

Dryverl: a Flexible Erlang/C Binding Compiler

Romain Lenglet and Shigeru Chiba
Tokyo Institute of Technology

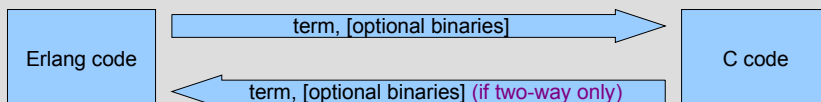
5th ACM Erlang Workshop
2006-09-16

The problem: Erlang/C bindings

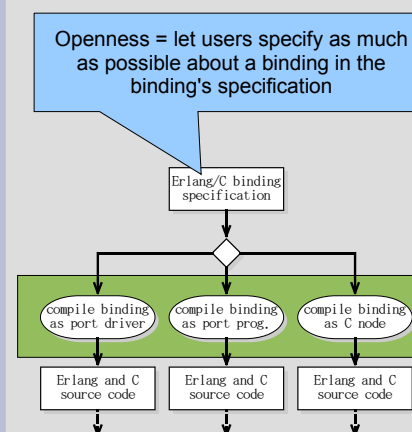
- How to integrate any Erlang and C code?
 - Dryverl generates all the Erlang and C code that implements a binding
- Purpose: offer maximum openness
 - Programmers control much of the implementation of a binding
- While also meeting those requirements:
 - Transparency / available mechanism
 - Hide ≠ and tricky implementation details
 - Cope with little openness / flexibility
 - Efficiency of generated code

What is an Erlang to C binding?

- No standard terminology for cross-language integration (?)
 - 1 binding = 1 Erlang function implemented in C
 - Information transmitted to/from C code:
 - 1 Erlang term
 - + 1 optional list of binaries (the port driver mechanism allows to pass binaries by reference)
 - Interactions can be two-way (interrogations) or one-way (announcements)
- 3 available mechanisms: driver, port, node



Openness = expressiveness of the specification language

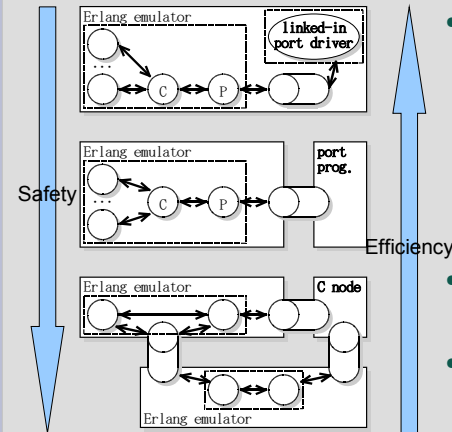


- Purpose of Dryverl
 - Generates all the code given a binding specification
 - Generate code for any available mechanism
 - Openness: offers maximum control over a binding's implementation

Why is openness important?

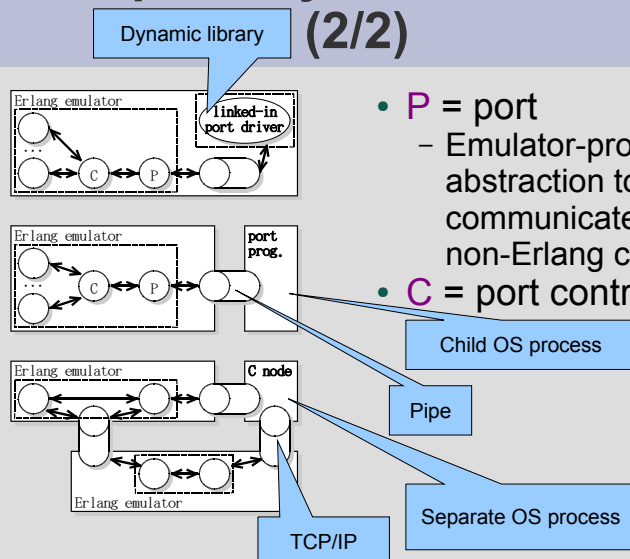
- **Integration of legacy Erlang APIs and C code and Adaptation of idioms and type systems**
 - Most existing compilers are not flexible enough and require **wrappers**
 - More verbose, more difficult to maintain
 - Using Dryverl: **no need for wrappers**
- **Improved performance**
 - **Terms must be encoded/decoded** in Erlang/C bindings
 - Fine control of encoding helps
 - E.g.: atoms encoded as integers
 - Static global optimizations: easier iff a spec contains everything, and in a structured format

Transparency / 3 mechanisms (1/2)



- **Strong similarities**
 - Erlang terms must be encoded/decoded
 - Same level of abstraction
 - Similar openness / level of control
- **But ≠ efficiency / safety trade-offs**
- **Transparency: hide differences in details**

Transparency / 3 mechanisms (2/2)



- **P = port**
 - Emulator-provided abstraction to communicate with non-Erlang code
- **C = port controller**

How is Dryverl open?

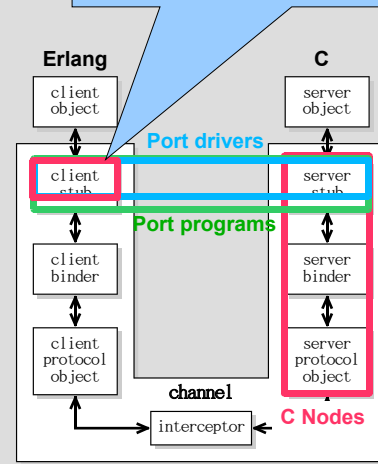
- **How to achieve openness?**
 - = expressiveness of the specification language
 - Mix of declarative and **programmatic approach**
 - Declarative where appropriate
 - Signatures of functions
 - Mostly **fragments of Erlang and C code + macros**
 - Encoding/decoding of terms
 - Dictionaries (“value maps” that map C data and integers)
 - More concise than using wrappers
 - Existing spec languages are declarative only
 - Simpler for simple cases
 - But more difficult for difficult cases
- Openness is **limited by transparency**
 - Dryverl opens only what can be opened using all three mechanisms

What can be open? (1/2)

- Erlang/C bindings are **distributed bindings**
 - Term must be encoded/decoded
 - ...
 - Similar to bindings in CORBA, Java RMI, etc.
- Model: ISO RM-ODP engineering viewpoint
 - General model of distributed bindings / **channels**

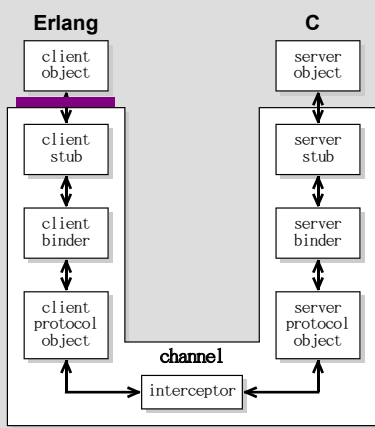
What can be open? (2/2)

"Artificial" client stub process will be added in the case of nodes, to get the same level of openness



- Erlang/C bindings are **distributed bindings / channels**
- Port drivers are the openness limiting factor
- Only **stubs** are open in all 3
- Open **except** for the term encoding / decoding part

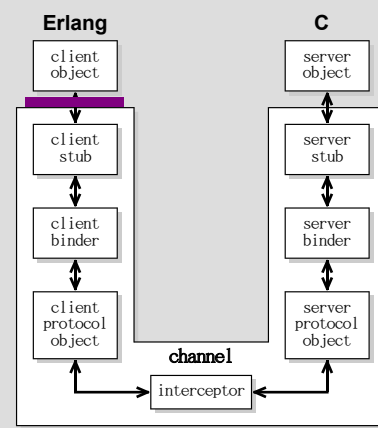
Binding specification: signature (1/2)



- Signature of the Erlang function
 - Arity
 - Two- or one-way
- Documentation
 - In OTP's edoc format
 - Type and name of arguments
 - Type of returned term

Binding specification: signature (2/2)

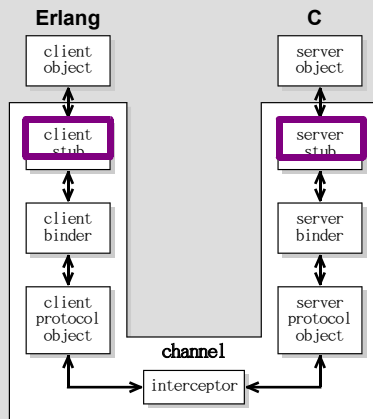
declarative specification



```

<def-erlang-input
  function-
  name="print_hello">
  <def-erlang-arg
    name="Message" type-
    doc="string()"/>
  </def-erlang-input>
  ...
  <def-erlang-output>
    <def-erlang-return
      type-
      doc="{ok,int()}" />
    </def-erlang-output>
  
```

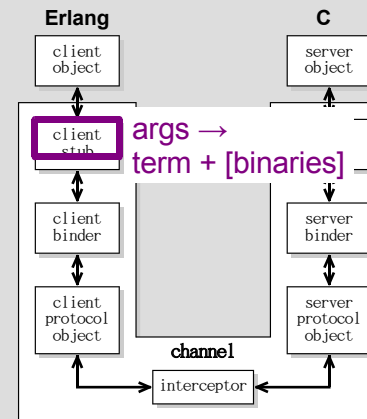
Binding specification: data transformation



- **Four data transformation parts** in Erlang and C stubs
 - Arguments into 1 term + [binaries]
 - E.g. atoms become integer constants
 - 1 term + [binaries] into C variables
 - And vice-versa

Binding specification: input data transformation (1/2)

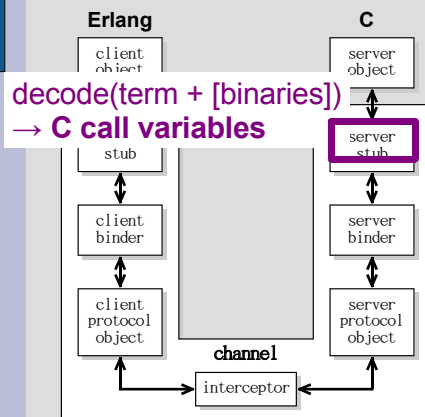
programmatic specification



```
<encode-input>
  <encode-input-main-term>
    string:strip(
      <erlang-arg
        name="Message"/>)
  </encode-input-main-term>
</encode-input>
```

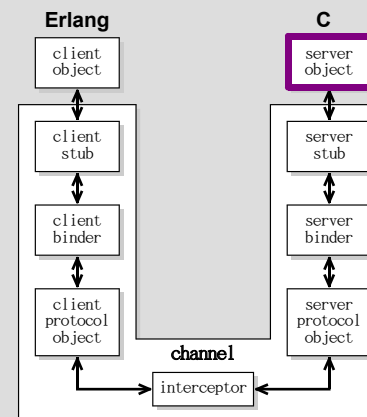
Binding specification: input data transformation (2/2)

programmatic specification



```
<decode-input>
  <assign-c-call-variables>
    /*...*/
  <decode-input-string-into>
    <c-call-variable
      name="msg"/>
  </decode-input-string-into>
</assign-c-call-variables>
</decode-input>
```

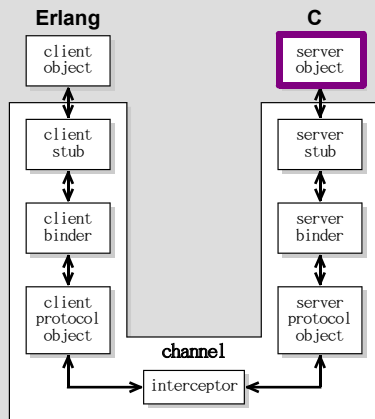
Binding specification: C implementation (1/2)



- **Arbitrary C code**
 - Typically, calls functions of a legacy C library
 - Processes values of the C call variables
 - Modifies the C call variables

Binding specification: C implementation (2/2)

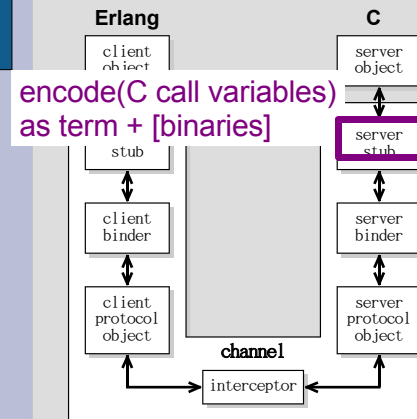
programmatic specification



```
<execute-body>
<process-c-call-variables>
  int i;
  i = printf("hello,
  %s\r\n",
  <c-call-variable
  name="msg"/>);
  <c-call-variable
  name="prt"/> = i;
</process-c-call-variables>
</execute-body>
```

Binding specification: output data transformation (1/2)

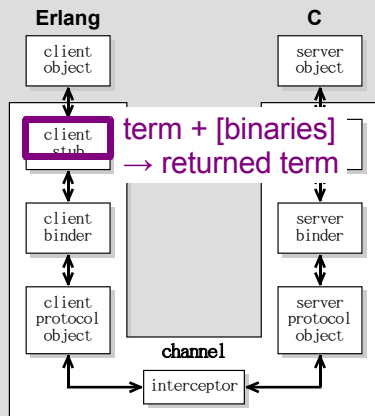
programmatic specification



```
<encode-output>
<encode-output-...>
  <encode-output-ulong>
    <c-call-variable
    name="prt"/>
  </encode-output-ulong>
</encode-output...>
</encode-output>
```

Binding specification: output data transformation (2/2)

programmatic specification



```
<decode-output>
<create-output-term>
  Prt = <erl-output-main-term/>,
  {ok, Prt}
</create-output-term>
</decode-input>
```

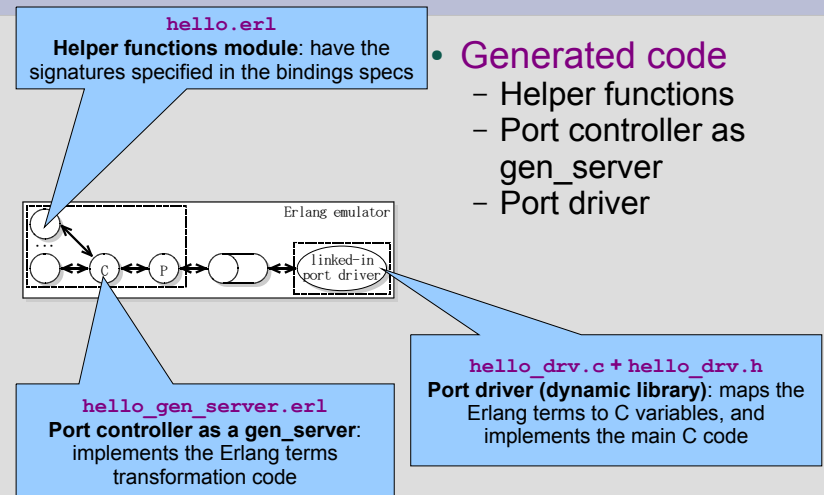
Dryverl's binding specification language

- The specification language allows specifying
 - The Erlang function signature
 - Data transformation Erlang and C code
 - Including code usually in wrappers
 - The executed C code
 - Full control is given on those parts
- The specification language is an XML dialect
 - Specified and documented in an **XML Schema**
- Dryverl is a set of **XSLT 1.0 stylesheets**

Bindings as port drivers (1/2)

- Only port drivers are currently supported as a target
- This was the top priority because:
 - **Best performance**
 - Only mechanism which allows passing binaries by reference
 - **Least open**: this was the limiting factor for the openness of Dryverl
 - **Most difficult** to deal with
 - Was designed for I/O drivers and fits well that purpose
 - But not adapted to integrate arbitrary C code

Bindings as port drivers (2/2)



Related works (1/2)

- Erlang/C binding generators
 - EDTK 1.1
 - The ancestor of Dryverl: many similarities
 - Dryverl is more open and powerful
 - Complete critical analysis on the Dryverl website
 - IG (Interface Generator)
 - Supports C-to-Erlang bindings
 - Little openness: no way to specify function signatures...
- Cross-language bindings for other languages
 - JNI, Python/C, GreenCard, etc.
 - Bindings between similar languages
 - Allow direct interactions without requiring encoding/decoding
 - Too different from the Erlang/C case

Related works (2/2)

- Open Distributed Processing frameworks
 - Standards: Java RMI, CORBA...
 - Open: xKernel, ObjectWeb Jonathan, FlexiNet...
 - Similarity
 - Stub compilers
 - Very similar architecture (cf. ISO RM-ODP)
 - Openness is much more limited in Erlang
 - No control of binders and protocol objects
 - Implemented in the “black-box” emulator
 - When control is offered (cf. inet_ssl...), impossible to control every binding separately

Conclusion

- Dryverl generates the complete implementation of Erlang/C bindings
 - **Openness**: offers full control over transformation of data, and the signatures of Erlang functions
 - **Can target transparently any mechanism**
 - **Efficiency**: automatic choice of best alternatives to perform a binding call
 - Drawback: XML is verbose, but Dryverl is a **backend for higher-level languages**
- Perspectives
 - Support port programs and C nodes
 - C-to-Erlang bindings

The Dryverl project

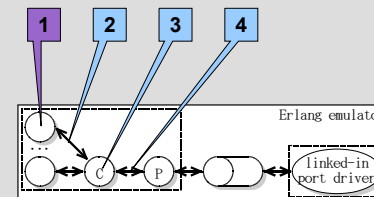


- Dryverl is Free Software (BSD license)
- It can be downloaded from:

<http://dryverl.objectweb.org/>

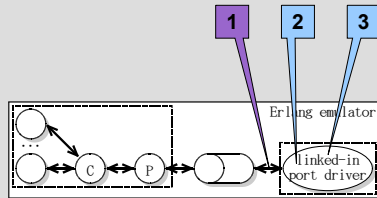
Bonus slides (^_^)

Bindings as port drivers



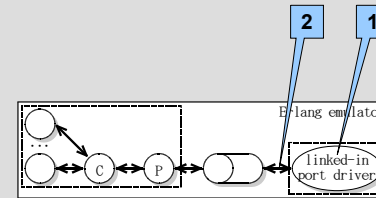
- **Invocation (1/2)**
 1. Helper function call
 - `print_hello()`
 2. `gen_server` "call"
 - Or "cast", if one-way binding
 3. Transforms args into 1 term + [binaries]
 4. Calls `port_call`
 - Or encodes term & calls `port_command` if [binaries] ≠ []

Bindings as port drivers



- **Invocation (2/2)**
 1. The emulator calls `outputv()` or `call()`
 - Whether we called `port_command()` or `port_call()`
 2. Decodes the Erlang term + [binaries] into C variables
 3. Main C code: processes and modifies those C variables

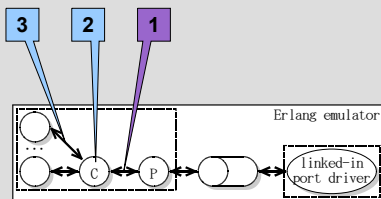
Bindings as port drivers



For two-way bindings only

- **Termination (1/2)**
 1. Encodes C variables into 1 Erlang term + [binaries]
 2. Calls `driver_output_term`
 - Or returns the term in the `port_call` call, if [binaries] = []
 - If [binaries] ≠ [] and `port_call` was called, returns `noreply`

Bindings as port drivers



For two-way bindings only

- **Termination (2/2)**
 1. Receives 1 term + [binaries] as a message
 1. Or the `port_call` call returns a term
 2. Transforms the term + [binaries] into one term to return
 3. Unblocks the `gen_server` "call"
 - `gen_server:reply()`

Bindings as port drivers

- Asynchronous operations
 - Multiple client calls can be executed simultaneously
 - `gen_server` "casts" for one-way bindings
- Problems with current implementation
 - Uses emulator's `driver_async` function to **start a concurrent task for every call**
 - Calls `driver_output_term` in tasks: **not allowed**, although worked in my small-scale tests
- Perspectives
 - **Start multiple ports**, and avoid `driver_async`
 - One `gen_server` will dispatch to port controllers
 - Transparent to clients