Typed Callbacks for More Robust Behaviours

Stavros Aronis 1  Konstantinos Sagonas 1,2

1 School of Electrical and Computer Engineering, National Technical University of Athens, Greece 2 Department of Information Technology, Uppsala University, Sweden
dialyzer@softlab.ntua.gr

Abstract

Behaviours are one of the most widely used features of Erlang/OTP. They offer a convenient and well-tested abstraction layer for frequently employed design patterns in concurrent Erlang programming. In effect, they allow programmers to focus on the functional characteristics of their applications without having to resort to Erlang’s concurrency-supporting primitives. However, when it comes to ensuring that behaviours are properly used and callbacks are as expected, the current Erlang/OTP compiler performs only minimal checks. This is no fault of the compiler though, because most/all of the callbacks’ API exists only in the documentation or the comments accompanying the code; as such, it cannot always be trusted and it is almost impossible to have it mechanically processed. In this paper, we propose a small extension to the language of function specifications of Erlang to allow the formal definition of the behaviours’ callback API. We have implemented this extension on the development branch of Erlang/OTP and provide evidence of how it can be leveraged by static analysis tools such as Dialyzer to detect behaviour misuses.

Categories and Subject Descriptors D.2.4 [Software Engineering]: Software/Program Verification—Reliability; Programming by contract; D.2.12 [Software Engineering]: Interoperability—Interface definition languages

General Terms Documentation, Languages, Reliability

Keywords abstract classes, behaviours, Erlang

1. Introduction

The key feature that sets Erlang [1][2] apart from any other functional programming language is its inherent support for concurrency. It is very easy for the developer to spawn a new process (even if only for executing a simple function) by making use of the built-in primitives, which allow the creation of light-weight threads with inherent support for message passing and process monitoring. Using these primitives, very sophisticated scalable systems and concurrent applications can be built.

Nevertheless, the direct use of these primitives is not always to the benefit of the developer. Besides being quite low-level, the use of those built-in mechanisms for process monitoring and supervision requires extra effort from the programmer, resulting in subtle differences in the application and, consequently, to minor or major bugs. This is where the Erlang/OTP implementation [5] comes to the rescue with the behaviours, an invaluable extension to the core language. Behaviours provide the means to wrap up all the tricky details that need to be taken into account in the development of concurrent software, letting the developer focus solely on the application’s desired functionality. This is achieved with the implementation of simple callback functions which plug into the behaviour’s inner machinery without using any of the concurrency primitives. Behaviours then take care of lost messages, individual process crashes and all the mundane errors that may happen.

Despite the use of behaviours, the developer might occasionally make trivial mistakes, such as forgetting to implement a particular callback. The compiler can detect such an omission, but this is all the help that is currently provided by Erlang/OTP. On more subtle errors, such as implementing a callback in way that is incorrect or incompatible with the rest of the behaviour’s code, there is no warning. A common case for that is when a callback returns a value that is incomprehensible by the behaviour’s inner workings. To amend this, popular text editors provide code templates that include boilerplate documentation for all the major behaviours and can be simply filled in to ensure correctness of the implementation. Still, there is no guarantee that this is done correctly by the developers who use such templates.

In this paper we will describe a better way to accurately document and describe the interface between a behaviour and its callback functions using the widely used language of function specifications that modern Erlang comes along with. We will demonstrate why this small language extension is useful in practice, by showing existing instances of misused behaviours in Erlang/OTP, both in their documentation and their usage by other applications. Finally, we will describe how tools like Dialyzer [3][7] can benefit from type information about callbacks to discover discrepancies in the implementation of behaviours’ callbacks that go well beyond the aforementioned simple checks which are carried out during compilation.
The rest of this paper is organized as follows. In Section 2, we describe behaviours in more detail, along with the problems that current callback specification is fraught with. In Section 3, we review how function specifications are currently used for purposes similar to those of our proposition. The main contribution of this paper is presented in Section 4, where we describe a simple language extension to describe the signatures of behaviour calls to the behaviour’s API plus the implementation of the callbacks.

Such segregation of code between the behaviour and callback modules improves robustness. This is mainly due to the generic part of the behaviour already being well-tested and compatible with much of the functionality that comes with OTP. The behaviour library code may be non-deterministic and its dynamic behaviour may prove a challenge for static analysis but all of this is external to the code that the programmer has to write. The latter constitutes only of the application-specific callback functionality (which should be in accordance with the specifications imposed by the declared behaviour). All sorts of message-passing or invocation of low-level concurrency primitives are delegated to the underlying behaviour implementation. Hence, callbacks become simple and deterministic in nature, aiding both the programmer and the static analysis tools to reason better about the code.

Table 1 lists the most common behaviours in Erlang/OTP, together with the callbacks required by each.

### 2. Behaviours in Erlang

**Behaviours** provide general, reusable solutions to commonly occurring tasks in concurrency-oriented programming. They divide the programming task in half, provide its generic part and require that the programmer writes the application-specific part. In this respect, they are similar to abstract classes as they appear in object-oriented languages such as Java, which come with part of their functionality readily available but need to be completed with the definition of the required abstract methods.

Behaviours are also the standard method to access the most fundamental of Erlang’s unique features: message-passing, fault-tolerance, concurrency and availability. The most commonly used behaviours in Erlang either implement abstract protocol patterns, (e.g. servers, finite state machines and event managers) or ensure that a system —or parts of it— are properly relaunched in cases of failure (supervisor and application behaviours). In all of these cases, the behaviour module contains the library functions required to implement the pattern or mechanism, plus some documentation outlining the expected functionality of the callbacks. What is needed to make a concrete application out of this pattern is merely some code that wraps calls to the behaviour’s API plus the implementation of the callbacks.

### 2.1 Current declaration of a behaviour

Users can also define their own behaviours. Initially, the code has to be separated into an infrastructure part (i.e. one that is shared between all uses of the behaviour) and a part specific for the current application. The interface between them also needs to be designated so that other modules are able to provide callbacks and utilize the behaviour. How this procedure is carried out is entirely a matter of software design. The current compiler supports this separation and warns about missing callbacks if the module describing a behaviour exports the aptly-named special function: behaviour_info/1. The behaviour_info(callbacks) clause of this function should return the expected callbacks in the form of a list of tuples containing the names of the callback functions as atoms and their arities as integers. The example in Listing 4 is taken from the generic server behaviour.

<table>
<thead>
<tr>
<th>Behaviour</th>
<th>Description</th>
<th>Module</th>
<th>Callbacks</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Generic server</strong></td>
<td>A server that can accept calls (that require a reply) and casts (that do not wait for a reply). These may modify an internal state.</td>
<td>gen_server</td>
<td>init/1, handle_call/3, handle_cast/2, handle_info/2, terminate/2, code_change/2</td>
</tr>
<tr>
<td><strong>Finite state machine</strong></td>
<td>A finite number of states exist along with the messages each state accepts, the replies that are sent and the state change that may follow. Each state has a function named after it that should be exported.</td>
<td>gen_fsm</td>
<td>init/1, handle_event/3, handle_sync_event/4, handle_info/3, terminate/3, code_change/4</td>
</tr>
<tr>
<td><strong>Generic event handler</strong></td>
<td>Event handlers register in a central event manager and are notified for any event that arrives.</td>
<td>gen_event</td>
<td>init/1, handle_event/2, handle_call/2, handle_info/2, terminate/2, code_change/3</td>
</tr>
<tr>
<td><strong>Application</strong></td>
<td>A collection of modules that implement some specific functionality and can be started and stopped as a whole.</td>
<td>application</td>
<td>start/2, stop/1</td>
</tr>
<tr>
<td><strong>Supervisor</strong></td>
<td>A process which supervises other processes (called child processes). A child process can either be another supervisor or a worker process.</td>
<td>supervisor</td>
<td>init/1</td>
</tr>
</tbody>
</table>

Table 1. Common Erlang/OTP behaviours
-module(gen_server).
-export([behaviour_info/1]).

behaviour_info(callbacks) ->
  [{init, 1}, {handle_call, 3}, {handle_cast, 2},
   {handle_info, 2}, {terminate, 2}, {code_change, 3}].

Listing 1. The behaviour_info/1 function in gen_server

Modules that use some behaviour should declare so by a behaviour attribute with the name of the behaviour. This attribute allows the erl_lint module to use the relevant list from the behaviour module to look up the required callbacks and emit warnings during compilation if any of them are missing. We should here note that in order to actually call the behaviour_info/1 function, it is required that the relevant behaviour be already compiled and accessible by the Erlang code loader.

Often, though not always, the behaviour module may contain some additional information, such as comments describing the required callbacks. So is the case with the excerpt shown in Listing 2 taken from the generic server behaviour again. The programmer has taken care to supply the relevant information for the init/1 and handle_call/3 callbacks for the generic server behaviour.

The problem with comments is that they are in free text form and cannot therefore be mechanically processed. Insofar as there is no systematic way of forcing comments to provide a rigid specification that remains in sync with the code they pertain to, discrepancies are hard to avoid. This is sometimes true even for long-standing pieces of software, such as the code of the behaviour in Listing 2 if one meticulously reads the documentation, its incompleteness soon becomes obvious. For once, the declaration of tuples seems haphazard: some elements are fully defined as specific atoms (ok, ignore, stop, reply, no_reply) and some variables also have proper, full definitions (Reason). From the remaining variables, some may indeed hold any term (Args, State, Msg, Reply, Tag, Term), whereas others appear unconstrained but are in fact more limited: Timeout should be a non-negative integer or the atom infinity, while From should be a process identifier. Neither constraint is mentioned in the comments, despite the diligent-looking effort. We would be safe to assume that the specification’s holes would not be obvious to a user skimming through the documentation in question.

What is more, at the time of writing of this paper, the published documentation of gen_server (for Erlang/OTP R13B04) has differences in both callbacks:

- init/1 may also return {ok, State, hibernate} and
- handle_call/3 may also return:
  {reply, Reply, NewState, hibernate}
  {noreply, NewState, hibernate}

We can see that the aforementioned comments have become outdated; this is expected. What is slightly surprising is that these comments have not been updated for some years now, circa R12B-1, when the ability to hibernate generic servers was added.

-----

%%% The user module should export:
%%% init(Args)
%%% => {ok, State}
%%%   (ok, State, Timeout)
%%% ignore
%%%   {stop, Reason}
%%% handle_call(Msg, (From, Tag), State)
%%% => {reply, Reply, State}
%%%   (reply, Reply, State, Timeout)
%%%   (noreply, State)
%%%   {noreply, State, Timeout}
%%%   {stop, Reason, Reply, State}
%%% Reason = normal | shutdown
%%% | Term terminate(State) is called
%%% ..... MORE COMMENTS FOR THE OTHER ..... 
%%% ..... FOUR CALLBACKS HERE ..... 

Listing 2. Documentation comments in gen_server.erl

3. Types and function specifications

The need to provide specifications for functions is well studied, especially in the realm of functional languages. The most widely-adopted method to specify the interface of a function is through a type signature. The programmer can either provide the signature themselves or have it automatically generated through a type inference algorithm. Most functional languages with static typing opt for the latter choice but some allow both methods to coexist, as is the case in Haskell: If the programmer chooses to provide a type signature for documentation purposes, the compiler will check that it be congruent with the one automatically inferred.

As Erlang is a dynamically typed language, programs written in it originally did not contain any explicit type information or function signatures. As of the current time, the situation has changed. Nowadays Erlang comes with a language for declaring types and function specifications and this language is used more and more in Erlang code, both in Erlang/OTP libraries and in applications. A static analysis tool, Dialyzer (Discrepancy AnalyZer for ERlang applications), can take this information into account and detect type discrepancies and other kinds of programming errors, often subtle ones, early in the development cycle.

Dialyzer has a unique type inference algorithm which is sound for defect detection instead for type safety (runtime checks in Erlang guarantee the latter). The algorithm assigns each function a success typing whose semantics are explained in the following statement:

“A success typing of a function \( f \) is a type signature, \( \tilde{a} \rightarrow b \), such that whenever an application \( f(\tilde{p}) \) reduces to a value \( v \), then \( v \in b \) and \( \tilde{p} \in \tilde{a} \).”

In this way, if Dialyzer finds that a function \( f \) has an application with a value that does not belong in \( \tilde{a} \) it can emit a warning. Dialyzer works by initially assuming that the arguments and the result may be any valid Erlang term and gradually constraining them using available information to infer a tighter success typing. Such information can be found in the patterns against which arguments are
behaviour_info(callbacks) ->
  [{init, 1, "-spec init(Args :: term()) ->
    {ok, State :: term()} |
    {ok, State :: term(), timeout() | hibernate} |
    ignore |
    {stop, Reason :: term()."},
  {handle_call, 3, "-spec handle_call(Request :: term(), From :: {pid(), Tag :: term()}, State :: term()) ->
    {reply, Reply :: term(), State :: term()} |
    {reply, Reply :: term(), State :: term(), timeout() | hibernate} |
    {noreply, State :: term()} |
    {noreply, State :: term(), timeout() | hibernate} |
    {stop, Reason :: normal | shutdown | term()}, Reply :: term(), State :: term()} |
    {stop, Reason :: term(), State :: term()."},
  ...].

Listing 3. An abandoned attempt: adding specs as strings to the information returned by behaviour_info/1

matched or in the type signatures of the core functions invoked.

Even though the success typing inference algorithm is by itself quite effective in detecting discrepancies, Dialyzer goes a step further and then takes into account type information provided by the programmers in the form of function specifications (or specs). However, it does so only when these specs are not in any conflict with the inferred success typing.

4. Declaring behaviour callbacks

Our goal is to have a better way of describing the interface between the behaviour and the callback modules. An obvious approach would be to pinpoint the types of the arguments and return values that are expected from each of the callback functions. This description can then be used to automatically generate documentation or detect possible discrepancies in the implementation of the callbacks by a module that uses the behaviour.

Note that this cannot simply be done by adding spec attributes in the module defining the behaviour. The reason for this is that the callbacks are expected to be defined in the callback module and the current compiler does not accept function specifications for functions that do not have a corresponding definition in the module. We could of course relax this constraint, but this would not be a good idea, even if restricted to behaviour modules, as it would mean that the compiler would accept specifications for functions that may not be defined anywhere. Such specifications are already dubious and have the added risk of being later “captured” by a newly introduced function that is not related to them.

4.1 An abandoned attempt: adding specs as strings

Our initial approach was to add a spec as a third element in each type-specifying tuple of the list that the behaviour_info/1 function returns for a behaviour module. Effectively, this change was as shown in Listing 3 where the spec is included inside a string. Apart from being simple, this strategy easily allows for backwards-compatibility: tools that do not need spec info can ignore it, while other tools that do need this information (such as Dialyzer) can simulate the compiler by also calling the behaviour_info/1 function, parsing the spec and using it for checks against the implementation of the callback module.

In hindsight, this approach is both ugly and inappropriate for numerous reasons:

- It introduces significant redundancy: the name of each callback exists both as the first element of the tuple and in the spec; a similar issue exists with the number of arguments.
- It requires special treatment from Dialyzer: normal specs are module attributes while these specs come as a result of a function.
- The validity (even syntactic) of these specs cannot easily be checked at compile time; doing so requires access to the set of types defined in the module.

The only benefit of this approach is that only minimal modifications to the compiler (and Erlang/OTP in general) were required. We decided to do better than that.

4.2 A proposal based on a callback attribute

Refining our initial approach, we next decided to implement the feature with the addition of a new special attribute named callback to the modules defining a behaviour. An example of the use of the new attribute is given in Listing 4. The syntax is kept identical to that of specs. The compiler should parse the new attribute and convert it to a form similar to the one used internally for the normal specs; this requires only slight modifications. The linter should also be extended to perform checks similar to those performed for specs, namely:

- only one callback attribute for each callback exists
- the types mentioned in the callback attributes have been defined

In this approach, the only difference between spec and callback attributes is that callbacks are defined for functions that do not exist in the module, while specs for non-existing functions produce compile-time warnings.
Note that the presence of the \texttt{callback} attributes has an extra bonus: there is actually no need anymore for an explicit definition of the \texttt{behaviour_info/1} function, as it can be automatically generated by collecting the names and arities of the \texttt{callback}s and generating a tuple list with them. Such an automatic function generation is already done by the compiler to add the \texttt{module_info/0,1} functions in every module, therefore it is trivial to add another function in the exported set. This ensures that the compiler can still use the \texttt{erl_lint} checks we presented in Section \ref{erl_lint} to detect missing \texttt{callback} definitions. Actually, an error is currently reported if a user has defined both \texttt{callback} attributes and an explicit \texttt{behaviour_info/1} function.

Having such \texttt{callback} information available, let us now see how we can take advantage of it.

5. Using \texttt{callback} information

5.1 Using \texttt{Dialyzer} to find errors

We first show how \texttt{Dialyzer} can be used to detect discrepancies in the implementation of a behaviour’s \texttt{callback}s.

As described in Section \ref{erl_lint} when provided with \texttt{specs}, \texttt{Dialyzer} can aid the developer’s job in two ways: it can trust the developer in that a certain function will not be used with broader types than specified in its arguments and further limit the type of both the function’s result and the argument variables that are passed to the function wherever it is applied. However, before doing so, \texttt{Dialyzer} checks that the types described in the \texttt{spec} have no contradictions with the ones inferred by the analysis of the actual code.

These checks can also be applied to \texttt{callback} attributes. Whenever a module declares the use of a particular behaviour with a \texttt{callback} attribute, \texttt{Dialyzer} can find the relevant \texttt{callback} attributes and, after consulting them, it can check whether:

\begin{itemize}
\item all \texttt{callback}s have been defined (the current compiler does that already);
\item the \texttt{callback} implementations accept the specified argument types;
\item the value returned by the \texttt{callback} is in accordance with the \texttt{callback} specification.
\end{itemize}

If any of these conditions does not hold, a \texttt{Dialyzer} warning is emitted. Notice that the implementation of a \texttt{callback} may accept any other arguments apart from the ones specified but should not return a result that does not belong in the range specified in its \texttt{callback} attribute.

5.2 An issue

As we will soon see, \texttt{callback} information is indeed useful for detecting behaviour misuses, but before that we will describe an issue we have run into. The problem is that \texttt{Dialyzer}'s inferred success typings have a tendency to over-approximate not only the domain of a function’s arguments but also—and much more importantly—the range of a particular call. This is completely acceptable for sound defect detection; it just means that some possible discrepancies are not reported by the tool. One of the reasons for this over-approximation is that \texttt{Dialyzer} does not analyze each clause of a function separately. When \texttt{Dialyzer} infers that the return type \texttt{b} of a certain function is a union of many different elements, every value in \texttt{b} will subsequently be assumed as a possible return value of a call to this function, regardless of whether certain values may be unobtainable with the provided combination of arguments.

The following function clearly illustrates that kind of problem:

\begin{verbatim}
answers(X) when is_integer(X) -> X * 42;
answers(X) -> X * 42.0.
\end{verbatim}

It is clear that this function can only be used with numbers, both integers and floats. For this function, the version of \texttt{Dialyzer} shipped with current \texttt{Erlang/OTP} infers the following success typing:

\begin{verbatim}
answers(number()) -> number()
\end{verbatim}

where \texttt{number()} :: \texttt{integer()} | \texttt{float()}, thereby foregoing the fact that any call with an integer will never re-
function with callback attributes and implementing the behaviour discrepancy detection in Dialyzer we got the experimental results presented in Table 2. Note that this table shows only verified errors. Some more discrepancies identified by Dialyzer need to be cross-checked by the Erlang/OTP team as the documentation on which the callback attributes were based might be outdated. We believe that these results prove the points made in the introduction, concerning the importance of having the behaviour interfaces formally described and the increased code robustness that can be hence achieved.

All the discrepancies detected correspond to cases where a callback has a wider return type than the one described in the relevant attribute. The most common warning was about the return value of generic server's handle_cast/2 and handle_info/2 callbacks which sometimes erroneously included {reply, ...}. An example of such an abuse in an Erlang/OTP application is given in Listing 7. Even with the use of a template that has a comment right on top of this code, errors can occur! This particular case is actually hopeless: not only is the code wrong but the comment is outdated in a manner similar to the handle_call/3 callback (described in Section 2.1).

We show another example in Listings 8, 9 and 10, this time using the generic event behaviour (gen_event). As can be seen in Listing 8 the callback function init/1 is expected to return only tuples tagged with ok. The problem here is that if the file:open/2 call in line 53 returns a value signaling an error (a tuple tagged with error), the same value will be returned by the callback, violating its specification. The code thus should not return this error but throw it instead.

6. Further work

The elimination of the false warnings mentioned in Section 5.2 has been the inspiration behind a major extension for Dialyzer’s type inference, which aims to accurately track dependencies between argument types and their results. This is a significant breakthrough as it empowers Dialyzer to employ a form of polyvariant control flow analysis and infer success typings similar to intersection types. Without this extension, warnings about behaviour (mis)-usage have to be very conservative.

The description of interface between the behaviour and the callback modules can also be further enhanced by providing associations between the API calls of a behaviour and its callback functions. Such information can aid tools that are confused by the non-deterministic nature of behaviours and cannot track calls that use the API back to the relevant callbacks. In Table 3 we show the im-
### Table 2. Behaviour discrepancies in Erlang/OTP applications

<table>
<thead>
<tr>
<th>Application</th>
<th>Description</th>
<th>Behaviour Used</th>
<th>Discrepancies</th>
</tr>
</thead>
<tbody>
<tr>
<td>inets</td>
<td>Internet clients and servers</td>
<td>gen_server</td>
<td>1</td>
</tr>
<tr>
<td>tftp</td>
<td></td>
<td>gen_server</td>
<td>1</td>
</tr>
<tr>
<td>dist_ac</td>
<td>distributed application controller</td>
<td>gen_server</td>
<td>1</td>
</tr>
<tr>
<td>mnesia</td>
<td>distributed DBMS</td>
<td>gen_server</td>
<td>2</td>
</tr>
<tr>
<td>ssh</td>
<td>SSH application</td>
<td>gen_server</td>
<td>2</td>
</tr>
<tr>
<td>error_logger</td>
<td>Stdlib’s error logger</td>
<td>gen_server</td>
<td>1</td>
</tr>
<tr>
<td>Total discrepancies</td>
<td></td>
<td></td>
<td>8</td>
</tr>
</tbody>
</table>

### Table 3. gen_server's API-callback interaction

<table>
<thead>
<tr>
<th>Call to API’s</th>
<th>... implies a call to callback</th>
</tr>
</thead>
<tbody>
<tr>
<td>start_link/(3,4), start/(3,4)</td>
<td>init/1</td>
</tr>
<tr>
<td>call/(2,3), multi_call/(2,3,4)</td>
<td>handle_call/3</td>
</tr>
<tr>
<td>cast/2, abcast/(2,3)</td>
<td>handle_cast/2</td>
</tr>
</tbody>
</table>

### Acknowledgments

We thank Nikolas Korasidis for comments and suggestions that have improved the presentation of our work.

### References


*https://github.com/erlang/otp/commits/pu*