Verifying Smart Contracts with Cubicle

Sylvain Conchon[1,2](✉), Alexandrina Korneva[1], and Fatiha Zaïdi[1]

[1] LRI (CNRS & Univ. Paris-Sud), Université Paris-Saclay, 91405 Orsay, France
Conchon@lri.fr

[2] Inria, Université Paris-Saclay, 91120 Palaiseau, France

Abstract. Smart contracts are usually conceived for an unkonwn (and potentially varying) number of users. From a theoretical point of view, they can be seen as parameterized state machines with multiple entry points, shared variables, and a message passing mechanism. To help in their design and verification, we propose in this paper to use Cubicle, a model checker for parameterized systems based on SMT. Our approach is a two-layer framework where the first part consists of a blockchain transactional model, while the second layer is a model of the smart contract itself. We illustrate our technique through the simple yet prime example of an auction. This preliminary result is very promising and lays the foundations for a complete and automatized framework for the design and certification of smart contracts.

Keywords: Blockchain · Smart-contracts · Model checking · MCMT

1 Introduction

The number of decentralized applications (DApps) running on top of blockchain networks is growing very fast. According to [3], there are now more than 3,000 DApps available on the Ethereum and EOS platforms which generate over 600 thousand transactions per day for a volume of 17 million USD. These DApps interact with the blockchain through around 4,700 smart contracts.

A smart contract is a stateful program stored in the blockchain with which a user (human or computer) can interact. From a legal perspective, a smart contract is an agreement whose execution is automated. As such, its revocation or modification is not always possible and worse than that, what the code of a smart contract does is *the law*... no matter what it may end up doing. Unfortunately, like any program, smart contracts may have bugs. Given the potential financial risks, finding these bugs before the origination of the contracts in the blockchain is an important challenge, both from economic and scientific points of view.

Various formal methods have been used to verify smart contracts. In [5], the authors present a shallow embedding of Solidity within F*, a programming language aimed at verification. Other similar approaches are based on deductive verification platforms like Why3 [10,12]. Interactive proof assistants (*e.g.* Isabelle/HOL or Coq) have also been used for modeling and proving properties about Ethereum and Tezos smart contracts [1,4].

© Springer Nature Switzerland AG 2020
E. Sekerinski et al. (Eds.): FM 2019 Workshops, LNCS 12232, pp. 312–324, 2020.
https://doi.org/10.1007/978-3-030-54994-7_23

A common thread here is the use of general-purpose frameworks based on sequential modeling languages. However, smart contracts can be considered as state machines [2,9] whose execution model, according to [13], is closer to that of a concurrent programming language rather than a sequential one. In this context, the use of model checking techniques becomes highly appropriate [6,11].

An important aspect of blockchains is that they are completely open. As a consequence, smart contracts are state machines that need to be conceived for an unknown (and potentially varying) number of users. This parameterized side of blockchains has not previously been taken into account.

In this paper, we propose a first ad-hoc attempt to model smart contracts into the declarative input language of Cubicle, a model checker for parameterized systems based on Satisfiability Modulo Theories (SMT) techniques. In our approach, smart contracts, as well as the transactional model of the blockchain, are encoded as a state machine on which safety properties of interest are encoded and verified. Our contributions are as follows:

- A two-layer framework for smart contract verification in Cubicle (Sect. 3). The first layer is a model of the blockchain transaction mechanism. The second layer models the smart contract itself.
- A description of how to express smart contract properties as Cubicle safety properties using both ghost variables and model instrumentation (Sect. 4)
- A way of interpreting Cubicle error traces as part of the smart contract development cycle (Sect. 5)

2 A Motivating Example

We illustrate our work with the example of an auction contract. The behavior of each client i is given by the automaton $\mathcal{A}(i)$ in Fig. 1.

Every client starts off in state *S1* where they can either bid a certain value v and go to state *S2*, or close the auction and stay in *S1*. In *S2*, a client can either withdraw their bid and end up back in *S1*, close the auction and stay in *S2*, or try to win the auction and go to *S3*.

Each state transition is guarded by certain conditions to ensure that the auction works correctly. To implement these conditions, each automaton needs to share some variables with the other automata and be able to send or receive messages through communication channels. The shared variables are:

- HBidder: the current highest bidder (winner)
- HBid: the above's bid amount
- Ended: a boolean variable for whether or not an auction is over
- Owner: the person who owns the contract
- PR_i: the amount a client i can withdraw,

while channels for message passing are bid, withdraw, end, win, and refund_i (a refund channel for each client i). These messages are synchronous and can have parameters. Sending a message with parameter c on channel ch is denoted by $\mathsf{ch}!(c)$, and the reception is written $\mathsf{ch}?(v)$.

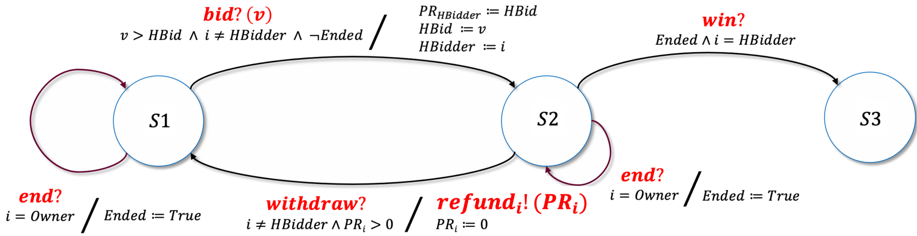

Fig. 1. Auction automaton $\mathcal{A}(i)$

In order to better explain how the automaton works, we'll look at states *S1* and *S2*, and the transitions between them.

The condition to trigger the transition from *S1* to *S2* is given by the formula $\mathsf{bid}?(v) \wedge v > \mathsf{HBid} \wedge i \neq \mathsf{HBidder} \wedge \neg\mathsf{Ended}$, which should be read as follows:

- a message on channel bid is received with a value v superior to the current HBid ($v > \mathsf{HBid}$);
- the client isn't the current winner ($i \neq \mathsf{HBidder}$);
- the auction is still open ($\neg\mathsf{Ended}$).

If these requirements are fulfilled, the transition is triggered, the client goes to *S2* and the state variables are modified in the following manner:

- $\mathsf{PR}_{\mathsf{HBidder}} := \mathsf{HBid}$ sets the pending returns for the old winner to their old winning bid;
- $\mathsf{HBid} := v$ sets the new top bid to v;
- $\mathsf{HBidder} := i$ sets the new winner to the client i.

Going from *S2* back to *S1* works the same way. The transition can be triggered when the condition $\mathsf{withdraw}? \wedge i \neq \mathsf{HBidder} \wedge \mathsf{PR}_i > 0$ is true, that is when a message on `withdraw` is received and the client i is not the highest bidder and has some amount to withdraw. When moving to *S2*, the variable PR_i is reset and a message $\mathsf{refund}_i!(\mathsf{PR}_i)$ is sent. The corresponding reception $\mathsf{refund}_i?(v)$ is part of a refund automaton $\mathcal{R}(i)$ run by each client, as seen in Fig. 2. The role of this one state automaton is to accept that kind of message and do whatever necessary to accept the refunded value v.

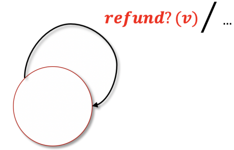

Fig. 2. Refund automaton $\mathcal{R}(i)$

Finally, the global system of the auction and the clients it interacts with can be expressed as a product of the auction automata and the refund automata

$$\Pi_i \mathcal{A}(i) \times \Pi_i \mathcal{R}(i)$$

A client of this contract, aka the automaton, needs to be sure that certain properties hold. These properties can be simple and visible in the transition requirements, or they can be more complex to the point where you can't prove them through the requirements alone. Consider two properties:

(a) "Each new winning bid is superior to the old winning bid"
(b) "I do not lose money"

Property (a) is easy to see as it's the requirement $v > \mathsf{HBid}$ of Fig. 1. Property (b) is less obvious as it requires not only a model of the contract itself, but also of the underlying blockchain semantics.

3 Modeling a Smart Contract with Cubicle

Anything done by a smart contract can be traced back to its blockchain. If HBidder is modified, that means that there was a message on channel bid with a sufficiently large value. Not being able to trace an action back to the blockchain implies a problem. Therefore, modeling a smart contract requires an accompanying model of the blockchain. We do this with the help of Cubicle, briefly introduced in the next subsection.

3.1 Cubicle

Cubicle is an SMT-based model checker for parameterized transition systems. For a more in-depth and thorough explanation, we refer the reader to [7,8]. In this section, we give a quick overview of the necessary aspects of Cubicle.

Cubicle input programs represent transition systems described by: (1) a set of type, variable and array declarations; (2) a formula for the initial states; and (3) a set of guarded commands (transitions).

Type, Variable and Array Declarations. Cubicle has several built-in data types, among which are integers (int), booleans (bool), and process identifiers (proc). Additionally, the user can define enumerations. For instance, the code

```
type location = L1 | L2 | L3
var W : location
var X : int
array Z[proc] : bool
```

defines a type location with three constructors (L1, L2, and L3), two global variables W and X of types location and int, respectively, and a proc-indexed array Z. The type proc is a key ingredient here as it is used to parameterize the system: given a process identifier i, the value Z[i] represents somehow the *local* variable Z of i.

Initial States. The content of a system state is fully characterized by the value of its global variables and arrays. The initial states are defined by an **init** formula given as a universal conjunction of literals. For example, the following declaration

```
init (i) { Z[i] = False && W = L1 }
```

should be read as: "initially, for all process i, Z[i] is equal to False and W contains L1". (Note that the content of variable X is unspecified, and can thus contain any value)

Transitions. The execution of a parameterized system is defined by a set of guard/action transitions. It consists of an infinite loop which non-deterministically triggers at each iteration a transition whose guard is true and whose action is to update state variables. Each transition can take one or several process identifiers as arguments. A guard is a conjunction of literals (equations, disequations or inequations) and an action is a set of variable assignments or array updates. For instance, the following transition

```
transition tr_1 (i)
requires { Z[i] = False }
{ W := L2;
  X := 1; }
```

should be read as follows: "if there exists a process i such that Z[i] equals False, then atomically assign W to L2 and X to 1".

Unsafe States. The safety properties to be verified are expressed in their negated form and characterize unsafe states. They are given by existentially-quantified formulas. For instance, the following unsafe formula

```
unsafe (i) { Z[i] = False && X = 1 }
```

should be read as follows: "a state is unsafe if there exists a process i such that Z[i] is equal to False and X equals 1".

Error Traces. All of the above allows Cubicle to verify a model. If it finds a way to reach an unsafe state, an error trace is printed, such as the following

```
Error trace: Init -> t2(#1) -> t3(#3) -> unsafe[1]
```

This lets the user check which sequence of transitions led to the unsafe state. A number preceded by # is a process identifier. This means that t2(#1) stands for process 1 activating that transition. If you have multiple unsafe states declared, the index next to **unsafe** lets you know which one was reached.

3.2 Blockchain Model

To model the blockchain we first need to model the elements that will constitute transactions seen in the blockchain. Consider the transactions as the message passing mechanism from Sect. 2.

```
type call = Bid | Withdraw | Send | Finish | None

var Cmd : call
var Value : int
var Sender : proc
var Recv : proc
```

The constructors of type call represent calls to smart contract entry points (message channels). Bid, Withdraw, Finish, Send correspond to the channels bid, withdraw, end, and refund, while None means *absence of transactions*. The elements of a transaction are defined by four variables:

- Cmd, the calls to an entry point;
- Value, the amount of money attached to a transaction;
- Sender, who calls the contract;
- Recv; the receiver, used in the case of Withdraw, where the contract calls a client.

Once the elements of a transaction are declared, the next step is to model the transaction mechanism of the blockchain. For that, we define three transitions to simulate transactions to the three smart contract entry points.

```
transition call_bid(i)
requires { Cmd = None }
{
  Cmd := Bid;
  Value := Rand.Int();
  Sender := i;
}

transition call_withdraw(i)
requires { Cmd = None }
{
  Cmd := Withdraw;
  Sender := i;
}

transition call_finish(i)
requires { Cmd = None }
{
  Cmd := Finish;
  Sender := i;
}
```

Each transaction has a parameter i which represents the client who called the corresponding entry point. The only requirement indicates that the contract can't be doing something else simultaneously (Cmd = None). The effects of these transitions are simple: Cmd is set to the corresponding constructor (Bid, Withdraw, or Finish, respectively) and the variable Sender is assigned to i. In call_bid, the variable Value is set to a (positive) random integer corresponding to the amount bid by i.

Once the blockchain has been modeled, we can move on to modeling the contract itself.

3.3 Smart Contract Model

To model the actual contract, we need to model its variables and its functions. These correspond to the state variables and the transitions from Sect. 2, respectively.

```
var HBidder : proc
var HBid : int
var Ended : bool
var Owner : proc
array PR[proc] : int
```

The transitions from state *S1* to *S2* and back are modeled as Cubicle transitions. These transitions serve as entry points for our contract.

```
transition bid(i)
requires { Ended = False && Cmd = Bid && i = Sender
           && i <> HBidder && PR[i] = 0 && Value > HBid }
{
  HBid := Value;
  HBidder := i;
  PR[HBidder] := HBid;
  Cmd := None;
}

transition withdraw(i)
requires { Cmd = Withdraw && i = Sender && PR[i] > 0 }
{ PR[i] := 0;
  Cmd := Send;
  Value := PR[i];
  Recv := i;
}
```

Transition bid is called by one process, i, who has to be the Sender, but not the current HBidder. The other requirements should be read as follows:

Ended = False: the auction is open
Cmd = Bid: the transaction in the blockchain is Bid
PR[i] = 0: the new bidder hasn't previously bid
Value > HBid: the new bid is bigger than the old winning bid

The effects are simple, HBidder and HBid are set to the new values, PR for the old winner who has now been outbid is set to his old bid value, and Cmd is reset to None to indicate that the contract is no longer occupied.

Similarly, the requirements of transition withdraw are the following:

Cmd = Withdraw: the transaction in the blockchain is a call to withdraw
i = Sender: the process i is the one that called the function
PR[i] > 0: the person previously bid and was outbid by someone

The effects of transition withdraw are slightly different since withdraw goes on to send money to whoever called the method. The receiver is now set to i (Recv := i), and the value that will accompany the transaction is set to the amount of money to be returned (Value := PR[i]). The pending return PR[i] is reset globally for client i (PR[i] = 0) and Cmd is set to Send, to indicate that the contract is calling the client's method. The transition which Send corresponds to can be seen below:

```
transition value(i)
requires { Cmd = Send && Recv = i }
{ Cmd := None; }
```

This transition checks that Send was in fact called (Cmd = Send), as well as the fact that the receiver is the currently active process (Recv = i). It then resets Cmd to None to free the contract.

4 Defining and Verifying Properties

Recall that we want to be sure of certain properties:

(a) "Each new winning bid is superior to the old winning bid"
(b) "I do not lose money"

Once defined informally, the properties need to be converted into safety properties. This is not always straightforward and might require additional information. It is done via a two-step process consisting of (i) defining extra logical formulas (*ghost* variables) and (ii) instrumenting the model with these formulas.

4.1 Ghost Variables and Model Instrumentation

Ghost variables, introduced below, do not appear in the contract's state variables, nor do they impact the Cubicle model outside of property verification.

```
array Out[proc] : int
array In[proc] : int
var Old_HBid : int
```

The variables Out and In are for property (b). In is an array storing how much each client (aka process) bids, and Out stores how much they get back if/when they call withdraw. Old_HBid tracks the old highest bid for property (a). The code below is the instrumented model. Transition withdraw has been omitted since it is not instrumented.

```
transition bid(i)
requires { Ended = False && Cmd = Bid && i = Sender
           && i <> HBidder && PR[i] = 0 && Value > HBid}
{
  HBid := Value;
  HBidder := i;
```

```
    PR[HBidder] := HBid;
    Cmd := None;
    Old_HBid := HBid;
    In[i] := In[i] + Value;
}
transition value(i)
requires { Cmd = Send && Recv = i }
{ Cmd := None;
  Out[i] := Out[i] + Value;
}
```

The *ghost* variables appear only in the action parts of the transitions. The bid transition updates HBid to set a new highest bid value. To keep track of what the old value was, Old_HBid is set to HBid's value. In is updated for the new bidder with their bid value. The transition value is instrumented instead of withdraw, since the most important action, giving the client back their money, happens during value. It uses Out to keep track of the money that's been returned.

The ghost variables are also part of the initial state declaration.

```
init (i) { Ended = False && HBid = 0 && Cmd = None
           && PR[i] = 0 && In[i] = 0 && Out[i] = 0
           && Old_HBid = 0 }
```

That is to say, the auction hasn't ended, there is no winning bid, the contract isn't doing anything, and no one has bid and subsequently withdrawn money.

4.2 Defining Properties

Once the code is instrumented, we can introduce the safety properties we want Cubicle to check.

Property (a): New bids are higher

The first property is "*Each new winning bid is superior to the old winning bid*". This property can be easily defined by the following unsafe formula which uses only the ghost variables Old_HBid and HBid.

```
unsafe () { Old_HBid > HBid }
```

Checking property (a) with the above formula simply means declaring Old_HBid being superior to HBid as unsafe, but only if the model was correctly instrumented with these variables.

Property (b): Do I lose money?

Defining this property is less obvious. While ghost variables have been introduced to keep track of money exchanges between users and the contract, another problem is the lack of precision of the sentence. When should we check that a user did not lose money? At the end of the auction? If so, when do we consider the auction to *really* be over?

We will make these issues more concrete in the next section. In particular, we shall explain how we arrived at the following formulation of property (b):

```
unsafe (i) { Ended = True && i <> HBidder && PR[i] = 0
             && Cmd = None && Out[i] < In[i] }
```

5 Interpreting Cubicle Error Traces

As stated previously, the tricky property is "*I do not lose money*". The logical implication is that if the auction is over, (Ended = True), then your Out isn't less than your In.

```
unsafe (i) { Ended = True && Out[i] < In[i] }
```

Except Cubicle prints the following error trace:

```
Error trace: Init -> call_bid(#1) -> bid(#1) ->
             call_finish(#1) -> finish_auction() -> unsafe
UNSAFE !
```

Upon further inspection, it becomes obvious why this state is reached. This is true for every client, even the winner, who technically does *lose* money, so to speak. We modify our unsafe state to the following by adding that the process cannot be the winner (HBidder <> i).

```
unsafe (i) { Ended = True && i <> HBidder &&
             Out[i] < In[i] }
```

However, Cubicle still says

```
Error trace: Init -> call_bid(#1) -> bid(#1) ->
             call_bid(#2) -> bid(#2) ->
             call_finish(#1) -> finish_auction() -> unsafe
UNSAFE !
```

as what's missing is checking whether or not a client withdrew their bid. We incorporate that check below.

```
unsafe (i) { Ended = True && i <> HBidder &&
             PR[i] = 0 && Out[i] < In[i] }
```

but Cubicle can still reach that state:

```
Error trace: Init -> call_bid(#1) -> bid(#1) ->
             call_bid(#2) -> bid(#2) ->
             call_finish(#1) -> finish_auction() ->
             call_withdraw(#1) -> withdraw(#1) -> unsafe
UNSAFE !
```

Once the smart contract has completely finished every action associated with a function (i.e. transition), it resets Cmd to None, which we haven't checked for. We add that to our unsafe state.

```
unsafe (i) { Ended = True && i <> HBidder && PR[i] = 0
             && Cmd = None && Out[i] < In[i] }
```

The above is the correct implementation of “I do not lose money”. This time Cubicle replies **Safe**

But error traces aren’t always the result of incorrectly written unsafe states. The automaton in Fig. 1 is incorrect. Modeling its bid requirements in Cubicle gives the following model and subsequent error trace:

```
transition bid(i)
requires { Ended = False && Cmd = Bid && i = Sender
          && i <> HBidder && Value > HBid}
...

Unsafe trace: call_bid(#1) -> bid(#1) -> call_bid(#3) ->
              bid(#3) -> call_bid(#1) -> bid(#1) ->
              call_bid(#2) -> bid(#2) -> call_finish(#2) ->
              finish_auction() -> call_withdraw(#1)->
              withdraw(#1) -> value(#1) -> unsafe
UNSAFE !
```

This is due to the requirements in Fig. 1 for bid not matching bid’s effects on the state variables. When a client is outbid, they can’t bid again without first having withdrawn their old bid. This is seen in Fig. 1 when bid sets PR_i to v with = instead of +=. This means that if an old bid wasn’t withdrawn, its value will be overwritten and the client will lose that amount. The way to fix this is to add $\mathsf{PR}_i = 0$ to bid’s requirements in Fig. 1

6 Related Work

By their nature, smart contracts lend themselves well to formal verification. Numerous approaches have been used to verify smart contracts. In [11], the authors create a three-fold model of Ethereum smart contracts and apply model checking in order to verify them. The authors of [9] introduce a finite-state machine model of smart contracts along with predefined design patterns, allowing developers to conceive smart contracts as finite-state machines, which can then be translated to Solidity. Deductive verification platforms like Why3 are used to verify specific properties of smart contracts. The authors of [10] use Why3 to verify RTE properties like integer overflow and functional properties, like the success of a transaction. Interactive proof assistants are also used in smart contract verification, such as Isabelle/HOL in [4] in order to check Ethereum bytecode. A major trend is the focus on a specific language, most notably Solidity, whereas we propose a more general framework, not tied to any concrete language. Our application of parameterized model-checking allows us to address the parametric aspects of smart contracts and treat them as concurrent programs.

7 Conclusion and Future Work

In this paper we proposed a two-layer framework for smart contract verification with the model checker Cubicle. This method implements a model of the smart contract itself and the blockchain transaction mechanism behind it. Our method introduces a way of verifying various types of functional properties linked to a smart contract as Cubicle safety properties. Since this is done through ghost variables and model instrumentation, it has no impact on the original smart contract itself, meaning it is independent of any particular smart contract language, and is therefore generalizable and usable for multiple smart contract languages. We also describe a way of interpreting potential error traces generated by Cubicle, and how they can aid in the development of a smart contract. An immediate line of future work is to automate this stepwise process. We need to define an abstract high-level language to express the properties to be checked by Cubicle. From this language, the ghost variables will be automatically generated to instrument the Cubicle code. Furthermore, we would also like to consider automatic translation of Solidity or Michelson code to Cubicle.

References

1. Mi-Cho-Coq: formalisation of the Michelson language using the Coq proof assistant. https://gitlab.com/nomadic-labs/mi-cho-coq
2. Solidity Common Patterns. https://solidity.readthedocs.io/en/v0.5.10/common-patterns.html#state-machine
3. State of DApps website. https://www.stateofthedapps.com/stats
4. Amani, S., Bégel, M., Bortin, M., Staples, M.: Towards verifying ethereum smart contract bytecode in Isabelle/HOL. In: Proceedings of the 7th ACM SIGPLAN International Conference on Certified Programs and Proofs, pp. 66–77. ACM (2018)
5. Bhargavan, K., et al.: Formal verification of smart contracts: short paper. In: Proceedings of the 2016 ACM Workshop on Programming Languages and Analysis for Security, pp. 91–96. ACM (2016)
6. Bigi, G., Bracciali, A., Meacci, G., Tuosto, E.: Validation of decentralised smart contracts through game theory and formal methods. In: Bodei, C., Ferrari, G.-L., Priami, C. (eds.) Programming Languages with Applications to Biology and Security. LNCS, vol. 9465, pp. 142–161. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-25527-9_11
7. Conchon, S., Goel, A., Krstić, S., Mebsout, A., Zaïdi, F.: Cubicle: a parallel SMT-based model checker for parameterized systems. In: Madhusudan, P., Seshia, S.A. (eds.) CAV 2012. LNCS, vol. 7358, pp. 718–724. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-31424-7_55
8. Conchon, S., Mebsout, A., Zaïdi, F.: Vérification de systèmes paramétrés avec Cubicle. In: JFLA, Aussois, France, February 2013
9. Mavridou, A., Laszka, A.: Designing secure ethereum smart contracts: a finite state machine based approach. In: Meiklejohn, S., Sako, K. (eds.) FC 2018. LNCS, vol. 10957, pp. 523–540. Springer, Heidelberg (2018). https://doi.org/10.1007/978-3-662-58387-6_28

10. Nehai, Z., Bobot, F.: Deductive proof of ethereum smart contracts using Why3. arXiv preprint arXiv:1904.11281 (2019)
11. Nehai, Z., Piriou, P.-Y., Daumas, F.: Model-checking of smart contracts. In: IEEE International Conference on Blockchain, pp. 980–987 (2018)
12. Reitwiessner, C.: Formal verification for solidity contracts
13. Sergey, I., Hobor, A.: A concurrent perspective on smart contracts. In: Brenner, M., Rohloff, K., Bonneau, J., Miller, A., Ryan, P.Y.A., Teague, V., Bracciali, A., Sala, M., Pintore, F., Jakobsson, M. (eds.) FC 2017. LNCS, vol. 10323, pp. 478–493. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-70278-0_30