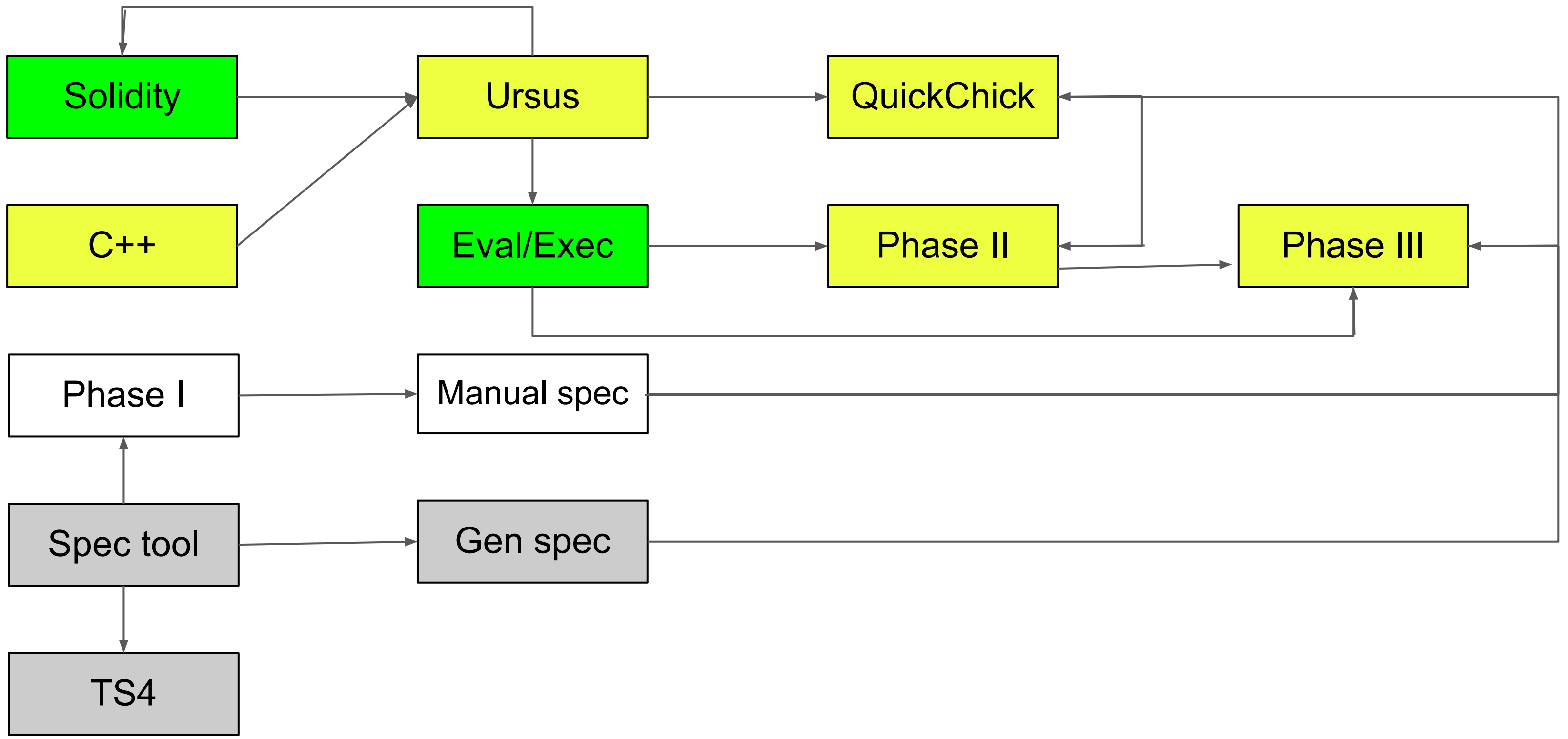
Ursus achievements for Q2CY22

Prepared by Pruvendo
06/09/22

Ursus designation

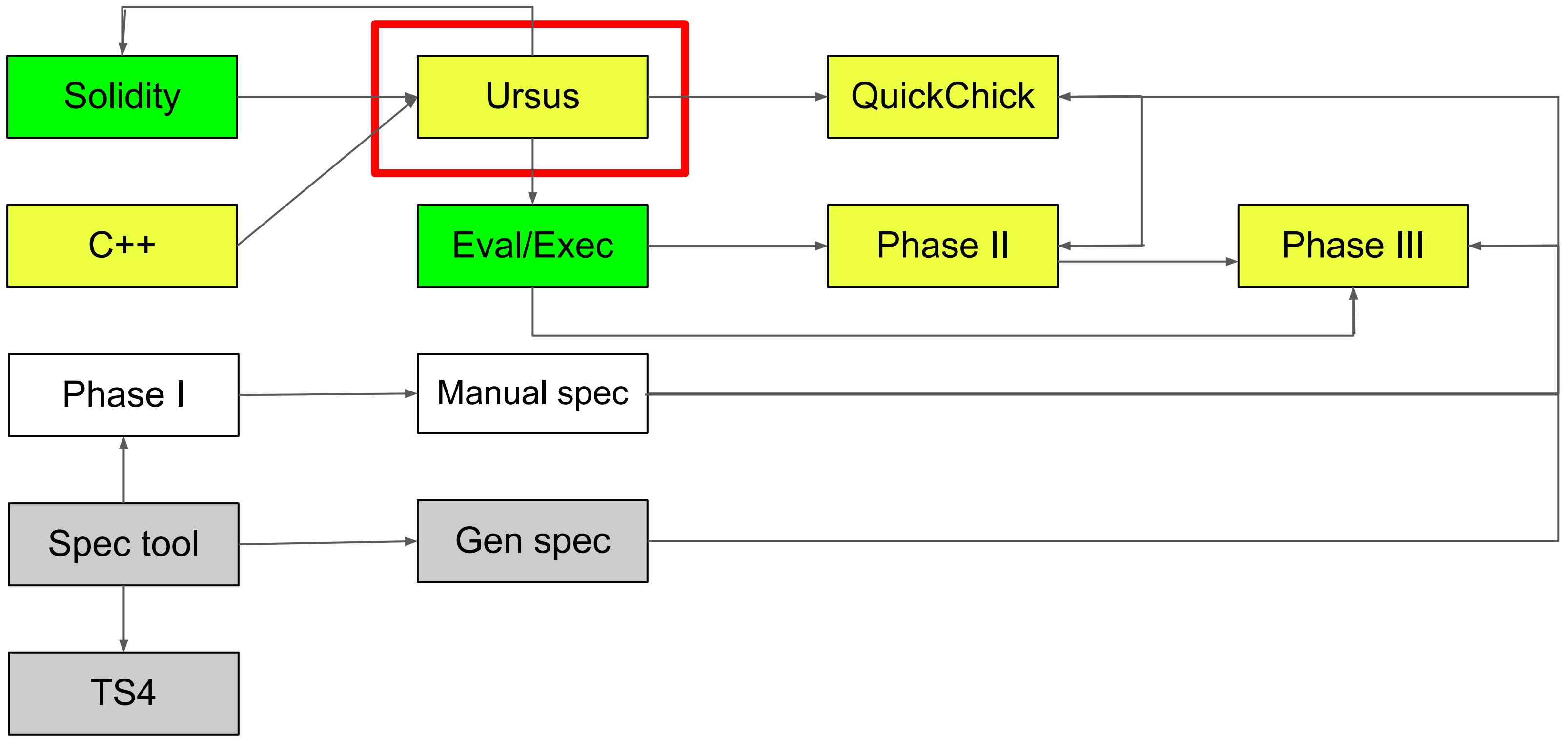
- Full-stack ecosystem for Everscale smart-contract verification
- Based on Coq custom grammar support
- Supports Solidity and C++
- Can be used as an independent language for smart-contract development
- Supports fast proofs with QuickChick
- Full fledged deductive verification in Coq

High-level architecture



Ursus language

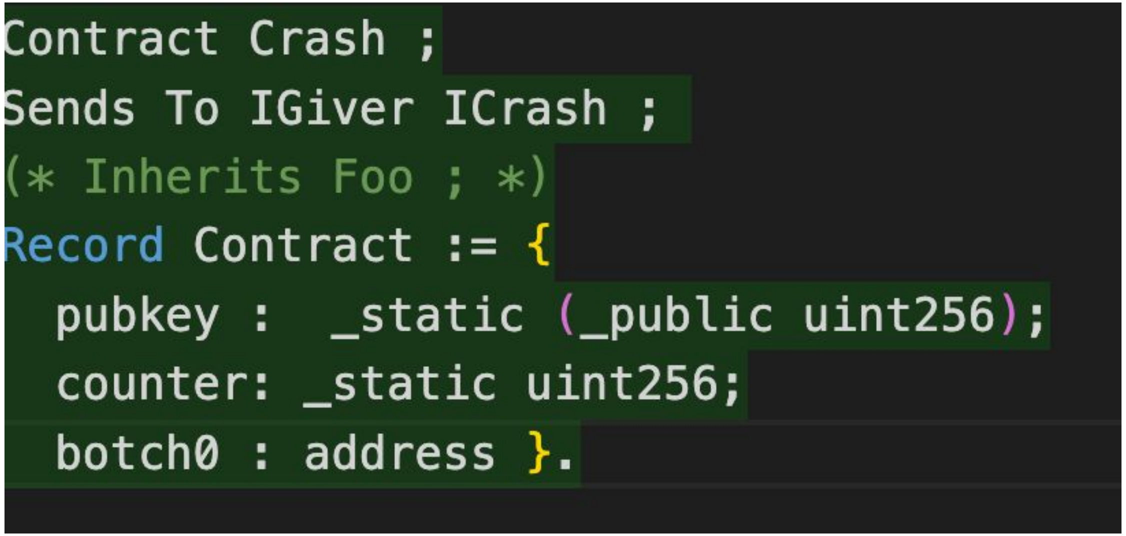
High-level architecture



Brief language description

- Ursus - embedded DSL language over Gallina, the declarative specification and programming language for Coq
- **Can be used to model imperative behavior (control flow) in a declarative environment**
- Has syntax similar to Solidity (with some modifications)

Language enhancements

- By the extension of the Vernacular (command) language in Coq
- Supports custom commands to automatically generate smart contract specific environment
- 

```
Contract Crash ;
Sends To IGiver ICrash ;
(* Inherits Foo ; *)
Record Contract := {
  pubkey :  _static (_public uint256);
  counter: _static uint256;
  botch0 : address }.
```

Standard library

- Standard library implements all the standard Solidity functions (refer API.md)
- The standard library is almost fully implemented
- Implemented parts are formally specified
- … and proven
- **The intention is to formally prove the specification of the whole standard library (with some simplifications)**

Standard library content

- Primitive types
- Maps
- Queues
- Vectors (finishing)
- Cells, Builders, Slices (almost done)
- Standard functions and TVM specifics

Operations on ℕ and ℤ

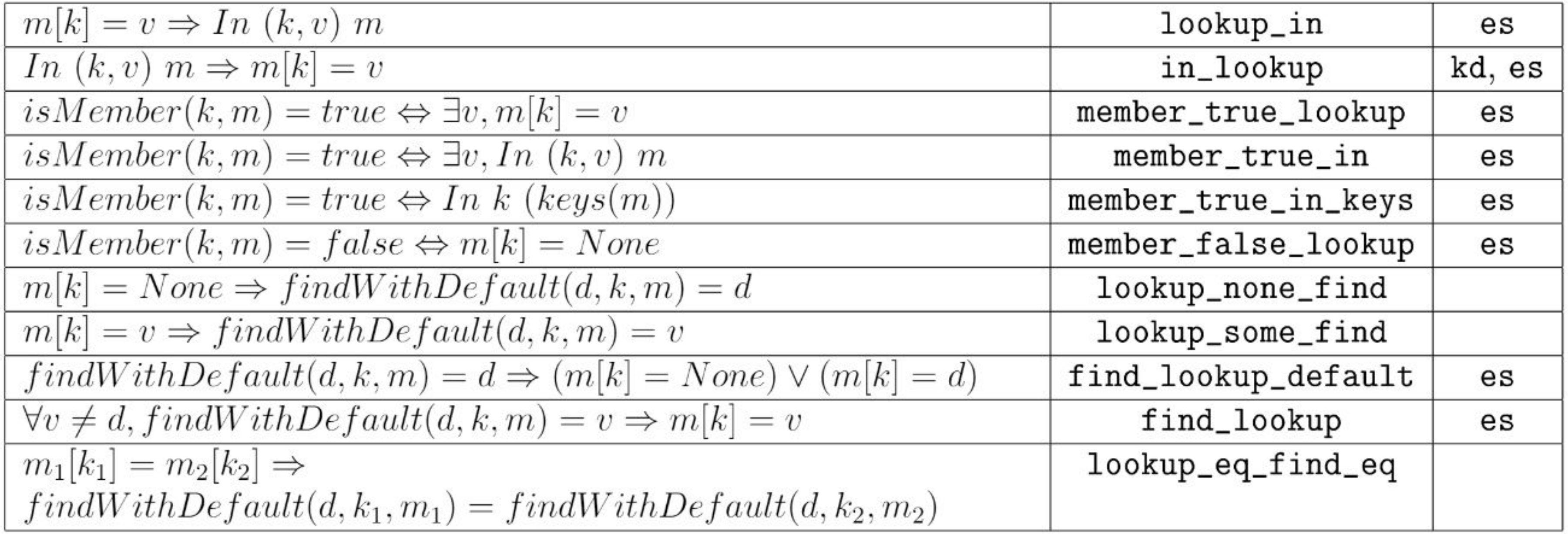
- Defined as functions from Coq standard library
- Coq tactic `lia` works in most cases
- However, division, `modulo` and bitwise operations still require manual care (need to be automated in future)
- The specification for these operations is partially ready
- … as well as proofs (based on standard library and in-house solutions)

Maps

- Looking up elements
- keysDistinct
- insert
- replace
- add
- delete, deleteMin, deleteMax
- min, max
- next, nextOrEq, prev, prevOrEq

Fully done - specified, implemented, proved

Example of maps (specification)



$m[k] = v \Rightarrow In\ (k, v)\ m$	`lookup_in`	es
$In\ (k, v)\ m \Rightarrow m[k] = v$	`in_lookup`	kd, es
$isMember(k, m) = true \Leftrightarrow \exists v, m[k] = v$	`member_true_lookup`	es
$isMember(k, m) = true \Leftrightarrow \exists v, In\ (k, v)\ m$	`member_true_in`	es
$isMember(k, m) = true \Leftrightarrow In\ k\ (keys(m))$	`member_true_in_keys`	es
$isMember(k, m) = false \Leftrightarrow m[k] = None$	`member_false_lookup`	es
$m[k] = None \Rightarrow findWithDefault(d, k, m) = d$	`lookup_none_find`	
$m[k] = v \Rightarrow findWithDefault(d, k, m) = v$	`lookup_some_find`	
$findWithDefault(d, k, m) = d \Rightarrow (m[k] = None) \vee (m[k] = d)$	`find_lookup_default`	es
$\forall v \neq d, findWithDefault(d, k, m) = v \Rightarrow m[k] = v$	`find_lookup`	es
$m_1[k_1] = m_2[k_2] \Rightarrow$ $findWithDefault(d, k_1, m_1) = findWithDefault(d, k_2, m_2)$	`lookup_eq_find_eq`	

Example of maps (code)

```
Lemma lookup_in: forall (m:listPair K V) (k:K) (v:V),

    hmapLookup k m = Some v -> In (k, v) m.

Proof.

intros. induction m. discriminate. unfold hmapLookup in H. simpl in
H.remember (eqb k (fst a)) as b. destruct b. simpl in H. simpl. left.
destruct a. simpl in Heqb. symmetry in Heqb. apply eqb_eq in
Heqb.inversion H. congruence. apply IHm in H. simpl. right. auto.

Qed.
```

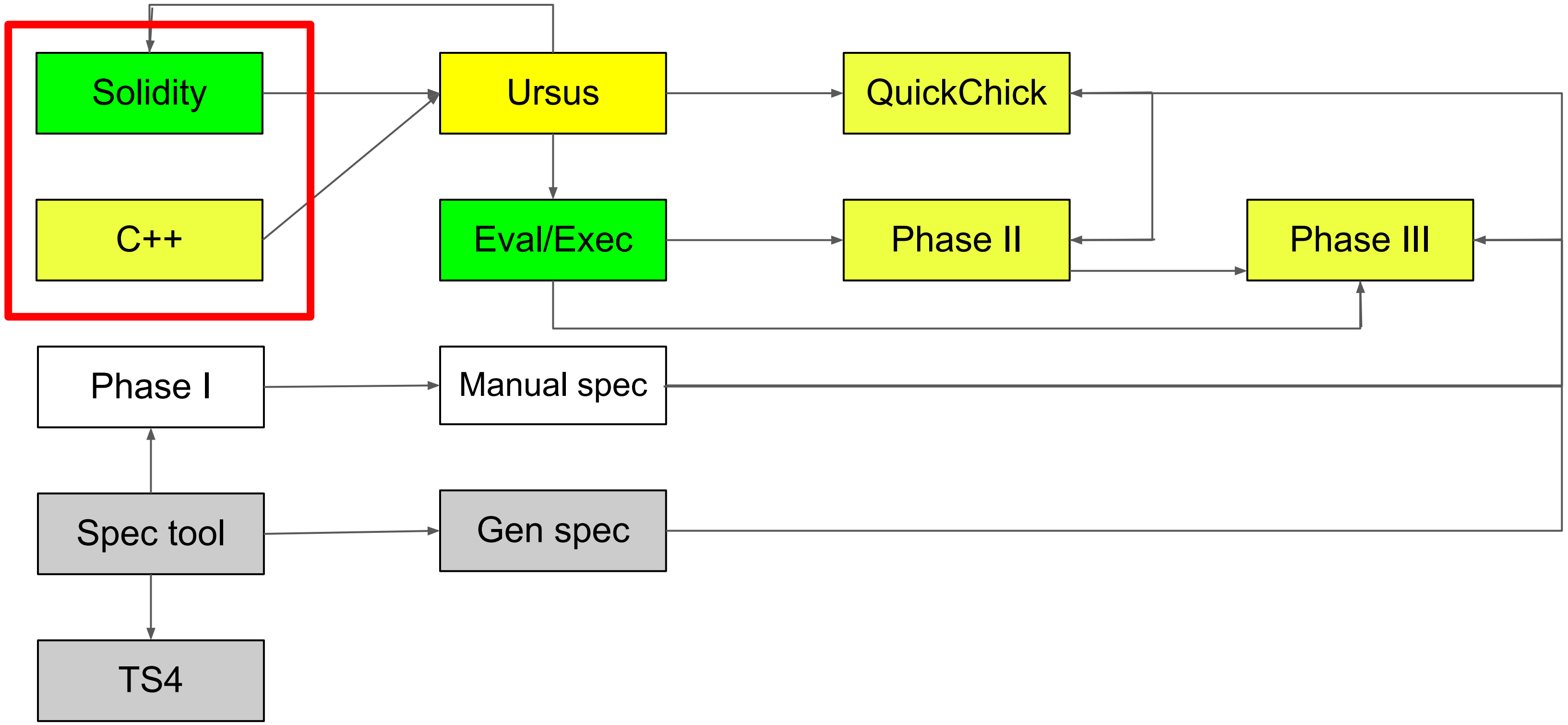
Queue

- push
- pop

These functions are fully done - specified, implemented and proved

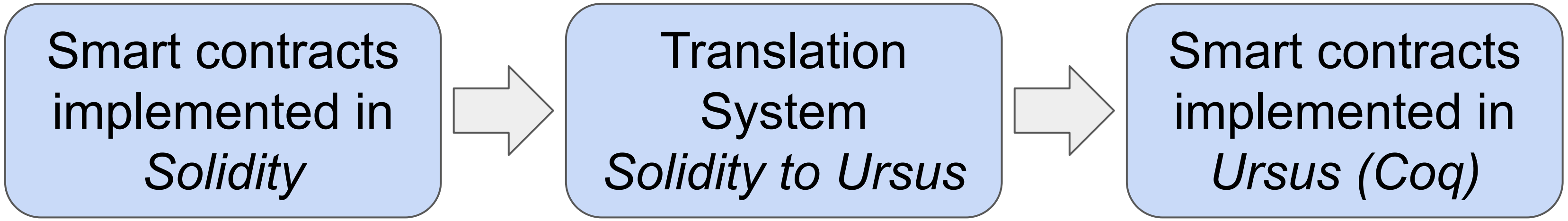
Translators

High-level architecture

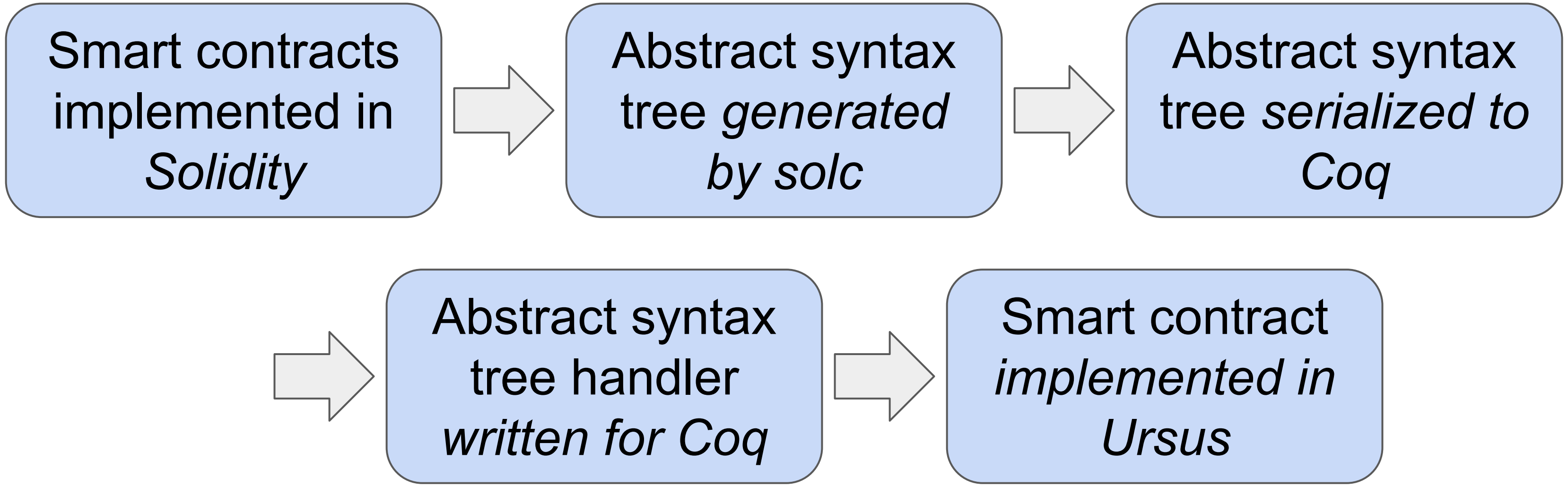


Solidity translator (alpha version)

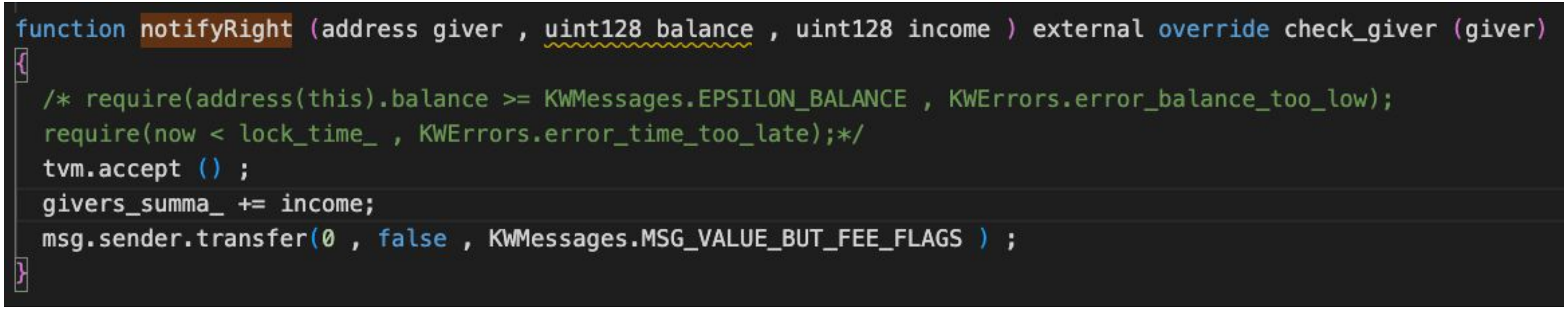
- Fully automated translator from Solidity to Ursus
- Has been tested with KW project


Smart contracts implemented in *Solidity*
Translation System *Solidity to Ursus*
Smart contracts implemented in *Ursus (Coq)*

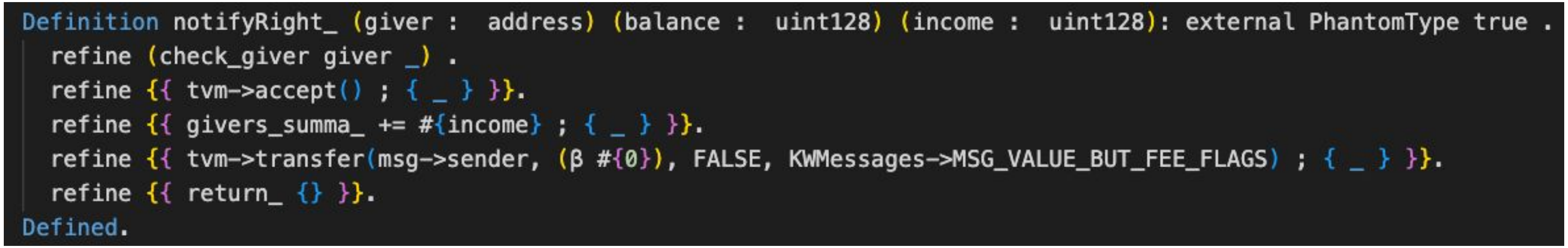
Translation pipeline


Smart contracts implemented in *Solidity*
Abstract syntax tree *generated by solc*
Abstract syntax tree *serialized to Coq*
Abstract syntax tree handler *written for Coq*
Smart contract *implemented in Ursus*

Translation example 1 (easy)

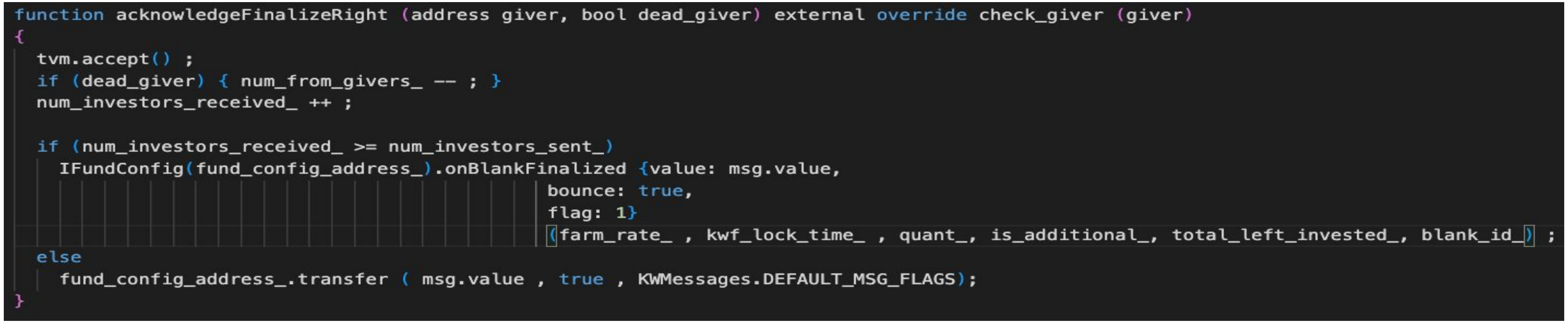


```
function notifyRight (address giver , uint128 balance , uint128 income ) external override check_giver (giver)
{
  /* require(address(this).balance >= KWMessages.EPSILON_BALANCE , KWErrors.error_balance_too_low);
  require(now < lock_time_ , KWErrors.error_time_too_late);*/
  tvm.accept () ;
  givers_summa_ += income;
  msg.sender.transfer(0 , false , KWMessages.MSG_VALUE_BUT_FEE_FLAGS ) ;
}
```



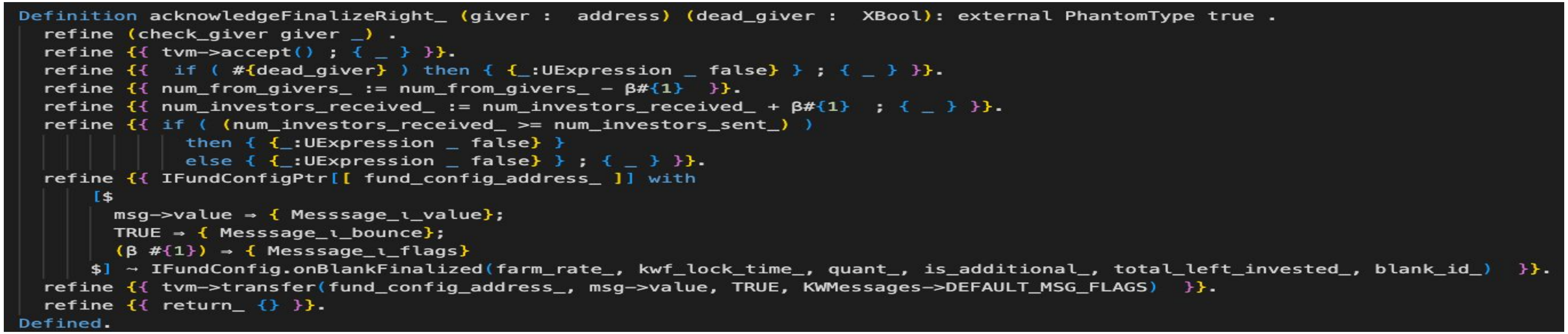
```
Definition notifyRight_ (giver :  address) (balance :  uint128) (income :  uint128): external PhantomType true .
  refine (check_giver giver _) .
  refine {{ tvm->accept() ; { _ } }}.
  refine {{ givers_summa_ += #{income} ; { _ } }}.
  refine {{ tvm->transfer(msg->sender, (β #{0}), FALSE, KWMessages->MSG_VALUE_BUT_FEE_FLAGS) ; { _ } }}.
  refine {{ return_ {} }}.
Defined.
```

Translation example 2 (medium)



```
function acknowledgeFinalizeRight (address giver, bool dead_giver) external override check_giver (giver)
{
  tvm.accept() ;
  if (dead_giver) { num_from_givers_ -- ; }
  num_investors_received_ ++ ;

  if (num_investors_received_ >= num_investors_sent_)
    IFundConfig(fund_config_address_).onBlankFinalized {value: msg.value,
                                                        bounce: true,
                                                        flag: 1}
                                                        (farm_rate_ , kwf_lock_time_ , quant_, is_additional_, total_left_invested_, blank_id_) ;
  else
    fund_config_address_.transfer ( msg.value , true , KWMessages.DEFAULT_MSG_FLAGS);
}
```



```
Definition acknowledgeFinalizeRight_ (giver :  address) (dead_giver :  XBool): external PhantomType true .
  refine (check_giver giver _) .
  refine {{ tvm->accept() ; { _ } }}.
  refine {{  if ( #{dead_giver} ) then { {_:UExpression _ false} } ; { _ } }}.
  refine {{ num_from_givers_ := num_from_givers_ - β#{1}  }}.
  refine {{ num_investors_received_ := num_investors_received_ + β#{1}  ; { _ } }}.
  refine {{ if ( (num_investors_received_ >= num_investors_sent_) )
              then { {_:UExpression _ false} }
              else { {_:UExpression _ false} } ; { _ } }}.
  refine {{ IFundConfigPtr[[ fund_config_address_ ]] with
      [$
        msg->value ⇒ { Messsage_ι_value};
        TRUE ⇒ { Messsage_ι_bounce};
        (β #{1}) ⇒ { Messsage_ι_flags}
      $] → IFundConfig.onBlankFinalized(farm_rate_, kwf_lock_time_, quant_, is_additional_, total_left_invested_, blank_id_)  }}.
  refine {{ tvm->transfer(fund_config_address_, msg->value, TRUE, KWMessages->DEFAULT_MSG_FLAGS)  }}.
  refine {{ return_ {} }}.
Defined.
```

Reverse translator

- Intended to:
  - Demonstrate the correct direct translation (until it formally verified)
  - Provide translation to Solidity if the contracts were originally implemented in Ursus
- Input is source code and **compiled** Ursus project (by Coq)
- Outcome is a set of Solidity files (at least one for each contract)

Reverse translator contains

- Function translator
- Interface translator
- Datatypes translator

Each translator gets the list of compiled Ursus-files generated by `make` configurator

Reverse translator execution example

at first run with: *./urs2solConf.native ~/devel/sample_twofiles_to_contracts/src/ursus*

to get *Makefile* with:

all:

./urs2solFunc.native Crash.v Giver.v

./urs2solIntrf.native ICrash.v IGiver.v

run the translation: make

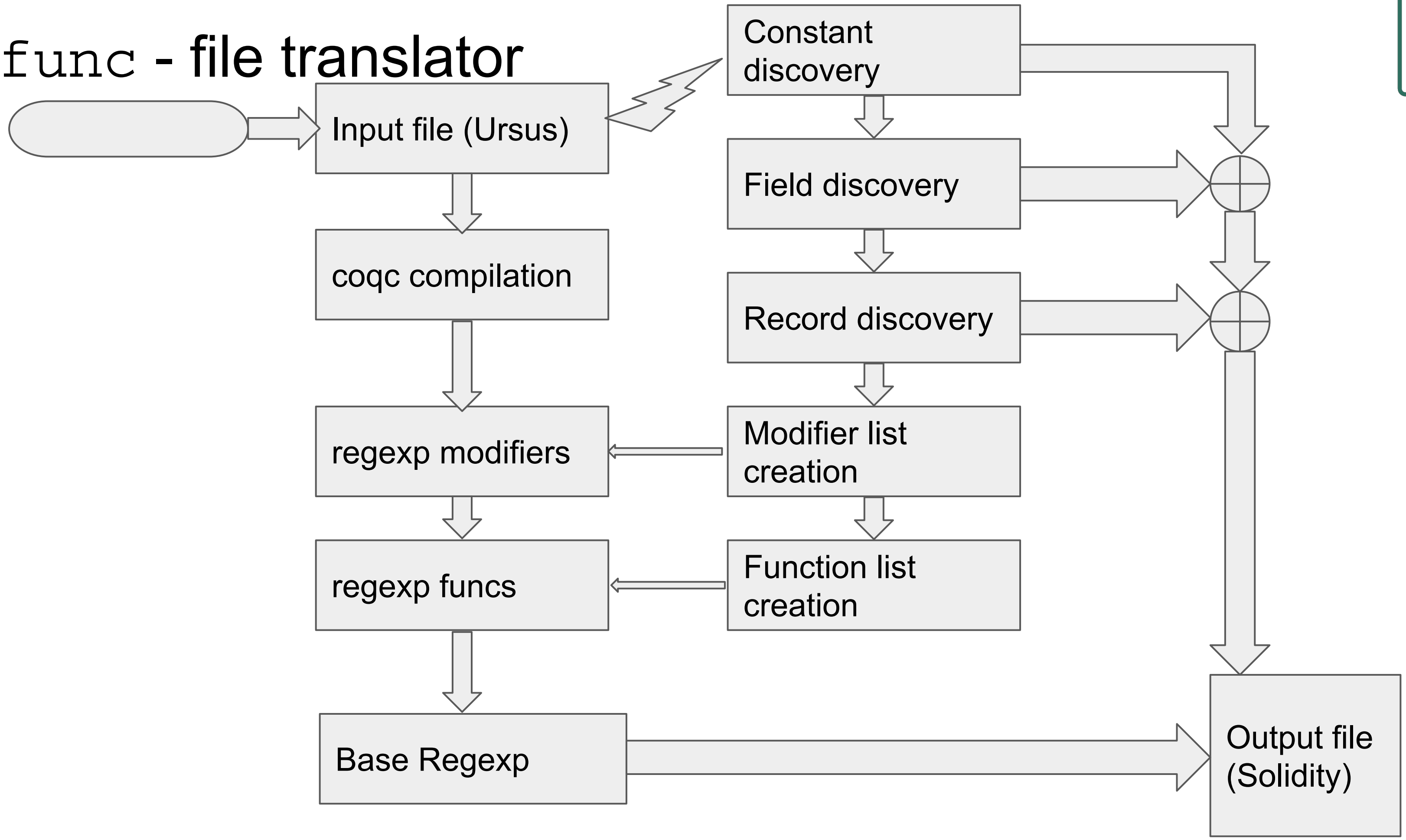
Reverse translator content

- `makefile` configurator
- `makefile` itself
- `func` - file translator
  - function marker script
  - modifier marker script
  - translation and I/O preparation (in Coq)
  - translator script
- interface - file translator

`func` - file content

- Constants
- Contract fields
- Records with fields used in functions
- Modifiers
- Functions themselves

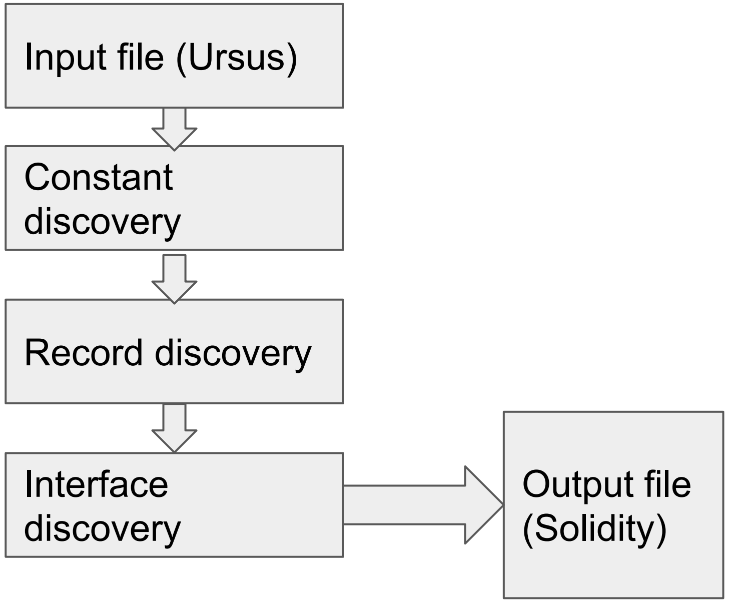
`func` - file translator



interface - file content

- Constants
- Records with fields used in contracts (not just the current contract as interfaces usually are needed for other contracts)
- Interfaces themselves

interface - file translator



C++ translator

- Was reworked after initial Flex 2
- Currently is able to parse most of the structures
- … but still needs some work to go
- Will be used at Flex 2 respin for the maiden flight
- Not presented throughout the current report

Evals/Execs

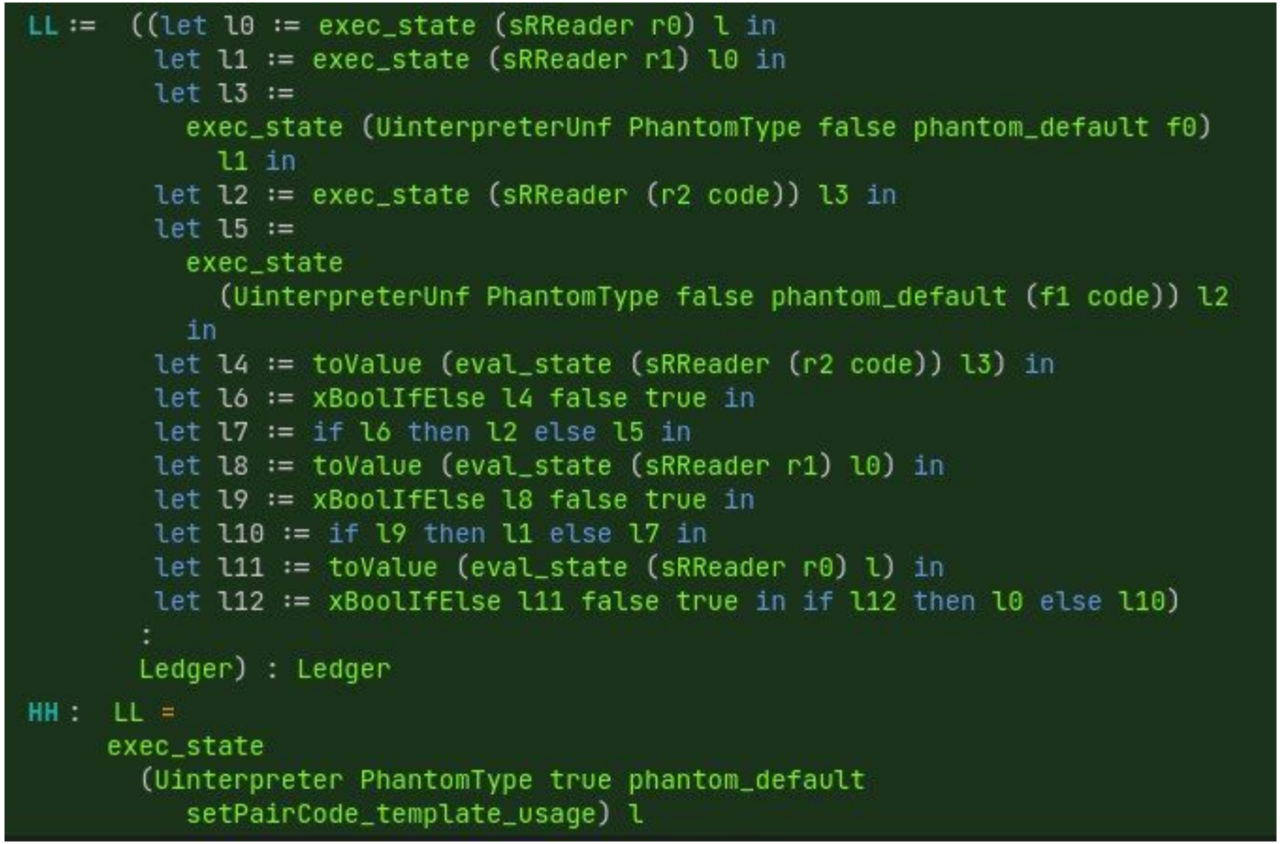
Purpose of Evals/Execs

- While built on top of functional language, Ursus remains an imperative one
- However, the effective Coq proving strategies require pure functional style
- The module turns imperative code into:
  - Evals - functions with input parameters as the input parameters of the "real function" and the current state while return value is a return value of the real function extracted from monad using the given state
  - Execs - functions with input parameters as the input parameters of the "real function" and the current state while return value is a modified state
- Thus, the properties can be proved in a native Coq way

Automated generation of Evals/Execs

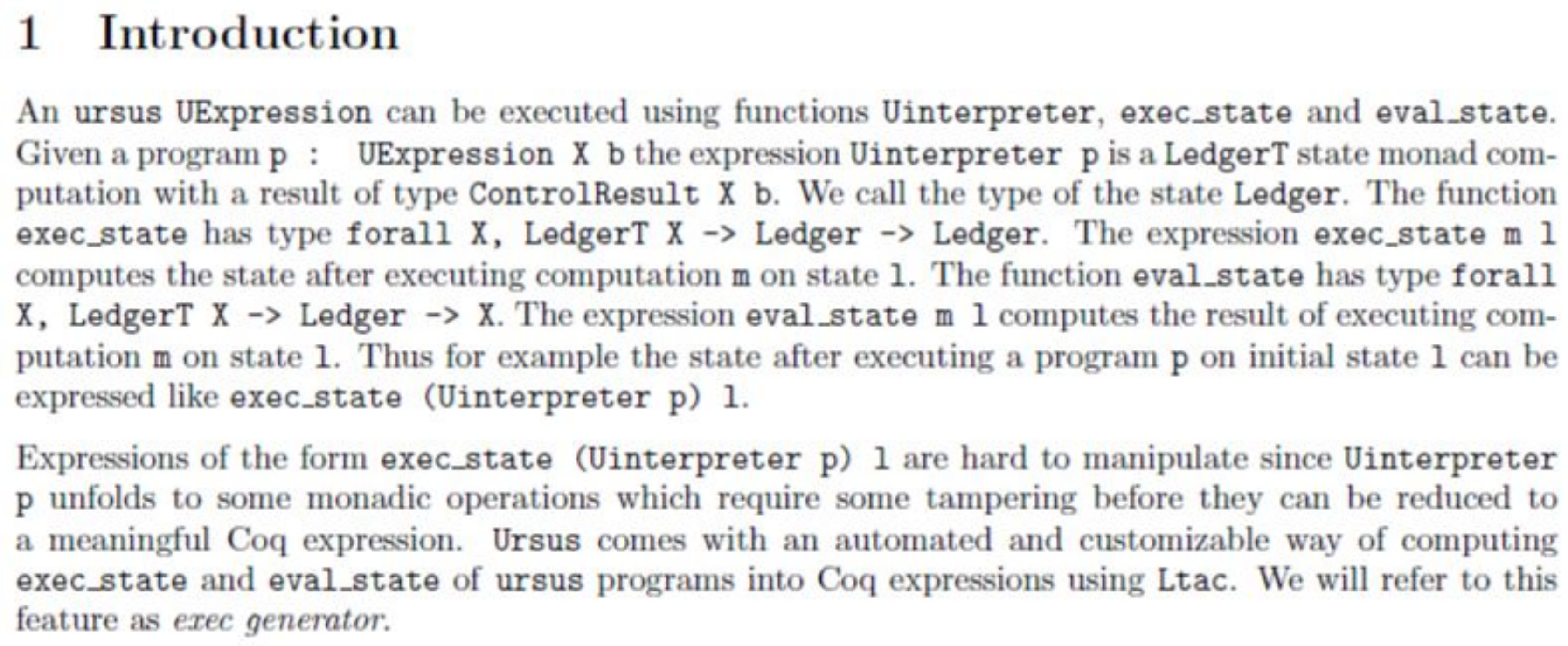
- Automated generation of Evals/Execs is a non-trivial task
- However, this task was completed that makes an important scientific and applied result
- The introduction section of the article (early preprint) is presented at the next slide while the current text is available at https://drive.google.com/file/d/15YODGcLWohGkoCGBY_dRwRiz9CoiSPrz/view?usp=sharing

Evals/Execs automatically generated



```
LL :=  ((let l0 := exec_state (sRReader r0) l in
        let l1 := exec_state (sRReader r1) l0 in
        let l3 :=
          exec_state (UinterpreterUnf PhantomType false phantom_default f0)
            l1 in
        let l2 := exec_state (sRReader (r2 code)) l3 in
        let l5 :=
          exec_state
            (UinterpreterUnf PhantomType false phantom_default (f1 code)) l2
          in
        let l4 := toValue (eval_state (sRReader (r2 code)) l3) in
        let l6 := xBoolIfElse l4 false true in
        let l7 := if l6 then l2 else l5 in
        let l8 := toValue (eval_state (sRReader r1) l0) in
        let l9 := xBoolIfElse l8 false true in
        let l10 := if l9 then l1 else l7 in
        let l11 := toValue (eval_state (sRReader r0) l) in
        let l12 := xBoolIfElse l11 false true in if l12 then l0 else l10)
       :
       Ledger) : Ledger
HH :  LL =
     exec_state
       (Uinterpreter PhantomType true phantom_default
          setPairCode_template_usage) l
```

Automated generation of Evals/Execs (intro section)

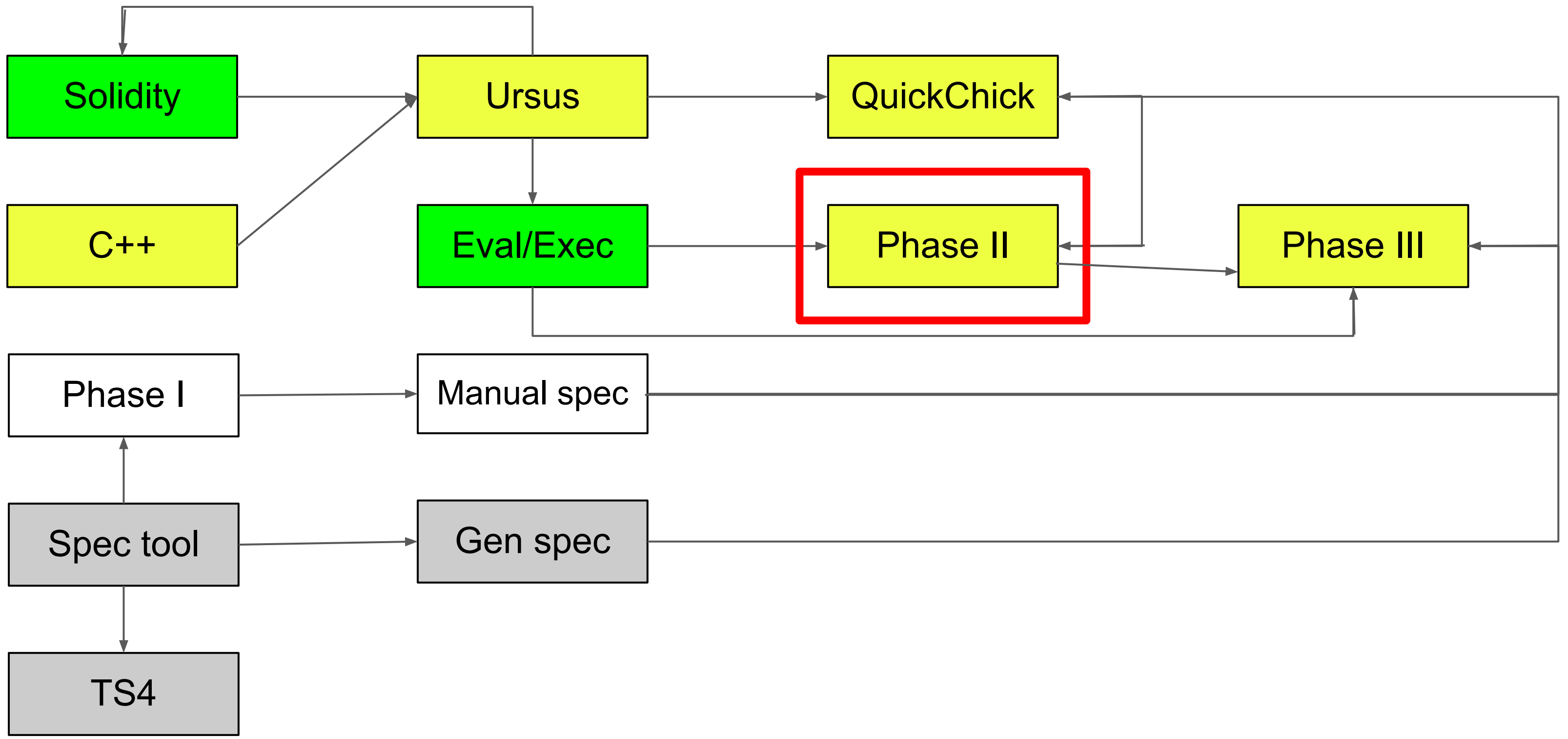


1 Introduction

An `ursus` `UExpression` can be executed using functions `Uinterpreter`, `exec_state` and `eval_state`. Given a program `p : UExpression X b` the expression `Uinterpreter p` is a `LedgerT` state monad computation with a result of type `ControlResult X b`. We call the type of the state `Ledger`. The function `exec_state` has type `forall X, LedgerT X -> Ledger -> Ledger`. The expression `exec_state m l` computes the state after executing computation `m` on state `l`. The function `eval_state` has type `forall X, LedgerT X -> Ledger -> X`. The expression `eval_state m l` computes the result of executing computation `m` on state `l`. Thus for example the state after executing a program `p` on initial state `l` can be expressed like `exec_state (Uinterpreter p) l`.

Expressions of the form `exec_state (Uinterpreter p) l` are hard to manipulate since `Uinterpreter p` unfolds to some monadic operations which require some tampering before they can be reduced to a meaningful Coq expression. `Ursus` comes with an automated and customizable way of computing `exec_state` and `eval_state` of `ursus` programs into Coq expressions using `Ltac`. We will refer to this feature as *exec generator*.

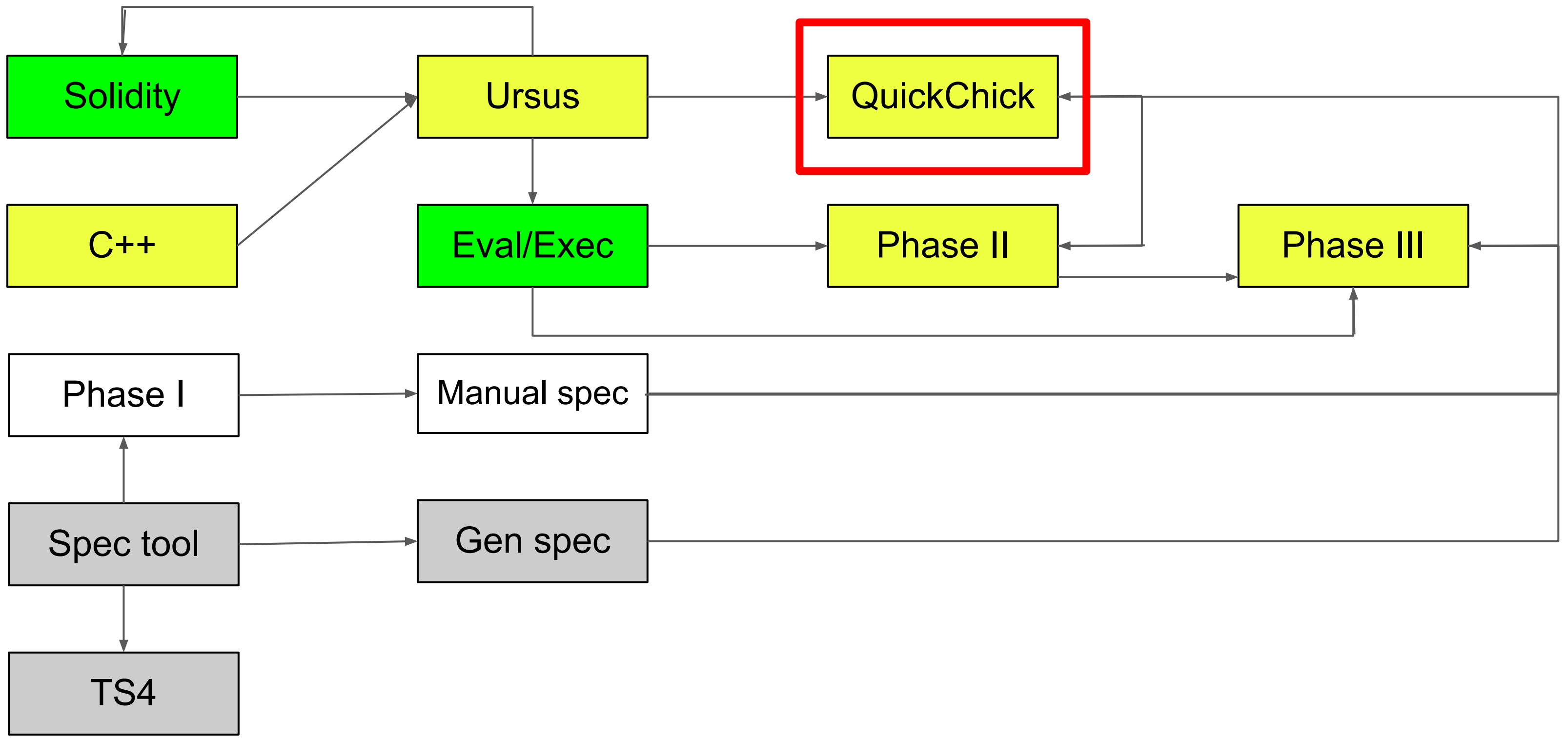
High-level architecture



Phase II

- Automation of Evals/Execs brings an opportunity to a maximal easy proving of Phase II
- So that all given inner functional (unit) specification can be directly proved just in a few lines (probably one) using evals/execs and automation tactics for integers and FOL
- The specification is still translated manually, … for a while
- An approach is to be validated throuought Flex 2 Respin

High-level architecture

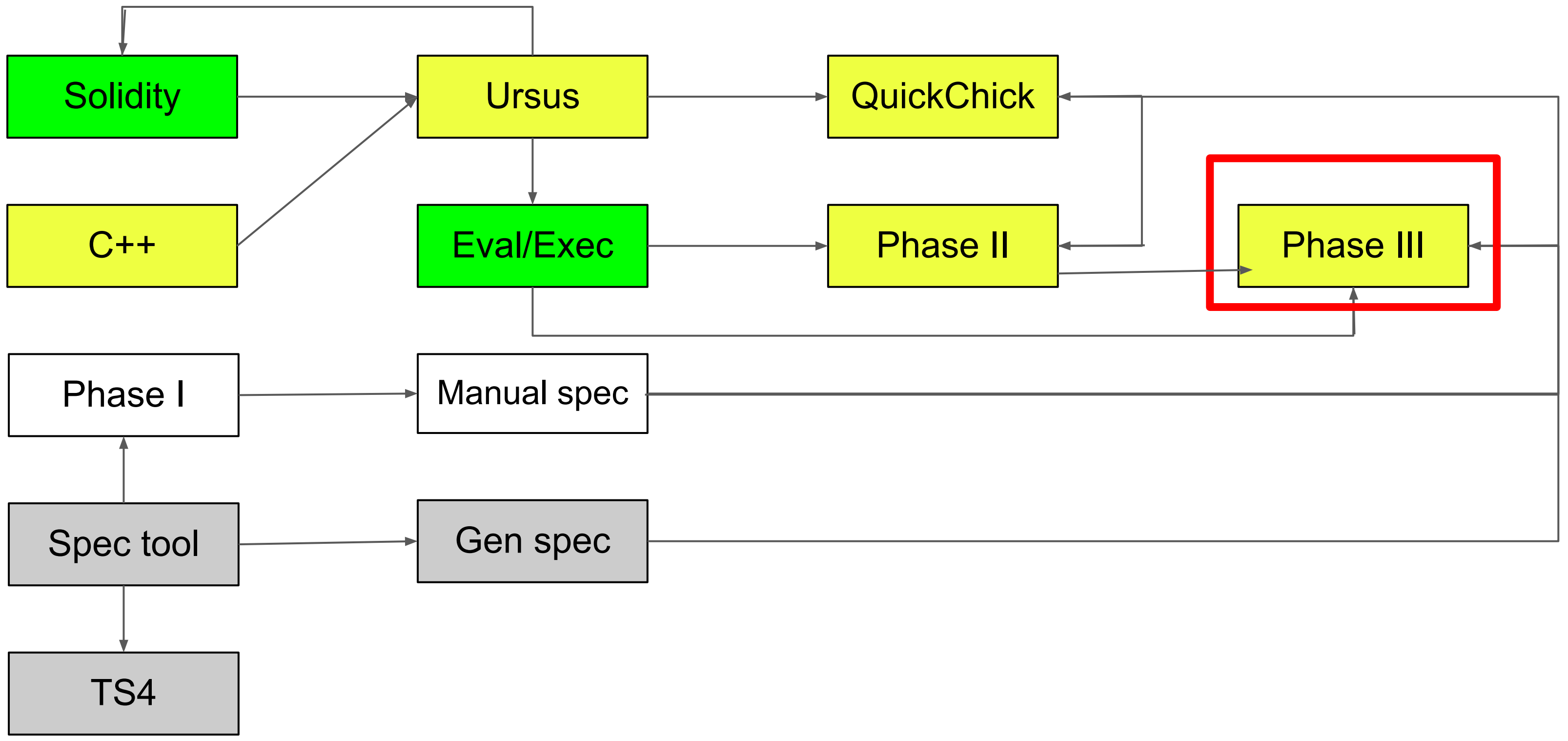


QuickChick

- Randomized automatic property verifier (analogous to QuickCheck in Haskell)
- An easy way to provide sufficient level of verification
- Less reliability
- … but high speed of implementation
- Can be a good competitor for audits
- Can be used immediately after phase I and automatic translation

Phase III

High-level architecture



Intentions

- Only scenario-based cross-functional verification is currently considered (others are correctness invariants, reverse scenarios)
- The goals are:
  - Develop a system for building a mathematical model of smart-contract scenarios.
  - Formulate and verify the specification of such a system.
  - Verify the scenarios of a real smart-contract

Scenario concept

- Scenario is a sequence of external messages considered as a trunk of a full message tree
- The extension of the trunk is performed automatically
- The representation of the scenario have to reflect:
    - The sequence of calls
    - Input parameters
    - Output parameters as functions of the state and input parameters
    - State data of each contract as an input parameter
- Based on SuperLedger concept (SuperMonad)

Scenario example

```
finalize(false, addrFromGiver)->addrBlank

|

'- addrBlank- notifyParticipant(true, 800, 0)->addrFromGiver

|

+- addrFromGiver- transfer->addrBlank

|

'- addrFromGiver- acknowledgeFinalizeRight(addrGiver, 317, true)->addrBlank

startVoting(690699, 0)-> addrBlank

vote(true, 0, 0) -> addrKWDPOOL

|

'- addrKWDPOOL-vote(10, 1, true, 1000, 0, 0) ->addrBlank

|

'- addrBlank- onVoteAccept(10) ->addrKWDPOOL
```

Message processing specification

- Message processing must satisfy the following properties:
  - If the function has failed, the message onBounce should be sent if there was a corresponding flag in the message parameters.
  - All generated messages must be internal
  - The contract address of the sender of each new vertex must match the address of the contract from which these messages were sent.
  - The received value corresponds to the sent value in deterministic way
- Gas support is in progress

Message tree specification

- The message tree must satisfy the following properties
  - If the tree is built correctly and the function with `bounce = true` failed at the root of some subtree, then there is a vertex inside containing the message `onBounce`
  - If the tree is built correctly, then its root is an external message, and its children are all internal messages.
  - All addresses in messages of a correctly constructed tree are correct.

Summary

What has been done (summary)

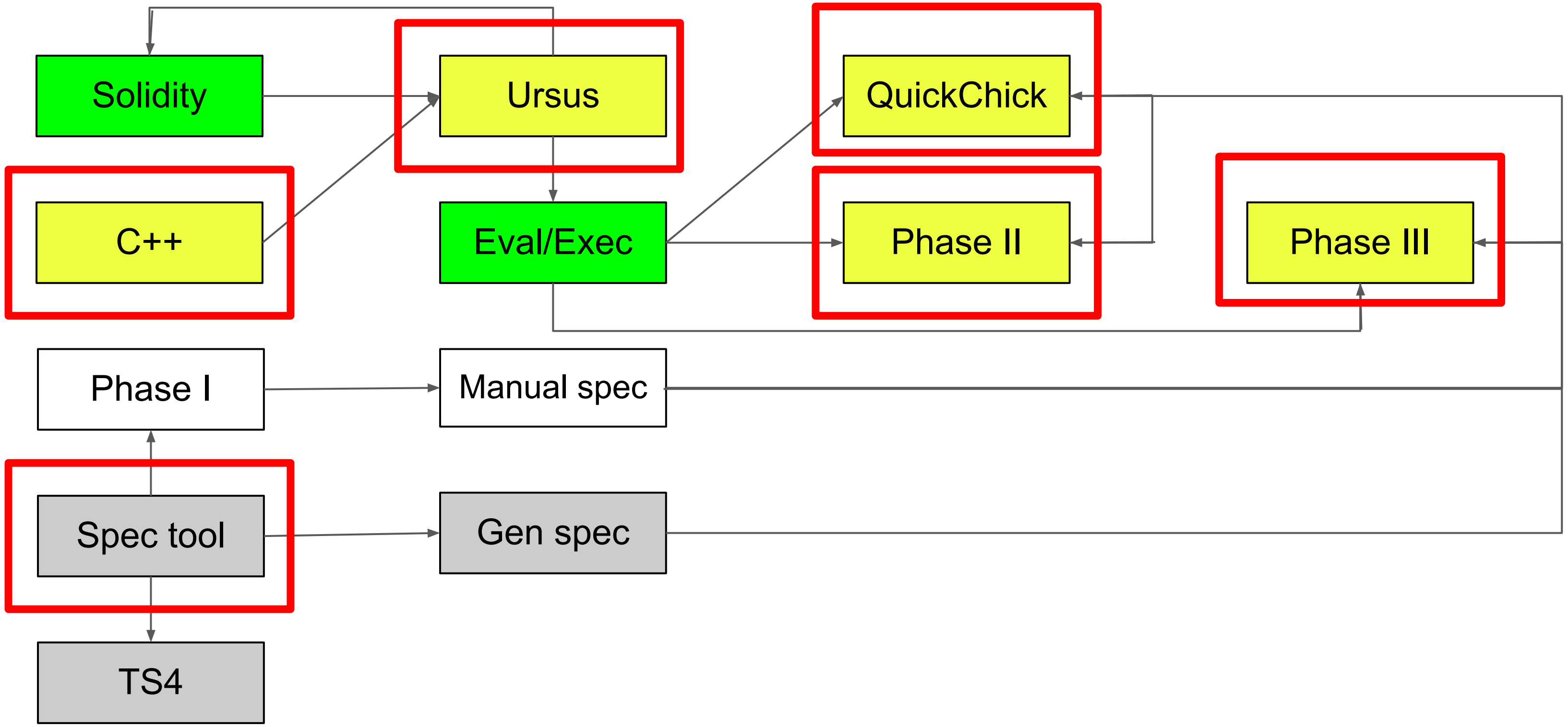
Since the last presentation (at Flex 2) the following results has been achieved:

- Major improvement of Ursus with Embedded Lambda Prolog automation
- Creation of Standard Library, with proving
- Alpha version of Solidity translator
- Alpha version of reverse Solidity translator
- Significant achievements towards C++ translator
- Automation of eval/exec generation and adopting the form needed for scenarios verification

What has been done (summary, continued), 2

- Implemented Solidity translator
- Implemented reverse Solidity translator
- Phase III scenario approach has been implemented

High-level architecture



Plans for the next quarter

- Full-scale solution for C++ translation
- Technology validation with Flex 2
- Improvement of Phase III technologies
- Proofs for Evals/Execs, initial technology’s invention
- Validation and correction of Phase II technologies
- Proofs and specification for the Standard Library, possibly uncompleted
- Ability to create basic specifications using the the scientific approach being invented

Thank you

For questions, comments and detailed answers please contact @andruiman or @SergeyEgorovSPb