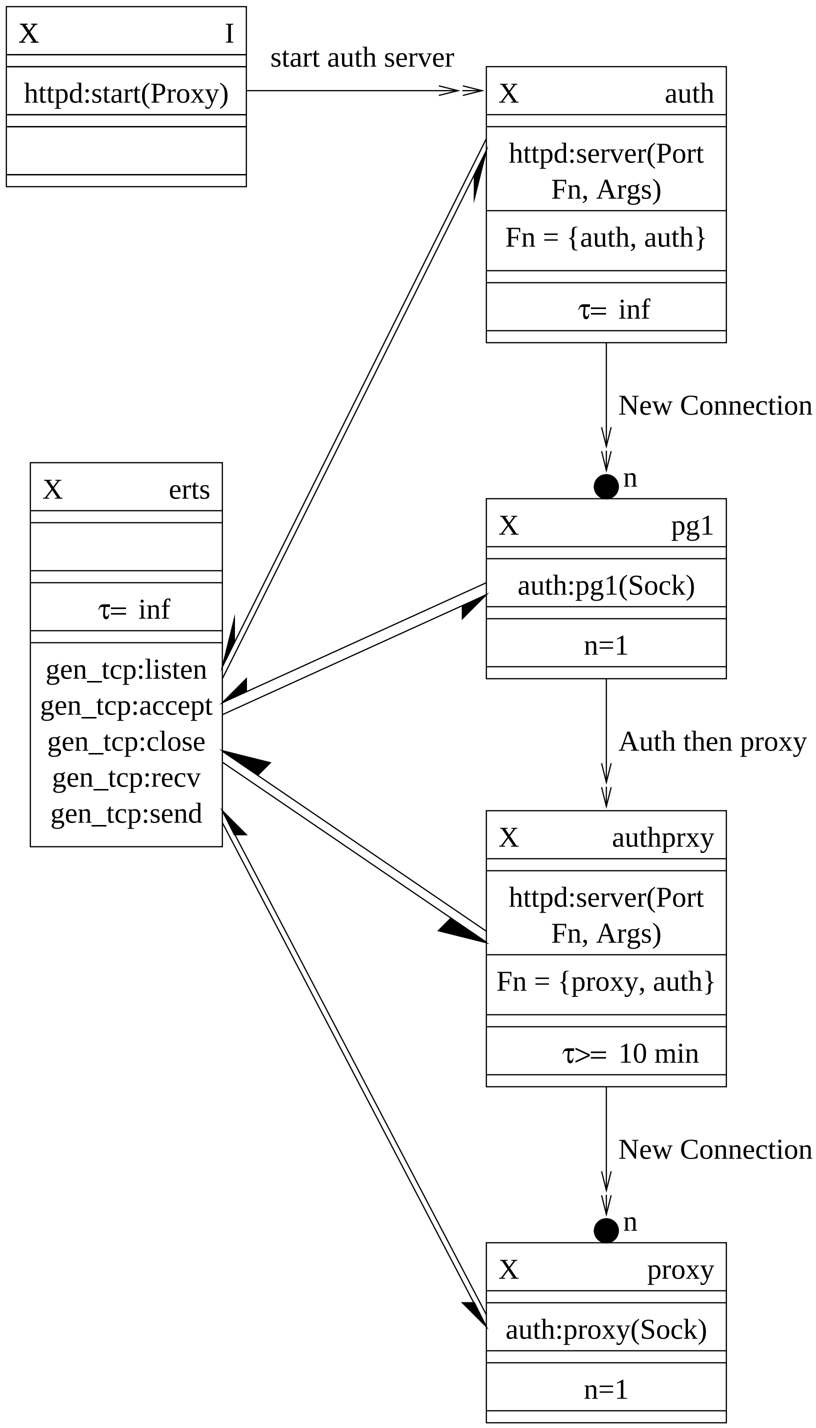

Figure 5: An authenticating WWW proxy server

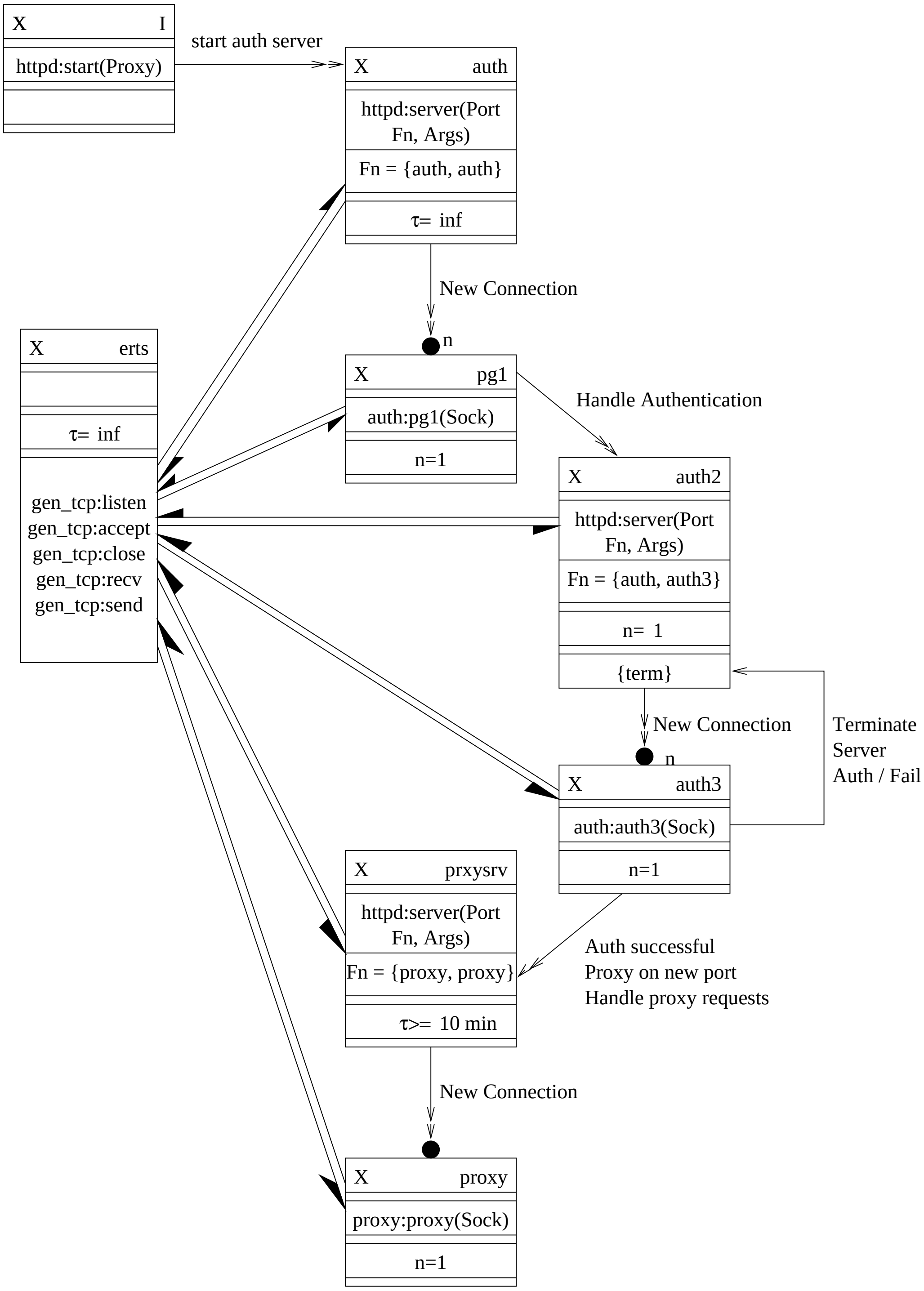

Figure 6: Modified authenticating WWW proxy server

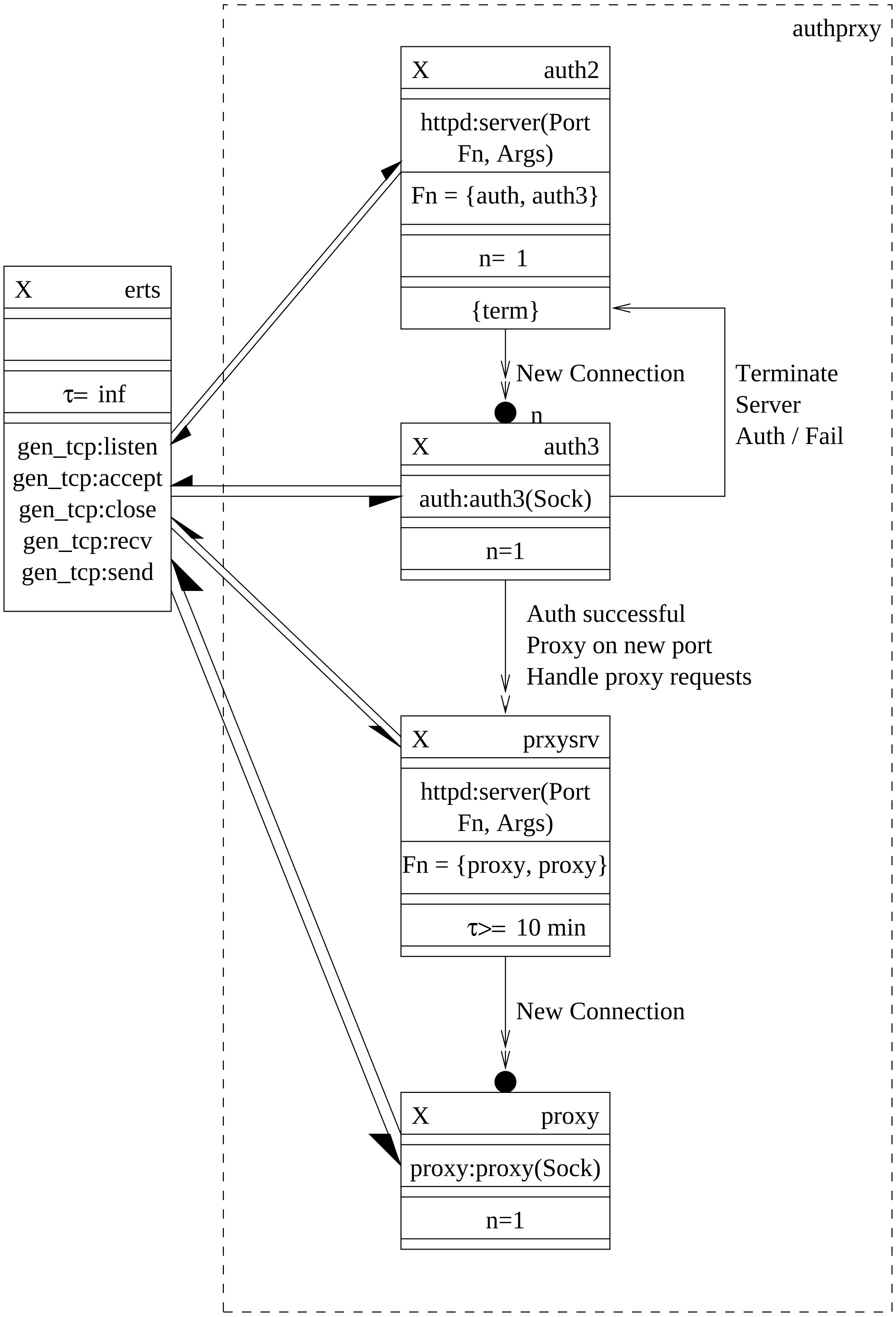

Figure 7: Detail of *authprxy*

9 Conclusion

The beginnings of a notation and a design technique for Erlang applications have been introduced. The notation is intended to assist both in the design and maintenance phases of project development. At this stage the notation has been used on only a few examples which have been strongly connected with WWW applications, however, the authors have found it helpful in both formulating and communicating their ideas. The notation has been shown to be suitable for reverse engineering existing code, and has been shown to be readily modifiable as design ideas change.

References

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

[2] Grady Booch. *Object-Oriented Analysis and Design with Applications.* The Benjamin/Cummings Publishing Company, second edition, 1994.

[3] Maurice Castro. *Erlang in Real Time.* Department of Computer Science, RMIT, 1998. ISBN 0864447434. Also available from http://www.serc.rmit.edu.au/~maurice/erlbk/.

[4] Klas Eriksson. *Graphics System Application (GS) version 1.3.* Ericsson Telecom AB, 5 1997.

[5] Magnus Fröberg. *Kernel Application (Kernel) version 2.1.* Ericsson Telecom AB, 11 1997.

[6] Johanna Isaksson and Elinor Sturesson. Design guidelines for erlang. Technical Report SERC-0083, Software Engineering Research Centre, 1 1999.

[7] James Rumbaugh, Michael Blaha, William Premerlani, Frederick Eddy, and William Lorensen. *Object-Oriented Modeling and Design.* Prentice-Hall, 1991.

[8] Rational Software, Microsoft, Hewlett-Packard, Oracle, Sterling Software, MCI Systemhouse, Unisys, ICON Computing, IntelliCorp, i-Logix, IBM, ObjecTime, Platinum Technology, Ptech, Taskon, Reich Technologies, and Softeam. *UML Semantics version 1.1.* Rational Software, 9 1997. http://www.rational.com/uml/ (ad/97-08-04).