Runtime verification in Erlang by using contracts*

Lars-Åke Fredlund[1], Julio Mariño[1], Sergio Pérez[2], and Salvador Tamarit[2]

[1] Babel Group
Universidad Politécnica de Madrid, Spain
{lfredlund,jmarino}@fi.upm.es
[2] Departament de Sistemes Informàtics i Computació
Universitat Politècnica de València, Spain
{serperu, stamarit}@dsic.upv.es

Abstract. During its lifetime, a program regularly undergoes changes that seek to improve its functionality or efficiency. However, such modifications may also introduce new errors. In this work we use the *design-by-contract* approach to allow programmers to formally state, in the code, some of the knowledge and assumptions originally made when the code was first written. Such contracts can then be checked at runtime, to ensure that modifications made to a program did not violate those assumptions. Applying these principles we have designed a runtime verification system for the Erlang language. The system comprises two kinds of annotations. The first one contains those needed to specify contracts in both sequential and concurrent code. The second kind of annotations is specifically intended for concurrent Erlang code. Details about the design and implementation of these contracts, as well as examples of its use are provided. The ideas presented in this paper have been implemented in a tool named `EDBC`. Its source code is available at `github.com` as an open-source and free project.

Keywords: Runtime Verification, Design-By-Contract, Program instrumentation, Concurrency.

1 Introduction

Developing software is not an easy task and, consequently, errors (*bugs*) are present in most software artefacts. Companies usually rely on program testing

* This work has been partially supported by MINECO/AEI/FEDER (EU) under grant TIN2016-76843-C4-1-R, by the *Comunidad de Madrid* under grant S2013/ICE-2731 (*N-Greens Software*), and by the *Generalitat Valenciana* under grant PROMETEO-II/2015/013 (*SmartLogic*). Salvador Tamarit was partially supported by the *Conselleria de Educación, Investigación, Cultura y Deporte de la Generalitat Valenciana* under grant APOSTD/2016/036.

to check that programs behave as expected. There are also programming languages, e.g., Haskell and Rust, that provide robust static type systems which can help discover errors during program compilation. Once a bug is found, the debugging process can start in order to find the *cause* or source of the bug. Even in a program language which lacks a static type system, programmers can make the process of locating the cause of a bug easier by annotating code with implicit assumptions made – e.g., concerning type of parameters to methods or functions. For instance, consider a function that will divide a number by zero if one of its parameters has a non-expected value. Normally such an error is manifest when the division by zero occurs. However, the true source of the bug is the function call. Defensive programming is a way to eliminate unexpected behaviours, e.g., division by zero, by checking the validity of arguments before operations are attempted. However, in mainstream programming this style is not a recommendable practice for various reasons, such as the need to add *boiler-plate* code which obscures the program logic, and because of the execution time overhead caused by such checks. In fact, the language considered in this article, Erlang, is infamous for its stance that defensive programming is to be avoided – "let it crash".

The Erlang programming has a number of interesting and innovative mechanisms for error detection and recovery,[3] but these features are supposed to be used only for errors that are hard to avoid (i.e., because static type systems are not strong enough to characterise full program behaviour). Unfortunately, not all the errors fall in this category. Some errors are rather easy to detect, and should ideally be detected at "compile time" instead of being detected and corrected when a software is operational. Erlang, as Python or JavaScript, is a dynamic programming language. This means that the program compiler, if it even exists, does relatively few checks during compilation to help prevent errors when the program is later run. For this reason, static analysis techniques, popularised primarily by the Dialyzer tool[12], have been successfully and widely adopted by Erlang practitioners. Dialyzer can analyse the code and report some errors without requiring annotating program code in any way. However, the capability of Dialyzer to detect program bugs can be considerably improved by the use of type contracts [10]. Note that such contracts are not used only by Dialyzer to implement static type checking, but also serve to document the developed code, which improves the maintainability of the resulting software.

Even when the Dialyzer tool is used, and when programmers provide e.g. EUnit test cases (an Erlang testing tool) to further check program behaviour, program bugs can still remain. In this work, we propose an mechanism to further structure and strengthen such "defensive" programming tasks, i.e., the Erlang Design-By-Contract (`EDBC`) system, a runtime verification framework based on the Design-By-Contract [14] philosophy. The `EDBC` system is available as free and open-source software at `https://github.com/tamarit/edbc`.

[3] For example, process links help structure fault detection and fault recovery in complex applications.

In typical design-by-contract frameworks there are different types of contracts, with most of them being related to program functions or methods. The most common contracts are *pre-* and *postconditions*. Preconditions are conditions that should hold before evaluating a function or method, while postconditions should hold after its evaluation. In addition to these, the `EDBC` system includes type contracts, decreasing-argument contracts to help analyse program termination, execution-time contracts to document bounds for the execution time of function, and purity contracts which prohibit side effects such as Erlang process-to-process communication. All these contracts can be used in any Erlang program, regardless whether the program is purely functional, or structured as a concurrent system, composed of a number of concurrent processes. To avoid the traditional execution overhead associated with the use of contracts, normally they are checked only during software production and maintenance. The `EDBC` library provides a mechanism to disable such checks for running product code.

The article is structured as follows: Section 2 describes the contracts supported by our system for annotating Erlang code, regardless whether the code is purely functional or implements concurrent behaviours. This section contains a number of examples (2.1), and also gives details on the implementation of contracts by means of program instrumentation in Section 2.2. There is a second type of contracts which is used in concurrent Erlang code, specifically, in programs using so called Erlang "behaviours". Annotations for such concurrent behaviours are described in Section 3.2, with examples in Sections 3.3 and 3.4. Finally, Section 4 presents an overview of related work, and Section 5 concludes and provides directions for future research and development.

2 Contracts in Erlang

In this section we first introduce the contracts provided by the `EDBC` system, and show how they can be used to check program behaviour at runtime in Section 2.1. Then, we present some implementation details in Section 2.2.

2.1 The contracts

Precondition contracts. With the macro[4] `?PRE/1` we can define a precondition that a function should hold. The macro should be placed before the first clause of the annotated function. The single argument of this macro is a function without parameters, e.g. `fun pre/0` or an anonymous function `fun() -> ... end`, that we call *precondition function*. A precondition function is a plain Erlang function. Its only particularity is that it includes references to the function parameters. In Erlang, a function can have more than one clause, so referring the parameter using the user-defined names can be confusing for both `EDBC` and for the user. In order to avoid these problems, in `EDBC` we refer to the parameters by their

[4] As an implementation decision, we have chosen to use Erlang macros to represent all contracts. The reason is that similar tools like EUnit also uses macros for assert definitions.

position. Additionally, the parameters are rarely single variables but can be more complex terms like a list or a tuple (since Erlang permits pattern matching). For these reasons we use the `EDBC`'s macro `?P/1` to refer to the parameters. The argument of this macro should be a number that ranges from 1 to the arity of the annotated function. For instance, `?P(2)` refers to the second parameter of the function. A precondition function should return a boolean value which indicates whether the precondition of the annotated function holds or not. The precondition is checked during runtime before actually performing the call. If the precondition function evaluates to `false`, the call is not preformed and a runtime exception is raised.

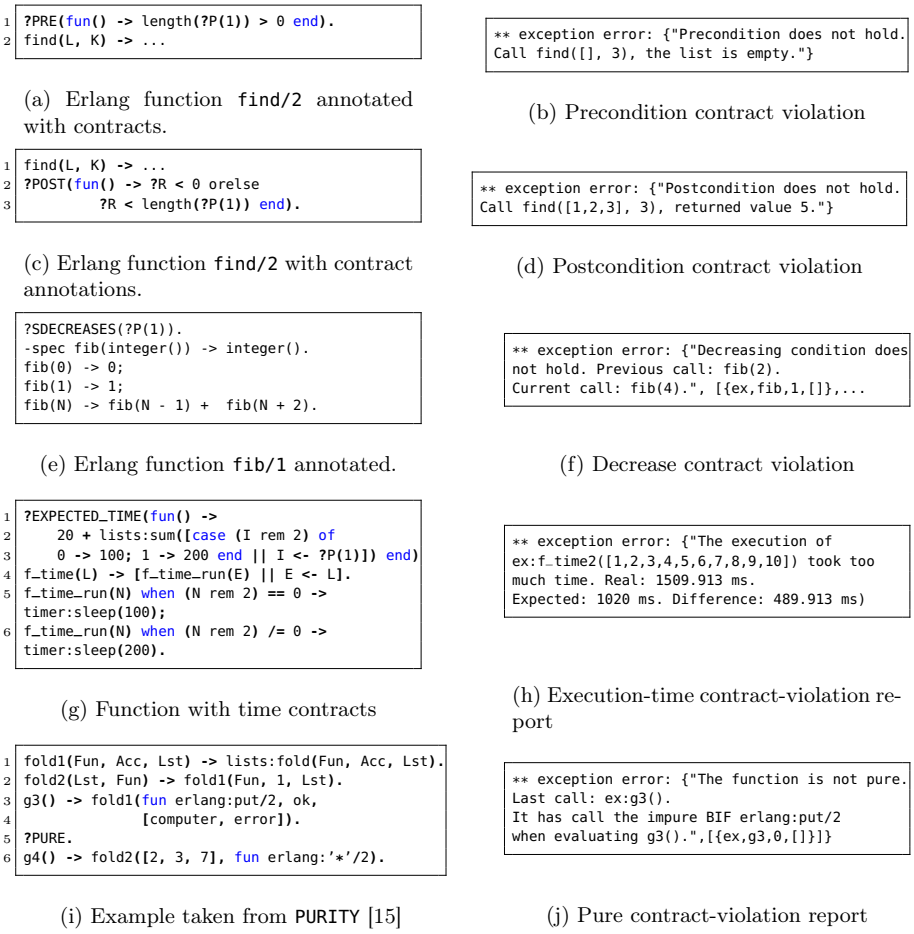
As an example, imagine a function `find(L,K)` which searches for the position of a value `K` in a list `L`, and returns `-1` if the value is not found. Figure 1a shows the usage of a precondition contract which expresses that the first parameter list should not be empty.

In case the precondition is violated, an Erlang exception is raised. For instance, if we attempt the function call `find([], 3)`, which fails the precondition check because the length of the list argument is 0, the resulting error message is shown in Figure 1b.

Postcondition contracts. Similar to preconditions, the macro `?POST/1` is used to define a postcondition that a function should satisfy. The macro should be placed after the last clause of the annotated function. Its argument is a function without parameters, which we call the *postcondition function*. For checking postconditions referring to the result of the function is essential; the macro `?R` permits this. Additionally, as in the `?PRE/1` precondition function the `?P/1` macros can be used to refer to the actual parameters of the annotated function. The result of a postcondition function is also a boolean value. Postcondition functions are checked after a call terminates, and a runtime error is raised if the postcondition function evaluate to `false`. Figure 1c shows an example of a postcondition contract associated with the function `find/2`. In this contract, an error is raised if the index returned by `find/2` is greater that the length of the list. Suppose an implementation of `find/2` returns the value `5` to the call `find([1,2,3],3)`. In this case, the execution would raise the error shown in Figure 1d.

Decreasing-argument contracts. These contracts are meant to be used in recursive functions, and check that (some) arguments are always decreasing in nested calls[5]. There are two types of macros to define these contracts: `?DECREASE/1` and `?SDECREASE/1`. They both operate exactly in the same way with the exception that the `?SDECREASE/1` macro indicates that the argument should be strictly smaller in each nested call, while the `?DECREASE/1` macro also permits the argument to be equal. The argument of both macros can be either a single `?P/1` macro or a list containing several `?P/1` macros. These contracts should be placed before the first clause of the function. Decreasing-argument contracts are checked each time a recursive function call is made, by comparing the arguments of the current call with the nested call just before performing the actual

[5] Note that decreasing contracts only guarantee termination if the sequence is strictly decreasing and well founded, i.e. values cannot go below a certain limit.



(a) Erlang function `find/2` annotated with contracts.

(b) Precondition contract violation

(c) Erlang function `find/2` with contract annotations.

(d) Postcondition contract violation

(e) Erlang function `fib/1` annotated.

(f) Decrease contract violation

(g) Function with time contracts

(h) Execution-time contract-violation report

(i) Example taken from `PURITY` [15]

(j) Pure contract-violation report

Fig. 1: Several examples of contract annotations.

nested recursive call. In case the argument expected to decrease is not actually decreasing, a runtime error is raised and the call is not performed. To exemplify the functionality of this contract, we use a wrong implementation of the Erlang program calculating the Fibonacci numbers shown (Figure 1e). When executing the function call `fib(2)`, the error message in Figure 1f is shown.

Execution-time contracts. `EDBC` introduces two macros that allow users to define contracts concerning execution times: `?EXPECTED_TIME/1` and `?TIMEOUT/1`. The macros should be placed before the first clause of the annotated function. The argument of these macros is a function without parameters called the *execution-time function*. An execution-time function should evaluate to an integer which defines the expected execution time in milliseconds. Within the body of an execution-time function we can use `?P/1` macros to refer to the arguments.

Permitting the execution-time function to refer to arguments is particularly useful when dealing with lists or similar structures where the expected execution time of the function is related to the sizes of arguments. Both macros have a similar semantics, the only difference is that with macro `?EXPECTED_TIME/1` the `EDBC` system waits till the evaluation of the call finishes to check whether the call takes the expected time, while with macro `?TIMEOUT/1` `EDBC` raises an exception if the function call does not terminate before the timeout limit is reached. As an example of time contracts, we consider a function which performs a list of tasks. Each task has its type (even or odd), and the allowed execution time is defined by this type (100 and 200 ms., respectively). Figure 1g shows the function and its associated time contract. Supposing we change the execution-time function to, for instance, `fun() -> 20 + (length(?P(1)) * 100) end`, we would obtain the contract-violation report shown in Figure 1h.

Purity contracts. When we say that a function is pure we mean that its execution does not cause any side effects, i.e., it does not perform I/O operations, nor does it send messages, etc. That a function is "pure" can be declared by using the macro `?PURE/0` before its first clause. The purity checking process is performed in two steps. First, before a call to an function declared to be pure is performed, a tracing process is started. Then, once the evaluation of the annotated function call finishes, the trace is inspected. If a call to an impure function or operation has been made, a runtime exception is raised. Note that due to the use of tracing we can provide exact purity checks, ensuring that there are neither false positives nor false negative reports.

Note that purity checking is not compatible with execution-time contracts, since checking execution times do require performing impure actions. In order to illustrate the checking of purity contracts we take a simple example used to present `PURITY` [15], i.e., an analysis that statically decides whether a function is pure or not. The example the authors presented is depicted in Figure 1i. We only added the contract `?PURE` in the test case `g4/0`, because the other test case, i.e. `g3/0`, performs the impure operation `erlang:put/2`. When `g4/0` is run, no contract violation is reported as expected. If we added the contract `?PURE` to `g3/0`, then the execution will fail showing the error in Figure 1j.

Invariant contracts. This contract is meant to be used in Erlang behaviours which has an internal state. An invariant contract is defined by using the macro `?INVARIANT/1`. This macro can be placed anywhere inside the module implementing the behaviour. The argument of the `?INVARIANT/1` macro is a function, named *invariant function*, with only one parameter that represents the current state of the behaviour. Then, an invariant function should evaluate to a boolean value which indicates whether the given state satisfies the invariant or not. The invariant function is used to check the invariant contract each time a call to a function which is permitted to change the state finishes, e.g., when a call by the `gen_server` behaviour to the `handle_call/3` callback function finishes (see Section 3.2 for a description of the behaviour). Note that invariant contracts can be used to check for the absence of problems in concurrent systems such as e.g. starvation. Examples of invariant contracts are presented in Section 3.2.

```
** exception error: {"The spec precondition does not hold.
Last call: ex:fib(a).
The value a is not of type integer().", ...}
```

Fig. 2: `spec` contract-violation report

```
method Find(a: array<int>, key: int)
   returns (index: int)
   requires a != null
   ensures 0 <= index ==> index < a.Length &&
           a[index] == key
   ensures index < 0 ==> forall k ::
      0 <= k < a.Length ==> a[k] != key
{...}
```

(a) Function `Find/2` annotated in `Dafny`

```
1  ?PRE(fun() -> length(?P(1)) > 0 end).
2  ?SDECREASES(?P(1))
3  find(L, K) -> ...
4  ?POST(fun() -> ?R < 0 orelse
5          (?R < length(?P(1))
6           andalso lists:nth(?R, ?P(1)) == ?P(2))
7        end).
8  ?POST(fun() -> ?R > 0 orelse
9          lists:all(fun(K) -> K /= ?P(2) end,
10         ?P(1))
11       end).
```

(b) Function `find/2` annotated in Erlang/EDBC.

Fig. 3: Contracts for function `find/2` in `Dafny` and `Erlang`

Type contracts. Erlang has a dynamic type system, i.e., types are not checked during compilation but rather at runtime. However, the language still permits to specify type contracts (represented by `spec` attributes) which serves both as code documentation, and as aid to static analysers like `Dialyzer` [12]. However, such type contracts are not checked at runtime by the Erlang interpreter, because of the potential associated cost in execution time. However, for programs still in production, checking such type contracts during runtime can be helpful to detect unexpected behaviour. For this reason, before a function is evaluated, `EDBC` checks the type contract of its parameters (if any), while its result is checked after its evaluation. If a type error is detected, a runtime exception error is raised. Note that `EDBC` does not use any special macro to check type contracts, the standard `spec` attributes are used instead. Figure 2 shows an error that would be shown in case of calling `fib(a)` for the program defined in Figure 1e.

Note that the `EDBC` system can be used to define quite advanced contracts. As a comparison point, the `Dafny` tool [11], which was an inspiration for `EDBC`, permits the use of quantifiers to define conditions for input lists. Figure 3a shows as an example of how quantifiers are used in `Dafny` to characterise the function `Find/2`.

Such contracts with quantifiers can be represented in `EDBC` too. Instead of using a special syntax like in `Dafny`, we can check conditions with quantifiers using a common Erlang function such as `lists:all/2`, which checks whether a given predicate is true for all the elements of a given list. Figure 3b shows how the contracts in Figure 3a are represented in `EDBC`. If we implemented this function as a recursive one, the list would be decreasing between calls. Then, we could also add the contract `?SDECREASE(?P(1))` to the function.

Contracts added by users can also be used to generate documentation. Erlang OTP includes the tool `EDoc` [6] which generates documentation for modules in `HTML` format.

We have modified the generation of `HTML` documents to also include information concerning `EDBC` contracts. As an example the `EDoc`-generated documentation for the function `find/2`, with information of its contracts (some in Figure 3b and some new), and its type specification, is depicted in Figure 4. Finally, it is important to note that the contract checking performed by `EDBC` does not cause incompatibilities with other Erlang analysis tools. For instance, users can both define `EDBC` contracts, and include EUnit [3] test case assertions, in the same function.

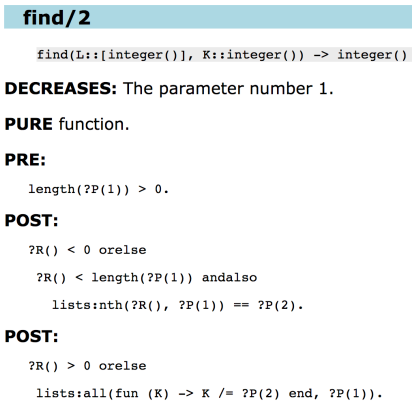


Fig. 4: `EDoc` for the annotated function `find/2`.

2.2 Implementation details

In this section we explain how the code is instrumented to support contract checking at runtime. Note that the code produced by the instrumentation process is normal Erlang code, which is executed by the standard Erlang runtime system in a completely normal fashion. Technically the instrumentation is performed using so called Erlang "parse transforms", which permits defining syntactic transformations on Erlang code.

Consider a module with a number of annotated functions. The instrumentation process replaces such annotated functions with a copy of the (possibly modified) original function, together with a number of helper functions which are synthesised from the contracts. The instrumentation is performed in three steps:

1. First, if a function has an associated contract, then an instrumentation to store the relevant information regarding function calls (function name, arguments and stack trace) is performed. This creates a new function which becomes the function entry point, and the original function is renamed. When the new function is called, it stores the call information, and proceeds to call the original function.
2. Then, contracts of type `?DECREASES/1` (including `?SDECREASES/1`) are processed. This instrumentation creates a function which checks if the size of its parameters have decreased between recursive calls. If they are decreased,

delayed calls are executed, and if they are not, a contract violation exception is raised. During the instrumentation the original function is also modified by replacing all the recursive calls to calls to the new created function. Note that, due to this instrumentation and the previous one, we have changed the call cycle of a recursive call from $\texttt{f}_{\texttt{ori}} \rightarrow \texttt{f}_{\texttt{ori}}$ to $\texttt{f}_{\texttt{si}} \rightarrow \texttt{f}_{\texttt{ori}} \rightarrow \texttt{f}_{\texttt{dc}} \rightarrow \texttt{f}_{\texttt{si}}$, where $\texttt{f}_{\texttt{ori}}$ is the original function, $\texttt{f}_{\texttt{si}}$ is the function that stores the call information, and $\texttt{f}_{\texttt{dc}}$ is the function that checks the decreasing of arguments.
3. Finally, the remaining contract types are processed distinguishing between contracts of type `?PRE`, and contracts of type `?POST`. All contracts except `?DECREASE` can in fact be generalised to one of of these two types of contract. Of course, each contract has its particularities, however, these particularities do not have any effect in the instrumentation process. The chain of calls becomes $\texttt{f}_{\texttt{si}} \rightarrow \texttt{f}_{\texttt{pre/post}}* \rightarrow \texttt{f}_{\texttt{ori}}$, where $\texttt{f}_{\texttt{pre/post}}*$ are a number of functions (maybe none) introduced by `?PRE`/`?POST` contracts. In the case of a recursive function which defines a `?DECREASE` contract, the call chain would be $\texttt{f}_{\texttt{si}} \rightarrow \texttt{f}_{\texttt{pre/post}}* \rightarrow \texttt{f}_{\texttt{ori}} \rightarrow \texttt{f}_{\texttt{dc}} \rightarrow \texttt{f}_{\texttt{si}}$. Further note that most of the helper functions have a call as its last expression enabling, in this way, so called last call optimisations to reduce the runtime cost of instrumenting code. The only exception is the functions generated for postconditions, which needs to be be stacked until internal calls are completely evaluated.

Finally, contract checking can be easily disabled or enabled using a special compilation flag, thus e.g. permitting production code to be compiled and run without instrumentation.

3 Concurrency

As we shall see, the contracts presented in the earlier Section 2 are highly useful for concurrent code too. The contracts for "normal" Erlang code in a sense help provide additional structure to Erlang functions; they serve to document assumptions how Erlang functions may be called, and, if the caller respects its contract, what can be expected of the behaviour of the function (i.e., concerning the computed result, side effects, termination properties, and so on). For concurrent Erlang code the situation is different: for the most part concurrent Erlang code is already highly structured. Most Erlang programmers do not write concurrent code from scratch, but rather rely heavily on proven concurrent Erlang behaviours (which can be considered a form of *design patterns*) present in the Erlang/OTP standard library.

3.1 Erlang Behaviours as Concurrent Contracts

The Erlang behaviours are formalisations of common programming patterns. The idea is to divide the code for a process in a generic part (a behaviour Erlang module), which is never changed, and a specific part (a callback Erlang module), which is used to tailor the process for the particular application being

implemented. Thus, the behaviour module forms part of the Erlang/OTP standard library, and the callback module is implemented by the programmer. Such behaviours provide standard mechanisms to implement concurrent behaviours: the generic part provides a proven implementation of an often complex concurrent coordination task, which permits programmers to focus on the easier task on how to adapt this generic behaviour to the particular application at hand.

Thus, the definition, and use of such behaviours, can be seen as a form of parametric software contract. The generic part of the guarantees the general behaviour of the behaviour, which to be able to function correctly, requires the specific part to function correctly, i.e., that callback functions function correctly (e.g. are computationally efficient, and terminate normally).

In the following section 3.2 we focus on an Erlang behaviour which is heavily used in industrial applications, the `gen_server` behaviour which is used to implement client–server architectures. In practice the behaviour has a number of shortcomings which are addressed in `EDBC` by extending it to handle client requests which arrive when the server is not capable of servicing them, a common situation in asynchronous concurrent systems. A programmer using the normal `gen_server` behaviour must manually handle such asynchronous requests by implementing a queuing policy in the specific behaviour part. In the `EDBC` a modified `gen_server` is provided instead, where the generic behaviour part handles such asynchronous requests according to a simple rule, and which implements a flexible queuing policy, thus providing a concurrent contract which is considerably easier to use.

3.2 The `gen_server` behaviour with contracts

One of the more commonly used Erlang behaviours is the `gen_server`, which provides a standard way to implement a client–server architecture. A callback module for this behaviour needs to define the specific parts of the server (process), e.g., what is the initial state of the server (e.g., implementing the Erlang function `init/0`), and handling specific client requests (e.g., implementing the Erlang functions `handle_call/3` for blocking client calls, and `handle_cast/2` for non-blocking client calls), etc.

Given the highly regular nature of specific parts of this behaviour, the use of normal contracts is highly useful. For instance, invariants (as expressed by the `?INVARIANT/1` contract explained in Section 2) constrain the (persistent) state of the underlying server process. Moreover, for the server to function correctly, the generic parts of the service require the programmer written specific parts of the behaviour to satisfy a number of properties expressible as contracts: calls to `handle_call/3` (or `handle_cast`) must normally be side effect free (as the generic part handles replying to server requests) as expressed by the `?PURE/0` contract, and and the code implementing `handle_call/3` should be efficient as expressed by the `?EXPECTED_TIME/1` contract.

As discussed earlier we have also extended the `gen_server` behaviour to handle asynchronous client requests to the server. This extension provides a mechanism for the server to postpone requests which it is not yet ready to

serve, but which should be served in the future when the server state changes. Concretely we add a new callback function that the behaviour specific part may implement: `cpre/3`. This function should return a boolean value indicating whether the server is ready or not to serve a given request. The rest of the `gen_server` callbacks are not modified. The three parameters of the callback function `cpre/3` are 1) the request, 2) the *from of the request*[6] and 3) the current server state. The function `cpre/3` should evaluate to a tuple with two elements. The first tuples element is a boolean value which indicates if the given request can be served. The second tuples element is the new server state.

The `gen_server_cpre` behaviour behaves in the same way as the `gen_server` behaviour except with a significant difference. Each time the server receives a client request, it calls to `cpre/3` callback before calling the actual `gen_server` callback, i.e., `handle_call/3`. Then, according to the value of the first element of the tuple that `cpre/3` returns, either the request is actually performed (when the value is `true`) or it is queued to be served later (when the value is `false`). In both cases, the server state is updated with the value returned in the second element of the tuple.

`EDBC` includes two implementations of the `gen_server_cpre` behaviour, each one treats the queued requests in a different way. The least complicated implementation resends to itself a client request that cannot be served in the current server state, i.e., a request for which function `cpre/3` returns `{false, ...}`). Since mailboxes in Erlang are ordered according to the arrival time of messages (i.e., in FIFO order), the postponed request will be the last request in the queue of incoming requests. This can be considered unfair, because, when once in the future the state of the server has changed thus potentially permitting the postponed client request to be served, the server could instead serve serve new client requests that have arrived later than the postponed client request.

The `EDBC` framework also provides a more fair version of the `gen_server_cpre` behaviour. In this version, three queues are used to ensure that older requests are served first: $\texttt{queue}_{\texttt{current}}$, $\texttt{queue}_{\texttt{old}}$, and $\texttt{queue}_{\texttt{new}}$. Each time the server is ready to listen for new requests, the $\texttt{queue}_{\texttt{current}}$ is inspected. If it is empty, then the server proceeds as usual, i.e., by receiving a request from its mailbox. Otherwise, if it is not empty, a request from $\texttt{queue}_{\texttt{current}}$ is served. Consequently, the served request is removed from $\texttt{queue}_{\texttt{current}}$. The queues are also modified by adding requests to $\texttt{queue}_{\texttt{old}}$ and $\texttt{queue}_{\texttt{new}}$. This is done when function `cpre/3` returns `{false, ...}`. Depending on the origin of the request it is added to $\texttt{queue}_{\texttt{old}}$ (when it comes from $\texttt{queue}_{\texttt{current}}$) or to $\texttt{queue}_{\texttt{new}}$ (when it comes from the mailbox). Finally, each time a request is completely served, the server state could have been modified. A modification in the server state can enable postponed requests to be served. Therefore, each time the server state is modified, $\texttt{queue}_{\texttt{current}}$ is rebuilt as follows: $\texttt{queue}_{\texttt{old}} + \texttt{queue}_{\texttt{current}} + \texttt{queue}_{\texttt{new}}$.

[6] The *from of the request* has the same form as in the `handle_call/3` callback, i.e., a tuple `{Pid,Tag}`, where `Pid` is the process identifier of the client issuing the request, and `Tag` is an unique tag.

3.3 Selective receives

In public forums such as `https://stackoverflow.com` and the `erlang-questions` mailing list[7], where Erlang programming is discussed, there have been a number of questions regarding the limitations of the standard `gen_server` implementation. Most of them concern how to implement a server which has the ability to delay some requests. For example, one question posted in stackoverflow.com[8] asks whether it is possible to implement a server which performs a selective receive while using a `gen_server` behaviour. None of the provided answers is giving an easy solution. Some of them suggest that the questioner should not use a `gen_server` for this, and directly implement a *low-level* selective receive. Other answers propose to use `gen_server` but delay the requests *manually*. This solution involves storing the request in the server state and returning a `no_reply` in the `handle_call/3`. Then, the request should be revised continually, until it can be served and use `gen_server:reply/2` to inform the client of the result. Our solution is closer to the last one, but all the management of the delayed requests is completely transparent to the user.

```
1 handle_call(test, _From, _State) ->
2   List = [0,1,2,3,4,5,6,7,8,9],
3   lists:map(fun(N) -> spawn(fun() ->
4       gen_server:call(?MODULE, {result, N}) end)
5     end, lists:reverse(List)),
6   {reply, ok, List};
7 handle_call({result, N}, _From, [N|R]) ->
8   io:format("result: " ++ integer_to_list(N) ++ "~n"),
9   {reply, ok, R}.
```

Fig. 5: `handle_call/2` for selective receive

Figure 5 shows the function `handle_call/2` of the `gen_server` that the questioner provided to exemplify the problem. When the request `test` is served, it builds ten processes, each one performing a `{result, N}` request, with `N` ranging from 0 to 9. Additionally, the server state is defined as a list which also ranges from 0 to 9 (Figure 5, lines 2 and 6). The interesting part of the problem is how the `{result, N}` requests need to be served. The idea of the questioner is that the server should process the requests in the order defined by the state. For instance, the request `{result, 0}` can only be served when the head of the state's list is also `0`. However, there is a problem in this approach. The questioner explains it with the sentence: *when none of the callback function clauses match a message, rather than putting the message back in the mailbox, it errors out.* Although this is the normal and the expected behaviour of a `gen_server`, the questioner thinks that some easy alternative should exists. However, as explained above, the solutions proposed in the thread are not satisfactory enough.

```
1 cpre(test, _, State) -> {true, State};
2 cpre({result, N}, _, [N|R]) -> {true, [N|R]};
3 cpre({result, N}, _, State) -> {false, State}.
```

Fig. 6: `cpre/3` for selective receive

With the enhanced versions of the `gen_server` behaviour we propose in this paper, users can define conditions for each request by using function `cpre/3`. Figure 6 depicts a definition of the function `cpre/3` that solves the questioner's problem without needing to redefine function `handle_call/3`

[7] see `http://erlang.org/mailman/listinfo/erlang-questions`

[8] see `https://stackoverflow.com/questions/1290427/how-do-you-do-selective-receives-in-gen-servers`

of Figure 5. The first clause indicates to the `gen_server_cpre` server that the request `test` can be served always. In contrast, `{result, N}` requests only can be served when `N` coincides with the first element of the server's state.

3.4 Readers-writers example

In this section we define a simple server that implements the readers-writers problem, as a second example of the use of the extended `gen_server_cpre` contract. We start by introducing an implementation of the problem using the standard `gen_server` behaviour. The server state is a record defined as `-record(state, readers = 0, writer = false)`. The requests that it can handle are four: `request_read`, `request_write`, `finish_read` and `finish_write`. The first two requests are blocking (because clients need to wait for a confirmation) while the latter two do not block the client (clients do not need confirmation). Figure 7 shows the handlers for these requests. They basically increase/decrease the counter `readers` or switch on/off the flag `writer`.

```
handle_call(request_read, _, State) ->
  NState = State#state{readers = State#state.readers + 1},
  {reply, pass, NState};
handle_call(request_write, _, State) ->
  NState =  State#state{writer = true}},
  {reply, pass, NState}.

handle_cast(finish_read, State) ->
  NState = State#state{readers = State#state.readers - 1},
  {noreply, NState};
handle_cast(finish_write, State) ->
  NState = State#state{writer = false},
  {noreply, NState}.
```

Fig. 7: Readers-writers request handlers

Having defined all these components, we can already run the readers-writer server. It will start serving requests successfully without any noticeable issue. However, the result in the shared resource is a mess, mainly because we are forgetting an important problem: its invariant, i.e. $!writer \vee readers = 0$. We can define an invariant for the readers-writers server by using the macro `?INVARIANT/1` introduced in Section 2. Figure 8 shows how the macro is used and the helper function which actually checks the invariant. Apart from the standard invariant, i.e., `(not Writer) orelse Readers == 0`, the function also checks that the state field `readers` is a positive integer and that the state field `writer` is a boolean value.

```
?INVARIANT(fun invariant/1).

invariant(#state{ readers = Readers, writer = Writer}) ->
    is_integer(Readers) andalso Readers >= 0
    andalso is_boolean(Writer)
    andalso ((not Writer) orelse Readers == 0).
```

Fig. 8: Readers-writers invariant definition

If we run the server with the invariant defined, we obtain feedback on whether the server is behaving as expected. In this case, the server is clearly not a correct implementation of the problem. Therefore, an error should be raised due to the violation of the invariant. An example of the errors is shown in Figure 9.

The error is indicating that the server state was `{state,0,true}` when the server processed a `request_read` which led to the new state `{state,1,true}` which clearly violates the defined invariant. The information provided by the

```
=ERROR REPORT====
** Generic server readers_writers terminating
** Last message in was request_read
** When Server state == {state,0,true}
** Reason for termination ==
** {{"The invariant does not hold.",Last call: readers_writers:handle_call(
     request_read, ..., {state,0,true}). Result: {reply, pass,{state,1,true}}",
     [{readers_writers,handle_call,3,...},...]}, ...}
```

Fig. 9: Failing invariant report

error report can be improved by returning a tuple `{false, Reason}` in the invariant function, where `Reason` is a string to be shown in this contract-violation report after the generic message.

In order to correctly implement this feature, we use the function `cpre/3` to control when a request can be served or not. Figure 10 shows a function `cpre/3` which makes the server's behaviour correct and avoids violations of the invariant. It enables `request_read` requests as long as the flag `writer` is switched off. Similarly, the `request_write` requests also require the flag `writer` to be switched off and also the counter `readers` to be 0. If we rerun now the server, no more errors due to invariant violations will be raised.

Although this implementation is already correct, it is unfair for writers as they have less chances to access the shared resource. The `EDBC` code repository[9] includes a number of implementations of the example, which implement various fairness criteria.

```
1 cpre(request_read, _, State = #state{writer = false}) ->
2   {true, State};
3 cpre(request_read, _, State) ->
4   {false, State};
5 cpre(request_write, _,
6      State = #state{writer = false, readers = 0}) ->
7   {true, State};
8 cpre(request_write, _, State) ->
9   {false, State}.
```

Fig. 10: Readers-writers `cpre/3` definition

4 Related Work

Our contracts are similar to the ones defined in [1], where the function specifications are written in the same language, i.e., Curry, so they are executable. Being executable enables their usage as prototypes when the real implementation is not provided. Their contracts are functions in the source code instead of macros, so it is not clear whether they could be removed in the final release. One of the authors extended this work in [8], where static analysis performed by an SMT solver at compile time was used to check the contracts. This analysis discharged the overhead produced by the dynamic verification of these contracts. In these works, there is not any mention about whether their contracts are integrated with a documentation tool like our contracts are with EDoc. Moreover, they only allow to define basic precondition and postcondition contracts, while we

[9] `https://github.com/tamarit/edbc/tree/master/examples/readers_writers`

are providing alternative ones like purity or time contracts. Finally, our contracts for concurrent environments has a completely different approach.

The work in [15] presents a static analysis which infers whether a function is pure or not. Since the focus on the article is on static analysis whereas ours is on dynamic analysis, the purity checking is performed in completely different ways in each work. However, we can benefit from their results by, for instance, avoiding to execute functions that are already known to be impure, reporting earlier to the user a purity-contract violation. In the same way, our system can be used in their approach to check the validity of statically-inferred results.

The type contract language for Erlang [10] allows to specify the intended behaviour of functions. Their usage is twofold: i) as a documentation in the source code which is also used to generate EDoc, and ii) to refine static analyses provided by tools such as Dialyzer. The contract language allows for singleton types, unions of existing types and the definition of new types. However, these types and function specifications do not guarantee type safety. This guarantee comes with Erlang which incorporates strong runtime typing with type errors detected and reported at runtime. Although such a static analysis is quite capable in detecting typing violations, strong typing usually detects unexpected behaviour too far from its source. Therefore, when debugging a program, providing the feature to detect violations of such type contracts at runtime can be a useful aid to provide more precise error location.

The contracts proposed for the concurrent environments follow the same philosophy as the specifications defined in [9,7]. Indeed, our function `cpre/3` takes its name from these works. Although these works were more focused on enabling the use of formal methods, or testing techniques, to verify nontrivial properties of realistic systems, in this paper we demonstrate that they can be used to concisely program server applications which are forced to deal with asynchronous requests that must be delayed, and moreover are also useful for runtime verification.

Dafny [11] is a language which allows to define invariants and contracts in their programs. The main difference between their approach and ours is that their contracts are not checked during runtime, but during compile-time. Additionally, as we have explained in Section 2, we can replicate the same type of contracts in `EDBC`. However, our approach does not need an extra syntax or functionality to define complex contracts as Dafny does.

The aspect-oriented approach for Erlang (`eAOP`) presented in [4] shares some similarities with our work. eAOP allows the instrumentation of a program with a user-defined tracing protocol (at an expression level). This is able to report events to a monitor (asynchronous) as well as to force some part of the code to block waiting for information from the monitor (synchronicity). Our system could be used to a similar purpose but only at the function level. Additionally, thanks to the functionality freedom allowed in our contracts, `EDBC` enables the definition of synchronisation operations at the user-defined contracts. More complex modifications of our system, such as the ones done in [13], can transform our work into a complete aspect-oriented system.

Also in Erlang, the work [5] defines a runtime monitoring tool which helps to detect messages which do not match a given specification. These specifications are defined through an automaton, which requires an extra knowledge from the user concerning both the syntax of the specification, and in the whole system operation. We propose a clear and easy way to define the conditions for when to accept a request without needing any user input.

Finally, JErlang [16] enables so called joins in Erlang. This is achieved by modifying the syntax of Erlang receive patterns to permit expressing matching of multiple subsequent messages. Although our goal and theirs are different, both approaches can simplify the way programmers solve similar kinds of problems. Indeed, we could simulate joins by adding a forth parameter to the function `cpre/3`. This additional parameter would represent the still unserved pending requests. When the last request of a user-defined sequence (*join*) is received, the pending requests should be examined to check whether the required join can be served. A similar modification is needed to the callback `handle_call/3` interface so that the pending requests could be served using the `gen_server:reply/2` call.

5 Conclusions

We have developed a new framework `EDBC` which permits annotating Erlang code (functions) with a number of different code contracts, ranging from classical ones such as function pre- and post-conditions to more novel ones, e.g., whether functions are side-effect free, and which can express limits on the execution time of function calls. Such contracts help programmers to formally document otherwise undocumented contextual assumptions regarding how functions may be used. Such a feature can, we believe, for instance reduce the number of bugs introduced when are programs are rewritten to add new functionality, as contracts permit checking that new code interfaces correctly with the old code. An additional advantage that code contracts provide is improved API documentation. Our framework includes a feature to include runtime contract as part of the automatically generated documentation of Erlang functions.

The code contracts supported by `EDBC` are checked at runtime, and can be easily be disabled for deployed code to avoid the runtime overhead of checking contracts, if so desired. Moreover our contracts use plain Erlang, without the need for defining new syntax or requiring supporting libraries.

As a second contribution of the article we have also provided an improved contract for an important class of client–server based concurrent systems. In Erlang such concurrent contracts are manifest as so called “behaviours”, a form of design patterns. Our improved concurrent contract permit programmers to state, in a declarative fashion, when asynchronous client requests can be served by the server, and when they must be postponed as the server is incapable of responding to them in its current state. The `EDBC` provides an implementation of this contract, essentially freeing an Erlang programmer from having to explicitly manage a set of queues to properly coordinate client requests, a nontrivial task.

There are multiple extensions of this work. For instance, contracts can be modified to return a default value instead of an error when a contract is violated, to permit more flexible policies for how contract violations are signalled, and recovered from. Another item for future work is to generalise the "decrease" contracts, which are used to ensure that recursive functions progress.

We can also use our contracts to control starvation of concurrent systems. The idea is to use a mapping from types of request to waiting requests which represents the fact that a delayed request is waiting for a concrete event to occur. An invariant can then be used to control starvation. Another useful extension concerning contracts for concurrent systems would be to cleanly communicate the fact that a server process fails a precondition check to the client process that issued the failing request, thus protecting the server process while communicating the error to the responsible party.

More generally, we can extend our system to translate `EDBC` contracts to EUnit tests cases, to or property test generators for property-based testing tools like Quviq QuickCheck [2]. Finally, we are trying to establish a relation between so called liquid types [17] and our approach as there are a number of similarities.

References

1. S. Antoy and M. Hanus. Contracts and specifications for functional logic programming. In C. V. Russo and N. Zhou, editors, *Practical Aspects of Declarative Languages - 14th International Symposium, PADL 2012, Philadelphia, PA, USA, January 23-24, 2012. Proceedings*, volume 7149 of *Lecture Notes in Computer Science*, pages 33–47. Springer, 2012.
2. T. Arts, J. Hughes, J. Johansson, and U. T. Wiger. Testing telecoms software with quviq QuickCheck. In *Proceedings of the 2006 ACM SIGPLAN Workshop on Erlang, Portland, Oregon, USA, September 16, 2006*, pages 2–10. ACM, 2006.
3. R. Carlsson and M. Rémond. EUnit: a lightweight unit testing framework for Erlang. In M. Feeley and P. W. Trinder, editors, *Proceedings of the 2006 ACM SIGPLAN Workshop on Erlang, Portland, Oregon, USA, September 16, 2006*, page 1. ACM, 2006.
4. I. Cassar, A. Francalanza, L. Aceto, and A. Ingólfsdóttir. eAOP: an aspect oriented programming framework for Erlang. In N. Chechina and S. L. Fritchie, editors, *Proceedings of the 16th ACM SIGPLAN International Workshop on Erlang, Oxford, United Kingdom, September 3-9, 2017*, pages 20–30. ACM, 2017.
5. C. Colombo, A. Francalanza, and R. Gatt. Elarva: A monitoring tool for Erlang. In S. Khurshid and K. Sen, editors, *Runtime Verification - Second International Conference, RV 2011, San Francisco, CA, USA, September 27-30, 2011, Revised Selected Papers*, volume 7186 of *Lecture Notes in Computer Science*, pages 370–374. Springer, 2011.
6. Ericsson AB. EDoc. Available at: `http://erlang.org/doc/apps/edoc/chapter.html`, 2018.
7. L. Fredlund, J. Mariño, R. N. Alborodo, and Ángel Herranz. A testing-based approach to ensure the safety of shared resource concurrent systems. *Proceedings of the Institution of Mechanical Engineers, Part O: Journal of Risk and Reliability*, 230(5):457–472, 2016.

8. M. Hanus. Combining static and dynamic contract checking for curry. In *Proceedings of the 27th International Symposium on Logic-Based Program Synthesis and Transformation (LOPSTR 2017)*, volume 10855 of *Lecture Notes in Computer Science (LNCS)*, pages 323–340. Springer, 2017.
9. Á. Herranz-Nieva, J. Mariño, M. Carro, and J. J. Moreno-Navarro. Modeling concurrent systems with shared resources. In M. Alpuente, B. Cook, and C. Joubert, editors, *Formal Methods for Industrial Critical Systems, 14th International Workshop, FMICS 2009, Eindhoven, The Netherlands, November 2-3, 2009. Proceedings*, volume 5825 of *Lecture Notes in Computer Science*, pages 102–116. Springer, 2009.
10. M. Jimenez, T. Lindahl, and K. Sagonas. A language for specifying type contracts in erlang and its interaction with success typings. In S. J. Thompson and L. Fredlund, editors, *Proceedings of the 2007 ACM SIGPLAN Workshop on Erlang, Freiburg, Germany, October 5, 2007*, pages 11–17. ACM, 2007.
11. K. R. M. Leino. Dafny: An automatic program verifier for functional correctness. In E. M. Clarke and A. Voronkov, editors, *Logic for Programming, Artificial Intelligence, and Reasoning - 16th International Conference, LPAR-16, Dakar, Senegal, April 25-May 1, 2010, Revised Selected Papers*, volume 6355 of *Lecture Notes in Computer Science*, pages 348–370. Springer, 2010.
12. T. Lindahl and K. Sagonas. Detecting Software Defects in Telecom Applications Through Lightweight Static Analysis: A War Story. In *Programming Languages and Systems: Second Asian Symposium, APLAS 2004, Taipei, Taiwan, November 4-6, 2004. Proceedings*, volume 3302 of *Lecture Notes in Computer Science*, pages 91–106. Springer, 2004.
13. D. H. Lorenz and T. Skotiniotis. Extending Design by Contract for Aspect-Oriented Programming. *CoRR*, abs/cs/0501070, 2005.
14. B. Meyer. Applying "Design by Contract". *IEEE Computer*, 25(10):40–51, 1992.
15. M. Pitidis and K. Sagonas. Purity in Erlang. In J. Hage and M. T. Morazán, editors, *Implementation and Application of Functional Languages - 22nd International Symposium, IFL 2010, Alphen aan den Rijn, The Netherlands, September 1-3, 2010, Revised Selected Papers*, volume 6647 of *Lecture Notes in Computer Science*, pages 137–152. Springer, 2010.
16. H. Plociniczak and S. Eisenbach. JErlang: Erlang with Joins. In D. Clarke and G. A. Agha, editors, *Coordination Models and Languages, 12th International Conference, COORDINATION 2010, Amsterdam, The Netherlands, June 7-9, 2010. Proceedings*, volume 6116 of *Lecture Notes in Computer Science*, pages 61–75. Springer, 2010.
17. P. M. Rondon, M. Kawaguci, and R. Jhala. Liquid types. In *Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '08, pages 159–169, New York, NY, USA, 2008. ACM.