contractLarva v0.2α
User Manual

Gordon J. Pace and Joshua Ellul and Shaun Azzopardi
{gordon.pace|joshua.ellul|shaun.azzopardi}@um.edu.mt

5 December 2017, updated 11 October 2019

> *This is an alpha version of* contractLarva. *If you are using the tool on smart contracts which will be deployed on Ethereum, it is recommended that you inspect the code created by the tool before deployment. The authors accept no responsibility of any losses, direct or otherwise, incurred due to the use of the tool.*

1 Overview

The notion of smart contracts has enabled anyone to write code which handles potentially large volumes of digital assets, typically in the form of a cryptocurrency. The price to pay for the ease in which this can now be achieved is that many smart contracts are written without necessary safeguards to ensure that they work as envisaged and do not malfunction. In addition, the language adopted to express such smart contracts on a number of distributed ledger systems has been designed to be Turing-complete — a language powerful enough able to express any computation. But with power comes responsibility, since more complex computations are more likely to have a higher incidence of bugs.

In order to ensure the correct behaviour of a smart contract, one would want to write a specification of what it should be performing, and ensure that it behaves accordingly. In contrast to the implementation, which specifies *how* to calculate the result, the specification only specifies properties which the system behaviour and results should satisfy. While a smart contract implementing a complex voting system might explain how voting, vote delegation, vote weights, etc. work, a property might simply specify that from the moment an issue is put to the vote and the moment a decision is taken, the total number of votes in the system (counting both cast and uncast) will not change. Another property might state that no one will lose their vote unless they invoke a method to delegate it. Similarly, in a wallet management smart contract, one might specify properties such as that a wallet may not change ownership, or that a wallet cannot be initialised more than once. Such properties are typically far easier to state than it is to program a system which satisfies them. Such specifications can be used in various ways e.g. as oracles in testing or as properties for static analysis. Another use is in runtime verification, where the runtime behaviour of the system is compared to the specification in order to identify violations as soon as they happen.

contractLarva is a runtime verification tool for smart contracts written in Solidity. The tool takes a specification using dynamic event automata (see Section 2) and a Solidity contract, and produces a new Solidity contract which acts just like the original smart contract, but in addition checks the dynamic (runtime) behaviour against the specification, allowing for reparations to be performed in case of a violation.

One of the challenges in the case of smart contracts lies in what to do if a violation does occur. The tool allows for the person writing the specification to decide how to react to violations. At a simplest level, one can, for instance, block the smart contract from executing further (other than emptying its content). However, one may also use a more sophisticated approach. For instance, in Section 3.3 we show an example of specification reparation based on a stake-placing design pattern, in which any party that can potentially violate the contract pays in a stake before running the contract, which will be given to the aggrieved party in case of a violation, but returned to the original owner if the contract terminates without violations.

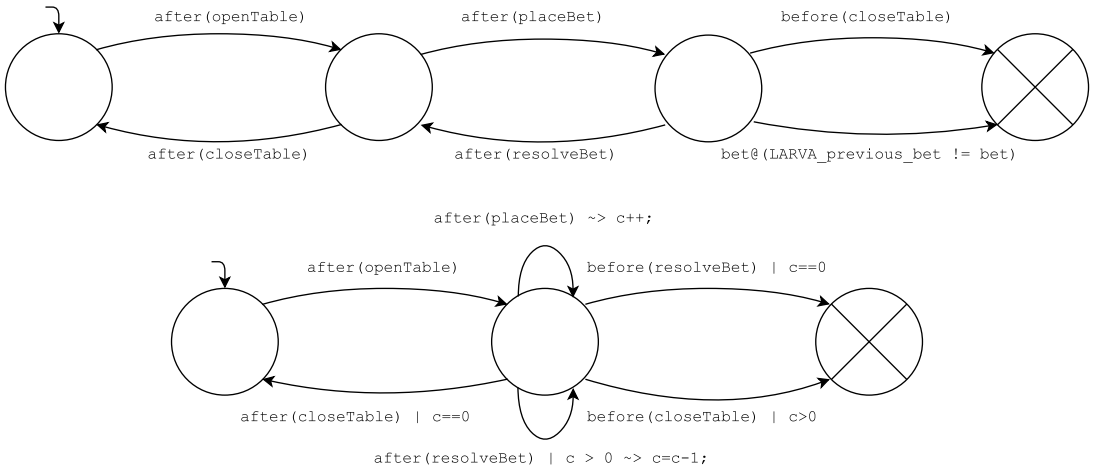

Figure 1: Contract specification examples (a) a betting table may not be closed when there is a placed wager which has not been resolved; (b) generalisation of the previous specification to allow for multiple simultaneous wages to be placed on the table.

contractLarva is a tool still in active development and planned to be extended in the future, with some planned extensions being listed in Section 5. However, the tool is being released as open source to encourage independent extensions and offshoots to the tool.

2 Specifying Properties using Dynamic Event Automata

In order to specify properties, contractLarva used *dynamic event automata* (DEA) — an automaton-based formalism, based on that used in the Larva runtime verification tool for Java[1].

Such specifications monitor for events (the choice of the term *event* is unfortunately overloaded with the notion of events in Solidity — in contractLarva, the use of the term is limited to the notion of event triggers as used in DAEs unless explicitly otherwise noted) over the contract and enable the specification of event traces which are not desirable. The tool allows capturing two types of events: (i) control-flow events corresponding to entry and exit points of functions defined in the smart contract, written `before(f)` and `after(f)` to refer to the entry and exit point of function `f` respectively; and (ii) data-flow events corresponding to changes in values of variables, written as `v@e` to denote the event when variable `v` is changed and expression `e` (which can also refer to the previous value of `v` as `LARVA_previous_v`) holds e.g. `winout@(winout < LARVA_previous_winout)` identifies points in the execution of the contract when the win-out amount is decreased.

At their most basic level, our specifications will be expressed as (deterministic) automata, listening to contract event triggers. States annotated with a cross denote that a violation has occurred. For instance, consider a smart contract which allows for its initiator to open the gambling table (`openTable`), on which other users may place a wager (`placeWager`), and then resolve it (`resolveWager`) any number of times. The table creator may close down a table (`closeTable`) as long as it has no unresolved wagers. The automaton shown in Fig. 1(a) ensures that a violation is identified if the table is closed when a wager has been placed but not resolved.

The automata used are, however, symbolic automata — in that they may use and manipulate variables. Transitions are annotated by a triple: `e | c ~> a`, where `e` is the event which will trigger the transition, `c` is a condition over the state of the contract (and additional contract property variables) determining whether the transition is to be taken, and finally `a` is an executable action (code) which will be executed if the transition is taken. Fig. 1(b) is a generalisation of the previous property, to handle the case when multiple wagers may be simultaneously placed and resolved on the same table. The actions typically impact just a number of variables local to the monitors (i.e. not the state of the system itself), although in some cases, however, specifically in the case of a property violation, one may choose to change the system state in order to make up for violated invariants.

[1] `http://www.cs.um.edu.mt/svrg/Tools/LARVA`

3 Tool Syntax

The tool contractLarva allows the specification of properties using DEAs over method calls and setting of global variables in a smart contract written in Solidity. In addition, the tool allows for the specification of what happens when a specification is violated.

3.1 Properties

Each property in a contractLarva specification file is written in the form of a DEA as described in the previous section. The top level structure of a DEA is as follows:

```
DEA ⟨PropertyName⟩ {
  states {
    ...
  }
  transitions {
    ...
  }
}
```

The property name is simply an identifier to be able to refer to the DEA by name. States are declared in the `states` block of the script, and are written as a semi-colon terminated sequence. Initial, bad and accepting states can be tagged by following the state name by a colon and the type of the state. There can only be one initial state, but multiple bad and accepting states are possible. For example, the states of a four state DEA can declared as follows:

```
states {
  TableOpen: initial;
  TableClosed: accept;
  CasinoClosed: accept;
  BetPlaced;
  PotReduced: bad;
}
```

Transitions are also written as a semicolon terminated sequence, where each transitions is written as: `src -[label]-> dst`, where (i) `src` is the name of the source state of the transition, (ii) `dst` is the destination state, and (iii) `label` is the guarded event on the transition.

The syntax for labels is as follows: `event | guard ~> action`, where (i) `event` is the trigger of the transition, (ii) `guard` is a Solidity expression which will be evaluated to decide whether or not to take the transition; and (iii) `action` is a Solidity statement (or block) which will be executed if the transition is taken. Both the condition and action can be left out, in which case, one can also leave out the separators: for instance `e | c ~>` can be written as `e | c` while `e | ~> a` can be written as `e ~> a` and `e | ~>` simply as `e`.

There are two types of event triggers which can be used in contractLarva:

(i) *Control-flow triggers* which trigger when a function is called or returns a value: (a) `before(function)`, triggers whenever `function` is called and before any of its code is executed[2]; and (b) `after(function)`, triggers the moment `function` terminates successfully (i.e. not reverted). In both cases, if you want to read the value of the parameters, they can be specified in the event e.g. `before(payAmount(value))` can be written to have a condition dependent on the parameter. Parameters may be left out altogether if you do not wish to access them e.g. `before(payAmount)`.

(ii) *Data-flow triggers*, trigger when an assignment on a global variable occurs (even if the value of the variable does not change) — `var@(condition)` triggers whenever variable `var` is assigned to (just after the assignment is performed), with the `condition` holding. The condition is optional, in which case any assignment to the variable will trigger the event. Furthermore, the condition in variable assignment triggers can refer the value of variable `var` before the assignment using the syntax `LARVA_previous_var` e.g. to trigger when a variable `x` is assigned to a larger value, one would use the event `x@(x > LARVA_previous_x)`.

[2]The checks and the code are instrumented as a modifier triggering before the function body is executed.

The following is an example of a complete DEA property, using a global variable `total` to keep count of the running total of bets (we will see how to declare such variables in the script in the next section):

```
DEA NoReduction {
  states {
    TableOpen: initial;
    TableClosed: accept;
    BetPlaced;
    PotReduced: bad;
  }
  transitions {
    TableOpen -[after(closeTable) | pot == 0 ]-> TableClosed;
    TableOpen
      -[after(placeBet(_value)) | _value <= pot ~> total += _value;]->
        BetPlaced;
    BetPlaced -[after(timeoutBet)]-> TableOpen;
    BetPlaced -[after(resolveBet)]-> TableOpen;
    BetPlaced -[pot@(LARVA_previous_pot > pot)]-> PotReduced;
  }
}
```

3.2 Script Structure

At the top level, a script is a specification of a number of monitors, each of which is to be applied to a named smart contract. Each monitor specification looks as follows:

```
monitor ⟨ContractName⟩ {
  declarations {
    ...
  }

  initialisation {
    ...
  }

  reparation {
    ...
  }

  satisfaction {
    ...
  }

  // DAE properties are written below
  ...
}
```

⟨*ContractName*⟩ should be replaced by the name of the contract to be monitored, exactly as it appears in the Solidity code. You may not have more than one monitor block referring to the same contract in a specification file. It is also worth mentioning at this stage, that CONTRACTLARVA will create a new smart contract with a different name (a contract named `Abc` will be replaced by a new one named `LARVA_Abc`).

Within a monitor specification, one can include the following parts in order (although they are all optional):

- Declarations: Within the `declarations` block one can include any declarations (functions, modifiers, variables, etc.) written in Solidity, which are to be added to the original smart contract in order to perform the functionality required. For instance, any variables which are required to keep track of additional state which is manipulated by the condition and actions of the DEA transitions should be declared here.

- Initialisation: Within the `initialisation` block one is to add Solidity code that will make up the constructor of the instrumented smart contract. If left out (and the declarations block does not contain any special initialising logic, as discussed further ahead), the constructor will simply contain a call to the original constructor of the contract.
- Reparation: The `reparation` block should include code which is to be executed as soon as any property which the monitor is checking is violated (reaches a bad state). Since sometimes one may prefer to have code handling reparation in the actions on transitions leading into a bad state, this block may be left out.
- Satisfaction: Similarly, the `satisfaction` is to contain code which is executed when all the DEAs in the specification reach an accepting state.

After these declarations, the monitor block can then have any number of DEAs which will be monitored for that contract.

contractLarva automatically provides functionality to be able to disable and enable the original contract functionality. By default, the functionality of the original smart contract starts off disabled i.e. all calls to functions in the original smart contract are reverted (via a `require`). However, the functions `LARVA_EnableContract()` and `LARVA_DisableContract()` can be used to enable and disable it respectively. It is worth noting that even if enabled, the functionality of the original contract is blocked until the original constructor is called.

It is also important to note that `LARVA_DisableContract()` does not stop a function call already in progress. For example, if you just disable the contract upon identifying a violation triggered by an event labelled by `before(function)`, the call to the function will continue normally. If this is not the desired behaviour, one must ensure to `revert` execution through the action on the transition or in the `reparation` function. Also note that if an event causes multiple DAEs to move into a bad state, each one will trigger a call to the reparation call even if `LARVA_DisableContract()` is called.

An important aspect of the contractLarva instrumentation is how it deals with the contract initialisation. Consider that using the `LARVA_EnableContract()` and `LARVA_DisableContract()` one may want to introduce another layer of logic before the original contract is enabled, e.g. the payment of some ether for insurance purposes as further discussed in 3.3. This means that one may not wish for the contract to be initialised upon deployment. contractLarva deals with this in the following manner in the given cases: (i) if both the initialisation and declarations block do not call the `LARVA_EnableContract()` then the contract starts as enabled, with the original constructor if any; (ii) if only the initialisation block contains enabling logic then this code is prepended to the original constructor, if any; while (iii) if the declarations block also contains such logic the original constructor, if any, is transformed into a normal function (with the name ¡contract-name¿Constructor) that must be called to enable the original contract. The third case is more expensive than the others, since it requires the original constructor code to be saved to the blockchain, while in the other cases the required initialisation code is simply part of the constructor (which executed during deployment but not deployed).

3.3 Use Case: Stake-based Verification

Consider a smart contract which allows for a service to be provided by the creator of the contract, to which users may, one at a time, access against a payment. Assume that we want to add a layer of protection over the smart contract, so that the the service provider guarantees that if any property is violated, the current user of the service gets 1 ether compensation — this stake is kept as an escrow in the contract, and has to be paid by the creator of the contract before anything else happens.

The structure of the specification will look as follows:

```
monitor Service {

  ⋮

  DEA {
    ...
  }

  DEA {
    ...
```

```
  }
}
```

The properties which are to be monitored are added in as DEAs in the specification. The stake handling mechanism is then added in the first part of the monitor script. We start by adding some functions and variables to the declaration block of the specification to be able to be able to get information about who is providing insurance for whom and by how much:

```
declaration {
  address private insurer_address;
  function getStake() private { return (1 ether); }
  function getInsurer() private { return insurer_address; }
  function getInsured() private { return current_user_address; }

  ⋮
}
```

The stake will be defined to be a constant of 1 ether, the insured party is taken from the variable `current_user_address`, which we will assume is managed by the original smart contract, and the insurer will always be the original creator of the smart contract, whose address we will store in a variable `insurer_address`. The initialisation block ensures this takes place:

```
initialisation {
  insurer_address = msg.sender;
}
```

Returning to the the declaration block, we will also add code to keep enabling paying of the stake and enabling the original contract:

```
declaration {
  ⋮

  enum STAKE_STATUS { UNPAID, PAID }
  STAKE_STATUS private stake_status = UNPAID;

  function payStake() payable {
    require (stake_status == UNPAID);
    require (msg.value == getStake());
    require (msg.sender == getInsurer());

    stake_status = PAID;
    LARVA_EnableContract();
  }
}
```

We can now handle specification (i) satisfaction (when the properties may no longer be violated), in which case we simply return the stake to the insurer; and (ii) violation, in which case the stake is paid to the insured party and the original smart contract is disabled:

```
satisfaction {
  getInsurer().transfer(getStake());
}

reparation {
  getInsured().transfer(getStake());
  LARVA_DisableContract();
}
```

Needless to say, this simple instrumentation leaves much to be desired: (i) it gives no way for the insurer to close down the smart contract and recover the stake; (ii) upon violation, the contract is locked and any resources locked within it can no longer be accessed. Such features can, however, easily be added onto the reparation strategy and incorporated into the specification.

4 Tool Usage

The tool can be compiled using any recent version (version 8.2.1 at the time of writing) of the Glorious Glasgow Haskell Compilation System[3] (GHC). The tool can then be invoked at the command line as:

`contractlarva` ⟨*DAE file*⟩ ⟨*input Solidity file*⟩ ⟨*output Solidity file*⟩

The output file created will have the same functionality as the input one, but augmented as specified in the DAE file.

5 Missing Features

The following are a number of restrictions and limitations in contractLarva v0.2α, but which we plan to add in the future:

Monitorable triggers: Currently, contractLarva only supports triggers on (named) function calls and variable value updates. Smart contracts provide various other points-of-interest one can (and perhaps should) hook onto, including asset transfers (e.g. via `send` or `transfer`), other non-user defined functions (e.g. `selfdestruct`), fallback function invocations, reverted transactions, etc. The plan is to add the capability to observe these (and other) triggers in future versions of contractLarva.

Variable assignment triggers: Events firing on variable assignments are currently limited to global variables, and cannot be arrays, array elements, structures or structure fields. Furthermore, the variable name may not be reused as a local variable or a function parameter name (since assignments to these would also be wrongly changed). Finally note that initialisation of that variable (as part of its declaration) does not trigger DAE transitions.

Function return value: contractLarva currently provides no means of referring to function return values in guards and actions in an `after(function)` event.

Structured reparation control: Further options to control how reparation happens (e.g. whether the global reparation code is executed once or upon every violation, whether execution continues, etc.), and a library of reparation schemata to avoid having to program them from scratch are to be added to the tool.

Initialisation parameters: The initialisation code (which is effectively the body of the constructor of the new contract created) is currently restricted to be a parameterless public function.

Specifications across contracts: contractLarva currently can only specify properties over a single contract. DAEs spanning over events across different contracts are currently not supported.

Parametrised properties: The current version of the tool does not support parametrised properties i.e. DAEs quantified over different slices of the contract behaviour e.g. one cannot have a property which is instantiated per user address.

6 Version History

0.1α released 29 November 2017 (First release)

0.2α released 5 December 2017

- Added parameters to function calls.

[3] GHC is available from `https://www.haskell.org/ghc`. To compile contractLarva, just execute at the command line: `ghc -o contractlarva Main`.

v0.2.1α released 22 August 2019

- Added full support for v0.4.* Solidity.

v0.2.2α released 1 September 2019

- Added full support for <=v0.5.11 Solidity.