1 DDMon testing DSL and test scenarios

In *Section 7* of the companion paper we evaluate DDMon's performance in different test scenarios involving various forms of deterministic and non-deterministic deadlocks, as well as executions that terminate successfully. Each one of these test scenarios is described as a "scenario file" written in a domain-specific language (DSL) which declares a number of services and then specifies how they exchange queries and responses with each other (possibly leading to a deadlock).

The execution of a test scenario begins with a number of initial `gen_server` calls (a.k.a. queries, in the terminology of the companion paper) to selected services starting concurrent *sessions*. In this context, a "session" means the collection of all service actions that were (possibly indirectly) caused by a single initial call. For instance, if service `A` initiates a call to `B`, which then in turns invokes `C`, we consider that a single session.

The data transmitted in each call contains instructions on how the data is supposed to be processed, including making further calls, waiting, or sending a response. In this way, a single `gen_server` instance created by the testing DSL can reproduce a variety of possible behaviours and deadlocks that may occur in real systems.

See *Appendix A.1* of the companio paper for additional information.

1.1 Scenario file format

A test scenario is specified as an Erlang `conf` file which follows the following format:

```
%%% The scenario file. Describes all sessions that are to be executed in parallel.
-type scenario() :: {sessions, list(session())}.

%%% Single session identified as `[Name]`. `[Initial]` is the process id that is
%%% called first when the session is initialized. `[Schedule]` describes further
%%% logic of the session, which shall be executed by `[Initial]`.
-type session() :: {Name :: atom(), {Initial :: proc_id(), Schedule :: schedule()}}.

%%% Description of the computation shape.
%%% - `{E0, E1, ..}` indicates sequential execution within the process, i.e.
%%%   execute `E0`, then `E1` and so on.
%%% - `[I :: proc_id() | S]` makes a blocking call to `I` and orders it to execute `S`
%%% - `{wait, X}` waits for `X` miliseconds
%%% - `{wait, X, Y}` waits randomly between `X` and `Y` miliseconds
%%% - `[]` does nothing
%%% After all actions are executed, the service sends a response.
-type schedule()
    :: tuple(schedule())
    |  [proc_id() | schedule()]
    |  {wait, non_neg_integer()}
    |  {wait, non_neg_integer(), non_neg_integer()}
    |  []
    |  schedule_sugar()
    .

%%% - `42` as a `schedule()` acts as `[42]`
%%% - `[{wait, X} | S]` (and its variants) act as `{{wait, X}, S}`
-type schedule_sugar()
    :: proc_id()
    |  [{wait, non_neg_integer()} | schedule()]
    |  [{wait, non_neg_integer(), non_neg_integer()} | schedule()]
    .

%%% Services are identified by integers. Can be selected at random from a list.
-type proc_id() :: non_neg_integer() | {random, list(proc_id())}.
```

1.1.1 Example

The following example illustrates a scenario of 4 services (numbered `0`, `1`, `2` and `3`) involved in two independent sessions (`left` and `right`).

- `left` begins with a call to the service `0`, which will call `1` for an immediate reply, wait for 50 milliseconds, and then call `1` again.
- `right` starts in parallel with a call to `3` which immediately calls `2`. Upon receiving the call, `2` will first wait for some random time between 0 and 100 milliseconds, and then call `1`, which will then call `0`.

```
{sessions,
 [ {left,
    { 0
    , { 1
      , {wait, 50}
      , 1
      }
    }
   }
 , {right,
    { 3
    , [2, {wait, 0, 100}, 1, 0]
    }
  }
 ]
}.
```

A deadlock may occur depending on how long service `2` will wait after receiving a call from `3`:

- If `2` waits long enough for `left` to receive a response, then the scenario will terminate with `0` sending a response to `1`, `1` to `2`, `2` to `3` and `3` finishing the session `right`.
- Instead, if `2` sends a query to `1` after `0` receives the initial call from `left`, but before `0` sends its final query to `1`, then `1` and `0` become locked on one another, resulting in a deadlock.

1.2 Running scenarios

To evaluate a particular scenario, build the project with `make` and use the generated escript as follows:

```
./ddmon ./scenarios/example.conf
```

Alternatively, use docker:

```
docker build -t ddmon .
docker run --rm ddmon ./ddmon scenarios/example.conf
```

See `./ddmon --help` for more information about the CLI interface.

1.3 Reading the log

Unless `--silent` is set, `ddmon` will print out a coloured log of certain events happening in the system.

By default, the output is formatted in multiple columns (one for each service running in the scenario); this column formatting can be turned off by setting `--indent=0`.

The entries in the log are presented in the format:

```
<TIMESTAMP> | <WHO>: <EVENT> |       |  ...
```

Where:

- `<TIMESTAMP>` is the timestamp of an event
- `<WHO>` is the process that reported the event
- `<EVENT>` describes the event

The `<EVENT>` can be either of the following:

- `X ? Q(s)` — Received a query from `X` in session `s`.
- `X ? R(s)` — Received a response from `X` in session `s`.
- `X ! Q(s)` — Sent a query to `X` in session `s`.
- `X ! R(s)` — Sent a response to `X` in session `s`.
- `%[ Q(s) ]` — Monitor handles an incoming query and forwards it to its service.
- `%[ R(s) ]` — Monitor handles an incoming response and forwards it to its service.
- `%[ ?!Q(s) ]` — Monitor handles an outgoing query sent by its service.

- `%[ ?!R(s) ]` — Monitor handles an outgoing response sent by its service.
- `=> UNLOCK` — Monitor enters unlocked state.
- `=> LOCK(p)` — Monitor enters locked state and initiates probe `p`.
- `=> ### DEADLOCK ### DL` — Monitor detects a deadlock; `DL` is the list of services which form a deadlock cycle.
- `=> foreign_deadlock` — Monitor receives a deadlock notification (for more details, see IMPLEMENTATION.md under "Deadlock reporting"). This happens when the monitored service is dependent on a deadlock detected by another monitor.
- `release X` — message indicating that a worker of a replicated service has finished its task and is ready to receive a query. (The encoding of replicated services in our formal model is described in *Appendix B* of the companion paper.)
- `waiting N ms` — process is waiting `N` milliseconds.

Erlang processes are identified as follows:

- `I0` — the session initiator
- `Px` — service `x` (`P` stands for "process": each service is an Erlang process)
- `Px(y)` — worker `y` of the replicated service `x` (the encoding of replicated services in our formal model is described in *Appendix B* of the companion paper)
- `Mx` — monitor of the service `x`
- `Mx(y)` — monitor of the worker `y` of the replicated service `x`

For example, the provided scenario `scenarios/supersimple.conf` produces the following log: (the timestaps may vary)

```
00:000:020  | M0:   I0 ? Q(s)
00:000:039  | M0:   %[ Q(s) ]
00:000:044  | M0:   P0 ! Q(s)
00:000:062  | P0:   M0 ? Q(s)
00:000:069  | P0:   waiting 10ms
00:012:988  | P0:   M0 ! R(s)
00:013:011  | M0:    ? R(s)
00:013:023  | M0:   %[ ?! R(s) ]
00:013:034  | M0:   I0 ! R(s)
```

Here, the service `0` is programmed to response to the initial call after a short delay. In more details:

1. Monitor of service `0` receives a call from the initiator of session `s`
2. Monitor of service `0` picks the message from its inbox
3. Monitor of service `0` forwards (sends) the message to the service `0`
4. Service `0` receives the message
5. Service `0` waits for 10 milliseconds
6. Service `0` sends back a response to its monitor
7. Monitor of service `0` receives the response from its service
8. Monitor of service `0` picks the response from its inbox
9. Monitor of service `0` sends the response back to the session initiator

1.4 Scenario templates for benchmarking

Aside from the scenario interpreter described above, we provide a utility to generate random scenarios based on pre-defined *templates* and benchmark them to obtain various performance statistics. We used those statistic to obtain figures in *Section 7* of the companion paper. Such benchmarks are described as Erlang `.conf` files with the following format:

```
{bench, list(scenario_template())}
```

Currently, the only supported template is `{conditional, <NCopies>, <Size>}` which generates scenarios that occasionally run into deadlocks similar to the one presented in the *Figure 7* of the companion paper. Here, `<Size>` defines the number of services, while `<NCopies>` specifies how many experiments should be conducted with this template.

To conduct an example benchmark, run

```
./ddmon ./scenarios/bc-small.conf
```

Since multiple scenarios are evaluated in parallel, the log described in the section above is replaced with an animation depicting status of each experiment. While experiments are conducted, the output should be similar to

```
[ . T D D @ D D @ D @ D @ D O O @ O D @ . . . D D D D . @ @ @ . D D D D O O @ .
  D . @ @ @ D D @ . D @ @ o @ D . @ D . @ D @ @ D . . . . D D @ D . @ @ @ D o D
  @ @ D . D D D D @ @ @ D D o o D @ D . . @ D @ D . D @ @ D . D D @ @ @ @ @ D @
  D D D ]
```

Each character (except `[`, `]` and) shows the status of one experiment:

- `.` — experiment not started yet
- `o` — experiment is in preparation (e.g. the scenario is being generated)
- `O` — experiment is running
- `@` — experiment terminated successfully with no deadlocks
- `D` — experiment terminated successfully with deadlocks reported
- `T` — experiment timed out
- `!` — an unexpected error happened during the execution

After that, the tool produces aggregated statistics from each experiment in CSV format. The statistics include numbers of messages exchanged between different types of entities, numbers of deadlocks reported, numbers of different types of messages, etc.

1.5 Overview of provided scenarios

Below we provide descriptions of other scenarios that we used in testing. All can be found in the `scenarios` directory as `.conf` files.

1.5.0.1 `supersimple`

Simple test with only one service that just responds to the initial call.

```
{sessions, [ {s, {0, [{wait, 10}]}} ] }.
```

1.5.0.2 `deadlock`

Scenario modelling a certain deadlock where service `0` calls service `1`, `1` calls `2` and `2` calls `0`, creating a locked-on dependency loop.

```
{sessions,
 [ {s, { 0
       , [ 1, 2, 0]
       }
   }
 ]
}.
```

1.5.0.3 `nodeadlock`

Scenario that always terminates without deadlocking. Two parallel sessions finish successfully:

- Service `0` calls `1`, `1` calls `2`, `2` responds to `1`, `1` responds to `0`
- Service `4` calls `3`, `3` calls `0` (optionally waiting for it to receive response from `1`), then `0` responds to `3`, and `3` responds to `4`

```
{sessions,
 [ {left, { 0
          , [1, 2]
          }
   }
 , {right, { 4
           , [3, 0]
           }
   }
 ]
}.
```

1.5.0.4 `random`

Approximately 50% chances to deadlock or not. (This scenario is also described in the "Example" section above.)

This scenario starts two independent sessions: `left` and `right` with 3 services.

- `left` begins with a call to the service `0`, which will wait for 50 milliseconds and then call `1`, which then responds immediately.
- `right` starts in parallel with a call to `2` which waits for some time randomly between 0 and 100 milliseconds, then calls `1`, which then calls `0`.

A deadlock may occur depending on how long `2` waits:

- If `2` waits long enough for `left` to terminate, then `1` will not be blocked and the scenario terminates `0` responding to `1`, `1` to `2`, and `2` finishing the session.
- If `2` does not wait long enough, then `2` may block `1` expecting a response from `0` while session `left` blocks `0` until `1` is unblocked, implying a deadlock.

```
{sessions,
 [ {left, { 0
          , [{wait, 50}, 1]
          }
   }
 , {right, { 2
          , [{wait, 0, 100}, 1, 0]
          }
   }
 ]
}.
```

1.5.0.5 `routing` An example of a replicated service implemented as described in *Appendix B*. Here, service `0` has two workers (`{router, {0, 2}}`). The session begins with service `1` calling `0`, which then calls service `2`, which calls `0` again. Without replication, this scenario would cause a deadlock; however, since service `0` schedules queries between two workers, the deadlock is avoided.

```
{router, {0, 2}}.

{sessions,
 [ {left,
    { 1
    , [0, 2, 0]
    }
   }
 ]
}.
```

1.5.0.6 `seq` Service `0` calls `1` which immediately sends a response. Then, `0` waits for 3 milliseconds and calls `1` again, but this time `1` waits for between 10-50 milliseconds and calls service `2` twice, after which it replies back to `0`. Finally, `0` calls `1` again.

This is a test of sequential operations and calling the same service multiple times.

```
{sessions,
 [ {sess, { 0
          , { 1
            , {wait, 3}
            , [ 1, {wait, 10, 50} | { 2, 2 } ]
            , [1]
            }
          }
   }
 ]
}.
```

1.5.0.7 `reply_and_dead` Service `0` calls `1` which responds immediately. Then, `0` calls `1` again, but this time `1` calls `0` causing a deadlock.

```
{sessions,
 [ {init, { 0
          , { [1], [1, 0] }
          }
   }
 ]
}.
```

1.5.0.8 `bc` and `ts` These scenarios describe large benchmarks used to obtain *Figures 15 and 16* in the companion paper, respectively. We used two different scenario configurations because the measurements in *Figure 15* require modelling a high amount of `gen_server`-based systems with different sizes, whereas the time series in *Figure 16* is generated by running fewer but larger systems (which improves the clarity of the resulting plot).