Verification-Led Smart Contracts

Richard Banach[1]

[1]School of Computer Science, University of Manchester,
Oxford Road, Manchester, M13 9PL, U.K.
richard.banach@manchester.ac.uk

Abstract. Turing complete smart contract formalisms (e.g. Solidity) are conceptually appealing, but leave the door open to the problems of verifying completely arbitrary code, a task which can be of arbitrarily high complexity or can be undecidable. We argue that a more structured approach, in which smart contract families are designed *ab initio* with efficient verifiability in mind, provide a much more practical way forward. We emphasise that the boundary between on-chain and off-chain information, which must always be determined in an application specific manner, is crucial in determining the practicability of smart contract verification. We discuss the role of refinement technologies in breaking down the complexity of smart contract verification, and illustrate the argument using the Event-B formal modelling framework and Solidity as implementation vehicle.

Keywords: Blockchain · Smart contract · Solidity · Verification · Event-B · Refinement · Rodin

1 Introduction

The introduction of Turing complete smart contract formalisms, the prime example being Solidity [35, 36, 34], made the use of distributed applications and smart contracts running on the blockchain much more appealing than hitherto. However, Turing completeness, taken literally, brings with it a host of difficulties. Not the least of these is the familiar fact that deducing anything non-trivial about *arbitrary* programs is undecidable, as enunciated for example in Rice's theorem [33, 18, 23], a textbook result.

In the present author's opinion, the popularity of smart contracts that grew markedly after the introduction of Solidity had a lot more to do with the convenience and expressivity of the source language and with the intuitive and familiar nature of its execution model in the Etherium Virtual machine [20] than with Turing completeness *per se*.

Of course, Solidity can keep the potential complexity of verification under control by charging enough gas for transactions that turn out to be difficult to verify. Such an approach is feasible and is fail-safe (insofar as questionable transaction features could simply cause the pre-deposited gas supply to run out), though it forces decisions to be made at runtime, i.e. at the time that contracts/transactions are created/executed.

In this paper, we promote an approach to smart contracts that starts with abstract formal models, and proceeds through formal refinement stages, to a concrete level that can be implemented in a smart contract execution environment. Such an approach automatically keeps verification complexity within bounds, because each model in the process has to be proved sound and consistent, and has to be proved to be a valid refinement of its predecessor. Moreover, the approach opens the door to formulating smart contracts using simpler formal frameworks, such as finite state machines or pushdown automata, potentially enhanced by measured amounts of parameterisation and more general computation (such as limited amounts of arithmetic performed at the individual automaton states). The significant amount of static verification that such an approach involves, implies that significantly less verification would need to be applied at runtime, and would be sufficient for assurance. We illustrate our proposal using the Event-B formalism, with Solidity as target on-chain implementation vehicle.

The rest of the paper is as follows. Section 2 briefly overviews some key elements of Event-B. Section 3 looks at Event-B refinement. Section 4 reviews essential elements of Solidity. We point out the similarities and differences between the two formalisms. Section 5 introduces a small example based on a payment scenario. It is first presented at an abstract level, and then Event-B refinement develops the details to a more concrete model. Section 6 then outlines the implementation of the concrete model in Solidity. Section 7 looks back, and considers variations and generalisations that the presented example suggests. Section 8 reviews related work and places the present work in context. Section 9 concludes.

Although, in this paper we focus on Solidity and Event-B for specificity and simplicity, it is clear that the same ideas can be explored in many other formalisms for smart contracts and for formal development.

2 An Overview of Event-B

In this section we recall a few essential features of Event-B, omitting a large number of facts not needed for our exposition. See [4, 31, 30, 5, 37] for a fuller exposition. The Rodin toolkit [30, 5] provides extensive mechanised support.

Event-B is a formalism for defining, refining and reasoning about discrete event systems. Its relatively uncluttered design makes it useful in many kinds of application. The syntactic unit that expresses self-contained behaviour is the MACHINE. A machine can refer to a static CONTEXT which contains arbitrary mathematical constructs to be used in the machine.[1] It declares the VARIABLES of the machine, which embody the dynamical behaviour of the machine, and crucially, it contains the INVARIANTS, which are predicates in the state variables

[1] In practice, the mathematics needs to be capable of being reasoned about by the reasoning tools within the Rodin toolset [30], which curtails the usable expressivity quite firmly.

that must remain true during all runs of the machine. Machine runs are specified implicitly via successions of EVENTS, each being of the syntactic form:

$$\mathit{EvName} \mathrel{\hat{=}} \text{WHEN} \;\; \mathit{grd} \;\; \text{THEN} \;\; xs := es \;\; \text{END}$$

In this syntactic form, *grd* is a guard, i.e. a boolean expression in the variables and constants, the truth of which enables the event to execute. (If the guard is false, then the event cannot execute, and then some other event can be selected to execute; if no event's guard is true, the machine deadlocks.) Provided the guard is true, the THEN clause defines a set of parallel updates $xs := es$ to the variables, all executed in a single atomic action. Of course, there are many additional forms of event syntax in the more definitive [4].

For machine M to be *correct*, the following proof obligation (PO) schemas must be provable:

$$\mathit{Init}(u') \Rightarrow \mathit{Inv}(u')$$

$$\mathit{Inv}(u) \land \mathit{grd}_{\mathit{Ev}}(u) \Rightarrow \exists u' \bullet \mathit{BApred}_{\mathit{Ev}}(u, u')$$

$$\mathit{Inv}(u) \land \mathit{grd}_{\mathit{Ev}}(u) \land \mathit{BApred}_{\mathit{Ev}}(u, u') \Rightarrow \mathit{Inv}(u')$$

Event-B treats *Init*ialisation as an event (with arbitrary initial state) and so the first schema demands that its after-value (indicated by priming the state variable(s) u) must establish the machine invariants, denoted by Inv. The second schema is the feasibility of event Ev. It says that if the guard $\mathit{grd}_{\mathit{Ev}}$ of event Ev and the invariants hold, then there is a well defined after-state that the event can establish, where $\mathit{BApred}_{\mathit{Ev}}$ is the before-after predicate of the event (i.e. specifying in logical form, the relation that captures the pairing between before-states and after-states of Ev). The third schema states that if a feasible update of Ev is actually executed, any after-state it might establish must satisfy the invariants Inv. When these schemas are proved to hold for all events of the machine, it follows by a straightforward induction that Inv is true in all states reachable by the machine.

3 Event-B Refinement

In Event-B, refinement is the technique by which additional detail, of both behaviour and of the state the new behaviour requires for its expression, can be added to a more abstract system model. In this manner, an early abstract view of the desired system evolves towards implementation in a provably correct way. Of course, the phrase 'provably correct' needs to be precisely defined. The details are as follows.

The Event-B notion of refinement is based on the action refinement concept [7–9, 11]. Suppose then that a(n abstract) machine M, of the form described above, is refined to a (concrete) machine MR, of a similar form. Let M have variables u and MR have variables v. Suppose the u and v state spaces are related by a refinement invariant $R(u, v)$ (also referred to as a joint invariant or gluing invariant). Suppose abstract event Ev_A of M is refined to concrete event Ev_C of MR. Then correct refinement requires the following PO schemas to hold:

$$Init_C(v') \Rightarrow \exists u' \bullet Init_A(u') \land R(u', v')$$
$$R(u, v) \land grd_{Ev_C}(v) \Rightarrow grd_{Ev_A}(u)$$
$$R(u, v) \land grd_{Ev_C}(v) \land BApred_{Ev_C}(v, v') \Rightarrow \exists u' \bullet BApred_{Ev_A}(u, u') \land R(u', v')$$

In the first of these, initialisation is refined. For each concrete initial state v', there must be an abstract intial state u' related to v' via the joint invariant. The second PO is guard strengthening. If a concrete event Ev_C is feasible, which means that its guard grd_{Ev_C} holds for a concrete state v that is the joint invariant image of an abstract state u, i.e. $R(u, v)$ holds, then u itself enables the abstract counterpart Ev_A of Ev_C. The third PO is the simulation property of refinement. If a concrete event Ev_C, enabled as just described, makes a step to an after-state v', then the abstract counterpart Ev_A of Ev_C can also make a step to an after-state u' such that the joint invariant holds for the two after-states $R(u', v')$.

The PO just described covers the 1-1 case of refinement, in which abstract steps and concrete steps are forced to correspond in the manner described. To allow greater flexibility for the introduction of detail via refinement, Event-B refinement also permits the presence of 'new' events in the refining machine *MR*. These are events whose steps have no abstract counterpart. For this to make sense, such events have to refine '`skip`', i.e. the null abstract state update, and this in turn forces the joint invariant to be, in essence, a projection from concrete states to abstract states. The PO that formalises this is the following:

$$R(u, v) \land grd_{NewEv_C}(v) \land BApred_{NewEv_C}(v, v') \Rightarrow R(u, v')$$

We see in this that because no change in u is envisaged, R must project away any difference between v and v', as stated.

If all of the above are provable for a pair of machines M and MR, then an inductive proof of simulation of any concrete execution by some abstract execution follows relatively easily.

4 A Bare Outline of Solidity

In this section we give an outline of Solidity, with a scope similar to that of the Event-B outline above. Solidity has many of the features of a typical stack + heap + inheritance based programming language, enriched with a collection of facilities for operating on the Ethereum blockchain. We structure the account in a way that parallels Section 2, since otherwise, confusion of terminology can, unfortunately, arise. (We use `teletype` font to emphasise Solidity meanings.)

The Solidity construct that corresponds to the machine is the `contract`, intentionally similar to a class in object-oriented languages. This contains the usual declarations of state variables and of `constructor`s (the analogues of intialisation in Event-B). There are also declarations of user-designed `struct`ured and `enum`erated types. The inception of a `contract` also refers to any `pragma`tic information and `import`ed constructs.

Corresponding to the Event-B event is the Solidity `function`. (There are also Solidity `event`s, which should not be confused with the Event-B usage;

we say more about them below.) A `function` defines a change of state of the `contract`. Analogously to an event's guard in Event-B, the `function` has one or more `require` clauses, which stipulate preconditions that must hold when the `function` executes. With these considerations, a `function` appears thus:

```
function name(i-params) visibilitySpecifier (o-params) {
    require(
        predicate
    )
    body
}
```

The *body* of a Solidity `function` permits a relatively standard range of sequential programming constructs to be used. Ethereum specific facilities are also provided. These permit, for example, the interaction with other contracts via external calls (accompanied by the gas to pay for their execution), and provisions for the sending and receiving of ether between `address`es or other `contract`s. And as well as the preconditions expressed in the `require` clauses, `assert` clauses can enforce the checking of state predicates in the interior of a *body*.

Solidity also allows a `contract` to declare one or more `modifier`s, which are consulted during `function` execution. These are a bit like a `function` without a *body*, the missing body being replaced by '`_;`'. When a `function` stipulates a `modifier`, its text replaces the '`_;`' in the `modifier`, thereby redefining the `function`. In particular, the `require` clauses of the `modifier` become conjoined to those of the `function`. This technique permits, for example, the consistent imposition of a collection of restrictions on when a family of `function`s might be permitted to execute.

Returning to Solidity `event`s, they are events in the style of concurrent languages, in that they have a name (and may contain some parameters). When they are `emit`ted during the course of executing a `function` *body*, the name (and its parameter collection, if any) is posted to the `contract`'s log for the transition executing the *body*, and listeners who have subscribed to the event can be informed of its presence. So, unlike Event-B events, Solidity `event`s do not change the `contract` state.

4.1 On Guards and Preconditions

The alert reader will have noted that when we spoke of Event-B events, we talked about guards, whereas for Solidity `function`s, we talked about preconditions. The distinction is not ephemeral, and hinges on the detailed formal semantics of these two concepts; see e.g. [10] for a good textbook discussion. Both concepts check a condition on entry to a state changing action, and if the condition evaluates to true, the action takes place. The distinction arises if the condition evaluates to false. For a guard, the semantics is as if the action had not been invoked at all — nothing changes, and the semantics only speaks of actions 'at runtime' whose guards are true. For mechanised reasoning systems, being able to simply ignore action occurrences whose guards are not true is an enormous convenience and leads to great efficiency.

For an action's precondition which evaluates to false for some runtime occurrence, the conventional semantics of the execution *abort*s, because the precondition expresses assumptions that are necessary for the action to make sense, and they have been found to fail. In Solidity, whose `function`s can be called within externally invoked transactions, there is no guarantee that any necessary assumptions are bound to hold for any runtime `function` invocation, so a precondition semantics is at first sight appropriate. In practice, failure of a `require` check throws an exception, and the exception implements a `revert`, which restores the state to its value before the `function` execution. Thus the semantics is close to that of a guard after all, albeit that the null action of a `function` execution that fails a `require` check is visible externally via a return code, etc.

The preceding might be viewed as something of a meandering detour, were it not for the fact that the the current Solidity documentation contains the sentence: "*Catching exceptions is not yet possible.*" which carries with it at least the suggestion that exception catching might be available in future and would be regarded as desirable. From our standpoint, of aiming to align formal development with practical implementation in systems like Solidity, being able to implement anything less trivial than skip on a failing `require` check would be regarded as retrogressive, as it would severely complicate establishing formal correspondence between a formal model of a contract and its Solidity implementation.

5 A Simple Payment Contract in Event-B

We illustrate the ideas above with a simple payment protocol case study. A *supplier* S agrees with a *customer* C to supply some goods or services, in exchange for payment. Upon completion of the work, S is due payment by C. They have agreed that if C pays by a given deadline, there is a discount of 10%. If the deadline is not met, but a later one is, the discount decreases to 5%. If payment is still missed, the full price becomes due, to be resolved by conventional means.

We create an abstract Event-B model as follows. Some parts appear in blue, which we discuss in Section 7.

$INITIALISATION_A \ \hat{=}$
  BEGIN $\ st_A := \mathsf{Working}_A \ || \ discs_A := \{0 \dots 10\}$ END

The $INITIALISATION_A$ launches the model. There are two abstract variables, the state variable st_A initialised to $\mathsf{Working}_A$ and the permitted range of discounts $discs_A$ initialised to the set $\{0 \dots 10\}$.

Event $Signal_A$ indicates the completion of the work and the demand for payment. The state becomes $\mathsf{Completed}_A$ and the discounts are unaltered.

$Signal_A \ \hat{=}$
  WHEN $\ st_A = \mathsf{Working}_A$
  THEN $\ st_A := \mathsf{Completed}_A \ || \ discs_A := \{0 \dots 10\}$
  END

The contract is completed in one of two ways. Either it completes as intended, via event $CollectY_A$, and a non-zero discount in the range $\{5 \dots 10\}$ is applied, or the demand for payment times out, and the available discounts are narrowed to $\{0\}$ in event $CollectN_A$.

$CollectY_A \ \widehat{=}$
WHEN $st_A = \mathsf{Completed}_A$
THEN $st_A := \mathsf{Done}_A \ || \ discs_A := \{5 \dots 10\}$
END

$CollectN_A \ \widehat{=}$
WHEN $st_A = \mathsf{Completed}_A$
THEN $st_A := \mathsf{Forfeit}_A \ || \ discs_A := \{0\}$
END

In Event-B we would expect to increase the trust in the model by adding invariants that express correctness properties of the model. In our case, we can write invariants that express the expected coupling between st_A and $discs_A$ under correct operation. An example of such an invariant is:

$$st_A = \mathsf{Completed}_A \Rightarrow discs_A = \{0 \dots 10\}$$

which is evidently provable given the definition of the model.

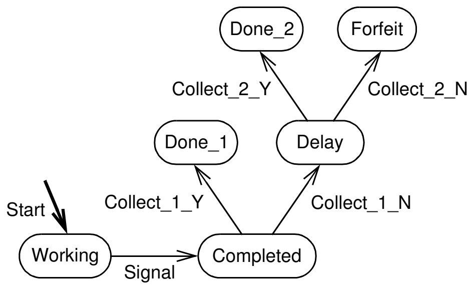

Fig. 1. State machine for the payment smart contract at the concrete level.

Once satisfied that the abstract model is correct and consistent with the requirements addressed at that level, the model is refined to include more of the originally described detail. This, more concrete model, is subscripted with $_C$, and its state machine appears in Fig. 1. The model itself is as follows.

The $INITIALISATION_C$ event is quite routine. Notice though, that the concrete variable $disc_C$ is an individual value, and not a range, as in the abstract model.

$INITIALISATION_C \ \widehat{=}$
BEGIN $st_C := \mathsf{Working}_C \ || \ disc_C := 0$ END

In accordance with the more detailed requirements at this level, the concrete model contains more detail in its states, and in its structure.

$Signal_C \ \widehat{=}$
WHEN $st_C = \mathsf{Working}_C$
THEN $st_C := \mathsf{Completed}_C \ || \ disc_C := 10$
END

As before, Signal_C indicates the completion of the work and the demand for payment. The state becomes $\mathsf{Completed}_C$ and the full discount is still available.

```
$\mathit{Collect\_1\_Y}_C \;\hat{=}$
    WHEN  $st_C = \mathsf{Completed}_C$
    THEN  $st_C := \mathsf{Done\_1}_C \;||\; disc_C := 10$
    END
$\mathit{Collect\_1\_N}_C \;\hat{=}$
    WHEN  $st_C = \mathsf{Completed}_C$
    THEN  $st_C := \mathsf{Delay}_C \;||\; disc_C := 5$
    END
```

This time, the model is more sensitive to the individual deadlines, so expiry of the first does not prompt the end of the contract. Instead, the discount reduces and a further delay is permitted.

```
$\mathit{Collect\_2\_Y}_C \;\hat{=}$
    WHEN  $st_C = \mathsf{Delay}_C$
    THEN  $st_C := \mathsf{Done\_2}_C \;||\; disc_C := 5$
    END
$\mathit{Collect\_2\_N}_C \;\hat{=}$
    WHEN  $st_C = \mathsf{Delay}_C$
    THEN  $st_C := \mathsf{Forfeit}_C \;||\; disc_C := 0$
    END
```

We claim that the concrete machine just defined is a refinement of the abstract one. Without listing all the details exhaustively, we indicate the essential points. The refinement relation is a function that maps $(st_C, disc_C)$ pairs to $(st_A, discs_A)$ pairs, mapping individual st_C values to individual st_A values, and demanding membership of $disc_C$ in the set $disc_A$. Most st_C values map to identically (or almost identically) named st_A values in the obvious way, except for Delay_C, which maps to $\mathsf{Completed}_A$. The latter enables the $\mathit{Collect\_1\_N}_C$ event to be a 'new' event as described in Section 2. It changes the concrete state —thus st_C changes from $\mathsf{Completed}_C$ to Delay_C, and $disc_C$ changes from 10 to 5— but in a way that is not visible to the abstract state through the refinement relation.

With this understood, checking the various simulation POs becomes fairly routine. So, $\mathit{INITIALISATION}_C$ is simulated by $\mathit{INITIALISATION}_A$; Signal_C is simulated by Signal_A; $\mathit{Collect\_1\_Y}_C$ and $\mathit{Collect\_2\_Y}_C$ are simulated by $\mathit{CollectY}_A$; and $\mathit{Collect\_2\_N}_C$ is simulated by $\mathit{CollectN}_A$.

These simulations ensure that the invariants proved to hold for the abstract model, continue to hold in the concrete model, provided that they are suitably interpreted through the refinement relation. For example, our abstract invariant $st_A = \mathsf{Completed}_A \Rightarrow discs_A = \{0 \ldots 10\}$ becomes (taking the model properties appropriately into account) $st_C = \mathsf{Completed}_C \Rightarrow disc_C \in \{0, 5, 10\}$. Also, we can add extra invariants pertaining to the concrete model alone, such as $st_C = \mathsf{Delay}_C \Rightarrow disc_C = 5$. Verifying all such properties can be done using Rodin.

6 Implementation in Solidity

On the assumption that all the necessary detail, along with the verification that corroborates it, has been included in the final concrete Event-B model and in the development path to it, the final implementation step in our approach is to translate it into a Solidity `contract`, which we now sketch. Thus, $INITIALISATION_C$ translates into a constructor:

```
constructor() public {
    st = Working;
    disc = 0;
}
```

The other events of the concrete model translate in like fashion:

```
function signal() external {
    require( st == Working );
    st = Completed;
    disc = 10;
}

function collect_1_Y() external {
    require( st == Completed );
    st = Done_1;
    disc = 10;
}

function collect_1_N() external {
    require( st == Completed );
    st = Delay;
    disc = 5;
}

function collect_2_Y() external {
    require( st == Delay );
    st = Done_2;
    disc = 5;
}

function collect_2_N() external {
    require( st == Delay );
    st = Forfeit;
    disc = 0;
}
```

It is clear that the translation just given is predicated on a number of assumptions (for instance, that the states are encoded as an `enum`erated type). We discuss such issues more fully in the next section.

7 Variations and Generalisations

The above small example raises a number of issues which we take some space to discus now.

We concede immediately, that for the sake of expository simplicity, we omitted any checks on the identities of the participants in the contract and on all other matters connected with security, things which in reality, do of course have the highest importance. Adding these, could be viewed as another refinement of the models we gave above.

We observe that nowhere in the contract is the actual value to be transacted mentioned (neither the currency to be used, whether ether or other). We are assuming, for the sake of the example, that such details are kept off-chain, for the sake of confidentiality. This alludes to a perennial tension in the smart contracts ecosphere, namely that while the contract code is 'public' (i.e. visible at least to any full nodes that might need to execute its code, and usually, more widely than that), the parties engaging in a contract usually prefer to keep their business confidential. In the present author's opinion, squaring this circle will prove to be a fertile ground for future innovation in the smart contracts world, particularly as smart contracts and conventional legally enforceable contracts begin to merge.

Continuing in the same vein, if, for the sake of argument, we regarded the numerical values of the discounts involved in the example above as already disclosing too much, we could remove all the blue parts from the Event-B models and the Solidity code, to yield a terser smart contract scheme. If such a view were adopted, restoring the blue parts (for example, in the off-chain part of the whole system) would again be a refinement of the terser models.

Noting that, in even the most concrete of our models the states are finite in number, as are the allowed discounts, we observe that our whole system consists of finite state machines and their refinements, much as discussed in [26]. In a non-Solidity, more bespoke and more special purpose private blockchain ecosystem, such more simply structured contracts could be encoded in much simpler ways, adapted to the application specific purpose.

Evidently, the positioning of the boundary between the on-chain and off-chain parts of a smart contract can have a significant effect on the efficiency of the runtime verification and execution of its functions. Our simple finite state example provides the most trivial instance of verification, but in a more elaborate scenario, the finite states could be connected to off-chain computations of arbitrary complexity. The fact that they would be off-chain, means that the complexity does not impact blockchain performance (provided it is executed by non-chain computational resources). The integrity of the connection between on-chain and off-chain components can be ensured, for example, by recording suitable hashes generated from the two computations in the on-chain elements. Of course, doing that in a distributed system will not be possible in a single atomic action in the general case. Achieving this properly can be formulated as another level of refinement. The techniques for refining abstract atomic actions into protocols that implement them in a distributed manner are relatively well known [12].

A contract is normally an arrangement between more than one agent. However, the Event-B models and the Solidity code above were monolithic, in the sense that there was no indication which agent (*customer* or *supplier*) was expected to execute which event/`function` (though this is relatively evident from context). This is connected with the absence of checks on identity and security details in our example, that we have mentioned already. Event-B and Rodin [4, 30] provide facilities for decomposing a monolithic refinement development into separate machines, relevant to the separate agents in a system, again engineered using refinement. This provides a principled way of integrating such concerns into the formally verified development.

For the sake of simplicity, the Solidity code given above assumed that the various `function`s would be called by the appropriate agents at the appropriate time. In particular, monitoring of the timeouts mentioned in the informal description of the contract is assumed to be delegated to the *customer*, being the agent in whose interest it is to receive early payment. To aid this approach, suitable `event`s could additionally be `emit`ted by the Solidity `function`s to assist in the monitoring of the progress of the contract by the various agents involved. Looking to the future, and to increasingly sophisticated smart/legal contract schemes, a much tighter integration of external oracles working entirely autonomously can be built into blockchain architectures. The Oraclize service [28] contributes substantially to this already.

Another source of complication, even in seemingly deterministic contracts like our example, is the potential nondeterminism that arises due to the inevitably unpredictable nature of distributed systems working, such as the PoW scheme that underpins Ethereum, Bitcoin and other blockchain architectures.

Considering the increasingly sophisticated smart/legal contract schemes just mentioned, it is not realistic to presume that in all cases, all possible playouts of the contract scheme can be foreseen in advance. So, to cope with unanticipated eventualities, the automated contract schemes will need to make provision for a number of tidy exit points that allow for termination of automated working and resolution of the contract by conventional human-mediated means.

These observations lead to an architectural model for verified smart contract development which is schematically illustrated in Fig. 2. A contract scheme is developed through a series of refinements $M \rightarrow MR \rightarrow \ldots$ etc. The various levels elaborate more complex but more implementable representations of the goals of the contract, as well as catering for a variety of successive concerns, as indicated above. In particular, representing the decomposition of the monolithic version of the contract (which enables correctness concerns to be expressed in the most direct way possible) into a version in which the various agents and their individual roles are clearly identified, should form part of this development. At a certain point, a level of detail suitable for on-chain implementation is reached, at which juncture an implementation in Solidity or other system can be built. The implementation on-chain may only represent part of what is needed, so further refinement can encompass the off-chain part of the contract, which, at an appropriate point, may itself be implemented.

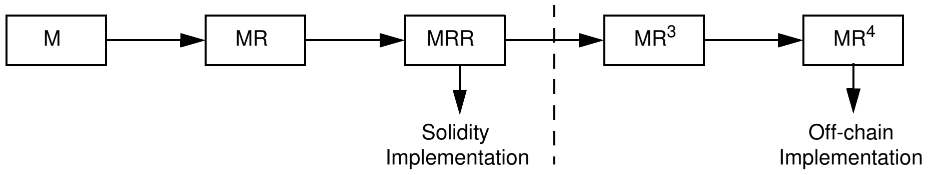

Fig. 2. A schematic illustration of a refinement chain for a formally developed smart contract. Machine M is refined to MR, which is further refined to MRR. These develop the on-chain elements of the contract, along with their correctness properties, up to the point that they capture all the required on-chain information in a sufficiently implementable way. Machine MRR can then be implemented as a `contract` in a practical system such as Solidity. Beyond the dashed line, MRR can be further refined to incorporate off-chain information relevant to the smart contract. The off-chain elements of machine MR^4 can then be implemented too.

It is worth observing that a lot of the issues to be brought into the refinement chain are independent from each other. In such cases, the relevant refinements can usually be performed in any order, and this gives rise, conceptually, to a multidimensional grid of models connected by refinements, which may be navigated in many ways. However, tools do not support such structures well, due to the syntactic complexity of capturing the multitude of ways that entities are connected together. So, for practical purposes, a linear refinement organisation is normally demanded in practice.

8 Related Work

After an early recognition that the smart contract world would profitably lend itself to, and would benefit from, formal verification approaches, the last two years have seen a vigorous grown of interest in the area, witnessed by an increasing number of publications. Many papers have appeared in the WTSC proceedings [2] and in the FC proceedings [1] since their inception, as well as a variety of other places.

Verification of the Ethereum Virtual Machine (EVM) and programs for it has been one of the key topics of interest from the beginning. The attention is warranted, since, for example, in [21] there is a table of financially significant losses ($>$ 400 USD in value) in the Ethereum smart contract system due to flaws in the implementation of the ERC20 API. The paper [21] gives a formal semantics of the EVM via the K system. In [29], the work in [21] is leveraged to create a verification tool for EVM bytecode. In [22] there is an account of a formal definition of the EVM in the Lem [27] language, which translates easily into a range of standard theorem provers. The range of work indicated promises to bring much needed precision to the implementation level of smart contracts, since without that, little reliance can be placed on how the execution of a contract might pan out.

Verification of smart contracts has attracted predictable attention, for reasons similar to those motivating [21]. In [19], an approach to verified smart contracts using runtime verification techniques is described. In [14], an approach via the F* functional language and targetted at not only runtime behaviour but also functional correctness, is perhaps closest to our approach (if in ours, all modelling were to be restricted to what we called the concrete level). In [32] it is argued that smart contracts are just concurrent systems, which is of course true, which pulls in all the verification ideas from the concurrent systems world. Also, we have already noted the approach to simpler contract structure via finite state machines, etc. [26].

An aspect not present in our simple example, but that arises in problems that are commonly treated via blockchains is the game theoretic one. Various kinds of gambling problem, auctions, elections etc., open the door to the invention of strategies to ensure the correct functioning of a contract, or to subvert its correct functioning; [16] is representative.

We cite also the interaction between on-chain and off-chain working, mediated by Oraclize [28] or otherwise, alluded to above, [32, 13], which is a developing topic. Finally, different dimensions of the interplay between smart contracts and conventional legally enforceable contracts, which we also have discussed above, attract the interest of various other authors, e.g. [6],

9 Conclusions

In the preceding sections we have outlined an essentially top-down approach to smart contracts. Of course the approach is overkill for the toy example we showed, but as contracts become more complex, and fuse with the legally enforceable kind, the dependability that formal development techniques can bring to the whole process will become needed more and more. The table of exploited Solidity and EVM vulnerabilities in [21] only strengthens this view.

The proposed approach brings to the fore a number of tensions which are worth exploring — the B-Method provides a suitable allegory. Originally, the classical B-Method [3] pioneered the formal and automated production of safety critical code, and nowadays the Atelier B tool [24, 25], maintained by Clearsy [17], is certified for such use. Certification is a sufficiently painful process that it has precluded the evolution of the underlying formalism and the adoption of new technologies in the core tool. To some extent this is circumvented by surrounding the core tool with helper tools that work with newer ideas and translate them into the older core formalism, although this is a bit ungainly [15]. One example of this is the development by Clearsy of a dialect of Event-B that can be interfaced to the core Atelier B tool.

Formalisms for dependable smart contracts face the same dilemmas. On the one hand, the desire for greater usability and acceptance create a strong pressure to evolve and improve the basic languages and frameworks used, which militates for rapid language change and enrichment, with the evident risk that unanticipated feature interaction can introduce vulnerabilities that undermine dependability. On the other hand, the desire for dependability of the system creates a pressure to *not* evolve or change the languages and frameworks used, precisely to avoid such problems.

Such issues aside, there would be no barrier to creating a dialect of a formalism such as Event-B that was aligned specifically to the formal development of smart contracts. This could include facilities for directly generating code at implementation level in a system such as Solidity (or at a lower level in the EVM). However, what this requires is stability of the underlying system, and confidence that it is sufficiently watertight. At the least, it requires precision in the semantics of any underlying system relied on, so that the precision built in to the formal layer is not undermined lower down the stack. We have noted above the ongoing evolution of the Solidity language, which prompts caution in our proposed approach. In particular, from our standpoint, catching exceptions and dealing with them in any more elaborate way other than `skip` (i.e., doing anything more than `revert`ing to the starting state of a transaction), would be considered harmful, as mentioned earlier. Nevertheless the graduated but rigorously controlled proposed approach to the development of large complex contracts could certainly yield benefits as the scale of automated contracts gets bigger in future years.

References

1. Conference on Financial Cryptography and Data Security (FC). Springer, LNCS (1997 onwards)
2. Workshop on Trustworthy Smart Contracts (WTSC). Springer, LNCS (2016 onwards)
3. Abrial, J.R.: The B-Book: Assigning Programs to Meanings. CUP (1996)
4. Abrial, J.R.: Modeling in Event-B: System and Software Engineering. CUP (2010)
5. Abrial, J.R., Butler, M., Hallerstede, S., Hoang, T., Mehta, F., Voisin, L.: Rodin: An Open Toolset for Modelling and Reasoning in Event-B. Int. J. Soft. Tools for Tech. Trans. 12, 447–466 (2010)
6. Al Khalil, F., Butler, T., O'Brien, L., Ceci, M.: Trust in Smart Contracts is a Process, As Well. In: Proc. WTSC-17. vol. 10323, pp. 510–519. Springer, LNCS (2017)
7. Back, R., Kurki-Suonio, R.: Decentralisation of Process Nets with Centralised Control. In: Proc. PODC-83. pp. 131–142. ACM (1983)
8. Back, R., Sere, K.: Stepwise Refinement of Action Systems. In: Proc. MPC-89. vol. 376, pp. 115–138. Springer, LNCS (1989)
9. Back, R., von Wright, J.: Trace Refinement of Action Systems. In: Proc. CONCUR-94. vol. 836, pp. 367–384. Springer, LNCS (1994)
10. Back, R., von Wright, J.: Refinement Calculus. Pringer (1998)
11. Back, R., Sere, K.: Superposition Refinement of Reactive Systems. Form. Asp. Comp. 8(3), 324–346 (1996)
12. Banach, R., Schellhorn, G.: Atomic Actions and their Refinements to Isolated Protocols. Form. Asp. Comp. 22, 33–61 (2010)
13. Bartoletti, M., Pompianu, L.: An Empirical Analysis of Smart Contracts: Platforms, Applications, and Design Patterns. In: Proc. WTSC-17. vol. 10323, pp. 494–509. Springer, LNCS (2017)

14. Bhargavan, K., Delignat-Lavaud, A., Fournet, C., Gollamudi, A., Gonthier, G., Kobeissi, N., Kulatova, N., Rastogi, A., Sibut-Pinote, T., Swamy, N., Zanella-Beguelin, S.: Formal Verification of Smart Contracts. In: Proc. PLAS-16. pp. 91–96. ACM (2016)
15. Burdy, L., Deharbe, D.: Teaching an Old Dog New Tricks: The Drudges of the Interactive Prover in Atelier B. In: Proc. ABZ-18. vol. 10817, pp. 415–419. Springer, LNCS (2018)
16. Chen, L., Xu, L., Shah, N., Gao, Z., Lu, Y., Shi, W.: Decentralized Execution of Smart Contracts: Agent Model Perspective and its Implications. In: Proc. WTSC-17. vol. 10323, pp. 468–477. Springer, LNCS (2017)
17. ClearSy: http://www.clearsy.com/
18. Davis, M., Weyuker, E.: Computability, Complexity and Languages. Academic Press (1983)
19. Ellul, J., Pace, G.: Runtime Verification of Ethereum Smart Contracts. In: Proc. EDCC-18, Workshop on Blockchain Dependability. pp. 158–163. IEEE (2018)
20. Ethereum: https://www.ethereum.org/
21. Hildenbrandt, E., Saxena, M., Rodrigues, N., Zhu, X., Daian, P., Guth, D., Moore, B., Park, D., Zhang, Y., Stefanescu, A., Rosu, G.: KEVM: A Complete Formal Semantics of the Ethereum Virtual Machine. In: Proc. CSFS-18. pp. 204–217. IEEE (2018)
22. Hirai, Y.: Defining the Ethereum Virtual Virtual Machine for Interacting Theorem Provers. In: Proc. WTSC-17. vol. 10323, pp. 520–535. Springer, LNCS (2017)
23. Hopcroft, J., Ullman, J.: Introduction to Automata Theory, Languages and Computation. Addison Wesley (1983)
24. Lecomte, T.: Atelier B has Turned 20. In: Proc. ABZ-16. vol. 9675, p. XVI. Springer, LNCS (2016)
25. Lecomte, T., Deharbe, D., Prun, E., Mottin, E.: Applying a Formal Method in Industry: A 25-Year Trajectory. In: Proc. SBMF-17. vol. 10623, pp. 70–87. Springer, LNCS (2017)
26. Mavridou, A., Laszka, A.: Designing Secure Ethereum Smart Contracts: A Finite State Machine Based Approach. In: Proc. FC-18. Springer, LNCS (2018)
27. Mulligan, D., Owens, S., Gray, K., Ridge, T., Sewell, P.: Lem: Reusable Engineering of Real-World Semantics. In: SIGPLAN Not. vol. 49, pp. 175–188. ACM (2014)
28. Oraclize: http://www.oraclize.it
29. Park, Y., Zhang, Y., Saxena, M., Daian, P., Rosu, G.: A Formal Verification Tool for Ethereum VM Bytecode. In: Proc. ESEC/FSE-18. pp. 912–915. ACM (2018)
30. RODIN Tool: http://www.event-b.org/
http://sourceforge.net/projects/rodin-b-sharp/
31. Sekerinski, E., Sere, K.: Program Development by Refinement: Case Studies Using the B-Method. Springer (1998)
32. Sergey, I., Hobor, A.: A Concurrent Perspective on Smart Contracts. In: Proc. WTSC-17. vol. 10323, pp. 478–493. Springer, LNCS (2017)
33. Sipser, M.: Introduction to the Theory of Computation. Thomson (2006)
34. Solidity: https://en.wikipedia.org/wiki/Solidity
35. Solidity Documentation: https://solidity.readthedocs.io
36. Solidity Github: https://github.com/ethereum/solidity
37. Voisin, L., Abrial, J.R.: The Rodin Platform has Turned Ten. In: Proc. ABZ-14. vol. 8847, pp. 1–8. Springer, LNCS (2014)