THOMSON REUTERS PRACTICAL LAW

Understanding Smart Contract Mechanics

by Ari Juels and William Marino, Jacobs Institute, Cornell Tech, with Practical Law Finance

Maintained • USA (National/Federal)

A Practice Note providing a practical explanation of how blockchain smart contracts work and an examination of issues presented by the application of traditional contract principles to smart contracts. This Note also provides a technical illustration of amendment and termination of smart contracts on the Ethereum blockchain smart contract platform.

It seems like **blockchain** is everywhere. Also known as distributed ledger technology (DLT), its uses in finance and financial services have already begun to proliferate. Among the blockchain initiatives already underway have been a number of DLT **smart contract** test cases (see Legal Updates, Firms Complete Successful Blockchain Equity Swap Smart Contract and Successful Blockchain Credit Default Swap Smart Contract Test Completed by DTCC Working Group).

DLT smart contracts are a form of computer code run in a framework that resembles execution by a trusted third party. They can accomplish a range of goals, one of which is helping enforce the terms of traditional legal agreements.

Assuring a client that a smart contract does what it purports to do could prove challenging for an attorney who is not familiar with the unique legal and mechanical issues these contracts present or with basic smart contract coding. Certain attorneys, especially in areas where test cases have been prevalent, such as swaps and derivatives, may need to become comfortable communicating with and speaking the language of contract code-writers or programmers.

This Note provides a practical explanation of how DLT smart contracts work and an examination of issues presented by the application of traditional contract principles to smart contracts, as well as a detailed nuts-and-bolts illustration of coding amendment and termination of smart contracts on Ethereum, a popular smart contract platform.

Smart Contract Basics

There are countless ways to embed promises in technology so as to make breach infeasible or unduly expensive. The classic example is the vending machine that promises to deliver a candy bar to anyone inserting a dollar. Smart contracts may soon include the blockchain-dwelling bytecode of an evergreen loan contract embedding a creditor's promise to issue a new cryptocurrency loan to a debtor who repays a prior one.

This Microsoft, Inc. video walks through the execution of a **total return swap** (TRS) "cryptoswap" on the Ethereum blockchain.

Though DLT smart contracts are still in their developmental stages, it does not seem that the question is so much whether DLT smart contracts will proliferate, but when. Certain blockchain consortia involving major banking and financial institutions have begun testing smart contract templates on blockchains.

A well-designed smart contract drives the probability of opportunistic breach toward zero as this behavior becomes impossible or at least expensive for the breacher.

The code for a smart contract is injected onto the blockchain when a personal account sends contract code to the data field of an unaddressed transaction. After this, the contract is added to a block and assigned an address, at which point its code becomes immutable. What is not immutable is the contract's state.

What differentiates smart contracts from traditional contracts is that they include protocols within which the parties perform on promises. These protocols create the ability to automatically enforce themselves, which eliminates the need for:

- Trusted intermediaries.
- Court enforcement.

As such, smart contracts have the potential to reduce transaction costs.

When promises are embedded in technology, one (perhaps the only) way to breach them is to disrupt that technology. Most smart contracts include security measures aimed at deterring this type of breach. To breach the vending machine's smart contract, you must break into its lockbox. To breach the blockchain loan contract, you must compromise the blockchain's consensus protocol.

In this sense, security mechanisms of smart contracts allow them to surpass the capabilities of paper contracts. The problem, however, is that securing contracts against disruption for the purpose of breach often means securing them against disruption of any sort, which is not always a desirable result.

It may come to light that there is a typo in the contract or that one party was defrauded during its creation. When these events arise, the parties (and sometimes courts or even the public) may find themselves wanting the contract to be performed differently or not at all. Contract law has a well-honed set of tools for undoing and altering contracts, including termination and rescission (for undoing contracts) as well as modification and reformation (for altering contracts).

However, these traditional tools often fail when applied to smart contracts. While they may successfully undo the legal agreement that a smart contract manifests, the contract may still automatically perform.

Application of Traditional Contract Principles to Smart Contracts

Termination and Rescission of Smart Contracts

Contract rescission may occur in one of three ways:

- **Termination by right.** This is where a right to take an action is reserved to either or both of the parties in the contract itself (see Termination by Right).
- **Rescission by agreement.** This is the discharge of both parties from any outstanding legal obligations admittedly existing thereunder (see Rescission by Agreement).
- **Rescission by court.** One of the parties may ask a court to set aside the contract (see Rescission by Court).

Termination by Right

If a smart contract is terminated by one of the parties as permitted under the agreement and nothing more is done, the smart contract will still automatically perform (auto-perform) the parties' promises, as it is designed to do, negating the effect of the termination.

Therefore, for a smart contract to function properly:

- When a party duly exercises a right of termination under the smart contract, auto-performance must cease. Due exercise of termination by right in a smart contract therefore halts all auto-performance.
- The smart contract must also ensure that a termination by right occurs if and only if the party holding the right exercises it. Termination by right therefore is enabled in a smart contract if and only if a party holding that right exercises it.
- Before triggering a termination by right, the smart contract's machinery must ensure that all partial performance that has occurred is compensated. For example, partial payments sent by either party must be returned. If this is not done, parties will resort to courts to enforce restitution of that partial performance, undoing one of the primary efficiency benefits of smart contracts. Termination by right in a smart contract therefore automatically compensates partial performance under the agreement.
- The smart contract's machinery must ensure that all conditions placed on the termination right are met before termination is triggered. For example, if payment of a termination fee is a condition of the right, the contract must pay this fee to the appropriate party (or otherwise ensure that it is paid) before initiating termination. Termination by right in a smart contract therefore is enabled if and only if any termination conditions are satisfied.

Rescission by Agreement
For a smart contract to function properly:

- As with termination by right, automated performance must cease upon rescission by agreement. Smart contract rescission by agreement therefore halts all auto-performance.
- Unlike with termination by right, rescission by mutual agreement must be conditioned under the smart contract on mutual agreement, an offer to rescind by one party and acceptance of that offer by all other parties. Smart contract rescission by agreement therefore is enabled only if all parties mutually agree to it.
- Smart contract rescission by agreement, like smart contract termination by right, must include restoration of any partial performance. Smart contract rescission by agreement therefore automatically compensates partial performance.

Rescission by Court
Of the grounds for rescission, unilateral mistake (when one party *thinks* the smart contract does one thing, while the other party *knows* it does another) is of particular interest in relation to smart contracts. Due to the introduction of code to the agreement-making process, unilateral mistake may be a greater danger than ever before.

For a smart contract to function properly:

- When there is a unilateral mistake or when any of the other bases for rescission by court exist and a court orders a smart contract rescinded, auto-performance must cease. Rescission of smart contracts by court therefore halts all auto-performance.
- The power to order rescission by court may only lie with and be exercised by the appropriate court. Rescission of smart contracts by court therefore is enabled if and only if triggered by an appropriate court.
- Upon rescission by a court, restoration must occur, just as it would in the case of termination by right or rescission by agreement (see Termination by Right and Rescission by Agreement). If partial performance is not automatically compensated, parties would need to petition the court to enforce restitution, erasing one of the primary benefits of smart contracts. Rescission of smart contracts by court therefore automatically compensates partial performance.

Modification of Smart Contracts

Parties may not wish to wholly discard an agreement but merely to alter some of its terms. Like the undoing of a contract, a contract may be modified in one of three ways:

- **Modification by right.** Where unilateral-modification clauses give one party the right to amend the contract without the consent of the other party. Note that some courts will uphold this right, while others will not (see Modification by Right).
- **Modification by agreement.** Contracting parties have a well-established right to modify their original contract by mutual agreement. Such a modification is itself a contract and must be based on mutual assent and supported by its own consideration (see Modification by Agreement).
- **Reformation.** A court may in some cases order a modification of a contract even over the objection of one or more parties. It may do so based on three grounds:
  - mutual mistake;
  - fraud; and
  - unconscionable terms (where there is an absence of meaningful choice for one party and the terms are unreasonably favorable to the other).

  (See Reformation.)

Modification by Right
Smart contract modification must halt auto-performance of only the terms that are intended to be modified while simultaneously initiating auto-performance of any new terms intended to replace them.

For a smart contract to function properly:

- Upon modification by right of a smart contract term, auto-performance of that term's original iteration must cease, while auto-performance of its new iteration must concurrently initialize. Modification of smart contracts by right therefore simultaneously halts auto-performance of original terms and initiates auto-performance of new terms.

- Modification of a term can be initiated only if a party holding the right to modify that term exercises that right. Modification of smart contracts by right therefore is triggered only if the party holding the right exercises it. If the modification is conditioned on the occurrence of events, such as the payment of a fee, those conditions precedent must occur before modification is triggered. Modification of smart contracts by right therefore automatically compensates partial performance of *deleted terms*.
- A smart contract must automatically compensate for any partial performance that has occurred and which is tied to obligations embedded in the terms being removed during modification. It need not compensate for partial performance of any terms that, though modified, remain active. Those terms will be compensated through the performance of the contract. Modification by right therefore is enabled only if conditions precedent are satisfied.

Modification by Agreement
As noted, smart contract modification must halt auto-performance of only the terms that are intended to be modified while simultaneously initiating auto-performance of the new terms intended to replace them.

The key difference between modification by right and modification by agreement is that a modification by agreement must be approved by all parties.

Modification of smart contracts by agreement therefore must:

- Simultaneously halt auto-performance of original modified terms and initiates auto-performance of new terms.
- Be enabled only when all parties mutually agree to it.
- Automatically compensate partial performance of *deleted terms*.

Reformation
Reformation is of special interest in relation to smart contracts. Grounds for reformation that are particularly important to smart contracts include mutual mistake, which covers the scrivener's error, or accidental deviation from the parties' agreement made while recording the agreement in writing. In smart contracts, the risk of this error is high because of the introduction of code to contracting.

Fraud and unconscionability (also grounds for reformation) are risks of particular relevance to smart contracts because code-savvy parties are in a position to defraud or force unconscionable terms on code-naive parties.

For these reasons, reformation of smart contracts is likely to be relatively common.

For a smart contract to function properly:

- Reformation of smart contracts must simultaneously halt auto-performance of original reformed terms and initiate auto-performance of new terms.
- As with rescission by court, the power to reform the contract must lie strictly with the court. Reformation of smart contracts therefore is enabled only if triggered by an appropriate court.
- Once triggered, the reformation must compensate for partial performance of any terms that are being deleted. Reformation of smart contracts therefore automatically compensates partial performance of *deleted terms*.

Amending and Terminating Smart Contracts on Ethereum

Ethereum Generally

Ethereum has been described as arguably the most ambitious crypto-ledger project. It is built on the Ethereum blockchain (there are many blockchains, each developed and owned by a different company). The Ethereum blockchain stores both:

- Transaction data concerning its native cryptocurrency, called Ether.
- The code and state of computer programs called contracts.

As noted, the code for a smart contract is injected onto the blockchain when a personal account sends contract code in the data field of an unaddressed transaction. After this, the contract is added to a block and assigned an address, at which point its code becomes immutable.

But the contract's state is not immutable. It must remain dynamic. Specifically, the blockchain nodes in the Ethereum network (the computers connected to the network), besides being able to add transactions to the ledger, also run contract code and maintain and adjust contract states in a virtual machine called the Ethereum Virtual Machine.

Contracts on Ethereum can hold balances of Ether. Contracts on Ethereum can also have variables and functions that, if called, adjust those balances and can do other things like send Ether to other contracts or accounts on Ethereum. These functions cannot "wake" on their own and, to execute, must be "called" – or activated – by parties to the contract, third parties, or other contracts.

One of Ethereum's high-level languages, Solidity, has been described as a cross between JavaScript and C++ but with a number of syntactic additions to make it suitable for writing contracts within Ethereum. The example below will use Solidity.

Undoing Contracts on Ethereum

There are at least two ways to undo contracts on Ethereum. These methods are applicable for:

- Termination by right.
- Rescission by agreement.
- Rescission by court.

The two primary ways to undo a smart contract pursuant to the foregoing on Ethereum are:

- The "global selfdestruct function," which is easy to implement and effective but imprecise (see Undoing Contracts on Ethereum Using Selfdestruct).
- Turning the entire contract " off" at the function level using a combination of Solidity's modifiers and enums (see Undoing Contracts on Ethereum Using Modifiers and Enums). These are programming tools that can be used to add conditions to the functions in the contract.

Undoing Contracts on Ethereum Using Selfdestruct

As stated, Ethereum contract code, once on the blockchain, cannot be altered. But it can be deleted. The global selfdestruct function, if called from inside a contract, sends the contract's Ether balance to a specified address, then deletes the contract's code from the blockchain going forward. This means the contract's functions cannot be called.

Since Ethereum contract functions cannot self-wake, this halts auto-performance and therefore satisfies this requirement for smart contract termination by right, rescission by agreement, and rescission by court.

It is also easy on Ethereum to grant the power to selfdestruct a contract to only those entities that should have it. If that is a single party (which is the case for termination by right and rescission by court), this can be done by wrapping selfdestruct function inside a conditional statement that checks if the address calling it belongs to the rightful exerciser:

```
function terminate() {
if (partyWithTerminationRightOrCourt == msg.sender) {
selfdestruct( partyReceivingContractBalance ); }}
```

If multiple parties must approve the undoing, as in rescission by agreement, there are a few ways to achieve this. One is to use Solidity's modifiers and enums to log the consent of parties and then throw exceptions when selfdestruct is called and those states do not reflect the necessary values of all parties required to consent:

```
contract Undoable {
address partyWithTerminationRight;
address partyWithoutTerminationRight;

enum State {RescissionByAgreementSuggested}
State public state;

modifier inState(State _state) {
if (state != _state) throw;
       _ }

function suggestRescissionByAgreement() {
  if ( partyWithoutTerminationRight == msg.sender) {
state = State.RescissionByAgreementSuggested; }}

function approveRescissionByAgreement()
inState(State.RescissionByAgreementSuggested) {
  if (partyWithTerminationRight == msg.sender) {
selfdestruct(partyWithoutTerminationRight); }}}
```

Next is the third standard, shared by each of these tools, that demands that any partial performance that has occurred be compensated automatically before the contract is undone. With selfdestruct, all that is needed is a variable that tracks the level of performance, a function that lets one party suggest a new value for that variable, and a second function that lets the counterparty approve the new value. When the contract is undone, the latest value for the variable will be paid out to the appropriate party.

Termination by right is the only version of contract undoing with a fourth standard (which is that the smart contract's machinery must ensure that all conditions placed on the termination right are met before termination is implemented). These conditions come in many shapes, but the simplest is where the right is conditioned on the payment of a termination fee. To satisfy this standard, a more streamlined version of the approach used to satisfy the third standard may be used (making sure that the termination fee has been paid before permitting termination by selfdestruct).

The contract code below ties together all of the above, satisfying the conditions for the three methods of undoing contracts by creating functions for termination by right, rescission by agreement, and rescission by court, giving the power over those functions to the right parties and paying out termination and partial performance fees where required.

To simplify things, partial performance is only assumed possible for one party:

```
contract Undoable {
address partyWithTerminationRight;
address partyWithoutTerminationRight;
address partialPerformanceApprover;
address court;
uint terminationFee;
uint suggestedPartialPerformanceCompensation;

enum State {RescissionByAgreementSuggested,
PartialPerformanceCompensationSuggested,
PartialPerformanceCompensationApproved}
State public state;

modifier inState(State _state) {
if (state != _state) throw;
       _ }

function terminateOrRescind(uint _terminationFee, address
_partyWithTerminationRight, address
_partyWithoutTerminationRight, address _court, address
_partialPerformanceApprover) {
terminationFee = _terminationFee;
partyWithTerminationRight = _partyWithTerminationRight;
partyWithoutTerminationRight =
_partyWithoutTerminationRight;
partialPerformanceApprover = _partialPerformanceApprover;
court = _court;}

function suggestPartialPerformanceCompensation(uint
_suggestedPartialPerformanceCompensation) {
if ( partyWithoutTerminationRight == msg.sender) {
suggestedPartialPerformanceCompensation =
_suggestedPartialPerformanceCompensation;
state = State.PartialPerformanceCompensationSuggested;}}

function approvePartialPerformanceCompensation()
inState( State.PartialPerformanceCompensationSuggested){
if (partyWithTerminationRight == msg.sender) {
state = State.PartialPerformanceCompensationApproved ;}}

function terminate()
inState(State.PartialPerformanceCompensationApproved){
if (partyWithTerminationRight == msg.sender) {
   partyWithoutTerminationRight.send(terminationFee);
   partyWithoutTerminationRight.send(
suggestedPartialPerformanceCompensation);
selfdestruct(partyWithoutTerminationRight);}}

function rescindByCourt()
inState(State.PartialPerformanceCompensationApproved){
if (court == msg.sender) {
   partyWithoutTerminationRight.send(
suggestedPartialPerformanceCompensation);
selfdestruct(partyWithoutTerminationRight); }}

function suggestRescissionByAgreement()
inState(State.PartialPerformanceCompensationApproved){
if (partyWithoutTerminationRight == msg.sender) {
state = State.RescissionByAgreementSuggested; }}

function approveRescissionByAgreement()
inState(State.RescissionByAgreementSuggested){
if (partyWithTerminationRight == msg.sender) {
   partyWithoutTerminationRight.send(
   suggestedPartialPerformanceCompensation);
selfdestruct(partyWithoutTerminationRight); }}}
```

This code does not cover cases where conditions placed upon termination by right represent the occurrence of real world events. But this example suggests that traditional contract principles may reasonably be applied and satisfied by smart contracts using one of the methods for undoing smart contracts (selfdestruct) on Ethereum.

However, there is another, arguably superior, method for undoing smart contracts on the Ethereum platform.

Undoing Contracts on Ethereum Using Modifiers and Enums

The selfdestruct function is a convenient one-stop solution for undoing contracts on Ethereum. But Solidity's modifiers and enums are a more nuanced tool for this and mesh well with existing tools for altering contracts.

The above example used modifiers and enums in conjunction with selfdestruct. The same strategy can be extended to implement termination by right, rescission by agreement, and rescission by court without selfdestruct. Specifically, two states may be created:

- One state for a contract that has been undone (ContractUndone).
- One state for a contract that is not undone (ContractNotUndone).

The state can initially be set as ContractNotUndone.

Next, a function may be created that enables the state to be toggled to ContractUndone (but not toggled in the other direction).

Lastly, we can cause all other functions to turn off or "throw" if the ContractUndone state exists. This will halt performance of the contract, satisfying the first standard above for the three methods of undoing contracts. The other standards for undoing smart contracts may be satisfied in the same ways as set out above for selfdestruct (see Undoing Contracts on Ethereum Using Selfdestruct).

Modifying Contracts on Ethereum

Modifying contracts on Ethereum (that is, implementing modification by right, modification by agreement, and reformation) is more nuanced than undoing contracts on Ethereum. The three basic ways to achieve smart contract modification on Ethereum are:

- Modification of variable (referred to as variable-captured) terms (see Modifying Variable-Captured Terms).
- Deletion of function-related (referred to as function-captured) terms (see Deleting Function-Captured Terms).
- Addition or alteration of function-captured terms (see Adding or Modifying Function-Captured Terms).

Modifying Variable-Captured Terms

Variable contract terms, such as price or rate terms that fluctuate, are often captured as variables in smart contract code. Modifying these variables is as simple as assigning a new value to the variable using a set-type function (which assigns new values to a variable). If such a function exists in the original contract, this method of modification halts performance of the old term and initiates performance of the new one.

This method of modification also satisfies the second standard for termination by right, which is that the scope (that is, which party or parties can exercise) of termination by right must be hard-coded into the smart contract during formation. Satisfying the remaining standards for modification by right, modification by agreement, and reformation can all be accomplished in much the same way they were accomplished when undoing a smart contract for termination by right, rescission by agreement, and rescission by court (see Undoing Contracts on Ethereum).

Modifying Function-Captured Terms

Sometimes, contract terms are captured by functions and not variables. Take, for example, a function which captures a promise to send a certain amount of currency to a certain address after a certain date; if called after that date, it sends the currency to that address. In that case, modification (of, for example, the currency amount) means deleting, adding, or swapping the relevant function(s). This must be handled differently than variable-level modification because, while variables can be changed freely, the functions in an Ethereum contract code are immutable once issued to the blockchain.

Deleting Function-Captured Terms

Of the types of function-level changes, the easiest to implement is deletion – that is, subtraction of terms. For that, the approach taken for termination by right, rescission by agreement, and rescission by court may be used: using modifiers and enums to create and toggle states, then causing functions to throw exceptions, or turn off the functionality, if the required states do not exist.

Using this method, functions can be programmed to turned off on demand, if the parties agree to a deletion-style modification. This will halt performance much as it would for termination by right, rescission by agreement, and rescission by court, satisfying the first standard for undoing a contract. The remaining standards can be satisfied as they were above for variable-captured functions.

Adding or Modifying Function-Captured Terms

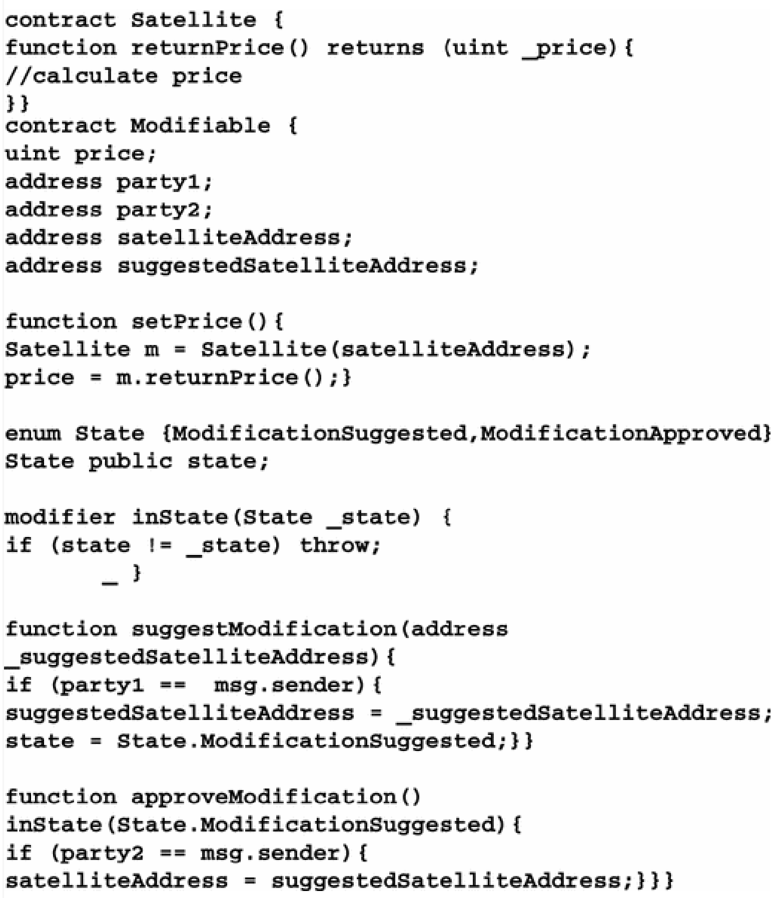
Adding wholly new function-captured terms and replacing existing functions is accomplished in a similar fashion to deleting function-captured terms. The difference between the two is that, if a function is being replaced, the initial version of it must also be turned off. This can be accomplished using the methods described above for deletion of functions (see Deleting Function-Captured Terms). There are at least two ways to add or swap functions in an Ethereum smart contract.

The first is to use modifiers and enums, which can be used to turn functions off and on. Of course, in order to be turned on, those functions must be included in the contract to begin with. Since contract code is immutable after initialization, this means that functions the parties suspect they may later wish to turn on during a modification must be included in the initial contract in an "off" state. If this can be accomplished, then the standards for all three types of contract modification (modification by right, modification by agreement, and reformation) can be satisfied much as they would be for variable-captured functions.

A second way to add or modify function-captured terms — and, seemingly, the way endorsed by Ethereum’s architects — is to create at the outset satellite contracts that capture certain function-specific terms. The addresses of these satellite contracts can be stored in address variables or an array of address variables in the central contract. Using these pointers, the central contract can call out to the satellite contracts when it needs to reference certain terms. If this is structured properly, modifying function-specific terms is as simple as changing the pointers.

As an example, suppose the parties wish to build flexibility into their price term. They can initialize a central contract with pointers to a satellite price-calculation contract. Changing the price calculation method (swapping price datafeeds or formulas) is as simple as changing the pointers from the central contract.

The code for such a contract might look like the code below, which includes a pair of functions that let one party suggest a new satellite contract and let the other approve the suggestion before making the change (note that, in order to call an outside contract's functions on Ethereum, the code for the outside contract must appear in the code for the present contract):



```
contract Satellite {
function returnPrice() returns (uint _price){
//calculate price
}}
contract Modifiable {
uint price;
address party1;
address party2;
address satelliteAddress;
address suggestedSatelliteAddress;

function setPrice(){
Satellite m = Satellite(satelliteAddress);
price = m.returnPrice();}

enum State {ModificationSuggested,ModificationApproved}
State public state;

modifier inState(State _state) {
if (state != _state) throw;
        _ }

function suggestModification(address
_suggestedSatelliteAddress){
if (party1 ==  msg.sender){
suggestedSatelliteAddress = _suggestedSatelliteAddress;
state = State.ModificationSuggested;}}

function approveModification()
inState(State.ModificationSuggested){
if (party2 == msg.sender){
satelliteAddress = suggestedSatelliteAddress;}}}
```

This contract satisfies the first standard for all three types of contract modification (modification by right, modification by agreement, and reformation). By de-linking the original satellite contract and linking the new one, it simultaneously halts auto-performance of the original versions of modified terms and initiates auto-performance of the new versions.

If it includes code to ensure that the party initiating the pointer switch is the correct one and also includes code that tracks and compensates partial performance in the event of a modification, then it can satisfy the second and third standards of all three types of modification as well. It can satisfy the fourth standard of modification by right by including additional code that prohibits modification unless certain conditions have been met.

The original publication is available at Springer via Setting Standards for Altering and Undoing Smart Contracts.

PRODUCTS

PLC US Finance, PLC US Financial Services, PLC US Intellectual Property and Technology, PLC US Law Department