Holistic Specifications for Robust Contracts on RChain

S. Drossopoulou & S. Eisenbach, Imperial College London

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 spec. languages do not adequately address robustness. 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 are *explicit* about contracts' overall behaviour.

Holistic Specifications

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

Control: $Call(\mathtt{x},\mathtt{y}.\mathtt{f}(\mathtt{zs}),\mathtt{n})$ expresses that currently, $\mathtt{x}$ is the caller, $\mathtt{y}$ the callee, $\mathtt{f}$ the function, $\mathtt{zs}$ the arguments, and $\mathtt{n}$ 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: $\mathtt{x}$ **obeys** A says that $\mathtt{x}$ behaves according to the specification A.

Control allows us to describe the causes of effects. Time allows us to reflect on all possible execution traces. Space is used to distinguish permission and authority [8]. Trust allows us to distinguish the expectations in the case where $\mathtt{x}$ behaves according to A, from the risks in the case where it does not [6].

An example: ERC20 transfers

ERC20 [12] 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 $\mathtt{p}'$ to be authorized to spend $\mathtt{m}$ tokens on behalf of $\mathtt{p}$: At some point in the past, $\mathtt{p}$ gave authority to $\mathtt{p}'$ to spend on their behalf a number which is larger or equal $\mathtt{m}$ plus the sum of tokens spent so far by $\mathtt{p}'$ on the behalf of $\mathtt{p}$.

We also require that any decrease in a participant's balance (i.e., $Next(\mathtt{e.balance(p)})$=...) is caused either by a transfer instigated by the account holder themselves (i.e., $Call(\mathtt{p},...)$), or by a transfer instigated by another participant $\mathtt{p}''$ (i.e., $Call(\mathtt{p}''...)$ 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 $\mathtt{p}''$ 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

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

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

"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 [10] we outline *Chainmail* and specify robustness aspects of popular patterns from object-capabilities [8] and smart contracts: membrane [13], Mint-and-Purse [9, 11], DOM-wrappers [5], ERC20 [12], and DAO [2].

This project aims to develop tools to express and ascertain robustness of RChain contracts. Deliverables:

3rd Month: Complete formal semantics of *Chainmail* for COO_0, a minimal object-oriented-contract calculus,

6th Month: Tool for *Chainmail* validation of COO_0 contracts through model checking [1], and corresponding test suite.

9th Month: Adaptation of previous test suite to Rholang, and new tests. Extend *Chainmail* semantics for Rholang: cater for channels, patterns, and asynchroneity.

12th Month: Tool for *Chainmail* validation of RL_0 contracts through model checking.

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/, 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, February 2007.

[8] M. S. Miller. *Robust Composition: Towards a Unified Approach to Access Control and Concurrency Control*. PhD thesis, Baltimore, Maryland, 2006.

[9] M. S. Miller, C. Morningstar, and B. Frantz. Capability-based Financial Instruments: From Object to Capabilities. In *Financial Cryptography*. Springer, 2000.

[10] S. Drossopoulou. Holistic Specifications for Robust Code. Invited Talk at FTfJP and Codemesh, //www.doc.ic.ac.uk/~scd/Holistic_Specs.pdf, July 2018.

[11] Solidity Community. Subcurrency Example. //solidity.readthedocs.io/en/develop/introduction-to-smart-contracts.html.

[12] The Ethereum Wiki. ERC20 Token Standard. https://theethereum.wiki/w/index.php/ERC20_Token_Standard.

[13] T. van Cutsem. Membranes in JavaScript, 2012. 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 no `ValidClient` is 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 – green part left informal