Smart Contracts – A Killer Application for Deductive Source Code Verification

Wolfgang Ahrendt, Gordon J. Pace, Gerardo Schneider

Abstract Smart contracts are agreements between parties which, not only describe the ideal behaviour expected from those parties, but also automates such ideal performance. Blockchain, and similar distributed ledger technologies have enabled the realisation of smart contracts without the need of trusted parties — typically using computer programs which have access to digital assets to describe smart contracts, storing and executing them in a transparent and immutable manner on a blockchain. Many approaches have adopted fully fledged programming languages to describe smart contract, thus inheriting from software the challenge of correctness and verification — just as in software systems, in smart contracts mistakes happen easily, leading to unintended and undesirable behaviour. Such wrong behaviour may show accidentally, but as the contract code is public, malicious users can seek for vulnerabilities to exploit, causing severe damage. This is witnessed by the increasing number of real world incidents, many leading to huge financial losses. As in critical software, the formal verification of smart contracts is thus paramount. In this paper we argue for the use of deductive software verification as a way to increase confidence in the correctness of smart contracts. We describe challenges and opportunities, and a concrete research program, for deductive source code level verification, focussing on the most widely used smart contract platform and language, Ethereum and Solidity.

Wolfgang Ahrendt
Chalmers University of Technology, Sweden, e-mail: `ahrendt@chalmers.se`

Gordon J. Pace
University of Malta, Malta, e-mail: `gordon.pace@um.edu.mt`

Gerardo Schneider
University of Gothenburg, Sweden, e-mail: `gerardo@cse.gu.se`

1 The Blockchain and Smart Contracts

Blockchain refers to a specific data structure as well as to an architecture for maintaining that data structure. The blockchain *data structure* is essentially a list ('chain') of lists ('blocks'), but augmented with a combination of hierarchical and chained crypto hashing, in order to (i) ensure that appending blocks can only be performed by consensus; and (ii) making changes on previous too computationally expensive to be tractable. The blockchain *architecture* is an open, distributed ledger that can record transactions between untrusted parties, in a permanent, transparent, and cryptographically secured way, without relying on any central authority. Bitcoin [31] was the first instantiation of blockchain, and was used for what has probably become the most widely recognised application of blockchain — that of cryptocurrencies. However, blockchain technology has a much wider and rapidly growing set of applications which are likely to play an important role in the future of the digital society. On the forefront of these are *smart contracts*.

A *smart contract* is intended to digitally facilitate and enforce the negotiation or performance of an agreement between all parties which choose to engage with it. Effectively, many smart contract implementations are computer programs which, using the blockchain, are stored in a manner that ensures immutability, i.e., they cannot be changed by any of the parties (unless mutability is implemented as part of the smart contract itself) and transparency, i.e. visible by the parties involved. The execution of smart contracts is performed in the blockchain network, by workers which earn some cryptocurrency in return, but in a manner which depends on no individual point of trust.

The by far most popular smart contract platform is *Ethereum*[1]. It was created by Vitalik Buterin and is now a community effort coordinated by the Ethereum Foundation. The Ethereum blockchain features its own cryptocurrency, Ether. Smart contracts are executed by the *Ethereum Virtual Machine* (EVM), a distributed virtual machine interpreting a bytecode-level smart contract language, called *EVM bytecode*. In order to have the code executed by workers in the blockchain network, the workers earn *'gas'* (which is traded with Ether). Each EVM instruction costs a fixed amount of gas. The caller pays for the execution by paying for the gas (in fact, in addition they also choose what gas price, in Ether, they are willing to pay), to support its execution. If a transaction runs out of gas payment, the transaction is aborted and leaves no side effect, though still losing the money used to pay for gas.

EVM code is too low-level (for most developers) to program in it directly. Similarly, it is too low level to allow inspection by (most) potential users of a contract. Instead, it is the target language for compilation from higher level languages, of which *Solidity* is the one which is used most widely. Ethereum smart contracts are largely written in Solidity, and inspected in that form by users considering to engage with a contract. The language borrows some syntactic flavour from JavaScript syntax. For instance, consider the smart contract snippets written in Solidity and shown in Listing 1. The `Auction` contract provides a protocol for regulating how the auc-

[1] `www.ethereum.org`

```
contract Auction {
  bool public auctionOpen = true;
  uint public currentBid = 0;
  address private auctionOwner;
  address private currentBidder = address(0);

  function Auction() public {
    auctionOwner = msg.sender;
    ...
  }

  function placeBid() public payable {
    // The auction must still be open
    require (auctionOpen);

    // The new bid must be higher than the current one
    require (msg.value > currentBid, "bid too low");

    // Remember the current bidder
    address previousBidder = currentBidder;
    uint previousBid = currentBid;

    currentBidder = msg.sender;
    currentBid = msg.value;

    // If there was previous bid, return the money to that bidder
    if (previousBidder != address(0)) {
      previousBidder.transfer(previousBid);
    }

  }

  function closeAuction() public {
    require (msg.sender == auctionOwner);
    auctionOpen = false;
    ...
  }

  ...
}
```

Listing 1: Snippets from a smart contract regulating an auction.

tion will take place. Once the code is set up on the blockchain (in compiled form), the participants are guaranteed that the logic of the auction process as described in the contract will be adhered to, thus ensuring certain guarantees e.g. only the auction creator may decide to close the auction.

A Solidity contract offers typically several *functions* (comparable to *methods* in the object-oriented setting) which can be called by anyone via the underlying blockchain system. For instance, in the auction smart contract, there is the function `placeBid` is called to make a new bid and `closeAuction` is used to close down the auction. More precisely, the caller is either an external account (signing the call with the private key of the account) or another contract. Any kind of information can be sent as parameter of the call, but in particular, a call can send cryptocurrency to the contract. The contract will then execute the called function, manipulating the local book-keeping data as well as transferring value, or any other kind of information, to other accounts or other contracts. For instance, the `placeBid` function will receive funds (hence marked `payable`) when called, and its logic will then (i) ensure that the new bid is higher than the current one; (ii) the previous bidder (if any) will have their bid returned; (iii) the new bid and the bidder's address are recorded. It is worth noting that if the argument passed to `require` does not hold, the whole execution fails and is reverted, thus not allowing the funds transfer. Reverting the execution results in rolling back the state of the smart contract to its original state to when a function was called from outside (i.e. if a function in a smart contract calls another which fails, any execution already done by the original function is also reverted). Contracts strongly encapsulate their local data. Even if a contract variable is labelled as `public`, it is still *not writable* from outside the contract. It only means that the variable is (indirectly) *readable*, through a getter-method that is generated during compilation.

The whole purpose of smart contracts is to describe and automate an agreed exchange of values and information over the internet, in a transparent way, for instance, any user knows that the auction is fair in the sense that a higher bid by *any* user is always accepted as the new winning bid. The smart contract, once enacted acts in itself as an entity on the blockchain, being able to receive or dispense funds (as regulated by its code).

The different participants are identified solely by their respective public key, which makes it easy to securely pass around encrypted or electronically signed information wherever appropriate. Whenever a function of a contract is called, the identity of the caller is sent along implicitly, and can be used by the contract in its internal bookkeeping, for call-backs, or for passing the caller identity on to some other contract.

To look further into the logic of the auction example, the code keeps track of whether the auction is open, the current winning bid, the bidder and the person who owns the auction, who corresponds to the one who enacted the contract (thus triggering the constructor of the contract). As long as the auction is open, any user may place a bid higher than the current winning one, until the owner of the auction decides to close it. Additional logic may guarantee that, for instance, the auction can only be closed after a certain period of inactivity. Transparency is the key attraction here, since inspection and analysis of the code shows that, for instance: (i) once closed, an auction may never be reopened; (ii) only the owner of the auction may close it; (iii) the funds stored in the auction smart contract match the value stored in the `currentBid` variable.

Relating the smart contract to the equivalent natural language legal contract one could have enacted in the real world instead, we note that: (i) unlike a legal contract, the smart contract does not only regulate behaviour, but also *performs* it — it guarantees, rather than makes illegal properties such as the fact that everyone gets to see the real highest bid, and that indeed the highest bidder (rather than a close relative) is selected as winner; and (ii) the execution of the contract, in a way corresponds to a *negotiation* process, i.e., the final mutual obligations of the seller (the auction owner) and the selected winner may include or excluding certain warranties for the item sold, a warranty timeout, pre-payment before and full payment after pick-up, and so on. (At the same time, smart contracts lack many concepts which important for real legal contracts, like, for instance, prohibitions.)

Many smart contracts were created, mainly in the very recent years. The blockchain of the most popular smart contract framework, Ethereum, contains around one million smart contracts (970,898 as of December 26, 2017). Clearly, this still young technology has reached a wide spread in a short time. The applications are virtually endless, and include even integration with the Internet of Things (IoT). For instance, the Swiss company slock.it[2] offers renting of apartments, where smart contracts organise not only the agreement and payment, but also physical access through door locks that are connected to the internet and controlled by the smart contract.

There is a growing number of sectors, private as well as public, which are heavily investigating the future exploitation of blockchain and smart contracts, for innovative ways of doing business, of sharing and tracing data, of executing advanced transactions, of digital governance, of exploiting to the Internet of Things, and of executing agreements between parties, to name a few. The Enterprise Ethereum Alliance (EEA)[3] connects several hundreds of companies with Ethereum subject matter experts. (Note that this is only a fraction of the creators and users of Ethereum smart contracts.) EEA organises organisations with a particular, mostly commercial, interest in smart contracts, and include many prominent companies like American Family Insurance, AMD, BP, CISCO, Credit Suisse, HP, ING, Intel, J.P.Morgan, MasterCard, Microsoft, Rabobank, Samsung, Shell, TIBCO, and UBS, to name a few. But there are also other groupings, like R3[4], a consortium of over 200 companies and regulators which build their own blockchain, Corda, with an according smart contract language, in order to, how they put it, *'transform the way the world does business'*. Members of the R3 partner network include (notably overlapping with the above) Amazon, HP, Intel, LG, Microsoft, and Oracle. All these developments are strong indicators that smart contracts are here to stay, and that smart contract safety is a significant issue.

2 Faulty Smart Contracts

Just like all pieces of software, smart contracts can, and do, suffer from programming errors, meaning that the code can deviate from the expected behaviour. Unlike in many software domains, the code of smart contracts is openly readable, and can be inspected by everyone before using it. And yet, it is well known that many errors are difficult to spot by inspection only. Most existing smart contract programming languages are Turing-complete[5], giving expressiveness and power, but making it

[2] `slock.it`

[3] `entethalliance.org`

[4] `www.r3.com`

[5] As opposed to usual Turing complete languages, executions of (Ethereum) contracts always terminate, because each external call specifies a 'gas limit', which effectively is an upper bound for the computational effort to be spent. However, as opposed to primitive recursive functions, Ethe-

more difficult to always understand the code fully, or to verify its correctness. There are many potential causes of programming errors, like numbers getting out of range, unintuitive semantics of certain language features, or intricate mismatches between internal bookkeeping (in the local data) and external bookkeeping (in the blockchain), to name just a few.

Erroneous behaviour may not be intended by the creator nor by the user of a contract. It is also possible that a malicious contract creator writes code to build expectations with obfuscated means to ensure that they will not be fulfilled. However, most errors are probably not intended by the creator of the contract, but discovered by a malicious user who then exploits them. In all of these scenarios, what is special in this application domain is that the parties using an erroneous smart contract can loose substantial value (typically cryptocurrency) at once, in big volumes, and that in an irreversible way (as blockchain transactions are permanent, and no authority has the power to undo them). Errors in smart contracts have already caused substantial financial damage, in some cases millions of dollars. Some famous bugs that have made the news include the DAO [28] and the Parity Wallet [9], and the two recent multi-million Ethereum bugs have led to losses equivalent to millions of dollars [18, 34]. Many more bugs have been detected and reported elsewhere [4], and some analysts claim that there are more than 30,000 buggy smart contracts on the Ethereum network [27]. All these reports just witness what many where afraid of: that it is easy to get smart contracts wrong, and that the consequences of errors can be severe.

The research community and practitioners, have already started to react to this problem by proposing different solutions. Some solutions go into the direction of creating new programming languages which are less expressive and more verification-friendly (see for instance [26] and references therein), while others propose to adapt existing or develop new verification techniques for existing programming languages. We give an overview of the latter in the next section.

3 Approaches to Smart Contract Verification: The Landscape

As argued in the previous section, given the finance-critical nature of many smart contracts, the need for verification of smart contracts is crucial, and interestingly although there is still limited foundational work and academic results addressing the challenge (perhaps because smart contracts are perceived to be no different than normal software), tool-development in the field to support smart contract developers is surprisingly active. In this section we look at the spectrum of verification techniques and tools developed for smart contract analysis and verification going beyond traditional testing and debugging support.

Dynamic analysis or runtime verification [24] have now long been touted as practical verification techniques which scale up to be used on real-world systems. There

reum contracts do not themselves imply any limit on the computation. It is only the caller of the contract who provides the limit.

is limited research and tools applying these techniques for smart contracts, perhaps due to the overheads which runtime verification introduces on the system at runtime. On smart contract platforms such as Ethereum, these overheads translate to additional gas consumption, and hence the cost of executing the smart contract. Ellul et al. [13] have developed ContractLarva[6], a tool which allows for automated injection of runtime monitors into an existing smart contract written in Solidity to verify correctness at runtime. Related techniques have been developed by Idelberger et al. [22, 37, 16, 22], where the monitors are synthesised from declarative descriptions to regulate events typically coming from real-world systems rather than regulate smart contracts themselves. Similarly, Garcia et al. [15] have developed techniques using BPMN-based specifications on Ethereum, while Prybila et al. [33] have a similar solution for Bitcoin. Both approaches allow the regulation of business process models using smart contracts. Technically, although this and the previous approaches can be used to monitor events resulting from other smart contracts, they provide no automated means of instrumenting synchronisation between their monitors and the monitored contracts.

In the context of smart contracts, runtime overheads do not only cost time, but, more importantly, gas, i.e., money. In order to avoid the overheads induced by runtime monitoring, compile-time techniques may be more attractive. Moreover, compile-time techniques analyse all possible executions, rather than only the ones which where observed. Many of the tools available out there fall under the class of syntactic analysis, analysing the structure of the code with little or no semantic information to identify features which may indicate vulnerabilities in a Lint-like manner. There are various such tools out there for smart contracts written in Solidity, including Solcheck[7], Solint[8], Solium[9] and Solhint[10], but many of these tools appear to simply replicate known syntactic analysis techniques from imperative languages in the context of smart contracts.

Static analysis techniques, which enrich this analysis using semantic information can be more effective in identifying potential problems with a system but require more effort to scale up for the verification of large systems. One major challenge here is that the semantics of smart contract languages are, at best, informally explained, and typically by resorting to explaining how they work at the level of the virtual machine on which they are executed. A formal semantics for the Ethereum platform is the KEVM formal semantics [19], which formalises the bytecode assembly on the Ethereum Virtual Machine (EVM). Another formalisation was recently developed by Grishchenko et al. [17], also at the bytecode level, giving a small-step semantics in F*. Another semantics at the virtual machine level was given in [20], allowing reasoning about smart contracts to be performed using the interactive theorem prover Isabelle/HOL.

[6] See `https://github.com/gordonpace/contractLarva`.

[7] See `https://github.com/federicobond/solcheck`.

[8] See `https://github.com/SilentCicero/solint`.

[9] See `https://github.com/duaraghav8/Solium`.

[10] See `https://github.com/protofire/solhint`.

One can categorise these static techniques into two: (i) approaches which use static analysis to identify a particular class of typical vulnerabilities (e.g. gas leaks, reentrancy problems); and (ii) specification-specific static analysis, particularly useful for the verification of smart contracts against a business-logic specification.

The former, typically addressing non-functional properties have been successfully deployed in many domains since they have been shown to scale up more readily — and this shows in the domain of smart contracts, where one finds a plethora of such tools. Different approaches have been taken to try to identify different types of vulnerabilities. For instance Fröwis et al. [14] try to identify cases where the control-flow of a smart contract is matable, which is typically not desirable. Luu et al. [25] have developed a tool OYENTE which uses symbolic execution to identify a whole class of possible issues, including reentrancy detection. Similarly, Mythril[11] [30] uses concolic analysis, taint analysis and control-flow analysis for security vulnerability detection. SmartCheck[12] uses a combination of lint-like and static analysis techniques to find common vulnerabilities. Many of these approaches work at the bytecode level, thus also allowing the verification of compiled contracts. In contrast, Bhargavan et al. [8] start at the source (Solidity) level and translate into F*, although they also use decompilation techniques to go from bytecode to F*. The motivation is to allow verification within F*. However, in contrast to what the title of the paper suggests, no actual verification of resulting F* is reported in the paper. The authors label their work as preliminary.

In contrast, static analysis of smart contracts at a business-logic level is still a largely neglected field of study, whether it is analysis at a code structure level (e.g. checking pre-/post-conditions or invariants of a system) or at a system-level (e.g. checking temporal logic properties which should hold along all execution paths of the system). Bai et al. [5] take a model checking approach, building a model of a particular smart contract and verifying it using the model checker SPIN. Although using this approach one can prove general temporal properties of the system, there is a huge gap between the level of abstraction of the smart contracts and the manually constructed model used for verification. Abdellatif et al. [1] model smart contracts using timed automata and verify their correctness, but take an ambitious approach of also modelling the underlying blockchain, including the mining process. Using probabilistic model checking, they verify properties such as the likelihood of a hacker using transaction ordering attacks. In this manner, this approach goes one step further in that they do not assume immediate writing of the information to the underlying blockchain. On the other hand, just as in the previous work, there is a gap between the smart contract model used for verification and the actual smart contract code.

While system-level temporal properties are ideal to reason at a system-wide level from an external perspective, most analysis is done post-development, with the system organisation and logic already in place. The system's architect and developers would have an understanding of how the individual parts fit together to guarantee

[11] See `https://github.com/ConsenSys/mythril`.

[12] See `https://tool.smartdec.net`.

the overall (integrated) logic of the system. Thus, finer-grained implementation-specific specifications at the code structure level, such as pre- and post-conditions and system invariants, are typically also desirable to allow developers to understand whether the parts are working as expected, and if not which parts are, in some way, broken and leading to failures. The relationship between the internal data and the transaction history is particularly desirable in the context of smart contracts, especially since smart contracts act like API calls which may be invoked independently of each other. Contrast this with monolithic systems which have a predetermined control-flow (the main function) and where user interaction affects which branches of the structure to follow. Despite this, to date we are not aware of any work done in functional verification of smart contracts at this code structure level, and this is where we position our work in this paper.

4 Towards a Deductive Source Code Verifier for Smart Contracts

Deductive program verification has been around for nearly 50 years, although a number of developments during the past 15 years have brought dramatic changes to what can be achieved. Contemporary verification tools support main-stream programming languages such as C [23], Java [3], or C# [6]. They reason directly on the source code level, support source code level specification languages (with pre/post-conditions and invariants), feature high automation (in contrast to verifiers based on higher-order logics), and provide rich graphical user interfaces.

In this section, we propose a research agenda which will provide the artefacts and tools for specification and deductive verification of smart contracts on the source code level. At this point, we aim at the by far most widely used smart contract framework, Ethereum, and at the most widely used programming language, Solidity. Clearly, these choices will have to be re-evaluated as we move forward, in the light of the very dynamic developments in this domain. Generally, targeting a wide spread platform and language is likely to boost the impact of this agenda on future smart contract practice, even more so as the Ethereum/Solidity community outspokenly asks for the involvement and contribution of formal methods.

In summary, we aim at a new specification language, a new program logic, and a new verification system for a concept (smart contract) and language (Solidity) for which comparable artefacts do not yet exist.

4.1 Challenges

Smart (Ethereum) contracts in general and Solidity in particular present various challenges to verification. We discuss these challenges in the following. Note that many of the Solidity features discussed here are also features of the underlying

EVM bytecode, and partly also (at least in similar form) of other smart contract frameworks.

The Ethereum blockchain has its own *built-in cryptocurrency*, called Ether, currently the world's second biggest cryptocurrency after Bitcoin. Solidity (and the underlying EVM) supports the transfer of cryptocurrency between users and contracts, as well as among contracts. This is different from passing around other pieces of information. The attempt to transfer Ether is only accepted if the (block solving) worker can validate that the sender has a non-negative Ether balance in the blockchain after the transfer. If the validation fails, the transfer will not take place, and the entire surrounding transaction (see below) is aborted. Another difference is that the currency balance of accounts and contracts is stored as global sate, whereas other data is encapsulated in the contracts.

Solidity features a *transaction* mechanism. Each external call of a contract function starts a transaction. If during the execution of the transaction (which may include local computations, contract-triggered calls, and successful Ether transfers) some Ether transfer fails, this aborts the entire transaction. Also running out of gas, or the failure of an assertion (see `assert` in Listing 1), cause abortion of the ongoing transaction. All *local and global effects* of that transaction will be *undone*. Modelling the reverting of arbitrary computations poses a particular challenge to the proof system.

Gas analysis is an interesting challenge in smart contract verification. The limited gas budget which a smart contract user sends along with each call means that, in this domain, the resource consumption very directly affects the functionality of the contract. However, the precise consumption is defined on the level of EVM code instructions, not on the level of higher level languages translating to EVM, like Solidity. Therefore, for a quantitative gas analysis, the exact compiler version has to be considered, and analysed, to predict gas consumption of any higher-level (also Solidity) code. At the same time, out-of-gas exceptions do not generally indicate errors in the smart contract itself[13], but rather they are caused by an insufficient gas budget provided by the caller. As the gas budget is given from outside the smart contracts, it cannot be used in static analysis.

Cryptographic features are available as primitives in Solidity (and EVM). The Solidity programmer can heavily use (implicitly and explicitly) cryptographic primitives without mastering underlying cryptography. Our agenda does not address the verification of the underlying cryptographic algorithms. (That would belong to a different agenda, namely the verification of the underlying blockchain mechanisms). Nevertheless, we have to formalise the guarantees cryptographic primitives make to the application level, and use them in the verification. Examples of such guarantees are the deterministic behaviour of a hash function, uniqueness of hash values, the accuracy of (the authentication with) cryptographic signatures, and so on[14].

Solidity has *richer built-in data types* than other languages with comparable source code verification support, like Java or C. One example is mappings. (In C

[13] Although they may be the result of a *gas leak* in the code.

[14] To be precise, some of these properties do are not strictly guaranteed, but hold with sufficiently high probability to justify relying on them.

and Java, such data types are available only as libraries.) These data types require special reasoning support.

Solidity features *many* more *numeric types* than most languages. For instance, the programmer can use unsigned integers, the range of which can be freely configured, all the way from 2^8 to 2^{256} (in steps of 8 in the exponent). Overflow and underflow is silent, such that, for instance, $35-42$ results not in -7, nor is any exception thrown, but it results in $2^{256}-7$. Both the flexible size and the silent under- and overflow pose challenges to the verification. Admittedly, silent under- and overflow are also an issue in Java or C verification. However, underflow due to a lower bound of zero (as in Solidity) happens more easily in practice than underflow with respect to MININT (as in Java or C). Moreover, under- and overflows are particularly critical in smart contracts, where most numbers subject to arithmetic operations represent real value, like for instance amounts of cryptocurrency which one party owes another party. It makes a big difference whether A owes B a total of -7 Ether, as opposed to $2^{256}-7$.

Solidity features a *mixture of different call mechanisms*. In addition to usual calls (building a context stack), Solidity offers some low-(EVM)-level call mechanisms. The first, `call`, is a generic function where the name of the called function is sent as an argument, together with the proper function arguments. This mechanism is not type safe. Another variant is `delegatecall`, which is similar to `call`, but effectively imports code from the called contract syntactically, thus executing it in the local, calling context. This way, some contracts act as libraries for other contracts, compensating for the lack of real libraries in the blockchain infrastructure. Furthermore, `delegatecall` is effectively a macro expansion mechanism — not type-safe, and prone to name capture. Solidity also features two different *function return mechanisms*, value return and call-by-reference (and writing to that reference instead of returning). This mixture of different call and return mechanisms poses interesting challenges to an according program logic and calculus.

As mentioned before, a smart contract strongly encapsulates its state. The contract variables can only ever be changed by other contracts or external accounts through calls to *local* functions. At the same time, there is also contract external, global state, notably the current cryptocurrency balance of all (external and contract) accounts. Therefor, the verification needs to reason about a *combined message passing and shared memory* paradigm. At the same time, the stronger data encapsulation as compared to, say, Java or C++, is an advantage when developing compositional verification techniques.

On the other hand, it is a challenge for compositional verification that it is *not possible to pass Ether to another contract without calling it*. For instance, in Listing 1, if the address stored in `currentBidder` happens to be another contract (we cannot control whether that is the case), then `currentBidder.transfer(..)` executes the code of that contract[15]. This passing of control via seemingly elementary Ether transfer makes it more difficult to control effects locally. For instance, the exe-

[15] In general, if `c` is a contract programmed in Solidity, `c.transfer(..)` behaviour can be overridden by using a fallback function which handles any function calls not defined in that contract.

cution of the Ether receiving contract may call back into the Ether sending contract. A verification methodology has to take this into account.

4.2 Approach

As argued above, the agenda we propose builds—and expands—on the state-of-the-art of deductive software verification. Concretely, we choose the KeY approach and system [3, 2] as a starting point and blueprint for a verification approach and system for Ethereum smart contracts written in Solidity. The most elaborate KeY version, KeY-Java, allows precise reasoning about practically all language features of (sequential) Java. Recently, a bug in the main sorting routine of the OpenJDK distribution of Java, `Collection.sort()`, was identified using KeY [10]. The same bug was then found to be present also in Oracle's Java and in Android. Another KeY version in the picture is KeY-ABS [12], a verification system for the distributed object language ABS. The choice of KeY is attractive because (a) KeY is among the approaches which have proven to master verification of *feature-rich mainstream languages* (like Java), (b) the KeY approach targets the object-oriented paradigm (Java, ABS) which the *contract-oriented paradigm* is building on, and (c) KeY has been used for *compositional verification of distributed objects* (ABS), which have similarities to communicating contracts with their strong data encapsulation. Note, however, that the agenda we present in this paper does not include the translation of smart contracts to another language for which a KeY version already exists. (Actually, such a work is also under the way, but will be reported elsewhere.) Rather, we describe here the version of a new KeY approach and system, for Ethereum, probably targeting Solidity, performing *source code level verification*. We give it the working title SolidiKeY. At the core, we aim at a new specification language, program logic, calculus, and proof system for Solidity.

Such an endeavour has to take the *examination of real smart contracts* as a point of departure. Luckily, smart contracts are openly available. (The EVM code is always stored in the blockchain. Moreover, often the corresponding source code is also publicly available, and via hash codes linked with the code in the blockchain.) One can start with those contracts which have known errors (e.g., [32]). Also, it is important to engage in discussions with the smart contract community. In the end, we want to offer to smart contract developers a method and tool which can be used *prior* to deploying a contract (irreversibly) in the blockchain.

We need to provide a new, business-logic level *specification language* for the targeted smart contract language, here Solidity. Its purpose is to formalise the desired functionality of the code units (functions and transactions), the integrity conditions on the stored data, and the relation between internal data and external communication. The specification language will share some design principles with the Java Modeling Language [21]: close integration of (non-destructive) Solidity language features into property descriptions, first-order quantification over data types, pre- and post-conditions of functions, state invariants over data stored in the contract, among others. In addition, the smart contract domain requires that the data stored in a contract reflects accurately, at each point in time, the communication between the contract and users (or other contracts). Formally, this boils down to a contract invariant, constraining the relation of the internal data to the communication history. For instance, `currentBidder` and `currentBid` (in Listing 1) must *invariantly* correspond to the sender and value of the highest bid in the *call history* of the contract. More precisely, these are the sender and value of the earliest of the calls to `placeBid()` which carries an Ether value greater or equal to the values of other calls to `placeBid()`. Such contract invariants are a cornerstone for compositional verification of a network of contracts. It must be possible to define them in the language, and the proof strategies need to support them.

From the contract and its specification, proof obligations must be generated, automatically. The logic may be a version of *dynamic logic* (DL), a modal logic for reasoning about programs on the source code level. DL extends first-order logic with two modalities, $\langle p\rangle\phi$ and $[p]\phi$, where p is a program (in source code) and ϕ is another DL formula. The formula $\langle p\rangle\phi$ is true in a state s if there *exists* a terminating run of p, starting in s, resulting in a state where ϕ holds. The formula $[p]\phi$ holds in a state s if *all* terminating runs of p, starting in s, result in a state in which ϕ holds. For deterministic programs p (like smart contract functions and transactions), the only difference between the two modalities is that termination is *stated* in $\langle p\rangle\phi$, and *assumed* in $[p]\phi$, such that the two modalities correspond to total and partial correctness, respectively. In the (Ethereum) smart contract world, a transaction will always terminate because of gas restriction. However, it is relevant to distinguish 'voluntary' termination by the business-logic and termination enforced by externally given gas limits. Also, one can redefine partial correctness to mean correctness in the absence of gas exceptions, and total correctness to mean correctness in the presence of gas exceptions. DL is the base logic for the KeY approach [7]. Hoare logic can be seen as a fragment of DL, because $\{\phi\}\texttt{foo}\{\psi\}$ can be expressed in DL as $\phi \rightarrow [\texttt{foo}]\psi$. DL and Hoare logic have in common that the logic and calculus is *specific* for the target language. We propose the development of a DL for Solidity, called Solidity DL.

For reasoning about this logic, we propose to develop a *sequent calculus*, covering all features of Solidity[16]. Given a set of formulae Γ, a program π and (post)condition ϕ the sequent $\Gamma \vdash \langle\pi\rangle\phi$ holds if π, when starting in a state fulfilling all formulae in Γ, terminates in a state fulfilling ϕ. The calculus uses the *symbolic execution* paradigm (by adding explicit substitutions to the logic, capturing the effects of a computation, see [7]). One advantages of this paradigm is that proofs advance through the source code (as opposed to flow backwards as in the weakest precondition calculus), making the proofs more intuitive. For real world languages, calculi capturing all language features tend to be large, with several hundreds of rules (including the axiomatisation of all data types). On the other hand, the full Java DL calculus realised as taclets in the KeY system provides a good starting point for axiomatising a language like Solidity. Several challenges of Ethereum/So-

[16] Some deprecated and discouraged features of the language may not be supported. However, one should not exclude features simply because they are challenging for verification.

lidity verification (see Sect. 4.1) need to be addressed in the development of such a calculus. For instance, the aforementioned transaction mechanism needs to be handled, to correctly model the roll back of all effects of a transaction once it is aborted. Here one can build on the fact that KeY-Java actually supports also JavaCard, a Java dialect featuring an abortable-transaction mechanism. Even if the transaction support of KeY-Java [29] is limited to method local transactions (whereas we need to support call-stack global transaction abortion), it provides a good starting point for smart contract transaction verification. A related issue is that the calculus must be able to verify robustness against gas-used-up exceptions. Another important aspect is that the calculus shall support *compositional* contract verification, by employing the *assume-guarantee* paradigm [11]. In our context, it means that a contract's compliance with its own specification is verified while assuming the other contracts' specification[17]. Here, one can build on concepts in KeY-ABS [12].

Finally, we aim for a *verification system*, able to perform practical source code level verification of smart contracts. Let us call that system SolidiKeY. For once, this requires the mechanisation of the aforementioned calculus. Here, one can take advantage of *taclets* [35], a domain specific language for writing and executing sequent calculus rules. In addition, what needs to be developed is the generation of proof obligations in Solidity DL from specifications, and strategies for high automation of the proof search. Such strategies are not only specific for the target language, but also for community specific programming pragmatics.

5 Discussion

In this paper, we have presented the opportunities brought forward by deductive analysis for smart contract verification. The case for the necessity of verification is increasingly being accepted in the software community, but in the case of smart contracts, the case becomes substantially stronger. Despite the typically small size of such programs (as compared to many software systems orders of magnitude larger), the fact that these contracts manipulate ownership of digital assets (typically in the form of cryptocurrency or tokens) and their immutability mean that bugs can be very costly. Ironically, unlike large systems which are typically built by teams of developers using mature software engineering practice, the small size of such contracts means that single, and not necessarily highly experienced developers, are sometimes responsible for their development.

We have argued for the need for verification at a business-logic level — ensuring that the software does what it is expected to do, and in particular, the desirability of code-structure level verification i.e. pre-/post-conditions and invariants. This is the level of abstraction at which deductive analysis gives added value. Let us contrast this with alternative means of approaching such verification.

[17] But not without a small 'delay' of the considered communication, to prevent circular reasoning.

Translate and Verify: Firstly, one may ask whether building verification techniques specific to a high level language such as Solidity is necessary. Why not translate to another high level language already supported by deductive analysis tools and perform the verification on the translation? Such an approach is easier to achieve, and would still allow for verification of the types of properties we discuss in this paper — over these past months, in fact, we have been exploring the use of a Solidity-to-Java translation to verify smart contracts using the KeY verification tool for Java source code. (This will be reported elsewhere.) However, this approach comes with a number of disadvantages: (i) Verifying the translation is a major undertaking, even more so when no complete formal semantics of the source language exists. It is easy to make errors in the translation due to assumptions which may or may not hold, e.g., are the semantics of assignments in Solidity and Java equivalent?; (ii) Solidity has a number of native domain-specific features, including failure and checkpointing (allowing a program to `revert` a transaction) and implicit resource management (like `payable` function calls and `transfer` of funds). Such domain specific features can be captured better by axiomatising them directly rather than coding them in syntactic sugar via another language.

Verification at a lower-level of abstraction: Given that high-level languages such as Solidity are compiled down to assembly code working on the underlying virtual machine, axiomatising the semantics of the EVM assembly and verifying at that level of abstraction comes with a number of benefits: (i) the verification is language-agnostic and can be performed on code compiled from any other high-level language; and (ii) most of the smart contracts available on Ethereum *are not* accompanied by their source code, but would still be amenable to verification. Despite these advantages, VM-level code loses the structure which the developers used and which typically carries correspondence with the program's correctness logic. For instance, the condition of a while loop typically carries information which can be used to derive loop invariants, while conditional statements encode correctness corresponding to the dilemma rule. Also, this approach widens the gap between developers and the verification activity, as the verification is not performed on the source code developers write and understand. For instance, using the verification facility of KEVM, assertions must be formulated on the EVM level (see [19], Sect. 5.2), which is very difficult, and hardly possible for source code developers. Another aspect often neglected in the literature is the need of specification languages which have at least the abstraction level of source code, ideally higher, but certainly not lower.

Despite the fact that our proposed approach will use the semantics of the language from a functional perspective, there are a number of limitations to correctness criteria which are covered by our approach. A certain class of smart contract attacks arise from transaction reordering, which may benefit a subset of the parties involved. There is little formal work addressing this issue [36, 1], but we note that it is difficult to encode within our proposed approach a formal model able to compare outcomes under malicious transaction reordering. (This would require analysis of quantitative hyper-properties, comparing the respective profit of different schedulings.) Similarly difficult to reason about (compositionally) are malicious attacks using non-functional aspects, particularly when accessing external contracts (e.g., reentrancy attacks which use gas consumption).

Despite these constraints, we believe that the correctness of smart contracts is a challenge for which no silver bullet exists. The functional correctness at the source code level is, however, an standing duck target for deductive reasoning. We believe that the limited degree of structural complexity of smart contracts, combined with the complex nature of correctness inherent to the interaction between different functions means that deductive verification can prove to be very effective in proving correctness of non-trivial intricate properties.

The outcome of the agenda we described here will contribute to the safety of the arising digital market places. By offering languages and methods for smart contract specification, users can understand better what a smart contract should do for them, and what it should not do, prior to using the contract. More importantly, the developed verification facilities provide strong guarantees to (potential) smart contract users that the specified properties are actually met, at the same time as they can warn users about incorrect contracts.

Acknowledgements The authors would like to thank Richard Bubel, Joshua Ellul, Raúl Pardo, and Vincent Rebiscoul for fruitful discussions about Solidity contracts and their verification.

References

1. T. Abdellatif and K. L. Brousmiche. Formal verification of smart contracts based on users and blockchain behaviors models. In *2018 9th IFIP International Conference on New Technologies, Mobility and Security (NTMS)*, pages 1–5, Feb 2018.
2. W. Ahrendt, B. Beckert, D. Bruns, R. Bubel, C. Gladisch, S. Grebing, R. Hähnle, M. Hentschel, M. Herda, V. Klebanov, W. Mostowski, C. Scheben, P. H. Schmitt, and M. Ulbrich. The KeY platform for verification and analysis of Java programs. In *STTE'14*, volume 8471 of *LNCS*, pages 55–71. Springer, 2014.
3. W. Ahrendt, B. Beckert, R. Bubel, R. Hähnle, P. H. Schmitt, and M. Ulbrich, editors. *Deductive Software Verification—The KeY Book*, volume 10001 of *LNCS*. Springer, 2016.
4. N. Atzei, M. Bartoletti, and T. Cimoli. A survey of attacks on Ethereum smart contracts (sok). In *Proceedings of the 6th International Conference on Principles of Security and Trust*, volume 10204 of *LNCS*. Springer, 2017.
5. X. Bai, Z. Cheng, Z. Duan, and K. Hu. Formal modeling and verification of smart contracts. In *Proceedings of the 2018 7th International Conference on Software and Computer Applications*, ICSCA 2018, pages 322–326, New York, NY, USA, 2018. ACM.
6. M. Barnett, B.-Y. E. Chang, R. DeLine, B. Jacobs, and K. R. M. Leino. Boogie: A modular reusable verifier for object-oriented programs. In F. S. de Boer, M. M. Bonsangue, S. Graf, and W. P. de Roever, editors, *Formal Methods for Components and Objects, 4th International Symposium, FMCO 2005, Amsterdam, The Netherlands, 2005, Revised Lectures*, volume 4111 of *LNCS*. Springer, 2006.
7. B. Beckert, V. Klebanov, and B. Weiß. Dynamic logic for Java. In *Deductive Software Verification—The KeY Book*, volume 10001 of *LNCS*. Springer, 2016.
8. K. Bhargavan, A. Delignat-Lavaud, C. Fournet, A. Gollamudi, G. Gonthier, N. Kobeissi, N. Kulatova, A. Rastogi, T. Sibut-Pinote, N. Swamy, and S. Zanella-Béguelin. Formal veri-

fication of smart contracts: Short paper. In *Proceedings of the 2016 ACM Workshop on Programming Languages and Analysis for Security*, PLAS '16, New York, NY, USA, 2016. ACM.

9. L. Breidenbach, P. Daian, A. Juels, and E. G. Sirer. An in-depth look at the parity multisig bug. Appeared at "Hacking, Distributed" `http://hackingdistributed.com/2017/07/22/deep-dive-parity-bug`, Jun 2016.
10. S. de Gouw, J. Rot, F. S. de Boer, R. Bubel, and R. Hähnle. OpenJDK's Java.utils.Collection.sort() Is Broken: The Good, the Bad and the Worst Case. In *Computer Aided Verification - 27th International Conference, CAV 2015, San Francisco, USA, July 2015*, 2015.
11. W.-P. de Roever, F. de Boer, U. Hannemann, J. Hooman, Y. Lakhnech, M. Poel, and J. Zwiers. *Concurrency Verification: Introduction to Compositional and Noncompositional Methods*. Cambridge University Press, 2001.
12. C. C. Din, R. Bubel, and R. Hähnle. KeY-ABS: A deductive verification tool for the concurrent modelling language ABS. In *Automated Deduction - CADE-25*. Springer, 2015.
13. J. Ellul and G. J. Pace. ContractLarva: Runtime verification of ethereum smart contracts. In *submitted for review*, 2018.
14. M. Fröwis and R. Böhme. In code we trust? In J. Garcia-Alfaro, G. Navarro-Arribas, H. Hartenstein, and J. Herrera-Joancomartí, editors, *Data Privacy Management, Cryptocurrencies and Blockchain Technology*, volume 10436 of *LNCS*, 2017.
15. L. García-Bañuelos, A. Ponomarev, M. Dumas, and I. Weber. Optimized execution of business processes on blockchain. In J. Carmona, G. Engels, and A. Kumar, editors, *Business Process Management*, volume 10445 of *LNCS*, 2017.
16. G. Governatori, F. Idelberger, Z. Milosevic, R. Riveret, G. Sartor, and X. Xu. On legal contracts, imperative and declarative smart contracts, and blockchain systems. *Artificial Intelligence and Law*, pages 1–33, Mar. 2018.
17. I. Grishchenko, M. Maffei, and C. Schneidewind. A semantic framework for the security analysis of ethereum smart contracts. In *POST*, volume 10804 of *Lecture Notes in Computer Science*, pages 243–269. Springer, 2018.
18. A. Hern. $300m in cryptocurrency accidentally lost forever due to bug. Appeared at The Guardian `https://www.theguardian.com/technology/2017/nov/08/cryptocurrency-300m-dollars-stolen-bug-ether`, Nov 2017.
19. E. Hildenbrandt, M. Saxena, X. Zhu, N. Rodrigues, P. Daian, D. Guth, and G. Rosu. KEVM: A complete semantics of the Ethereum Virtual Machine, 2017. White paper.
20. Y. Hirai. Defining the ethereum virtual machine for interactive theorem provers. In *Financial Cryptography Workshops*, volume 10323 of *Lecture Notes in Computer Science*, pages 520–535. Springer, 2017.
21. M. Huisman, W. Ahrendt, D. Grahl, and M. Hentschel. Formal specification with the Java Modeling Language. In *Deductive Software Verification—The KeY Book*, volume 10001 of *LNCS*. Springer, 2016.
22. F. Idelberger, G. Governatori, R. Riveret, and G. Sartor. Evaluation of logic-based smart contracts for blockchain systems. In J. J. Alferes, L. Bertossi, G. Governatori, P. Fodor, and D. Roman, editors, *Rule Technologies. Research, Tools, and Applications*, volume 9718 of *LNCS*. Springer, 2016.
23. N. Kosmatov, V. Prevosto, and J. Signoles. A lesson on proof of programs with Frama-C. Invited tutorial paper. In M. Veanes and L. Viganò, editors, *Tests and Proofs*. Springer, 2013.
24. M. Leucker and C. Schallhart. A brief account of runtime verification. *The Jour. of Logic and Algebraic Progr.*, 78(5):293 – 303, 2009. The 1st Workshop on Formal Languages and Analysis of Contract-Oriented Software (FLACOS?07).
25. L. Luu, D.-H. Chu, H. Olickel, P. Saxena, and A. Hobor. Making smart contracts smarter. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, CCS '16, pages 254–269, New York, NY, USA, 2016. ACM.
26. A. Miller, Z. Cai, and S. Jha. Smart contracts and opportunities for formal methods. In *ISoLA'18*, LNCS. Springer, 2018. To appear.

27. Mix. Ethereum bug causes integer overflow in numerous erc20 smart contracts (update). Appeared at HardFork `https://thenextweb.com/hardfork/2018/04/25/ethereum-smart-contract-integer-overflow/`, Apr 2018.
28. D. Z. Morris. Blockchain-based venture capital fund hacked for $60 million. Appeared at Fortune.com `http://fortune.com/2016/06/18/blockchain-vc-fund-hacked`, Jun 2016.
29. W. Mostowski. Verifying java card programs. In *Deductive Software Verification—The KeY Book*, volume 10001 of *LNCS*. Springer, 2016.
30. B. Mueller. Smashing ethereum smart contracts for fun and real profit. In *HITB SECCONF Amsterdam*, 2018.
31. S. Nakamoto. Bitcoin: A peer-to-peer electronic cash system. 2009.
32. I. Nikolić, A. Kolluri, I. Sergey, P. Saxena, and A. Hobor. Finding the greedy, prodigal, and suicidal contracts at scale, 2018. Unpublished, submitted, avaliable at arXiv:1802.06038.
33. C. Prybila, S. Schulte, C. Hochreiner, and I. Weber. Runtime verification for business processes utilizing the bitcoin blockchain. *CoRR*, abs/1706.04404, 2017.
34. H. Qureshi. A hacker stole $31M of Ether - how it happened, and what it means for Ethereum. Appeared at FreeCodeCamp `https://medium.freecodecamp.org/a-hacker-stole-31m-of-ether-how-it-happened-and-what-it-means-for-ethereum-9e5dc29e33ce`, Jul 2017.
35. P. Rümmer and M. Ulbrich. Proof search with taclets. In *Deductive Software Verification—The KeY Book*, volume 10001 of *LNCS*. Springer, 2016.
36. I. Sergey and A. Hobor. A concurrent perspective on smart contracts. In M. Brenner, K. Rohloff, J. Bonneau, A. Miller, P. Y. Ryan, V. Teague, A. Bracciali, M. Sala, F. Pintore, and M. Jakobsson, editors, *Financial Cryptography and Data Security*, volume 10395 of *LNCS*. Springer, 2017.
37. I. Weber, X. Xu, R. Riveret, G. Governatori, A. Ponomarev, and J. Mendling. Untrusted Business Process Monitoring and Execution Using Blockchain. In *Formal Techniques for Distributed Systems*, volume 9850 of *LNCS*. Springer, 2016.