Model-checking of Solidity Smart Contracts (1)

Sebti Mouelhi
August, 2021

Here are the notes on the verification of Solidity smart contracts. Section 1 is dedicated for the smart contracts modeling and verification problems. Section 2 is dedicated for some $\mathbb{K}$ framework exercises used as a proof assistant for the verification of Ethereum smart contracts.

1 Smart contracts

The manuscript's logical structure is gradual. First, raw implementations of the different functions and pseudo-codes are presented and guaranteed to be Solidity-compilable within a well-formed smart contract with justifiable coding choices faithful to specifications. Second, each smart contract is step-by-step extended by the required correctness/safety properties. All the contracts presented in the document are tested under the Solidity v0.7.6 compiler `solc`.

Smart contracts were verified using the source-level verification tool SOLC-VERIFY [1] (Github link) with its Microsoft Boogie back-end. The choice was quickly made after testing the latest version of the `solc`-builtin SMTChecker. After having tested a few examples, it seems that the CHC-proof (*Constrained Horn Clauses*) engine is a bit slow, sensitive to overflow properties, and has no advanced counter-examples display. The concern was to save time and avoid going in circle tests. In addition, the SMTChecker's expression power is poor: verifying blockchain/object-oriented code using a powerful Hoare-based formal annotations (function entry-point pre/post-conditions) and global (local loop-)invariants with quantified formulas is more interesting and comfortable. Incidentally, the usual `assume`/`assert` `solc`-statements are perfectly well-supported by the tool.

1.1 Basic verification

Here is the `payAll` raw function included in a working contract `Token`.

```
// SPDX-License-Identifier: UNLICENSED
pragma solidity >=0.7.0;
contract Token {
    mapping(address => uint) balances;
    // keyword memory added to parameter declaration of "address[] to"
    function payAll(address from, address[] memory to, uint val) public {
        uint amount = val * to.length; // total -> amount; overflow
        require(balances[from] >= amount);
        balances[from] -= amount;
        // int -> uint
        for (uint i = 0; i < to.length; i++){
            balances[to[i]] += val; // overflow
        }
    }
}
```

```
# solc-verify 1_token.sol --arithmetic mod-overflow
Token::payAll: ERROR
 - 1_token.sol:11:9: Invariant 'No overflow' might not hold on loop entry
 - 1_token.sol:11:9: Invariant 'No overflow' might not be maintained by the loop
Token::[implicit_constructor]: OK
```

Overflows

Obvious properties to be verified are those ensuring the **absence of overflows**. In `payAll`, `uint` variables are 256-bit unsigned, overflows might happen when using raw arithmetic operations.

SOLC-VERIFY only displays inner possible overflows within the function's statements without saying that the whole function may terminate with overflow. That's why only the loop error overflow messages are displayed in the output shown above. For the sake of argument, if the loop is commented a global error message is displayed saying: `1_token.sol:6:5: Function can terminate with overflow`.

The solution is to replace the builtin operators by overflow-free arithmetic functions, that could be defined in a separate library `SafeIntMath`.

```
// SPDX-License-Identifier: UNLICENSED
pragma solidity >=0.7.0;

contract Token {
    using SafeIntMath for uint;

    mapping(address => uint) balances;

    function payAll(address from, address[] memory to, uint val) public {
        uint amount = to.length.mul(val); // ok

        require(balances[from] >= amount);

        balances[from] = balances[from].sub(amount); // ok
        for (uint i = 0; i < to.length; i++) {
            balances[to[i]] = balances[to[i]].add(val); // ok
        }
    }
}
library SafeIntMath {
    function mul(uint256 a, uint256 b) internal pure returns (uint256) {
        uint256 c = a * b;
        require(a == 0 || c / a == b);
        return c;
    }
    function div(uint256 a, uint256 b) internal pure returns (uint256) {
        // assert(b > 0); // implicit
        uint256 c = a / b;
        // assert(a == b * c + a % b); // always holds
        return c;
    }
    function sub(uint256 a, uint256 b) internal pure returns (uint256) {
        require(b <= a);
        return a - b;
    }
    function add(uint256 a, uint256 b) internal pure returns (uint256) {
        uint256 c = a + b;
        require(c >= a);
        return c;
    }
}
```

```
# solc-verify 1_token.sol --arithmetic mod-overflow
Token::payAll: OK
Token::[implicit_constructor]: OK
No errors found.
```

Two properties could be defined as pre/post-conditions to check the correctness of the `payAll` function. The first ones target the `from` balance and the others are for the `to[i]` (`!= from`) balances. They could be defined by the following obvious (pseudo) post-conditions:

```
balances[from] == __old(balances[from]) - val * to.length && balances[from] >= 0
forall i; balances[to[i]] == __old(balances[to[i]]) + val
```

under some preconditions:

```
forall i; from != to[i]
forall i,j; i != j => to[i] != to[j]
forall i; 0 <= balances[to[i]] && balances[to[i]] <= max_val
```

They should be defined resp. as `/// @notice postcondition` and `/// @notice precondition` annotations (see the code below). `__old` (`__verifier_old_uint` for short) is a special function used by the model-checker to represent the state variable value at the function entry-point (before the call).

Loop unrolling

The proof was performed by enabling loop unrolling. The SOLC-VERIFY main script was slightly modified to include the switch `/loopUnroll:n` in the Boogie command. **The `n` is the depth applied, `n=2` was sufficient to prove the post-conditions** (following up to `n` back-edges, then some).

```
uint constant max_val = 2**128 - 1;

/// @notice precondition forall (uint i) !(i < to.length) || from != to[i]
/// @notice precondition forall (uint i, uint j) !(i < to.length && j < to.length && i != j) || to[i] != to[j]
/// @notice precondition forall (uint i) !(i < to.length) || balances[to[i]] <= max_val
/// @notice postcondition balances[from]  == __verifier_old_uint(balances[from]) - val * to.length && balances[from] >= 0
/// @notice postcondition forall (uint i) !(i < to.length) || balances[to[i]] == __verifier_old_uint(balances[to[i]]) + val
function payAll(address from, address[] memory to, uint val) public {
    uint amount = to.length.mul(val);
    require(balances[from] >= amount);
    balances[from] = balances[from].sub(amount);
    // this is a loop invariant (without effect on the proof)
    /// @notice invariant 0 <= i && i <= to.length
    for (uint i = 0; i < to.length; i++) {
        balances[to[i]] = balances[to[i]].add(val);
    }
}
```

The assertion (suffixed by `/// @notice invariant`) before the `for` loop is without effect but it shows how loop invariants could be defined. The proof command used to model-check the function is the following:

```
# solc-verify 1_token.sol --arithmetic mod-overflow --unroll 1 --verbose --output .
```

The switch `--output .` is to generate the `.bpl` Boogie input file. Notice that Z3 is the default SMT solver used by SOLC-VERIFY.

An extra global invariant `total_stacked == __sum(balances)` could be defined. For example, the sum of owners' balances is always equal to the total staked amount (if a the `payAll` loop invariant are properly defined).

```
/// @notice invariant total_staked == __verifier_sum_uint(balances)
contract Token {
    using SafeIntMath for uint;

    uint total_staked;
    mapping(address => uint) balances;

    constructor() public {
        total_staked = 100000;
        balances[msg.sender] = total;
    }

    //post-conditions ...
    function payAll(address[] memory to, uint val) public {
        uint amount = to.length.mul(val);

        require(balances[msg.sender] >= amount);
        balances[msg.sender] = balances[msg.sender].sub(amount);
        /// @notice invariant total_staked == __verifier_sum_uint(balances) + (to.length - i) * val
        /// @notice invariant 0 <= i && i <= to.length
        for (uint i = 0; i < to.length; i++) {
            balances[to[i]] = balances[to[i]].add(val);
        }
    }
}
```

Before, presenting the correctness proof of `transfert`, It is worth to mention how exceptions, like `revert()` and `require()`, are handled by SOLC-VERIFY. For that, let's first recall how thrown exceptions are handled by Solidity. A transaction contains the function to be called with arguments, and an execution fee called *gas*. Optionally, some value of Ether (the native currency of Ethereum) can also be transferred with transactions. The Ethereum network then executes the transaction by running the contract code in the context of the contract instance. During their execution, each instruction costs some predefined amount of gas. **If the contract overspends its gas limit, or there is a runtime error** (e.g., an exception is thrown, or an assertion fails), then **the entire transaction is aborted with no effect on the ledger** (apart from charging the sender the used gas) [1].

This exception/error property should be somewhere handled by the proof engine in order to ensure correctness of smart contracts specifications.

✍ Correctness

SOLC-VERIFY targets contracts functional correctness with respect to **completed transactions and different types of failures**. An **expected failure** is due to an exception deliberately thrown to guard from the user (e.g., `require()`, `revert()`). An unexpected failure is any other failure (e.g., `assert()`, overflow). A **contract is correct if all transactions (public function calls) that do not fail due to an expected failure also do not fail due to an unexpected failure and satisfy their specification.**

Error handling

Solidity **exceptions undo all changes made to the global state by the current call**. Deliberately **thrown exceptions** (`require()`, `revert()` and `throw()`) are therefore **mapped to `false` assumptions in Boogie**, which **stop the verifier without reporting an error**. Solidity assertions `assert()` are mapped to Boogie assertions, causing a reported error when their conditions are not true.

The proof of the `transfer` function is straightforward. It could be defined directly in the same `Token` contract as follows.

```
/// @notice precondition from != to
/// @notice precondition val <= balances[from]
/// @notice precondition 0 <= balances[to]
/// @notice postcondition balances[from]  == __verifier_old_uint(balances[from]) - val
/// @notice postcondition balances[to]  == __verifier_old_uint(balances[to]) + val
function transfer(address from, address to, uint val) public {
    uint updatedFrom;
    uint updatedTo;
    if (balances[from] >= val) {
        updatedFrom = balances[from].sub(val);
        updatedTo = balances[to].add(val);
    }
    else {
        revert();
    }
    balances[from] = updatedFrom;
    balances[to] = updatedTo;
}
```

Let's see now how throwing `revert()` in this examples is handled. For example, the function's post-condition on `balances[from]` above should have been formulated otherwise to **ensure total correctness** as follows:

```
__old(balances[from]) >= val ==> balances[from] == __old(balances[from]) - val
__old(balances[from]) < val ==> balances[from] == __old(balances[from])
```

The second condition guarded by $rev \triangleq$ `__old(balances[from]) < val` will always hold even in the case a false condition is putted after the implication sign `=>` since `revert()` is thrown when rev holds.

```
if (balances[from] >= val) {
    ...
}
else {
    revert();
}
```

In the Boogie intermediate `.bpl` file, the above condition control flow is translated as follows:

```
if ((balances#15[__this][from#86] >= val#90)) { // assumptions & updates & guarantees }
else { assume false; }
```

According to this translation mechanism, whatever is the assertion q, $rev \Rightarrow q$ holds since `false` $\Rightarrow$ `anything`.

Due to *gas* and runtime errors exception handling, total and partial correctness are equivalent. According to [1], the authors currently do not model gas: running out of gas does not affect correctness as the transaction is reverted. However, they might model it in the future in order to verify liveness and more advanced properties.

1.2 Transmuter model

Let's recall the main features of the naif Transmuter pegging model of stable xUSDs:

- users deposit xUSD into the Transmuter;
- each day, a certain amount of USD comes into the Transmuter (via the `exchange()` function) that credits users USD proportional to the amount they have staked;
- when a user chooses to claim the converted USD, an equal amount of xUSD will be burned;
- users can freely deposit more xUSD and withdraw the unconverted xUSD as well.

Currency datatype

In the pseudo-code, the xUSD currency amounts are represented by real values (the Solidity non-supported type `number`). Using floating point real numbers is warranted in finance. But, in this practical context, the choice to use real numbers instead of integers is motivated by the usage of the `fraction` and `delta` real rates applied by the `exchange()` function.

Given that fixed/floating point numbers are not yet fully (only syntax is) supported by (the latest version of) Solidity and SOLC-VERIFY, a possible simple solution is to use `uint`-percentage rate computations to automate the proof (see link for more details). We introduce for that the following `SafeIntMath` functions:

```
uint256 constant max_v = 2 ** 128 - 1;
uint256 constant min_pcent = 1;

function pcent(uint256 a, uint256 b) internal pure returns (uint256) {
    require(a <= max_v && b <= max_v);
    uint256 at100 = a * 100; require(a == 0 || at100 / a == 100);
    uint256 p = (at100 + b / 2) / b;
    if (p == 0 && a > 0 && b > 0) { // if a and b are + and p is null, then p = min_pcent
        p = min_pcent;
    }
    return p;
}
function pcentv(uint256 p, uint256 b) internal pure returns (uint256) {
    uint256 v = p * b;
    require(p == 0 || v / p == b);
    return (v / 100);
}
```

Mapping iteration

Mapping hash tables (`mapping` datatype) are not yet iterable in Solidity. An intermediate dynamic array could be used to iterate over the owner addresses of the mapping `accs` (`accounts`). An extra maps could be used to check the existence of a mapped owner as well.

```
struct Account {uint unex; uint exch;}
address[] owns; // dynamic array of owners
mapping(address => Account) accs;
mapping(address => bool) exist; // to check address existance
uint tb; // total_buffered

constructor() {
    if (!exist[msg.sender]) {
        exist[msg.sender] = true;
        accs[msg.sender].unex = 0; accs[msg.sender].exch = 0;
        owns.push(msg.sender);
    }
    tb = 0;
}
```

Proofs

The proof was conducted by first validating the different functions by pre/post-condition, and then the whole smart contract by safety invariants.

```
/// @notice postcondition accs[own].unex == __verifier_old_uint(accs[own].unex) + a
function deposit(uint a, address own) public {
    accs[own].unex = accs[own].unex.add(a);
}
/// @notice postcondition accs[own].unex == __verifier_old_uint(accs[own].unex) - a
function withdraw(uint a, address own) public {
    require(a <= max_v && accs[own].unex >= a);
    accs[own].unex = accs[own].unex.sub(a);
}
/// @notice postcondition accs[own].exch == __verifier_old_uint(accs[own].exch) - a
function claim(uint a, address own) public {
    require(a <= max_v && accs[own].exch >= a);
    accs[own].exch = accs[own].exch.sub(a);
}
/// @notice postcondition !(accs[usr].unex > 0) || s == accs[usr].unex + accs[usr].exch
/// @notice postcondition !(accs[usr].unex == 0) || s  == 0
function userStaked(address usr) public view returns (uint s) {
    s = 0;
    if (accs[usr].unex > 0) {
        s = accs[usr].unex.add(accs[usr].exch);
    }
    return s;
}
```

The function `totalStaked` as it was defined in the specification could not directly validated face to its usual post-conditions even with loop unrolling.

```
/// @notice postcondition exists (uint i) !(i < owns.length && accs[owns[i]].unex > 0) || t > 0
/// @notice postcondition forall (uint i) !(i < owns.length && accs[owns[i]].unex > 0) ||
///   new lines are not allowed --->            accs[owns[i]].unex + accs[owns[i]].exch <= t
function totalStaked() public view returns (uint t) {
    t = 0;

    for (uint i = 0 ; i < owns.length; i++) {
        t = t.add(userStaked(owns[i]));
    }
    return t;
}
```

The solution to validate them was simply i) to use an intermediate variable `s_at_i` to store the current value `userStaked(owns[i])`, and ii) to apply a loop unroll with back-edge depth equal to 2.

```
function totalStaked() public view returns (uint t) {
    t = 0;

    uint s_at_i = userStaked(owns[0]); // intermediate variable

    for (uint i = 0 ; i < owns.length; i++) {
        t = t.add(s_at_i);
        if (i < owns.length - 1){
            s_at_i = userStaked(owns[i+1]);
        }
    }
    return t;
}
```

The `exchange()` function was validated without applying the same logic as `totalStaked()` since there is no accumulators to be computed. The following usual post-conditions were defined for correctness (see the code below).

```
forall i; __old(accs[owns[i]].exch) <= accs[owns[i]].exch
forall i; __old(accs[owns[i]].unex) >= accs[owns[i]].unex
```

```
/// @notice postcondition forall (uint i) !(i < owns.length) ||
///                       (__verifier_old_uint(accs[owns[i]].exch) <= accs[owns[i]].exch)
/// @notice postcondition forall (uint i) !(i < owns.length) ||
///                       (__verifier_old_uint(accs[owns[i]].unex) >= accs[owns[i]].unex)
function exchange(uint a) public {
    tb = tb.add(a);
    ts = totalStaked();
    if (0 < tb && 0 < ts){
        uint pc = tb.pcent(ts); // uint percentage applied
        tb = 0;
        for (uint i = 0 ; i < owns.length; i++) {
            uint dlt = pc.pcentv(userStaked(owns[i]));
            if (accs[owns[i]].unex >= dlt) {
                accs[owns[i]].unex = accs[owns[i]].unex.sub(dlt);
                accs[owns[i]].exch = accs[owns[i]].exch.add(dlt);
            }
            else {
                tb = tb.add(dlt.sub(accs[owns[i]].unex));
                accs[owns[i]].exch = accs[owns[i]].exch.add(accs[owns[i]].unex);
                accs[owns[i]].unex = 0;
            }
        }
    }
}
```

Finally, some (non-exhaustive) invariants were defined in a special view function `invariants()`. Unfortunately, the special SOLC-VERFIER function `__verifier_sum_uint` could not be applied on mappings whose values are `struct`-composite. Due to this constraint, the main global `/// @notice invariant` safety properties couldn't be stated. Some other solutions were tested to cope with this obligation but they need a deep investigation to find out all the required loop invariants.

```
/// @notice invariant
/// forall (uint i, uint j)
///          !(i < owns.length && j < owns.length && i != j) || owns[i] != owns[j]
/// @notice invariant forall (uint i) !(i < owns.length) || exist[owns[i]]
function invariants() public view {
    uint tc = 0;
    uint ts = 0;
    uint t = 0;
    for (uint i = 0; i < owns.length; i++) {
        address at_i = owns[i];
        tc = tc.add(accs[at_i].exch);
    }
    for (uint i = 0 ; i < owns.length; i++) {
        address at_i = owns[i];
        uint s_at_i = accs[at_i].unex.add(accs[at_i].exch);
        t = t.add(s_at_i);
    }
    // (sum i; accs[owns[i]].exch)) <= (sum i; accs[owns[i]].unex + accs[owns[i]].exch)
    assert(tc <= t);

    for (uint i = 0 ; i < owns.length; i++) {
        uint s_at_i = userStaked(owns[i]);
        ts = curr_ts.add(s_at_i);
    }
    // totalStaked() <= (sum i; accs[owns[i]].unex + accs[owns[i]].exch)
    assert(ts <= t);
    /*
       the proof is not validated by directly assigning "ts" to totalStaked(),
       the solution was to inline its code directly in the body of invariants().
     */
}
```

1.3 Transmuter optimized model

The optimized version of Transmuter presented in the specification **is not equivalent to the naif version**. Here are some counter-examples/bugs and some fixes suggestions.

The `exchange()` body is never reached

```
// the default value of uninitialized variables is zero
...
number totalUnexchanged;
function exchange(number amount) {
    totalBuffered += amount;
    if (totalBuffered > 0 && totalUnexchanged > 0) {
        /*
           this conditioned body is never reached since totalUnexchanged
           is initialized to zero and never updated outside of it !
         */
        ...
            if (rate >= 1) {
                ...
                totalUnexchanged -= tick.unexchanged;
            }
    }
}
```

A possible fix would be to define a function `totalUnexchanged()` (as `totalStaked()` in the naif version) that sums the total amount of the ticks' unexchanged amounts.

The attribute `occupiedTick` of users accounts is initialized to 0

```
struct Account {
    number unexchanged;
    number exchanged;
    number occupiedTick; // this attribute is initialized to 0 in accounts
}
// the default value of uninitialized variables is zero
mapping(address => Account) accounts;
mapping(number => Tick) ticks;
number currentTick = 1; // ticks[0] is not used
...

function migrate(address owner) {
    // unauthorized access to tiks[0]
    ticks[accounts[owner].occupiedTick].unexchanged -= accounts[owner].unexchanged;
    accounts[owner].unexchanged -= newlyExchanged(owner);
    accounts[owner].exchanged += newlyExchanged(owner);
    accounts[owner].occupiedTick = currentTick;
    ticks[currentTick].unexchanged += accounts[owner].unexchanged;
}
function newlyExchanged(address owner) {
    // unauthorized access to tiks[0]
    if (ticks[accounts[owner].occupiedTick].unexchanged == 0) {
        return accounts[owner].unexchanged;
    } else {
        number rate = ticks[currentTick].rate - ticks[accounts[owner].occupiedTick].rate;
        return accounts[owner].unexchanged * rate;
    }
}
```

This bug could be fixed by initializing accounts and adding `require` guards as follows:

```
constructor() {
    accounts[msg.sender].unexchanged = 0;
    accounts[msg.sender].exchanged = 0;
    accounts[msg.sender].occupiedTick = 1;
}
function migrate(address owner) {
    require(accounts[owner].occupiedTick > 0);
    ...
}
function newlyExchanged(address owner) {
    require(accounts[owner].occupiedTick > 0);
    ...
}
```

At a given tick (day), only users who do transactions might have their balances updated

According to the naif specification, the users who do not execute transactions (functions `deposit()`, `withdraw()`, or `claim()`) could have their balances updated especially if they have sufficient deposits. This property is not fulfilled in the optimized specification: only the users who execute one of those functions could have their balances rate-updated by the function `migrate()`.

Ticks' rate

It seems also that the rate calculation is different form that applied in the naif specification.

2 $\mathbb{K}$ framework

Exercise 1

```
module SIMPLE-SPEC-SYNTAX
    imports CONTROL-FLOW-SYNTAX

    syntax Id ::= "$a" [token]
                | "$b" [token]
                | "$c" [token]
                | "$x" [token]
                | "$y" [token]
                | "$z" [token]
endmodule

module VERIFICATION
    imports SIMPLE-SPEC-SYNTAX
    imports CONTROL-FLOW

    rule maxInt(X, Y) => Y requires          X <Int Y [simplification]
    rule maxInt(X, Y) => X requires notBool X <Int Y [simplification]
endmodule

module SIMPLE-SPEC
    imports VERIFICATION

    claim <k> $a = A:Int ; $b = B:Int ;
              if (A < B) {
                $c = B ;
              } else {
                $c = A ;
              }
              => . ... </k>
          <mem> MEM => MEM [ $a <- A ] [ $b <- B ] [ $c <- ?C:Int ] </mem>
          requires A <=Int ?C andBool B <=Int ?C
endmodule
```

This specification simply states that once the program is fully executed (`=> .`), then i) the symbolic memory will be updated by writing `A` and `B` in the symbolic `Int`-sized spaces pointed resp. by the program variables `$a` and `$b`, and ii) an arbitrary (`?`) symbolic `Int`-sized value `?C` is assigned to `$c` such that `A <= ?C` and `B <= ?C` (`requires` clause) which is true according to the input program. The proof's output of this claim (without that defined at line 47) is the following.

```
understanding-k-framework/# kompile --backend haskell control-flow.k
understanding-k-framework/# kprove tests/control-flow/new-simple-spec.k
#Top
understanding-k-framework/# kprove tests/control-flow/new-simple-spec.k
```

which means that the claim is valid.

```
module SIMPLE-SPEC
    imports VERIFICATION

    rule maxInt(X, Y) => Y requires          X <Int Y [simplification]
    rule maxInt(X, Y) => X requires notBool X <Int Y [simplification]
    /*
      By the way, the above rules are already defined in compiled.txt by
      the following syntactic rule directly assigned to semantics, so
      they (above) could be removed (imported from domain.md).

      syntax Int ::= "maxInt" "(" Int "," Int ")" [function, functional, hook(INT.max),
                                                   smt-hook((ite (< #1 #2) #2 #1))]
    */
    claim <k> $a = A:Int ; $b = B:Int ;
              if (A < B) {
                $c = B ;
              } else {
                $c = A ;
              }
              => . ... </k>
          <mem> MEM => MEM [ $a <- A ] [ $b <- B ] [ $c <- maxInt(A, B) ] </mem>
endmodule
```

The proof output is the following:

```
#Not ( {
    MEM [ $a <- A:Int ] [ $b <- B:Int ] [ $c <- maxInt ( A , B ) ]
  #Equals
    MEM [ $a <- A:Int ] [ $b <- B:Int ] [ $c <- B:Int ]
  } )
#And
  <generatedTop>
    <k>
      _DotVar1
    </k>
    <mem>
      MEM [ $a <- A:Int ] [ $b <- B:Int ] [ $c <- B:Int ]
    </mem>
  </generatedTop>
#And
  {
    true
  #Equals
    A <Int B
  }
```

It means that `A < B` but `$c` is assigned to arbitrary value different from `B`. It seems that `maxInt(A,B)` cannot be evaluated in symbolic memory assignments. The following two possible solutions could be applied.

```
// maxInt is evaluated in a "require" clause
<mem> MEM => MEM [ $a <- A ] [ $b <- B ] [ $c <- ?C:Int ] </mem>
requires ?C ==Int maxInt (A, B)

//or

<mem>
  $a |-> A
  $b |-> B
  $c |-> (_ => maxInt(A, B))
</mem>
/*
  A context <mem> in which $c is bound to anonymous
  Int-sized symbolic variable _ existing somewhere in
  <mem>. The content of _ will be updated to the result
  of maxInt(A, B).
*/
```

Exercise 2

```
claim <k> while ( 0 < $n ) {
            $s = $s + $n ;
            $n = $n - 1 ;
          }
          => . ... </k>
          <mem>
             $s |-> (S:Int => S +Int ((N +Int 1) *Int N) /Int 2)
             $n |-> (N:Int => 0)
          </mem>
```

This claim states the property of the first n natural numbers summation: $\sum_{i=0}^{n} = \frac{n(n+1)}{2} =$ `sum_n`. According to the loop, `$s = $s + sum_n`. The output without the `requires` loop invariant is the following:

```
#Not ( {
    N
  #Equals
    0
  } )
#And
  <generatedTop>
    <k>
      _DotVar1
    </k>
    <mem>
      $n |-> N:Int
      $s |-> S:Int
    </mem>
  </generatedTop>
#And
  {
    false
  #Equals
    0 <Int N
  }
```

The proof is in a false situation where `$n` $\neq$ `0` and `0 > n` violating the summation property. The loop invariant ensures that this erroneous state is unreachable. It states that `0 <= n`.

```
<mem>
   $s |-> (S:Int => S +Int ((N +Int 1) *Int N) /Int 2)
   $n |-> (N:Int => 0)
</mem>
requires N >=Int 0
```

References

[1] Á. Hajdu and D. Jovanović. SOLC-VERIFY: A modular verifier for solidity smart contracts. In *Verified Software. Theories, Tools, and Experiments*, pages 161–179. Springer International Publishing, 2020.