Dryverl: a Flexible Erlang/C Binding Compiler *

Romain Lenglet Shigeru Chiba
Tokyo Institute of Technology
{lenglet,chiba}@csg.is.titech.ac.jp

Abstract

This article introduces Dryverl, an Erlang/C binding code generator. Dryverl aims at becoming the most abstract, open and efficient tool for implementing any Erlang/C bindings, as either C port drivers, C port programs, or C nodes. The most original feature of Dryverl is to provide users with *open* Erlang/C bindings, similar to distributed bindings in open distributed processing systems, to allow specifying programmatically the data transformations that must often be performed in Erlang/C bindings. Implementation details are hidden to developers, and implementation differences between port drivers, port programs, and nodes are abstracted by Dryverl, and Dryverl aims at generating the most efficient implementations possible for every target mechanism.

Categories and Subject Descriptors D.2 [*Software*]: Software Engineering

General Terms Design, Languages, Performance

Keywords Erlang, C, binding, stub, port, driver, compiler, ODP

1. Introduction and objectives

Easily and efficiently interfacing Erlang code with C libraries, such as graphics toolkits, communication libraries, or hardware drivers, is one of the remaining challenges of Erlang development.

Erlang provides three mechanisms to implement efficiently a binding between Erlang code and C code: 1) as a *C port driver*, i.e., as a library that is dynamically linked into an Erlang emulator OS process; 2) as a *C port program*, i.e., as a program executed as a subprocess of an Erlang emulator OS process, both communicating through pipes; and 3) as a *C node*, i.e., as a program executed as a separate OS process which communicates with other Erlang nodes using the same network protocols as "normal" Erlang nodes. However, the choice of a particular mechanism is a tradeoff between efficiency and safety, which can generally be made independently from a binding's functionality. For instance, implementing a C port driver allows exchanging binary data with Erlang code without copy, which is very efficient, but any failure in a driver may crash the whole Erlang emulator. Implementing C port programs and C nodes is safer, but communications are more expensive. The problems are that all those mechanisms are difficult to use, especially on the C sides of bindings, and that their implementation details differ.

More importantly, Erlang/C bindings must often cope with differences between the idioms and type systems of Erlang and C, by performing complex *transformations* of exchanged data. For instance, Erlang atoms should typically be transformed either into integer constants (e.g., defined in enumerations) or into strings. Using existing binding compilers, such as the Erlang/C EDTK [4] and IG [17] compilers, such transformations must generally be implemented in separate wrappers, which makes specifying and maintaining bindings as a whole more difficult.

There is therefore a need for Erlang/C binding compilers that would 1) *abstract* the implementation details and differences between the available implementation mechanisms, and 2) allow easily specifying any transformations of data exchanged between Erlang and C, by *opening* to developers the data transformation parts of bindings.

This article introduces such a compiler: Dryverl. Dryverl aims at generating the Erlang and C source code that implements bindings using any chosen implementation mechanism (port driver, port program or C node), from abstract specifications of the functional aspects of bindings in the Dryverl DSL (Domain-Specific Language), although currently only port drivers are supported. More originally, it allows completely specifying transformations of data in binding specifications. The development of Dryverl was initially motivated by a need to bind the GSSAPI (Generic Security Service API) standard [18], which is a very widespread, though non-trivial C API which cannot easily be bound to Erlang using other Erlang/C binding compilers. A binding development process using Dryverl is illustrated in Fig. 1.

* This work is sponsored by a JSPS Post-Doctoral Fellowship.

Erlang'06 September 16, 2006, Portland, Oregon, USA.

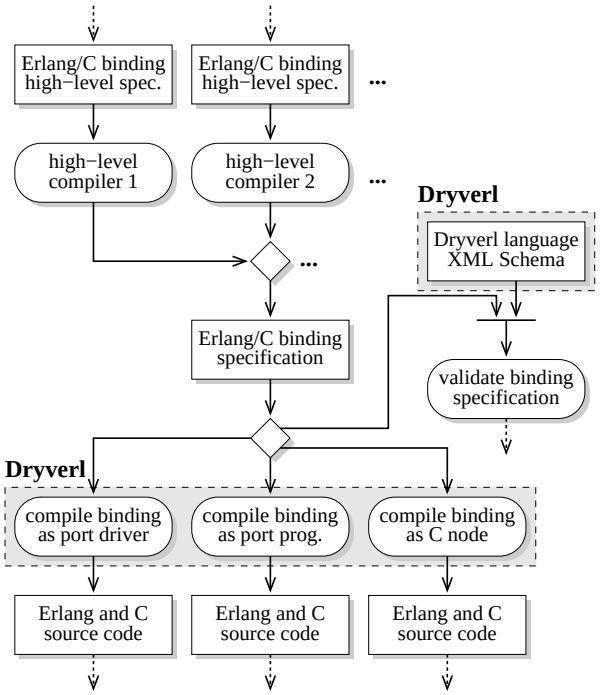

Figure 1. An Erlang/C binding development process using Dryverl

Although Dryverl can be used directly by developers, it is more specifically intended to be used as the back-end of higher-level compilers that would provide less general specification languages to developers, than the Dryverl DSL.

This article is organized as follows. Section 2 discusses the requirements for an Erlang/C binding compiler, and especially the *openness* and *abstraction* requirements, using the ISO RM-ODP standard [7, 8] and reusing ideas from open distributed computing frameworks, and using the concrete example of GSSAPI. Section 3 shows how those requirements are reflected into the Dryverl DSL. Section 4 shows how Dryverl provides *abstraction* and *efficiency* when bindings are implemented as C port drivers. Section 5 compares Dryverl with other language binding generators, regarding our requirements. Section 6 concludes this article.

2. Background and objectives

The simultaneous requirements for the design of Dryverl are the following:

- *openness*: the implementations of Erlang/C bindings must be as open as possible to let programmers control as much as possible in bindings, to adapt them to the applications and their environment;
- *abstraction*: the bindings specification language and the compiler must hide the implementation details of the different mechanisms (C port driver, C port program and C node), and the differences between them, to allow generating an implementation of a binding using any mechanism, without requiring to modify that binding's specification;
- *efficiency*: generated implementations of bindings must be as efficient as possible, as permitted by the features provided by the mechanisms they are generated to use.

The rest of this section discusses and justifies the most important requirements: *openness* and *abstraction*.

2.1 The scope of openness in Erlang/C bindings

The need for openness in Erlang/C bindings is similar to the need for openness in distributed bindings, in distributed systems. Openness in this context implies making the internal configuration of bindings explicit, and controllable from their environment. A general conceptual framework for the description of open distributed bindings is the *engineering viewpoint* of the ISO RM-ODP (Reference Model of Open Distributed Processing) standard [7, 8]. In RM-ODP, a *binding* supports interactions between two or more objects. The engineering viewpoint especially focuses on the internals of *channels*, that are configurations of objects that implement such bindings, to enable interactions between objects in different *clusters* [8], hence possibly in different computers in a network. A cluster is a group of objects that is an units of activation, checkpointing, migration, etc. There is then a distinction between *distributed bindings*, implemented using channels, and *local bindings*, between objects in the same cluster, which don't require channels. A channel is a configuration of the following kinds of objects [8], as illustrated in Fig. 2 for a channel that binds two objects:

- *stub*: interprets the interactions conveyed by the channel, and performs any necessary transformation (e.g., encoding of data into an external format) or monitoring;
- *binder*: maintains a distributed binding between basic objects;
- *protocol object*: handles communication issues to achieve communication between basic objects (possibly in different OS processes or physical systems);
- *interceptor*: performs checks to enforce policies, transformations of data representations, etc.

The RM-ODP engineering viewpoint was defined so that it can be used to describe any distributed binding implementation, such as ONC RPC [13], Java^{TM} RMI [15], or CORBA [11]. The interfaces between the stubs and the client and server objects are specified either directly using programming language abstractions, e.g., Java^{TM} interfaces in the case of Java^{TM} RMI, or using specific IDLs (Interface Definition Languages) such as CORBA's IDL. The transformations performed by stubs consist typically in marshalling data into an external representation such as RPC's XDR [2]. Binders resolve remote object identifiers, deal with object migration and distributed garbage collection, etc. Lazy binding can be implemented by a client stub, by asking the binder

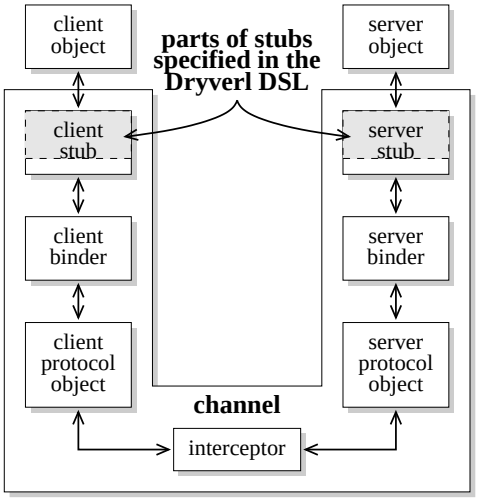

Figure 2. Engineering viewpoint – Channels, and Dryverl-generated stubs

to establish a binding only when the first interaction occurs. Protocol objects implement protocols such as TCP. Interceptors are optional objects that perform adaptations or transparent observations on interactions, such as QoS monitoring.

In addition, the motivation for introducing the engineering viewpoint into RM-ODP, and to specify a common general configuration of channels, was to emphasize the need for flexibility in distributed bindings, e.g., the need to replace protocol objects. There is a need to customize the configuration of channels for instance to perform complex one-to-many interactions, or to adapt the QoS of channels to application-specific requirements. This need for flexible channels has been fulfilled by distributed communication frameworks such as *x*-Kernel [6], Jonathan [1] and FlexiNet [5].

In addition to ODP (Open Distributed Processing) channels, we propose to use RM-ODP engineering viewpoint to describe, as channels, low-level cross-language bindings such as Erlang/C bindings. Therefore, it is natural, yet still original, to apply the objective of openness of channels, to cross-language bindings. ODP standards such as CORBA and ONC RPC on one hand, and cross-language binding compilers such as Dryverl, the Python/C API [3], and Java^{TM}'s JNI [14] on the other hand, are very similar. Notably, in both cases the interfaces of stubs are specified using similar programming language abstractions (e.g., Java^{TM} interfaces) or IDLs. The only difference is in their purposes. The purpose of the latter is to *reuse* existing ("legacy") code in one language, from another language. In that context, ODP systems such as CORBA (1) are too inefficient when networked distribution is not required, and (2) their IDLs are often not flexible enough to define interfaces to code to reuse. This difference does not invalidate the idea of opening cross-language bindings.

The rest of this section discusses how far the requirement of openness can go concerning Erlang/C bindings, i.e., which objects in such specific channels can be made open, and the tradeoffs that have to be made with the objective of abstraction.

As illustrated in Fig. 3, using either port drivers, port programs, or C nodes to implement Erlang/C bindings, the C part of an Erlang/C binding is executed in a different cluster or in a different OS process, than the Erlang processes that interact with it. Even C port drivers cannot access the internals of the emulator, like for instance using the Python/C API or Java^{TM}'s JNI. Therefore, Erlang/C bindings can be considered as channels.

However, only some parts of those channels are open. More precisely, the largest common open parts are the parts in stubs from the external interfaces of stubs (used to interact with client and server objects), down to (but not including) the parts of the stubs that encode data into an external format, as illustrated in Fig. 2. For the sake of *abstraction*, Dryverl allows only specifying such parts that are open in all mechanisms, but offers a high flexibility (*openness*) to specify those parts. Section 3 illustrates what can be specified using the Dryverl binding specification language (the Dryverl DSL).

In the case of port drivers and port programs, communication must take place through Erlang *ports*. A port is a binder, since it maintains a communication binding between the driver or program, on one hand, and an Erlang process called the *port owner* or *connected process*, on the other hand, which is the only process that can interact with the port. Every port is identified with a unique identifier in an Erlang node. A port is created by its port owner, and is destroyed by it (or along with it). Therefore, a port owner is part of the client stub in a channel. A port owner and a port program or port driver can exchange raw binary data (e.g., using the `erlang:port_command/2` BIF), in which case stubs generated by Dryverl must encode/decode Erlang terms into/from the external format. In addition, a port owner and a port driver can also exchange Erlang terms using the port-driver-specific `erlang:port_call/3` BIF, which encodes/decodes terms itself. In the latter case, the external format encoding part of stubs cannot be opened by Dryverl. Therefore, stubs generated by Dryverl either use directly the encoding/decoding functions provided by the chosen mechanism, or implement those encoding/decoding parts if none are provided by the emulator, or if the ones provided are too limited. The upper layers in stubs are left open to developers, to implement custom transformations of decoded data.

Erlang/C bindings implemented as C nodes are quite different. C nodes interact using channels with other Erlang (or C) nodes, using the same protocols as between Erlang nodes. Channels used to communicate are already completely implemented, including stubs, by the Erlang emulator on the Erlang side, and by Erlang/OTP's `erl_interface` library

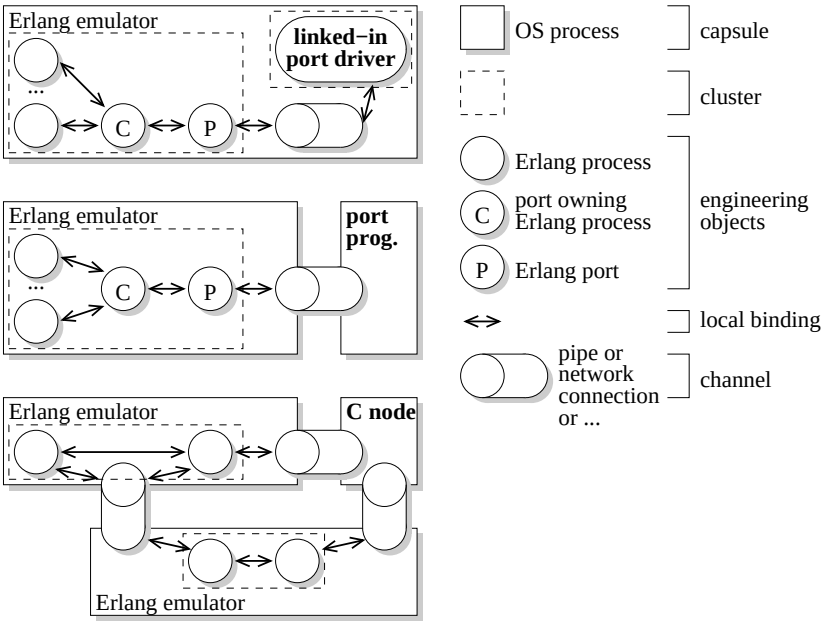

Figure 3. Engineering viewpoint – The three Erlang/C binding mechanisms

on the C side. However, the need for customized channels is still valid in that case, for instance to optimize communication by simplifying data structures (e.g., by converting atoms into integer values), etc. Therefore, Dryverl will also allow generating, from binding specifications, implementations of client stubs as Erlang processes, and of server stubs in C nodes, that use "normal" Erlang channels to communicate.

It must be noted that the C node approach is the only one that offers a degree of customization of binders and protocol objects in channels. For instance, it is possible to use different network protocols (e.g., SSH instead of TCP) to communicate between Erlang nodes. One perspective is to study how to customize channels, on a per-channel basis, when implementing Erlang/C bindings as C nodes. However, this is out of the scope of the Dryverl compiler, since it would be specific only to C nodes.

2.2 Why openness of stubs is essential

Although only parts of Erlang/C bindings' stubs are open by Dryverl, this openness is an essential feature, for two reasons:

- The idioms and type systems of Erlang and C are so different, that the data transformations that are necessary between Erlang and C function calls often cannot be automatically determined, or even specified declaratively in an IDL, but often have to be explicitly implemented by developers.
- Transforming sent and returned data can improve efficiency. For instance, although atoms are represented efficiently in memory in the Erlang emulator, they are encoded like a string in the external format, which is inefficient. It is better to transform atoms into/from small integer constants in the client stubs, since an integer term < 256 is encoded into only two bytes using Erlang's standard external representation.

The differences between idioms and type systems is the biggest issue. For instance, every Erlang function takes terms as arguments, and returns exactly one term. Arguments are always passed by value. On the contrary, C functions accept by-reference arguments: every C function takes values in "in" and "inout" arguments, and returns a value (for non-`void` functions) and values in "out" and "inout" arguments. Therefore, in an Erlang-to-C binding, the Erlang arguments must be mapped into the "in" and "inout" C arguments, and the C return value and the "out" and "inout" argument values must be mapped (or transformed) into the single term returned by the Erlang function. This mapping typically cannot be determined automatically, since the "in", "out" or "inout" nature of a C argument is typically specified unformally (e.g., in English in the GSSAPI standard [18]).

Only a few ODP IDLs allow specifying the "in", "out" or "inout" nature of arguments [11, 16], and IDL-to-language mappings specify how the "out" and "inout" arguments can be returned in languages that don't support them. For instance, CORBA's IDL-to-Java^{TM} mapping specifies that such arguments must be returned by a Java^{TM} method, together with the return value, in a compound object. However, ODP IDLs have little expressive power, which prevents their use to bind many existing ("legacy") complex C libraries such as GSSAPI. Moreover, more flexibility is required to specify complex mappings between Erlang and C arguments and returned values, for instance to fit an ex-

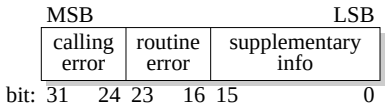

Figure 4. Major status 32-bit integer returned by all GSS-API C functions

isting Erlang behaviour. Among cross-language IDLs, only EDTK [4] allows identifying "out" and "inout" arguments, and specifying transformations, although that IDL's flexibility is limited. Dryverl goes beyond those purely declarative IDL approaches, by allowing specifying those mappings *programmatically*.

A concrete example of the difficulty to map Erlang and C data types is the *major status* 32-bit integer value that is returned by every C function defined in the GSSAPI standard. That value is composed of three fields, as illustrated in Fig. 4:

- The *calling error* is an integer value that can take four possible values (0, 1, 2, or 3) which, if non-zero, indicate low-level errors, e.g., that wrong C structures were passed. This is similar to a C enumerated type, but the non-zero values are instead defined as C macros: `GSS_S_CALL_INACCESSIBLE_READ`, `GSS_S_CALL_INACCESSIBLE_WRITE`, and `GSS_S_CALL_BAD_STRUCTURE`.
- If the *calling error* is zero, then the *routine error* may be non-zero to indicate a higher-level, logical error. This is also similar to an enumerated type, with the 18 possible non-zero values defined as C macros: `GSS_S_BAD_MECH`, `GSS_S_BAD_NAME`, etc.
- If the *routine error* is non-zero, then the *supplementary info* field is a bit field, in which bits are set to provide additional info. Bit masks for all bits are defined as C macros: `GSS_S_CONTINUE_NEEDED`, `GSS_S_DUPLICATE_TOKEN`, `GSS_S_OLD_TOKEN`, etc.

In Erlang, the canonical way to represent enumerations, such as the *calling error* or *routine error* fields, is as disjunctions of atoms. For instance, using the `edoc` notation (i.e., Erlang/OTP's standard code documentation format):

```
CallingError = inaccessible_read |
 inaccessible_write | bad_structure
```

However, existing ODP or cross-language IDLs compilers cannot determine such a transformation automatically, because they cannot identify fields in an integer value, but only the value as a whole. In addition, the enumerations in the GSSAPI are not defined using real C enumerated types, making it impossible for a generic IDL compiler to determine the possible values and their identifiers.

The same problem arises with the *supplementary info* bit field. The canonical way to represent a bit field in Erlang is as a list of atoms, e.g.:

```
InfoStatus = [continue_needed |
 duplicate_token | old_token | ...]
```

Likewise, no IDL allows considering bit fields, so that such transformations must be explicitly specified by developers.

It must be noted that those macros are not formally specified in the GSSAPI standard, but the `gssapi.h` header file that defining them had been written by hand and included as an annex in the standard. The only way to extract those integer constants automatically would be to hand-code a parser specific to the GSSAPI-specific macro naming conventions. Such a tool could not be reused for other bindings.

In addition, the GSSAPI specifies that every function can return only a limited subset of the possible values in the *calling error* field, and only some combinations of *calling error* values and *supplementary info* flags are valid depending on the function. As a consequence, all fields must be considered together to be transformed, and the *major status* field must be transformed as a whole as disjunctions of atoms and `{atom, list_of_flags}` tuples that are different for every function. For instance, the Erlang function in the binding for GSSAPI's function `gss_acquire_cred` must return the following status:

```
MajorStatus = inaccessible_read |
 inaccessible_write | bad_structure |
 complete | bad_mech | bad_nametype |
 bad_name | credentials_expired |
 no_cred | failure
```

But the Erlang function in the binding for GSSAPI's function `gss_init_sec_context` must return:

```
MajorStatus = inaccessible_read |
 inaccessible_write | bad_structure |
 complete | continue_needed | ... |
 bad_mech |
 {failure, ([]|InfoStatus)}
InfoStatus = [old_token |
 duplicate_token]
```

Note that `gss_init_sec_context` can return flags for status `failure`, unlike `gss_acquire_cred`. There is no way to have an IDL compiler determine automatically the right combinations to return for a function, since these are documented in natural language only in the GSSAPI standard, and are not reflected in the C function signatures.

Therefore, such limitations of existing ODP and cross-language IDLs often force developers to write data transformations by hand in *wrapping functions* (or wrappers, or "glue code"), either on the Erlang side or on the C side of a binding, or both. The problem is that wrappers are separate from the IDL specification, although for a binding all

wrappers and the IDL specification are strongly coupled. This makes the maintenance of a binding difficult, because several separate files have to be maintained consistently together.

3. Overview of the Dryverl DSL

The Dryverl DSL is an XML dialect. A complete *"Hello, world!"* example specification is detailed in the following. A specification document is essentially a sequence of binding specifications. Currently, only Erlang-to-C bindings specifications are supported, as `<def-erlang-to-c-binding>` elements. In the future, `<def-c-to-erlang-binding>` elements will also be supported.

```
<def-bindings>
 ...
 <def-erlang-to-c-binding>
  ...
 </def-erlang-to-c-binding>
 <def-erlang-to-c-binding>
  ...
 </def-erlang-to-c-binding>
</def-bindings>
```

3.1 IDL-like elements

The first and last elements (`<def-erlang-input>` and `<def-erlang-output>`) in a binding spec. specify the signature (name and arguments) of the Erlang function that is generated for clients to call the binding, and document the types of the arguments and of the returned term. An example is given in Fig. 5. Those elements functionally correspond to classical ODP and cross-language IDLs.

3.2 Data transformation elements

The elements in between are abstract Erlang and C code fragments that perform data transformation in the client and server stubs, on the Erlang and C sides of a binding, in up to five steps. This goes beyond the functionality of an IDL.

<encode-input> contains Erlang expressions that compute, from the arguments of the Erlang function, an Erlang term to pass by copy to the C part of the binding, and optionally a list of binaries to pass by reference (if supported by the implementation, e.g., the port driver mechanism, or by copy otherwise). **<decode-input>** contains C code that decodes the term passed by copy, and the optional list of passed-by-reference binaries, and assigns values to *call variables*, which are variables passed to the following steps in a call. **<execute-body>** contains C code that processes call variables values, typically by passing them in a call to a C function, and assigns the returned values to call variables. **<encode-output>** contains C code that encodes an Erlang term to return by copy, and optionally a list of binaries to return by reference (if supported, or by copy otherwise), from the values of call variables. **<decode-output>** contains Erlang code block that constructs the term returned by the Erlang function call, from the decoded term received by copy, and from the optional list of returned-by-reference binaries.

In the `<encode-*>` and `<decode-*>` steps, the details of external encoding / decoding of Erlang terms are abstracted to the programmer. In `<encode-input>` (resp., `<decode-output>`) elements, this is achieved by transparently encoding the term returned by the user-specified Erlang expressions (resp., by passing the transparently decoded terms to the user-specified Erlang expressions).

For instance, the `<encode-input>` code fragment in Fig. 5 removes spaces around the string argument, translates atoms into integer values, and then constructs the resulting passed-by-copy term as a tuple. In this example, no binary is sent by reference. How that tuple is encoded is hidden and specific to the implementation generated by Dryverl. The `<decode-output>` code fragment uses the `<erl-output-main-term/>` element, that is substituted at runtime by the decoded term received from the C server stub. This example also illustrates that any Erlang code can be executed.

In `<decode-input>` and `<encode-output>` elements, the details of external decoding / encoding of terms are abstracted by providing abstract *macros* in the form of XML elements that can be included in the user-specified C code.

For instance, the simplified `<decode-input>` code fragment in Fig. 6 decodes the term constructed in element `<encode-input>` in Fig. 5, and assigns the decoded values to call variables. Very similarly, `<encode-output-*>` macro-like XML elements can be used in `<encode-output>` C code fragments to encode the returned Erlang terms.

3.3 Transformation of GSSAPI's *major status*

As another example, the `<decode-output>` elements for GSSAPI functions should all be similar to the following code fragment, which decodes the returned *major status* value, as described in the previous section:

```
<decode-output>
 <create-output-term>
  {MajorStatus, ...} =
     <erl-output-main-term/>,
  {major_status_to_atoms(MajorStatus,
     false, false), ...}
 </create-output-term>
</decode-output>
```

The common decoding code is factorized into function `major_status_to_atoms/3` (cf. Fig. 7), whose 2nd and 3rd arguments are booleans that specify if status bits can be combined with a `complete` (resp. `failure`) status, which is specific to every GSSAPI function. This function is specified in a "verbatim code" element at the top of the binding specification file.

```
<def-erlang-to-c-binding
   name="hello">
 <def-erlang-input
    function-name="print_hello">
  <def-erlang-arg name="Message"
     type-doc="string()"/>
  <def-erlang-arg name="Count"
     type-doc="(integer()>0)|default"/>
 </def-erlang-input>
 <encode-input>
  <encode-input-main-term>
   {string:strip(
      <erlang-arg name="Message"/>),
    case <erlang-arg name="Count"/> of
     C when C > 0 -> C;
     default -> 5
    end}
  </encode-input-main-term>
 </encode-input>
 ...

 <decode-output>
  <create-output-term>
   {ok, PrintedBytes} =
      <erl-output-main-term/>,
   io:format("the end~n", []),
   {ok, PrintedBytes + 9}
  </create-output-term>
 </decode-output>
 <def-erlang-output>
  <def-erlang-return
     type-doc="{ok,PrintedBytes}"/>
  <def-erlang-type-doc>
     PrintedBytes=integer()>0
  </def-erlang-type-doc>
 </def-erlang-output>
</def-erlang-to-c-binding>
```

Figure 5. Erlang interface definition and encoding/decoding code

```
<def-erlang-gen-server-header>
major_status_to_atoms(MajStatus,
   CompBitsOK, FailBitsOK) ->
 CallingError =
    (MajStatus bsr 24) band 16#ff,
 case CallingError of
  0 -> RoutineError =
     (MajStatus bsr 16) band 16#ff,
   Atom = re_to_a(RoutineError),
   Bits = MajStatus band 16#ffff,
   add_atoms(Atom, Bits,
      CompBitsOK, FailBitsOK);
  _ -> ce_to_a(CallingError)
 end.

ce_to_a(1) -> inaccessible_read;
...
ce_to_a(3) -> bad_structure.

re_to_a(0) -> complete;
...
re_to_a(18) -> name_not_mn.

add_atoms(Atom, Bits,
   CompBitsOK, FailBitsOK) ->
 if (Atom==complete) and (Bits==1) ->
   continue_needed;
  ((CompBitsOK and (Atom==complete))
   or
   (FailBitsOK and (Atom==failure)))
  and (Bits band 1 == 0) ->
   {Atom, dec_bits(Bits bsr 1,1,[])};
  Bits == 0 -> Atom
 end.
dec_bits(_Bits, 5, Atoms) -> Atoms;
dec_bits(Bits, BitNum, Atoms)
   when BitNum > 0, BitNum &lt; 5 ->
 NewA = if (Bits band 1) == 1 ->
     [bit_to_a(BitNum)|Atoms];
   true -> Atoms end,
 dec_bits(Bits bsr 1,BitNum+1,NewA).
bit_to_a(1) -> duplicate_token;
...
bit_to_a(4) -> gap_token.
</def-erlang-gen-server-header>
```

Figure 7. Erlang code for decoding GSSAPI's major status values

```
<decode-input>
 <assign-c-call-variables>
  <decode-input-tuple>
   <decode-input-tuple-size-into>
    &amp;tuple_arity
   </decode-input-tuple-size-into>
   <decode-input-tuple-contents>
    <!- Decode the string: ->
    ...
    <decode-input-string-into>
     <c-call-variable
        name="message"/>
    </decode-input-string-into>
    <c-call-variable
       name="message_length"/> =
     string_length;
    <!- Decode the integer: ->
    <decode-input-ulong-into>
     &amp;<c-call-variable
             name="count"/>
    </decode-input-ulong-into>
   </decode-input-tuple-contents>
  </decode-input-tuple>
 </assign-c-call-variables>
</decode-input>
```

Figure 6. C code for decoding input

3.4 C body elements

`<execute-body>` C code fragments are typically simpler than the other elements. However, since any C code is allowed, complex computations can still be specified, for instance using loops, as illustrated in Fig. 8.

Even in the simpler case the `<execute-body>` only performs a C function call, this flexibility allows easily handling all "in", "out" and "inout" arguments in a uniform way: such differences are translated into which call variables are assigned decoded and transformed values in the `<decode-input>` step, and which ones have their values transformed and encoded in the `<encode-output>` step.

3.5 Summary

The examples above demonstrate that Dryverl allows specifying more concisely the parts of stubs that transform data, which are usually written in wrappers, altogether with IDL-level information in binding specifications. Since all those aspects of a binding are strongly coupled together, specifying them in a single document helps maintaining them mutually consistent, and therefore helps specifying complex bindings such as for GSSAPI.

In addition, knowing about all parts of a binding, including the data transformation parts of stubs, allows the Dryverl compiler 1) to perform global semantic checking of specifications, and 2) to perform global optimizations specific to the implementation mechanism (C port driver, C port program or C node). For instance, as detailed in the next section, when compiling a binding into a C port driver, if a client stub never transmits binaries by reference, the compiler can check that the server stub also never tries to receive binaries by reference, and in that case optimized implementations of client and server stubs are generated by Dryverl.

```
<execute-body>
 <process-c-call-variables>
  int i =
     <c-call-variable name="count"/>;
  int printed_bytes = 0;
  while (i > 0) {
   printf("hello, %s!\r\n",
      <c-call-variable
       name="message"/>);
   i--;
   printed_bytes += 10 +
      <c-call-variable
       name="message_length"/>;
  }
  free(<c-call-variable
        name="message"/>);
  <c-call-variable
   name="printed_bytes"/> =
      printed_bytes;
 </process-c-call-variables>
</execute-body>
```

Figure 8. C code body

4. Bindings as port drivers

Currently, Dryverl supports only the generation of bindings as C port drivers. This had the highest priority, since this is the most efficient form of binding, and C port drivers are the most difficult to deal with regarding *abstraction* and *efficiency*, two of Dryverl's requirements. There are several ways to communicate between a C port driver and its Erlang port owning process, of which three have been retained as the canonical ones used to implement Dryverl bindings:

- Erlang → driver: call to the `port_command/2` BIF or to the `port_call/3` BIF,
- driver → Erlang: call to the `driver_output_term/3` C function, or return of a value to a `port_call/3` call, which is possible only if that BIF was called from Erlang instead of `port_command/3`.

The problem is to choose which one to use in every case, since they have different limitations and efficiencies. The following details the possible cases, and the corresponding choices made in Dryverl.

4.1 Comparison of `port_command/2` and `port_call/3`

Since the primary purpose of Erlang's port driver mechanism is to implement efficient I/O drivers, e.g., for network or database access, one of its important features is to allow transfering binaries by reference between Erlang programs and drivers. However, the functions that support this feature, the `port_command/2` Erlang BIF and the `driver_output_term/3` C function (and similar functions), have limitations:

- `driver_output_term/3` uses a special format for encoding terms which supports only a few types of Erlang terms, and `port_command/2` does not encode terms (encoding must be done explicitly before calling it);
- they are inefficient, when used only to pass simple Erlang terms to and from a driver without needing to pass binaries by reference.

One cause of inefficiency is that using those functions is like asynchronous from the port owner viewpoint: when calling `port_command/2`, the only way to return data to Erlang is by calling C functions like `driver_output_term/3` (or similar), which actually sends an Erlang message which the port owner must then `receive`.

Using `port_call/3` is more efficient, and it is synchronous: data returned by the driver can be direcly returned in a `port_call/3` call. In addition, it uses Erlang's standard external format for encoding/decoding terms, which is also used to communicate between Erlang nodes and C nodes. However, it cannot transmit binaries by reference like `port_command/2` and `driver_output_term/3`.

In order to unify the external encoding format, when using `port_command/2` and `driver_output_term/3`, the stubs generated by Dryverl call the `term_to_binary/1` and `binary_to_term/1` Erlang BIFs transparently, and the `erl_interface` C library, to encode/decode terms into/from Erlang's standard external format.

4.2 "One-way" vs. "two-way" bindings

A Dryverl binding can be specified as "one-way", by omitting together the `<encode-output>`, `<decode- output>` and `<def-erlang-output>` elements in the binding's specification. In that case, calls do not return values and can be non-blocking, which simplifies the general optimization problem: the generated code is simpler, and only a choice between `port_command/2` and `port_call/3` has to be made.

4.3 Asynchronous calls

Erlang calls to `port_command/2` and `port_call/3` provoke the execution of C functions of the driver, in the same OS thread that executes the Erlang process that performs the call. Therefore, the execution of the emulator is suspended during the driver execution. In recent versions of Erlang/OTP, several OS threads are used to schedule all the Erlang processes, to improve performance on SMP systems. However, while it executes a driver function, an OS thread cannot schedule Erlang processes, which may still have a negative impact on reactivity and performance.

Therefore, the emulator maintains a pool of OS threads reserved for the asynchronous execution of driver tasks. The number of such threads is zero by default, and can be set using the `+A` option when starting the emulator. Every driver can optionally add tasks into its specific queue, which are then executed asynchronously by those threads. Every Dryverl-generated driver implementation provides an option to use that mechanism at runtime for all calls, to execute asynchronously the `<execute-body>` and (for "two-way" bindings) the `<encode-output>` steps for every binding.

However, asynchronous driver tasks can return data to Erlang only using the `driver_output_term/3` C function (and similar). It is not possible for a task to return data in a `port_call/3` call, since that BIF is synchronous.

4.4 Synthesis: Dryverl's optimization algorithm

At compile time, the Dryverl compiler can determine *static aspects* of a Dryverl binding specification: 1) if binary terms *may* be passed by reference or not, from Erlang to C and/or from C to Erlang, and 2) if calls return values ("two-way" calls), or not ("one-way" calls). *Dynamic aspects* are determined at runtime, for every call, by Dryverl-generated bindings: 1) if binary terms *have to* be passed by reference or not, from Erlang to C and/or from C to Erlang, and 2) if the driver is setup at runtime to execute calls asynchronously.

The runtime optimization algorithm, considering *dynamic aspects*, boils down to two rules: **1)** if binaries have to be passed by reference from Erlang to C, then call `port_command/2`, else call `port_call/3`, and **2)** if `port_call/3` was called, and no binary has to be returned by reference, and the driver is not set to asynchronous mode, then return data directly in the `port_call/3` call, else call the `driver_output_term/3` C function.

When `port_call/3` is called, but is not used to return data, a term is returned anyway, which either notifies the Erlang client stub to `receive` the data in a message instead, or is ignored in the case of "one-way" bindings.

The Dryverl compiler considers the *static aspects* in order to not generate code for cases that would never happen at runtime, and to not generate unnecessary tests for *dynamic aspects*. If the `<encode-input>` contains no `<encode-input-byref-binary-list>`, i.e., if no binary has ever to be passed from Erlang to C, then the generated binding always calls `port_call/3`, and no code is generated to test in the driver, at runtime, if `port_command/2` was called instead. If the binding is "one-way", then no code is generated to return data. If the `<encode-output>` contains no `<encode-byref-binary>`, i.e., if no binary has ever to be returned by reference, then no code is generated to test at runtime if such binaries have to be returned, and to actually return them.

5. Related works

Works related to Dryverl include distributed processing systems available in Erlang, such as Erlang/OTP's IC (Interface Compiler) CORBA IDL-to-Erlang compiler and the Orber CORBA ORB that are integrated into Erlang/OTP, and cross-language binding generators such as SWIG [12], Java^{TM}'s JNI (Java Native Interface) [14], and the Erlang/C-specific IG (Interface Generator) [17] and EDTK (Erlang Driver Toolkit) [4].

The main contribution of Dryverl, compared to those works, is the openness of the data transformation parts of stubs, as detailed in Section 2, which is essential to address the differences in type systems and idioms between Erlang and C (or other languages). All other approaches limit themselves to allow specifying the external interfaces of bindings' stubs, in the C syntax or a similar syntax (e.g., the CORBA IDL), and to generate bindings as black boxes. This approach is sufficient for bindings between semantically similar languages, for instance between C and Python which both allow "out" and "inout" arguments, but not between Erlang and other languages (e.g., C). In that case, complex transformations of exchanged data must often be performed in bindings.

Only EDTK goes further, by allowing specifying limited categories of transformations declaratively, e.g., to create Erlang terms to be returned by an Erlang client stub. However, that specification language is limited, so that performing complex transformations still requires writing complex wrappers by hand. Dryverl's approach to provide complete programmatic control over data transformations in stubs is more general, and allows handling difficult cases more easily. Another limitation is that EDTK supports only C port drivers, and IG only C port programs. EDTK also supports executing drivers as C port programs, but this does not fully use the capabilities of the C port program approach. Dryverl is actually a rewrite of EDTK and is intended as a continuation of that project, reusing many of its implementation ideas (value maps, etc.), while improving and generalizing it. For instance, Dryverl supports "one-way" (i.e., asynchronous) bindings, like some distributed processing IDL compilers [11, 16]. In addition, Dryverl and EDTK both support "out" and "inout" C arguments, which IG doesn't. However, Dryverl will also reuse ideas from IG as well, to support C-to-Erlang bindings.

A notable related work is GreenCard [9], a Haskell-to-C binding compiler. GreenCard's specification language is very similar to Dryverl's, as it also allows to specify for a binding both the Haskell interface, and the several steps of a call including all transformations of data between Haskell and C code. More precisely, GreenCard allows defining values of C variables using macros (called Data Interface Schemes) that transform the Haskell arguments and, likewise, allows defining the Haskell function return values using macros that transform the C variables. Those macros are rewritten by the Haskell compiler into source C code that is blended with the C source code generated from the Haskell source code. Since ultimately all code is in C, there is no need to encode/decode data to interact between generated C code and external C code. On the contrary, encoding/decoding of data is mandatory in Erlang/C bindings, and Erlang code is never compiled into C code, which makes it mandatory to separate the Erlang and C parts as stubs in bindings, and to clearly identify and control which data they exchange in encoded form, to optimize communication. Dryverl addresses those issues specific to Erlang.

The Dryverl DSL is more verbose than the other specification languages. One perspective is to implement additional, higher-level compilers, that accept such simpler specification languages, and that generate Dryverl specifications. This would make Dryverl the "assembly language" of Erlang/C binding development, as illustrated in Fig. 1. However, whenever specifying a given binding using another compiler would require writting wrappers to transform data, Dryverl should always be used directly instead, since it would make writing and maintaining that binding easier.

6. Conclusion and perspectives

Dryverl generates efficient implementations of Erlang/C bindings, from abstract specifications. The most important feature of Dryverl is the openness of stubs in bindings, to allow precisely specifying transformations of data that are exchanged between Erlang and C. This feature is essential to bridge the gap between C's and Erlang's idioms and type systems, without requiring developers to write a lot of wrappers ("glue code") around generated bindings. The interest of Dryverl is demonstrated in a (still under development) binding for the GSSAPI standard [18], which is used to secure many client-server applications.

Dryverl is an ongoing project. Dryverl currently supports only Erlang-to-C bindings implemented as C port drivers, but it will also support C-to-Erlang bindings, and implementations as C port programs and C nodes. Dryverl is Free Software, distributed under the BSD License, and can be downloaded from the Dryverl web site [10].

References

[1] B. Dumant, F. Horn, F. D. Tran, and J.-B. Stefani. Jonathan: an open distributed processing environment in Java. In *Proceedings of the IFIP International Conference on Distributed Systems Platforms and Open Distributed Processing (Middleware'98)*, pages 175–190. Springer–Verlag, 1998.

[2] M. Eisler. XDR: External data representation standard. RFC 4506 (Standard), May 2006.

[3] Python Software Foundation. Python/C API reference manual – `http://docs.python.org/api/api.html`.

[4] S. L. Fritchie. The evolution of Erlang drivers and the Erlang driver toolkit. In *Proceedings of the 1st ACM SIGPLAN Erlang Workshop*, pages 34–44. ACM, Oct. 2002.

[5] R. Hayton, A. Herbert, and D. Donaldson. FlexiNet – a flexible component oriented middleware system. In *Proceedings of the 8th ACM SIGOPS European Workshop: support for composing distributed applications (EW98)*, pages 17–24, Sept. 1998.

[6] N. C. Hutchinson, L. L. Peterson, M. B. Abbott, and S. O'Malley. RPC in the x-Kernel: evaluating new design techniques. In *Proceedings of the 12th ACM Symposium on Operating Systems Principles (SOSP'89)*, pages 91–101. ACM, Dec. 1989.

[7] ISO/ITU–T. Reference Model of Open Distributed Processing (RM-ODP), part 2: Foundations – international standard ISO/IEC 10746–2, ITU–T recommendation X.902, Nov. 1995.

[8] ISO/ITU–T. Reference Model of Open Distributed Processing (RM-ODP), part 3: Architecture – international standard ISO/IEC 10746–3, ITU–T recommendation X.903, Nov. 1995.

[9] S. P. Jones, T. Nordin, and A. Reid. Green Card: a foreign-language interface for Haskell. In *Proceedings of the ACM SIGPLAN Haskell Workshop*, June 1997.

[10] ObjectWeb. Dryverl – `http://dryverl.objectweb.org/`.

[11] OMG. Common Object Request Broker Architecture (CORBA): Core specification, version 3.0.3, Mar. 2004.

[12] Simplified Wrapper and Interface Generator. SWIG 1.3 doc. – `http://www.swig.org/Doc1.3/`.

[13] R. Srinivasan. RPC: Remote Procedure Call protocol specification version 2. RFC 1831 (Proposed Standard), Aug. 1995.

[14] Sun Microsystems, Inc. Java Native Interface Specification – `http://java.sun.com/j2se/1.3/docs/guide/jni/spec/jniTOC.doc.html`.

[15] Sun Microsystems, Inc. Java Remote Method Invocation specification – `http://java.sun.com/j2se/1.4.2/docs/guide/rmi/spec/rmiTOC.html`.

[16] TINA-C. TINA Object Definition Language manual, version 2.3, July 1996.

[17] T. Törnkvist. IG - the Interface Generator – `http://www.bluetail.com/tobbe/ig/doc.new/`.

[18] J. Wray. Generic Security Service API version 2 : C-bindings. RFC 2744 (Proposed Standard), Jan. 2000.