

# Holistic Specifications for Robust Smart Contracts

Sophia Drossopoulou, Imperial College London, UK

## Vision

We work on methodologies to deliver *robust* smart contracts. We call robust those smart contracts which behave correctly in the *open* world, i.e., when used in unanticipated ways, and by unknown participants.

We propose that robustness should be at the forefront of smart contract development: It should be an *explicitly* specified concern, and developers should use code-reviews and tools to guarantee adherence to it.

Traditional specification languages do not adequately express robustness, as they were designed for the *closed* world. They attach pre- and post-conditions to each function of a contract, and thus give *sufficient* conditions for some effect to take place. They are explicit about each individual function; but are *implicit* about the overall behaviour emerging from the combination of all functions. Instead, in the open world, we are primarily interested in *necessary* conditions (all possible causes) for effects, and in the behaviour of a contract as a *whole*.

## Holistic Specifications

We have developed *Chainmail*, a *holistic* specification language which extends traditional specification languages [7] with novel concerns about:

**Control:**  $Call(x, y, f(zs), n)$  expresses that currently,  $x$  is the caller,  $y$  the callee,  $f$  the function,  $zs$  the arguments, and  $n$  money was attached,

**Time:** *Next* and *Prev* describe the next and previous snapshot of execution,

**Space:**  $In(A, S)$  means that assertion  $A$  holds when considering only the objects from  $S$ .

**Trust:**  $x$  **obeys**  $A$  says that  $x$  behaves according to the specification  $A$ .

Time allows us to reflect on all possible execution traces. Control allows us to describe the causes of effects. Space is used to distinguish permission and authority [9]. Trust allows us to distinguish the expectations in the case where  $x$  behaves according to  $A$ , from the risks in the case where it does not [6].

### An example: ERC20 transfers

ERC20 [14] keeps track of participants' tokens; tokens may be transferred between participants, provided the transfer was instigated by the account holder, or somebody authorized by them.

In Fig. 1 we define what it means for  $p'$  to be authorized to spend  $m$  tokens on behalf of  $p$ : At some point in the past,  $p$  gave authority to  $p'$  to spend on

$$\begin{aligned}
 Authorized(e, p, p', m) \equiv & \left\{ \begin{array}{l} \exists m'. Prev( Call(p, e.authorize(p', m')) \wedge m' \geq m \\ \text{In the previous step, } p \text{ authorized } p' \text{ for some amount greater or equal } m \\ \vee \\ Prev( Authorized(e, p, p', m) \wedge \neg \exists p''. Call(p', e.transferFrom(p, p'', \_), \_) ) \\ \text{In the previous step, } p' \text{ was authorized for } m \text{ and did not transfer on behalf of } p \\ \vee \\ \exists m'', m'', p''. [ Prev( Authorized(e, p, p', m') \wedge Call(p', e.transferFrom(p, p'', m''), \_) ) \\ \wedge m'' \leq m' - m ] \\ \text{In the previous step, } p' \text{ was authorized for } m', \text{ and transferred on behalf of } p \\ \text{some amount smaller than } m' - m \end{array} \right. \\
 \\
 \forall e : ERC20. \forall p : Any. \forall m : Nat. \\
 [ \quad Next(e.balance(p)) = e.balance(p) - m \\
 \text{If, in the next step } p \text{'s balance decreases by } m, \\
 \longrightarrow \\
 \exists p', p'' : Any. \\
 [ \quad Call(p, e.transfer(p', m), \_) \quad \vee \quad Call(p'', e.transferFrom(p, p', m), \_) \quad \wedge \\
 Authorized(e, p, p'', m) ] \quad ] \\
 \text{then, in the current step either } p \text{ itself, or a } p'' \text{ authorized for } m, \text{ instigates a transfer of } m
 \end{aligned}$$

Figure 1: Authorization and ERC20 guarantees on transfer in *Chainmail*—informal explanations in yellow

their behalf a number which is larger or equal  $m$  plus the sum of tokens spent so far by  $p'$  on the behalf of  $p$ .

We also require that any decrease in a participant's balance (i.e.,  $Next(e.balance(p))=...$ ) is caused either by a transfer instigated by the account holder themselves (i.e.,  $Call(p, ...)$ ), or by a transfer instigated by another participant  $p''$  (i.e.,  $Call(p'', ...)$ ) who had been given authority earlier.

This holistic specification gives to account holders an "authorization-guarantee": their balance cannot decrease unless they themselves, or somebody they had authorized, instigates a transfer of tokens. Moreover, authorization is not transitive.

### Comparison with Traditional Specifications

As stated earlier, traditional specifications give *sufficient* conditions, e.g., if  $p$  is authorized and executes `transferFrom`, then the balance decreases. But they are *implicit* about the overall behaviour and the *necessary* conditions, e.g., what are all the possible actions that can cause a decrease of balance?

With traditional specifications, to obtain the "authorization-guarantee", one would need to inspect the pre- and post- conditions of all the functions in the contract, and determine which ones decrease balances, and then determine which ones affect authorizations. Moreover, with traditional specifications, nothing stops the next release of the contract to add, e.g., a method which allows participants to share their authority, and thus violate the "authorization-guarantee".

### Progress and Project Aims

In [12] we developed *Chainmail*, and specified robustness aspects of popular patterns from object-capabilities [9] and smart contracts: the membrane [15], Mint-and-Purse [10, 13], DOM-wrappers [5], ERC20 [14], and DAO [2].

We now want to apply our ideas to larger case studies. We also want to develop methodologies with which to demonstrate that contracts adhere to *Chainmail* specifications, including argued code-reviews, testing [1], inference systems, and tool-based program verification [8, 11]. Results on small, class-based languages and methodologies for code-reviews would take one and a half highly-qualified person-years, moving to delegation-based would take another person-year, the development of verification logics one person-year, first stab at verification tools another two person-years, and same amount for the testing infrastructure. We collaborate with Susan Eisenbach (Imperial), James Noble (VU Wellington) and Toby Murray (Univ. Melbourne).

## References

- [1] M. Christakis, P. Mueller, and V. Wuestholz. Guiding dynamic symbolic execution toward unverified program executions. In *ICSE*, 2016.
- [2] Christoph Jentsch. Decentralized Autonomous Organization to automate governance paper. <https://download.slock.it/public/DAO/WhitePaper.pdf>.
- [3] Coindesk. Understanding the DAO attack. [www.coindesk.com/understanding-dao-hack-journalists/](http://www.coindesk.com/understanding-dao-hack-journalists/), June 2016.
- [4] T. V. Cutsem and M. S. Miller. Trustworthy proxies: Virtualizing objects with invariants. In *ECOOP*, 2013.
- [5] D. Devriese, L. Birkedal, and F. Piessens. Reasoning about object capabilities with logical relations and effect parametricity. In *IEEE EuroS&P*, pages 147–162, 2016.
- [6] S. Drossopoulou, J. Noble, and M. Miller. Swapsies on the internet: First steps towards reasoning about risk and trust in an open world. In *(PLAS)*, 2015.
- [7] G. T. Leavens, E. Poll, C. Clifton, Y. Cheon, C. Ruby, D. R. Cok, P. Müller, J. Kiniry, and P. Chalin. JML Reference Manual. Iowa State Univ. [www.jmlspecs.org](http://www.jmlspecs.org), February 2007.
- [8] K. R. Leino. Dafny: An automatic program verifier for functional correctness. In *LPAR16*. Springer, April 2010.
- [9] M. S. Miller. *Robust Composition: Towards a Unified Approach to Access Control and Concurrency Control*. PhD thesis, Baltimore, Maryland, 2006.
- [10] M. S. Miller, C. Morningstar, and B. Frantz. Capability-based Financial Instruments: From Object to Capabilities. In *Financial Cryptography*. Springer, 2000.
- [11] L. D. Moura and N. Bjørner. Z3: an efficient smt solver. In *TACAS, ETAPS*. Springer, 2008.
- [12] S. Drossopoulou. Holistic Specifications for Robust Code. Invited Talk at FTfJP and Codemesh, [http://www.doc.ic.ac.uk/~scd/Holistic\\_Specs.pdf](http://www.doc.ic.ac.uk/~scd/Holistic_Specs.pdf), July 2018.
- [13] Solidity Community. Subcurrency Example. [//solidity.readthedocs.io/en/develop/introduction-to-smart-contracts.html](http://solidity.readthedocs.io/en/develop/introduction-to-smart-contracts.html).
- [14] The Ethereum Wiki. ERC20 Token Standard. [https://theethereum.wiki/w/index.php/ERC20\\_Token\\_Standard](https://theethereum.wiki/w/index.php/ERC20_Token_Standard).
- [15] T. van Cutsem. Membranes in JavaScript, 2012. [prog.vub.ac.be/~tvcutsem/invokedynamic/js-membranes](http://prog.vub.ac.be/~tvcutsem/invokedynamic/js-membranes).

**Appendix A: DAO** in a similar style as that of the ERC20 spec earlier, we can write a *Chainmail* specification requiring that DAO [2] holds as much ether as the sum of the its clients' balances, and that balances may only be affected by clients joining or leaving, and projects being approved or repaying. This precludes the famous re-entrancy bug [3].

**Appendix B: Escrow** Given clients  $c1$  and  $c2$  holding accounts in two currencies, an Escrow agent can exchange  $k$  of one currency against  $k'$  in the other, provided  $c1$  and  $c2$  had sufficient funds [4].

One expects that the clients behave according to spec `ValidClient` – e.g., do not try to withdraw more than entitled. But what are the risks if some clients do not adhere to `ValidClient`, and there is no central authority to certify them? We sketch a spec for this in Figure 2 (more in [6]): If the agent

reports success (by returning `true`), then either: (a) both clients satisfy `ValidClient` and had sufficient funds and the exchange did take place or, (b) neither satisfy `ValidClient`. Similarly, if the agent reports failure (by returning `false`), then either (c) they both satisfy `ValidClient` but did not have sufficient funds, or (d) exactly one satisfies `ValidClient`. Case (b) was surprising, esp. as there is no way to implement the agent so as to internally distinguish (a) from (b). Nevertheless, the risk is limited: in all cases except (a), we have the guarantee that all `ValidClient` are not affected.

```
e obeys ValidEscrow ≡
true
{ res := e.exchange(c1,c2,k,k') }
[ res = true ∧ c1 obeys ValidClient ∧ c2 obeys ValidClient
→ c1, c2 had sufficient funds & transfers between them took place, other ValidClient not affected ] ∧
[ res = true ∧ ¬(c1 obeys ValidClient) ∧ ¬(c2 obeys ValidClient)
→ c1, c2 may be affected, but all ValidClient not affected ] ∧
[ res = false ∧ c1 obeys ValidClient ∧ c2 obeys ValidClient
→ c1 or c2 had insufficient funds, c1 and c2 and all ValidClient not affected ] ∧
[ res = false ∧ ( c1 obeys ValidClient ∨ c2 obeys ValidClient )
→ all ValidClient not affected ]
```

Figure 2: Sketch of Escrow specification – yellow part described informally