Introduction

[license GPLv3 blue] [version v0]

`detectEr` is an inline monitoring tool that synthesises runtime monitors from correctness properties specified in syntactic subset of Hennessy-Milner Logic with recursion that is in its *normal form*. This logic fragment is used to specify *invariants* about the system under scrutiny; we will refer to this logic fragment as *safety HML*, or sHML_{nf} for short. `detectEr` assumes access to the source code of the program to be monitored. It instruments monitoring instructions into the target program via *code injection* by manipulating its parsed abstract syntax tree. The modified syntax tree is compiled by `detectEr` into an executable form which can then be run normally. The instrumented instructions perform the runtime analysis in *synchronous* fashion as the program executes. Our tool is developed in the Erlang language and for the Erlang eco-system.

Overview

Correctness properties in `detectEr` are scripted in plain text format using SHML_{nf} syntax. These are then parsed, and the corresponding executable runtime monitor is synthesised in its source or binary form. The resulting file bears the name of the properties script, and contains one function, `mfa_spec`, that encapsulates all the monitoring logic.

Script file

More than one property may be specified in a single script file, albeit separated by commas `,`; the last one must be terminated with a period `.`. A specification must target *one* Erlang function pattern, and may optionally include simple *guard statements*. The guard specification format follows the one used by Erlang, the only difference being that guard functions (*e.g.* `is_pid/1`, `is_alive/1`, *etc.*) are currently not supported. Patterns and guards on Erlang binaries, bitstrings and maps will be implemented in future releases. Specifications can only target function calls, *i.e.*, `Mod:Fun(Args)`, that have been exported from Erlang modules and invoked via `erlang:spawn/3` or `erlang:spawn/4`.

Example 1.

The function `test` in module `example` takes two parameters, `A` and `B`, where `A` is an integer in the closed interval [0, 10]. A property specification that targets the function `test` is specified as follows:

```
with
  example:test(A, B) when A >= 0, A =< 10
monitor
  property in SHML_nf.
```

The property itself, `property in` SHML_{nf}, is written using the grammar defined next.

The $SHML_{nf}$ grammar

The $SHML_{nf}$ grammar is defined by the following BNF:

```
<SHML_nf> ::= ff  ①
            | and(<SHML_nf list>)  ②
            | X  ③
            | max(X. <SHML_nf>).  ④
```

① Falsity, an atom

② A sequence of comma-separated **conjuncts** where each conjunct is a sub-formula `<SHML_nf>` that starts with a necessity `[<ACTION>]<SHML_nf>`.

③ Recursion variable specified as a standard Erlang variable

④ Maximal fix-point that specify recursive loops comprised of one variable and the sub-formula `<SHML_nf>`

A necessity, specified by `[]` contains the `<ACTION>` that is matched with the trace event exhibited by the system. Matching is performed both on the type of action as well as the data it carries in relation to the trace event. There are five actions types:

Fork	Process creation action, specified as PID_p `→` PID_c`, M:F(A)`. This action is exhibited by the parent process invoking fork.
Forked	Process initialisation action, specified as PID_p `←` PID_c`, M:F(A)`. This action is exhibited by the child process that has been forked.
Exit	Process termination action, specified as `PID ** CLAUSE`.
Send	Process send action, specified as `PID ! CLAUSE`.
Receive	Process receive action, specified as `PID ? CLAUSE`.
Any	Generic user-given process action, specified as `CLAUSE`.

where `PID` is a generic Erlang variable that binds to a process ID (PID). PID_p denotes the PID of the parent process and PID_c, the PID of the child (forked) process. `M:F(A)` denotes the function that was forked to execute in its own independent process: the variable `M` binds to the name of the *module* where the function resides, `F`, to the forked *function*, and `A`, binds to the *arguments* specified in `F`. `CLAUSE` represents a standard Erlang clause that may in turn contain generic data variables. Actions inside the necessity construct `[]` may optionally include guards, albeit with the restrictions mentioned above.

The following are some examples of scripted HML_{nf} properties.

Example 2.

This property checks that the parent process does not fork child processes with negative IDs. The property below reads as: "the parent process `P` cannot fork a child process `C` via the function `child:init(Id, StartCnt)` such that the `Id` assigned is negative".

```
with
  parent:start()
monitor
    and([P -> C, child:init([Id, _]) when Id < 0] ff).
```

The falsity `ff` states that when the necessity `[P → C, child:init([Id, _]) when Id < 0]` matches such a fork event, a *violation* is flagged. We use the Erlang anonymous variable `_` to bind `StartCnt` since this value is unimportant.

Example 3.

Apart from checking that the parent process does not fork child processes with negative IDs, this property also requires that it does not terminate. The property below reads as: "the parent process `P` cannot fork a child process `C` via the function `child:init(Id, StartCnt)` such that the `Id` assigned is negative **AND** it cannot terminate"

```
with
  parent:start()
monitor
  and([P -> C, child:init([Id, _]) when Id < 0] ff, [P ** aborted] ff).
```

The outer `and(⋯)` is comprised of a list with two necessities. Necessities in a list are conjoined. The first matches the action `P → C, child:init([_, _])` to the fork event exhibited by `P`. The second matches the action `P ** aborted` to the termination event exhibited by `P`.

Example 4.

This property checks that the parent process does not fork a child process and terminates immediately after with the reason `aborted`. The property below reads as: "the parent process `P` cannot fork a child process `C` via the function `child:init(Id, StartCnt)` and terminate immediately with the reason `aborted` ".

```
with
  parent:start()
monitor
  and([P -> C, child:init([_, _])] and([P ** aborted] ff)).
```

The outer `and(⋯)` construct consists of a **single necessity** that matches the action `P → C, child:init([_, _])` to the fork event exhibited by `P`. Similarly, the inner `and(⋯)` matches the only action `P ** aborted` to the termination event exhibited by `P`. We remark that, following the SHML_{nf} grammar given above, nesting `and` constructs, *i.e.*, `and([⋯] and([⋯], ⋯))`, enables us to encode necessity *sequences*.

Example

The example that we cover next assumes that Erlang is installed and that you are familiar with the Erlang REPL. Before proceeding, the source code should be compiled. This can be done from the command prompt using the make file target `make compile`. The Erlang REPL can be conveniently launched from the current directory by typing `make run`. Whenever necessary, exit the REPL using the key combination `CTRL+c`.

Sample System

We include a source code sample that models a simple client-server interaction. This sample can be found under the `./src/system/` directory. The server module (`server.erl`) exposes two functions, start and stop, that are used to launch and terminate the server process. It also implements three operations described below:

Operation	Request	Description
stop	`{From, Ref, stop}`	Server stop request
add	`{From, Ref, {add, A, B}}`	Addition request
multiply	`{From, Ref, {mul, A, B}}`	Multiplication request

The variables `From` and `Ref` bind to the PID of the sender process and reference respectively; `A` and `B` bind to the numbers that are operated upon. `Ref` is used for internal implementation purposes, and is unimportant in what follows. The function `stop/1` exposed by the `server` module sends a `stop` request to the server process to terminate it. Our server is started and stopped from the Erlang REPL as follows:

Starting and stopping the server

```
1> server:start(ok).
<0.81.0>
2> server:stop().
{ok,stopped}
```

Executing `server:start/1` returns the PID <0.81.0> assigned to server process by Erlang. We specified the option `ok` when starting the server to launch our server process in correct operating mode. Option `buggy` starts the server in buggy mode, and this is the mode we shall use to test our correctness properties with. The message on the last line, `{ok,stopped}`, shows the Erlang tuple the server sends to the caller of `server:stop/0` as a confirmation. Raw requests to the server process can be sent as follows:

Sending raw requests

```
1> server:start(ok).
<0.81.0>
2> server ! {self(), ref, {add, 9, 7}}.
{<0.79.0>,ref,{add,9,7}}
3> flush().
Shell got {ref,{add,16}}
```

The client module (`client.erl`) exposes two remote invocation stubs that encapsulate the sending and receiving of message requests to and from the server. These correspond to the add and multiply operations, and are used like so:

Adding and multiplying using the client API

```
1> server:start(ok).
<0.81.0>
2> client:add(9, 7).
16
3> client:mul(9, 7).
63
```

If we start the server using the `buggy` flag, the add and multiply operations used above return the wrong result.

Starting the server in buggy mode

```
1> server:start(ok).
<0.81.0>
2> client:add(9, 7).
17
3> client:mul(9, 7).
64
```

Monitoring the server

Suppose we would like to specify a correctness property in SHML_{nf} that verifies the addition functionality exposed by the server. For this particular case, we are only interested in the *addition functionality*. This property, found in `./examples/example_1.hml`, is explained below.

Example 5.

Our property should be interpreted from the point of view of the server process.

```
with
  server:loop(_)  ①
monitor
  and([Launcher <- Server, server:loop(_)]  ②
  max(X. ③
    and(  ④
      [Server ? {Client, _, {add, A, B}}] and(  ⑤ ⑥
        [Server ! {_, {add, AB}} when AB =/= A + B] ff,  ⑦
        [Server ! {_, {add, AB}} when AB =:= A + B] X  ⑧
      ),
      [Server ? {Client, _, {_, _, _}}] and(  ⑨
        [Server ! {_, {_, _}}] X  ⑩
      ),
      [Server ? {Stopper, _, stop}] and(  ⑪
        [Server ! {_, {ok, stopped}}] X  ⑫
      )
    )
  )
).
```

To facilitate our explanation, we break down the property into the following intuitive steps:

① Target the function `server:loop/1` with any argument (it can match the arguments `ok` or `buggy`)

② Match the forked initialisation event exhibited by the server

③ Start the maximal fix-point that allows us to encode looping via recursion on the variable *X*

④ Outer `and(⋯)` consists of a list with three conjuncts

⑤ **First** conjunct specifies the meat of the property that determines whether the server is buggy

⑥ Match the client request receive event `?` exhibited by the server, in this case `{add, A, B}`, continued by,

⑦ Match the response send event `!` to the client, `{add, AB}}` when the addition of `A` and `B` **does not** match the value `AB` returned by the server; **ff** signals a violation of the property, **AND**,

⑧ Match the response send event `!` to the client, `{add, AB}}` when the addition of `A` and `B` matches the value `AB` returned by the server; the recursive variable *X* is unfolded

⑨ **Second** conjunct matches any client request receive events, continued by,

⑩ Match any response send event to the client; the recursive variable *X* is unfolded

⑪ **Third** conjunct matches the stop request receive event `?` exhibited by the server, `stop`, continued by

⑫ Match the response send event `!` exhibited by the server, `{ok, stopped}`; the recursive

variable X is unfolded.

Synthesising the runtime monitor

To synthesise the runtime monitor, the following command can be run from the Erlang REPL:

Compiling the $SHML_{nf}$ property down to a monitor

```
1> hml_eval:compile("examples/example_1.hml", [{outdir, "ebin"}, v]).
ok
```

The corresponding file `example_1.beam` containing the executable monitor code is created in the directory `ebin`. Our compiler takes the following options:

Option	Description
`outdir`	Directory where the generated output monitor file should be written. If left unspecified, defaults to the current directory `.`
`v`	Inserts logging statements into the generated output monitor file. Only use for debugging purposes
`erl`	Instructs the compiler to output the generated monitor as Erlang source code rather than beam. If left unspecified, defaults to beam

We used the `v` flag so that the compiled monitor produces verbose output on the REPL.

Instrumenting the system

The system is instrumented by executing the weave function. We specify the source file (`server.erl`) to be weaved, together with the function `example_1:mfa_spec/1` encapsulating the synthesised monitor code corresponding to our property.

Instrumenting the server

```
2> weaver:weave_file("src/system/server.erl", fun example_1:mfa_spec/1, [{outdir,
"ebin"}]).
{ok,server,[]}
```

Readers can inspect the source in `launcher.erl` for more details. As before, the output directory is set to `ebin`; the instrumented server module `server.beam` is correspondingly compiled to this directory. Our code weaver can also instrument all the files in a given directory using `weaver:weave/3` The options supported by `weaver:weave_file/3` and `weaver:weave/3` are identical:

Option	Description
`outdir`	Directory where the generated weaved files should be written. If left unspecified, defaults to the current directory `.`
`i`	Directory containing include files that the source files in the source directory depend on
`filter`	filter function that suppressed events. If left unspecified, defaults to allows any
`erl`	Instructs the compiler to output the generated files as Erlang source code rather than beam. If left unspecified, defaults to beam

Running the correct server

We start by testing our monitor on the correct version of the server. Exit the REPL by using `CTRL+c` and type `make run` again on the terminal to *reload the instrumented* server. The monitor generated earlier with the verbose `v` flag set logs to the REPL the trace event it analyses. Log statements can be identified by the PID in the square brackets. In the excerpt below, the references in `#Ref<⋯>` are shortened for clarity.

Runtime analysis

```
1> server:start(ok).
[<0.81.0>] Analyzing event {trace,<0.81.0>,spawned,<0.79.0>,{server,loop,[0]}}.  ①
<0.81.0>
2> client:add(9, 7).  ②
[<0.81.0>] Analyzing event {trace,<0.81.0>,'receive',{<0.79.0>, #Ref<...>,{add,9,7}}}.
③
[<0.81.0>] Analyzing event {trace,<0.81.0>,send, {#Ref<...>,{add,16}}, <0.79.0>}.  ④
[<0.81.0>] Unfolding rec. var. '<em>X</em>'.  ⑤
16  ⑥
3> client:mul(9, 7).  ⑦
[<0.81.0>] Analyzing event {trace,<0.81.0>,'receive',{<0.79.0>, #Ref<...>,{mul,9,7}}}.
[<0.81.0>] Analyzing event {trace,<0.81.0>,send,{#Ref<...>,{mul,63}},<0.79.0>}.
[<0.81.0>] Unfolding rec. var. '<em>X</em>'.
63
```

The monitoring code weaved into the server effects this analysis:

① `spawned` event is analysed when the server is launched; `spawned` is the Erlang equivalent of forked

② User invokes `client:add/2` on Erlang REPL

③ `receive` event is analysed when the request sent by `client:add/2` is processed by the server

④ `send` event is analysed when the server replies back

⑤ Send request by the server matches the expected result, *i.e.*, `9 + 7 =:= 16`; the internal monitor branch that unfolds the recursive variable *X* is taken, and the monitor loops back to its starting

state

⑥ Correct result returned from server

⑦ User invokes `client:mul/2` on Erlang REPL and a similar analysis is performed by the weaved monitor.

Running the buggy server

We now test the buggy server when requesting additions from the server. Exit the REPL by using `CTRL+c` and type `make run` again on the terminal to *reload the instrumented* server.

Runtime analysis with `add`

```
1> server:start(buggy).
[<0.81.0>] Analyzing event {trace,<0.81.0>,spawned,<0.79.0>,{server,loop,[1]}}.  ①
<0.81.0>
2> client:add(9, 7).  ②
[<0.81.0>] Analyzing event {trace,<0.81.0>,'receive',{<0.79.0>,#Ref<...>,{add,9,7}}}.
③
[<0.81.0>] Analyzing event {trace,<0.81.0>,send,{#Ref<...>,{add,17}},<0.79.0>}.  ④
[<0.81.0>] Reached verdict '<strong>no</strong>'.  ⑤
17
3> client:add(9, 7).  ⑥
17
```

The monitoring code weaved into the server effects this analysis:

① `spawned` event is analysed when the server is launched

② User invokes `client:add/2` on Erlang REPL

③ `receive` event is analysed when the request sent by `client:add/2` is processed by the server

④ `send` event is analysed when the server replies back

⑤ Send request by the server **does not match** the expected result, *i.e.*, `9 + 7 =/= 17`; the internal monitor branch that flags a rejection is taken, and the monitor stops its analysis

⑥ Subsequent calls to `client:add/2` do not trigger the monitor henceforth.

Performing the same execution and requesting multiplications produces different results:

Runtime analysis with `mul`

```
1> server:start(buggy).
[<0.81.0>] Analyzing event {trace,<0.81.0>,spawned,<0.79.0>,{server,loop,[1]}}.  ①
<0.81.0>
2> client:mul(9, 7).  ②
[<0.81.0>] Analyzing event {trace,<0.81.0>,'receive',{<0.79.0>,#Ref<...>,{mul,9,7}}}.
③
[<0.81.0>] Analyzing event
{trace,<0.81.0>,send,{#Ref<0.3424250081.1934360577.92435>,{mul,64}},<0.79.0>}.  ④
[<0.81.0>] Unfolding rec. var. '<em>X</em>'.  ⑤
64
client:add(9, 7).  ⑥
[<0.81.0>] Analyzing event {trace,<0.81.0>,'receive',{<0.79.0>,#Ref<...>,{add,9,7}}}.
[<0.81.0>] Analyzing event {trace,<0.81.0>,send,{#Ref<...>,{add,17}},<0.79.0>}.
[<0.81.0>] Reached verdict '<strong>no</strong>'.
17
```

The following analysis is effected by the monitor:

① `spawned` event is analysed when the server is launched

② User invokes `client:mul/2` on Erlang REPL

③ `receive` event is analysed when the request sent by `client:mul/2` is processed by the server

④ `send` event is analysed when the server replies back

⑤ Send request by the server for `mul` does not match the `add` conjunct; the second conjunct that matches any request is taken, and the monitor unfolds the recursive variable *X*, looping back

⑥ Calling `client:add/2` at this point **still** triggers the analysis to flag a violation.

Conclusion

We invite readers to try specifying other properties on our client-server system. For instance, one could write a second property inside `example_1.erl` to flag a rejection once the server is terminated with the function `server:stop/0`. Should you have any questions, comments or spot any bugs, do not hesitate to contact us.