Extending ERLANG with structured module packages

Richard Carlsson
Computing Science Department
Uppsala University
Box 311, S-751 05 Uppsala, Sweden
richardc@csd.uu.se

December 2, 1999

Abstract

This article describes a way to extend ERLANG with structured program module packages, in a simple, straightforward and useful way.

1 Introduction

When ERLANG was conceived, it inherited a lot of its flavour from languages like Strand, Prolog and Parlog, which (at least in many implementations) have a similar concept of program modules: these are program files ("compilation units"), each assigned a globally unique name (in the system), and each declaring some or all of its functions as *exported*. Non-exported functions can only be referred to from within the same module, while exported functions are also accessible from any other module in the system. In ERLANG, files containing source or object code for a module must be given the same name as the module, plus a suffix which is ".erl" for source files, and *e. g.* ".jam" for object files for the JAM abstract machine.

The name space for modules in these languages is *flat*, *i. e.*, when a particular module is referred to, this is always done by its full name, regardless of the context in which the reference is made: there is no way to express a reference to another module in relation to the current module. Furthermore, since programmers like to keep names short, names such as "lists", "math", "queue", "shell", "random", etc., quickly become used (these examples all taken from the standard library). When code from different vendors is combined in the same system, each distribution possibly consisting of several hundred modules, the likelihood of one or more name clashes becomes large. Also, because of the meta-calls often used in ERLANG, it is not always an easy task in such cases to rename the clashing modules uniquely without introducing errors, even if the source code is available. To keep the risk of clashes down, some programmers resort to giving modules abbreviated names such as "gb", "rb" "dbg", etc., which is uninformative and could be considered bad programming style, or using prefixed names such as "snmp_supervisor", "snmp_error", "snmp_generic", etc., which more or less solves the problem, but in a way that is clumsy and limited by the maximum length of file names on the host operating system.

Section 2 of this article describes the package system of Java and introduces the basic workings of a similar system for ERLANG. Section 3 describes extensions to the ERLANG language in order to make life easier for the programmer using packages. Section 4 discusses possible pitfalls, and section 5 is a summary of the system and the necessary changes.

2 A package system

To solve this situation and bring order to the present chaos, I suggest a system of packages whose basic structure is shamelessly borrowed from that of Java [2].

In Java, each compilation unit is a publicly available *class*, which is similar to a module in ERLANG, the main difference being that ERLANG modules cannot have distinct *instances* and do not support inheritance. Java source and object files are, like ERLANG modules, given the name of the public class they contain plus the extensions ".java" and ".class", respectively.

In Java, however, classes can belong to a *package*: this is a way of structuring the *files*, and is orthogonal to the class hierarchy of Java. The same concept can therefore be applied to ERLANG, even though it is not an object-oriented language.

2.1 The structure of packages in Java

If a Java class file (having the suffix ".java") contains a *package declaration* stating a package name, then that class file belongs to the named package. Furthermore, the package name also indicates to the Java implementation where the object file is located.

A Java package name consists of a sequence of names separated by period characters, such as

```
java.rmi.server
```

The full name of the class `RemoteObject` in this package, then, is

```
java.rmi.server.RemoteObject
```

When the Java implementation tries to load the object file for a class by its full name, it uses its `CLASSPATH` setting. This is simply a sequence of file system paths in the host operating system, which are to be used for the search in the order they are given. The package name is then subdivided at each period character into a sequence of one or more names. This is interpreted as a relative path in the host operating system, in a way that is system-dependent: in a Unix-like system, the relative path corresponding to the above package name would be

```
./java/rmi/server/
```

An attempt is then made to, for each path `ROOT` listed in `CLASSPATH`, load the file whose name is the concatenation of `ROOT`, the relative path for the package, the class name, and the object file suffix (".class"); in this example:

```
ROOT/java/rmi/server/RemoteObject.class
```

until such a file is found for some ROOT, or all paths in CLASSPATH have been tried. With this approach, object files are thus located in a set of directory trees, rather than in a set of flat directories.

When a Java program refers to a class that is not defined in the same file, and by its class name only, the Java compiler will assume that the class is defined in the same package as that of the current file, and will not confuse it with classes of the same name in other packages. In this way, a package declaration creates a distinct name space for classes.

For simplicity, and often useful for testing or writing simple applets, if a Java source file does not contain a package declaration, it is automatically placed in the "unnamed default package", which is a flat name space just like the current ERLANG module system.

2.2 Packages in Erlang

Today, ERLANG module names seldom or never contain period characters; one reason for this is that since the module name declaration of an ERLANG module has the form

```
-module(<A>).
```

where <A> is an ERLANG atom,[1] such a module would have to be declared as *e. g.*

```
-module('foo.bar').
```

stating the module name within single-quotes. Therefore, it can be expected that the use of the period character as separator in module names can be adopted with few (if any) existing ERLANG programs needing rewriting.

My suggestion, then, is that ERLANG modules in packages be named similarly to full class names in Java: for instance, the full name of a module m in a package a.b.c should be a.b.c.m, while its object file would be named "m.jam" (we assume for simplicity from now on that all object files are for the JAM abstract machine) and reside in a directory ROOT/a/b/c/ for some ROOT in the search path of the ERLANG code server. The object file for a module whose full name does not contain any period characters, such as *e. g.* io_lib, is thus assumed to be located in some directory ROOT/, exactly as in the ERLANG implementations of today; for this example, the file would be ROOT/io_lib.jam.

To handle this first step, only the code server needs to be modified, and only when the name of a requested module does contain period characters will the behaviour differ from that of today. It should also be apparent that this convention is compatible with existing code: packaged code could pass a full module name (generally as an atom) to old-style code, which could use the name obliviously, even for making meta-calls, without errors; to the old code, a module name does not have structure.

Existing standard modules could easily be moved into packages without disturbing old code, by simply creating "stub" replacement modules in which all exported functions make a direct jump to the function of the same name in the corresponding packaged module; such stub modules will be small, and the extra

[1] Atoms are a primitive datatype in ERLANG; they can be seen as nullary constructors, and are identified by their print names. Unless surrounded by single-quotes, their names must begin with a lowercase letter, and not contain other characters than letters, digits, or underscore ('_'). Examples of atoms are foo, mad_hatter and 'foo@bar'.

call is a relatively small cost. In particular, the package `erlang` and all its sub-packages should be reserved for standard library functions. A library module such as `lists` could *e. g.* be renamed `erlang.list` – thus note that this would present a very good opportunity to restructure (by renaming, splitting, moving individual functions, etc.) the existing standard modules, preferably according to the suggestion for a new set of standard modules made by Jonas Barklund [1]. However, details of such a structuring of existing code into packages is outside the scope of this article.

3 Extensions to the Erlang language

So far, I have only described a structured way of storing object files in relation to module names. If the ERLANG language itself remained unchanged, this convention would force the programmer to write the full module names, always within single quotes, in all situations. This would be cumbersome and ugly, and miss one of the main points with a structured name space: to be able to make references relative to the current package.

3.1 A new form of module declarations

It is a simple task to extend the ERLANG grammar to not only accept module name declarations on the form

$$\texttt{-module(<A}_1\texttt{>).}$$

where $\texttt{<A}_1\texttt{>}$ is an atom, but also more generally on the form

$$\texttt{-module(<A}_1\texttt{>.<A}_2\texttt{>}\cdots\texttt{.<A}_n\texttt{>).}$$

for $n \geq 2$, where all $\texttt{<A}_i\texttt{>}$, $i \in [1, n]$, are atoms. Each such atom could of course be individually stated within single-quotes, and it is therefore necessary to check that the atoms do not themselves contain period characters, and that they are not the empty string (‘`''`’). It is also recommended that some other characters, such as *e. g.* ‘`$`’ be reserved for future use in module names, for example for auto-generated object files for sub-modules, if such a concept would be shown to be useful.

We introduce a little terminology:

- The *full module name* is the concatenation of the print names of the atoms $\texttt{<A}_1\texttt{>}\cdots\texttt{<A}_n\texttt{>}$ and the separating period characters. A full module name should not contain two adjacent period characters.
- The *module name* is the atom $\texttt{<A}_n\texttt{>}$.
- The *package name* is the concatenation of the atoms $\texttt{<A}_1\texttt{>}\cdots\texttt{<A}_{n-1}\texttt{>}$ and the separating period characters.

for instance, in a declaration `-module(fee.fie.foe_fum).`, the full module name is given by the atom `'fee.fie.foe_fum'`, the package name by the atom `'fee.fie'` and the module name by `'foe_fum'`. For a module whose full name contains no period characters, such as `io_lib`, the package name is the empty string, and the module name is the same as the full module name; thus, the meaning of old-style declarations does not change.

3.2 Package-relative compilation

The main advantage with the extended form of module name declarations, however, is not to relieve the programmer from writing single-quotes: it is to *signal that the source file is part of a package*, and that module references within it may therefore be interpreted as relative to the same package.

Note that it is still legal to use a declaration such as `-module('foo.bar.baz')` to give a full module name, but that this does not enable package-relative compilation. Compilation of modules with such old-style name declarations is not affected by the transformations described in this section.

3.2.1 Explicit remote calls

When a remote call on the form

```
<A>:<F>(...)
```

is encountered in a packaged module, where <F> is any expression and <A> is an *atom whose print name does not contain period characters*, then that atom is interpreted as the name of a module in the same package as the current module.

In this case, the compiler will automatically replace <A> with the corresponding full module name, by prepending the package name to <A>, separated by a period character. For example, if the call `fred:f()` occurs in a module whose package name is `foo.bar`, it will be replaced by the call `'foo.bar.fred':f()`.

This allows the programmer to *e.g.* create a module named `lists` in a package, and refer to that module directly by that name without confusion with the standard module of the same name.

To simplify calling modules in specific packages, it is easy to extend the Erlang grammar to allow remote calls on the form

$$\texttt{<A}_1\texttt{>.<A}_2\texttt{>.}\cdots\texttt{<A}_n\texttt{>:<F>(...)}$$

for atoms <A$_i$>, $i \in [1, n]$, $n \geq 2$, and an expression <F> (with the same restrictions on the atoms as in a module name declaration). Thus, a programmer is not forced to write a full module name within single quotes, but still has the possibility. For example, the calls

```
foo.bar.baz:f()
```

and

```
'foo.bar.baz':f()
```

are equivalent. Thus note that if the module specifier is a single atom, then that atom may contain period characters, but not otherwise.

3.2.2 Imported functions

Import declarations in packaged modules should be handled analogously to remote calls. If an import statement

```
-import(<A>, [...]).
```

is encountered in such a module, and <A> is an *atom whose print name does not contain period characters*, then that atom is interpreted as the name of a module in the same package as the current module, and the full module name is substituted by the compiler.

The ERLANG grammar should also be extended analogously to allow period-separated full module names to be written without surrounding single quotes in import declarations; *i. e.*, both

```
-import('foo.bar.baz', [...]).
```

and

```
-import(foo.bar.baz, [...]).
```

should be allowed, and be equivalent.

3.2.3 Forcing absolute module references

It is quite possible that a packaged module could need to refer to a module that is not packaged (*i. e.*, whose package name is the empty string). For this purpose, it is necessary to be able to refer to a full module name using a leading period character, as in the call

```
.lists:reverse(X)
```

This form (more generally described as $.\texttt{<A}_1\texttt{>}\cdots.\texttt{<A}_n\texttt{>}$, for atoms $\texttt{<A}_i\texttt{>}$, $i \in [1, n]$, $n \geq 1$), should therefore be included in the ERLANG grammar for remote calls and import declarations.

The same effect could be achieved by giving the module name with a prepended period character within single quotes:

```
'.lists':reverse(X)
```

however, this is *not recommended* in general, since two atoms 'm' and '.m' do not compare equal, but can be interpreted as references to the same object file.

3.3 Meta-call support

Meta-calls are often used in ERLANG, either for direct function calls as in the following examples:

```
<M>:<F>(...)
```

and

```
apply(<M>, <F>, [...])
```

where <M> and <F> are any expressions evaluating to atoms, or for the evaluation of a function call by a new process, as in:

```
spawn(<M>, <F>, [...])
```

```
spawn_link(<M>, <F>, [...])
```

or

```
spawn(<N>, <M>, <F>, [...])
```

where in addition, <N> is an expression evaluating to an atom that is taken to represent the name of an ERLANG node.

It must then be remembered that the atoms yielded by evaluating expressions `<M>` above must be *full module names*; the built-in functions `apply` and `spawn` (and their variants) *can and should not* be modified to interpret module names relative to the current module. As a simple example, consider a function

```
f(X) -> spawn(X, start, [...]).
```

defined in a packaged module `foo.m`, and a call

```
foo.m:f('my_server')
```

in some other module. It would then be impossible for the `spawn` in function `foo.m:f/1` to know if the module name `my_server` should be interpreted relative to package `foo` or if it is a full module name whose package name is the empty string. Thus, the `apply` and `spawn` functions should remain unchanged, always interpreting the given module name as a full module name.

3.3.1 Getting the module name

Since it is generally a source of errors to be forced to write things more than once, I suggest a new predefined (but not exported) function `this_module/0` which returns the *full module name* of the module in which it occurs. For instance, to ensure correct behaviour when spawning a process to execute a function `run` in the current module, one could write

```
spawn(this_module(), run, [...])
```

Another typical example is to pass the full name of the current module to some other function, for general use including making meta-calls, as in

```
gen_server:start(this_module(), ...)
```

The call `this_module()` could be defined as synonymous to

```
module_info(module)
```

using the already predefined function `module_info/1`.

It would also be possible to use the automatically defined preprocessor macro `MODULE` for the same purpose, but the use of the preprocessor for any purpose is strongly discouraged by this author.

3.3.2 Getting package-relative names

Where, in a situation similar to those above, it is necessary to refer to a module other than the current, but in the same package, it would be convenient to not have to specify the full package name. This could be accomplished by another predefined (not exported) function `this_package/1`. For example, assume that in a module `m` in package `foo.bar`, we are to spawn a function `start/3` in a module `server` in the same package. We could then write

```
spawn(this_package(server), start, [...]
```

which would be equivalent to

```
spawn('foo.bar.server', start, [...]
```

but not dependent on the actual name of the current package.

The call `this_package(<A>)` could be defined as a substitution of the print name of `<A>` for the last component of the value of `this_module()`, if `<A>` does not contain period characters and is not the empty string `''`.

3.3.3 No other predefined functions!

There should be no other additions to the set of predefined functions or ERLANG. It might for some purposes be necessary to find the package name (the full module name without the last segment) or the module name (the last segment of the full module name) of the current module or of another module, or to perform other operations on module names, such as concatenating a package and a module name, or to map a full module name or a package name onto a relative file path, but such functions would more suitably be placed in some separate support module.

It could also be argued that special versions of `call` and `spawn` should be added to handle this kind of name expansion automatically, but this is the wrong way to go. It would add extra predefined functions without solving the general problem when full module names need to be passed between functions, and is not even a big advantage. It is no doubt easier (but more opaque) to write, say

```
spawn(start, [...])
```

instead of

```
spawn(this_module(), start, [...])
```

but *e. g.*

```
spawn_this_package(server, start, [...])
```

is not really simpler than

```
spawn(this_package(server), start, [...])
```

Furthermore it can be argued that, in particular for `spawn`, local functions should if possible never be called via a meta-call, since this requires the target function to be exported from the module, even if it is not part of the official interface. A version of `spawn` which could initiate the evaluation of a named local function, *e. g.* `fun start/3`, or an anonymous local function (a so-called "fun expression") `fun (...) -> ... end`, by a new process, would be much cleaner.[2]

3.3.4 Passing names of modules in other packages

When the full name of a module that is not part of the same package as the current module is to be passed to some function for purposes as those described above, it should be given explicitly as an atom, within single-quotes if necessary. *E. g.*, if we were to give the full name of module `a.b.c` as argument to `gen_server:start/3`, from a module `x.y.z`, we would write

```
gen_server('a.b.c', ...)
```

[2]It has been hinted that this form of `spawn` will be included in a coming release of ERLANG.

The alternative would be to extend the syntax of period-separated atom sequences to be allowed as general expressions, so we simply could write

```
gen_server(a.b.c, ...)
```

That, however, is taking the idea too far; we would then have introduced a general kind of atom-concatenating operator in the language, which could be used regardless of context, but there is no really good reason for being able to write something like

```
{'R2'.'D2'}
```

where the period character could easily be mistaken for a comma, or what is worse, be mistakenly inserted instead of an intended comma.

3.4 Imported module names

The currently existing import declarations in ERLANG allow the programmer to use functions in other modules than the current by their function names only; *e. g.*, a declaration `-import(lists, [reverse/1]).` allows a function call `lists:reverse(X)` to be written more briefly `reverse(X)`, where the imported name overrides any locally defined function of the same name.

A more general form of such declarations would allow the definition of a *local alias* for a remote function, where also the actual name used locally for making a call could be individually selected by the programmer.

When a packaged module needs to refer to several modules that are not in the same package, it would then either have to specify the full module name in each call to those modules, or use import statements so that the individual functions can be called directly by name. However, there is then the possibility that the same function name exists in two distinct modules, where both modules have long full names, or that for some module, many functions are used but we do not wish to import them all, perhaps because of the risk of clashes with locally defined names.

3.4.1 Importing packaged modules

To support easier access to particular modules in cases such as the above, I suggest a new form of import declaration, on the following form:

```
-import(<M>).
```

where `<M>` is a *full module name.*

The occurrence of such a declaration in a module would allow the use of the *module name alone*, *i. e.*, without a package name, in calls to the imported module. For example, a declaration

```
-import(foo.bar.baz).
```

in module `a.b.c` would make a call

```
baz:f(...)
```

in the same module be synonymous to

```
foo.bar.baz:f(...)
```

However, the behaviour of `this_package(baz)` would not be affected, as its name suggests, yielding `'a.b.baz'`.

In particular, note that since the imported module name is always a full module name, a declaration `-import(lists).` would make a call `lists:f(...)` in the same module be synonymous to `.lists:f(...)` thus correctly referring to the module whose package name is the empty string.

It is important that all occurrences of this kind of import statement are processed *before any function-importing declarations are expanded*, since the latter should be interpreted relative to the former, as well as to the current package. Thus, the two declarations `-import(foo.bar.baz).` and `-import(baz, [fred/1]).` (given in any order) in a module `a.b.c` together make a call `fred(X)` synonymous to `foo.bar.baz:fred(X)`, and not to `a.b.baz:fred(X)`.

3.4.2 Similarity to imported packages in Java

Java has a similar form of import declaration, on the two forms

package.*class*;
package.*;

where the former imports a particular class, and the latter all classes in a particular package. This allows the programmer to refer directly to any imported class without its package name; however, if two classes with the same name are imported, then neither can be used without giving its package name.

Java is a statically typed language, and needs information about the types of all external classes referenced in a program file in order to compile that file. The Java compiler therefore searches for object files for such classes, recursively compiling source files where possible in order to produce any object files that are missing. This makes importing of all classes in a package possible, because the search order is well-defined, and all referenced classes must be present.

Erlang, however, is dynamically typed, and the compiler never actually needs to examine other source files in order to compile a particular file. It would not be in line with Erlang programming conventions to let the sets of existing object files in two distinct packages decide from which of these packages a particular module is imported; *e. g.*, if we would import all modules from packages `foo` and `bar`, then a reference to a module `m` would be resolved to either `foo.m` or `bar.m` depending on which package actually defines a module `m`. Therefore, a full module name must be given in the `-import(<M>).` declaration described above, and not just a package name.

3.5 Why no package-relative package references?

The reader may wonder why, in a "structured" package system, there are no language constructs that allow the programmer to refer to a module in a subpackage of the current package by a relative name, instead of by its full name.

For example, in a module `a.b.m1`, it would certainly be possible to let a call

```
c.m2:f()
```

be interpreted as equivalent to

```
'a.b.c.m2':f()
```

There are however several problems with this approach. Most importantly, it adds to the complexity of the package system, making programs difficult to understand and being prone to errors. For instance, all calls to functions in packaged modules that are *not* in subpackages of the current package would have to be written as

```
.x.y.z:f()
```

with a leading period character. This is annoying and could often result in mistakes.

When there is a need to refer to modules in subpackages by a short name, the general mechanism for importing packaged modules suggested above should be sufficient, and results in perspicuous programs. The same design decision was apparently made in Java.

4 Caveats

4.1 Program transformations

If a source code transformation should want to rewrite a meta-call expression such as

```
<M>:<F>(...)
```

if it can show that <M> will evaluate to a specific atom <A>, to

```
<A>:<F>(...)
```

(doing a so-called *constant propagation*), then it is important to keep in mind that the result of <M> should be interpreted as a *full module name*, and never relative to the package of the module containing the code.

In this case, either the resulting program must not be compiled in a package-relative way (this can be done by expanding all package-relative references and substituting an old-style module name declaration, which states the full module name within single quotes), or the program performing the transformation must be aware of this interpretation and instead substitute the expression

$$\texttt{.<A'}_{1}\texttt{>.<A'}_{2}\texttt{>}\cdots\texttt{<A'}_{n}\texttt{>:<F>(...)}$$

where the $\texttt{<A'}_{i}\texttt{>}$, $i \in [1, n]$, are the period-separated (nonempty) segments of <A>. Note the leading period character, which explicitly indicates a full module name.

4.2 Other possible problems

There exist some modules in the ERLANG distribution which make assumptions about the present way of storing object files in the file system, notably the module `filename`. Such modules may need to be updated to handle the tree substructure of the new object file storage.

5 Summary

I have described a system of structured module packages for ERLANG, which is conceptually simple and easy to implement, and which should be backwards compatible with practically all existing code. I have suggested straightforward extensions to the ERLANG language for easier programming with packages, all of which could be removed by a preprocessor pass if so desired. I have also discussed why no further extensions should be necessary, or even motivated.

In brief, the following things need to be implemented in order to support the package system as described:

- Extend the code server to analyse module names in order to find the search path substructure for an object file.
- Add the new, package-relative compilation enabling form of module declarations to the grammar.
- Allow period-separated full module names in remote calls and in import declarations, including names with a leading period. (Note that this is not dependent on package-relative compilation, and thus should be allowed regardless of the form of module name declaration being used.)
- Extend the compiler to, for packaged modules, expand package-relative remote calls to full module names.
- Add the predefined functions `this_module/0` and `this_package/1`.
- The preprocessor `epp` must be extended to handle the new form of module name declarations, in order to correctly support the automatically defined macro `MODULE`.
- Add -import(<M>). declarations for full function names <M>, and make the compiler expand these before ordinary function imports are processed.
- The stdlib function `filename:find_src` will probably also need to be made aware of the new structure of object file search paths.

References

[1] Jonas Barklund et al., *Proposal 15: Built-in functions of Erlang.* ERLANG specification project, June 1998,
http://www.ericsson.se/cslab/~rv/Erlang-spec/index.shtml.

[2] David Flanagan, *Java in a Nutshell.* O'Reilly & Associates, Inc., 1997.