DECODE language design patterns

Denis Roio, Dyne.org

Monday, 27 November, 2017

Abstract

This document explains the nature of smart rules in DECODE. It establishes guidelines and requirements for the implementation of an execution engine for a new domain specific language. DECODE's language is an external DSL implemented using a Syntax-Directed Translation. Its Semantic Model leads to coarse-grained tasks to be executed by the nodes on the peer to peer network. This is a living document and its latest version can be found on files.dyne.org/decode

Keywords— blockchain; smart; contract; rules; langsec; DSL

Contents

Introduction

The main way to communicate with a DECODE node and operate its functions is via a language, rather than an API. All read and write operations affecting entitlements and accessing attributes can be expressed in a smart-rule language, which we intend to design and develop to become a robust open standard for authorisation around personal data. The DECODE smart-rule language will aim to naturally avoid complex constructions and define sets of transformations that can be then easily represented with visual metaphors.

At this stage of the research, this document is split in 3 sections:

1. a brief "state of the art" analysis, considering existing blockchain based languages and in particular the most popular "Solididy" supported by the Ethereum virtual machine.
2. a brief enumeration of the characteristics of this implementation and abstract them from it, to individuate the fundamental features a smart-rule language should have in the context of permissionless, distributed computing.
3. a set of technical recommendations for the development of smart-rules in DECODE

A new memory model

In computing science the concepts of HEAP and STACK are well known and represent the different areas of memory in which a single computer can store code, address it while executing it and store data on which the code can read and write. With "virtual machines" the implementation of logic behind the HEAP and STACK became "virtualised" and not anymore bound to a specific hardware architecture, therefore leaving more space for the portability of code and creative memory management practices (like garbage collection). It is also thanks to the use of virtual machines that high level languages became closer to the way humans think, rather than the way machines work. This is an important vector of innovation for the language implementation in DECODE, since it is desirable for this project to implement a language that is close to the way humans think.

With the advent of distributed computing technology and blockchain implementations there is a raising necessity to conceive the HEAP and STACK differently (Pizka and Rehn, 2002).

The underpinning of this document, elaborated on the term "blockchain language", is that a new "distributed ledger", as collective and immutable memory space, can be addressed with code ran on different machines.

A "blockchain language" then is a language designed to interact with a "distributed ledger". A distributed ledger is a log of events in possession of all nodes being part of a network.

This document intentionally leaves aside considerations about the consensus algorithm of a blockchain-based network, which are very specific issues concerning the implementation of a blockchain and are covered by other research tasks in DECODE. While assuming an ideal condition for fault tolerance, we will continue focusing on the function that the distributed ledger has for the distributed computation of a language.

Blockchain languages

This section is a brief exploration of the main language implementations working on blockchains. Far from being an exhaustive overview, it highlights the characteristics of these implementations and most importantly the approach followed in building virtual machines that are based on assembler-like operation codes and languages that compile to these.

The conclusion of this section is that the blockchain languages so far existing are designed with a product-oriented mindset, starting from the implementation of a virtual machine that can process OP codes. Higher level languages build upon it, parsing higher level syntactics and semantics and compiling them into a series of OP codes. This is the natural way most languages like ASM, C and C++ have evolved through the years.

Arguably, a task-oriented mindset should be assumed when re-designing a new blockchain language for DECODE. The opportunity for innovating the field lies in abandoning this approach and instead build an External Domain Specific Language (Fowler, 2010) using an existing grammar to do the Syntax-Directed Translation. The Semantic Model can be then a coarse-grained implementation that can sync computations with blockchain based deterministic conditionals.

Bitcoin's SCRIPT

Starting with the "SCRIPT" implementation in Bitcoin (Nakamoto, 2008) and ending with the Ethereum Virtual Machine implementation, it is clear that blockchain technologies were developed with the concept of "distributed computation" in mind. The scenario is that of a network of computers that, at any point in time, can execute the same code on a part of the distributed ledger and that execution would yield to the same results, making the computation completely deterministic.

The distributed computation is made by blockchain nodes that act as sort of "virtual machines" and process "operation codes" (OP_CODE) just like a computer does. These OP_CODES in fact resemble assembler language operations.

In Bitcoin the so called SCRIPT implementation had an unfinished number of "OP codes" (operation codes) at the time of its popularisation and, around the 0.6 release, the feature was in large part deactivated to insure the security of the network, since it was assessed by most developers involved that the Bitcoin implementation of SCRIPT was unfinished and represented threats to the network. Increasing the complexity of code that can be executed by nodes of an open network is always a risk, since code can contain arbitrary operations and commands that may lead to unpredictable results affecting both the single node and the whole network. The shortcoming of the SCRIPT in Bitcoin were partially addressed: its space for OP_RETURN (Roio et al., 2015) became the contested ground for payloads (Sward et al., 2017) that could be interpreted by other VMs, as well the limit was partially circumvented by moving more complex logic in touch with the Bitcoin blockchain(Aron, 2012), for instance using the techniques adopted by Mastercoin (Willett, 2013) and "sidechains" as Counterparty (Bocek and Stiller, 2018) or "pegged sidechains" (Back et al., 2014) implementations. All these are implementations of VMs that run in parallel to Bitcoin, can "peg" their results on the main Bitcoin blockchain and still execute more complex operations in another space, where tokens and conditions can be created and affect a different memory spaces and distributed ledgers.

Languages implemented so far for this task are capable of executing single OP codes: implementations are very much "machine-oriented" and focused on reproducing the behaviour of a turing-complete machine (Wegner et al., 2012) capable of executing generic computing tasks.

The Ethereum VM

The Ethereum Virtual Machine is arguably the most popular implementation of a language that can be computed by a distributed and decentralised network of virtual machines that have all their own HEAP and STACK, but all share the same immutable distributed ledger on which "global" values and the code (contracts) manipulating them can be inscribed and read from.

Computation in the EVM is done using a stack-based bytecode language that is like a cross between Bitcoin Script, traditional assembly and Lisp (the Lisp part being due to the recursive message-sending functionality). A program in EVM is a sequence of opcodes, like this:

```
PUSH1 0 CALLDATALOAD SLOAD NOT PUSH1 9 JUMPI STOP JUMPDEST PUSH1
   32 CALLDATALOAD PUSH1 0 CALLDATALOAD SSTORE
```

The purpose of this particular contract is to serve as a name registry; anyone can send a message containing 64 bytes of data, 32 for the key and 32 for the value. The contract checks if the key has already been registered in storage, and if it has not been then the contract registers the value at that key. The address of the new contract is deterministic and calculated on the sending address and the number of times that the sending account has made a transaction before.

The EVM is a simple stack-based architecture. The word size of the machine (and thus size of stack item) is 256-bit. This was chosen to is a simple word-addressed byte array. The stack has a maximum size of 1024. The machine also has an independent storage model; this is similar in concept to the memory but rather than a byte array, it is a word- addressable word array. Unlike memory, which is volatile, storage is non volatile and is maintained as part of the system state. All locations in both storage and memory are well-defined initially as zero.

The machine does not follow the standard von Neumann architecture. Rather than storing program code in generally-accessible memory or storage, it is stored separately in a virtual ROM interactable only through a specialised instruction. The machine can have exceptional execution for several reasons, including stack underflows and invalid instructions. Like the out-of-gas (OOG) exception, they do not leave state changes intact. Rather, the machine halts immediately and reports the issue to the execution agent (either the transaction processor or, recursively, the spawning execution environment) which will deal with it separately (Wood, 2014).

The resulting implementation consists of a list of OP codes whose execution requires a “price” to be paid (Ethereum’s currency for the purpose is called “gas”). This way an incentive is created for running nodes: a fee is paid to nodes for computing the contracts and confirming the outcomes of their execution. This feature technically defines the Ethereum VM as implementing an almost Turing-complete machine since its execution is conditioned by the availability of funds for computation. This approach relies on the fact that each operation is “atomic”, meaning it is executed at a constant unit of speed.

Here below is the list of OP codes in use in Ethereum, which results immediately familiar to anyone who has done some assembler language:

```
# schema: [opcode, ins, outs, gas]

opcodes = {

    # arithmetic
    0x00: ['STOP', 0, 0, 0],
    0x01: ['ADD', 2, 1, 3],
    0x02: ['MUL', 2, 1, 5],
    0x03: ['SUB', 2, 1, 3],
    0x04: ['DIV', 2, 1, 5],
    0x05: ['SDIV', 2, 1, 5],
    0x06: ['MOD', 2, 1, 5],
    0x07: ['SMOD', 2, 1, 5],
```

```
    0x08: ['ADDMOD', 3, 1, 8],
    0x09: ['MULMOD', 3, 1, 8],
    0x0a: ['EXP', 2, 1, 10],
    0x0b: ['SIGNEXTEND', 2, 1, 5],

    # boolean
    0x10: ['LT', 2, 1, 3],
    0x11: ['GT', 2, 1, 3],
    0x12: ['SLT', 2, 1, 3],
    0x13: ['SGT', 2, 1, 3],
    0x14: ['EQ', 2, 1, 3],
    0x15: ['ISZERO', 1, 1, 3],
    0x16: ['AND', 2, 1, 3],
    0x17: ['OR', 2, 1, 3],
    0x18: ['XOR', 2, 1, 3],
    0x19: ['NOT', 1, 1, 3],
    0x1a: ['BYTE', 2, 1, 3],

    # crypto
    0x20: ['SHA3', 2, 1, 30],

    # contract context
    0x30: ['ADDRESS', 0, 1, 2],
    0x31: ['BALANCE', 1, 1, 20],
    0x32: ['ORIGIN', 0, 1, 2],
    0x33: ['CALLER', 0, 1, 2],
    0x34: ['CALLVALUE', 0, 1, 2],
    0x35: ['CALLDATALOAD', 1, 1, 3],
    0x36: ['CALLDATASIZE', 0, 1, 2],
    0x37: ['CALLDATACOPY', 3, 0, 3],
    0x38: ['CODESIZE', 0, 1, 2],
    0x39: ['CODECOPY', 3, 0, 3],
    0x3a: ['GASPRICE', 0, 1, 2],
    0x3b: ['EXTCODESIZE', 1, 1, 20],
    0x3c: ['EXTCODECOPY', 4, 0, 20],

    # blockchain context
    0x40: ['BLOCKHASH', 1, 1, 20],
    0x41: ['COINBASE', 0, 1, 2],
    0x42: ['TIMESTAMP', 0, 1, 2],
    0x43: ['NUMBER', 0, 1, 2],
    0x44: ['DIFFICULTY', 0, 1, 2],
    0x45: ['GASLIMIT', 0, 1, 2],

    # storage and execution
    0x50: ['POP', 1, 0, 2],
    0x51: ['MLOAD', 1, 1, 3],
    0x52: ['MSTORE', 2, 0, 3],
    0x53: ['MSTORE8', 2, 0, 3],
```

```
    0x54: ['SLOAD', 1, 1, 50],
    0x55: ['SSTORE', 2, 0, 0],
    0x56: ['JUMP', 1, 0, 8],
    0x57: ['JUMPI', 2, 0, 10],
    0x58: ['PC', 0, 1, 2],
    0x59: ['MSIZE', 0, 1, 2],
    0x5a: ['GAS', 0, 1, 2],
    0x5b: ['JUMPDEST', 0, 0, 1],

    # logging
    0xa0: ['LOG0', 2, 0, 375],
    0xa1: ['LOG1', 3, 0, 750],
    0xa2: ['LOG2', 4, 0, 1125],
    0xa3: ['LOG3', 5, 0, 1500],
    0xa4: ['LOG4', 6, 0, 1875],

    # arbitrary length storage (proposal for metropolis hardfork)
    0xe1: ['SLOADBYTES', 3, 0, 50],
    0xe2: ['SSTOREBYTES', 3, 0, 0],
    0xe3: ['SSIZE', 1, 1, 50],

    # closures
    0xf0: ['CREATE', 3, 1, 32000],
    0xf1: ['CALL', 7, 1, 40],
    0xf2: ['CALLCODE', 7, 1, 40],
    0xf3: ['RETURN', 2, 0, 0],
    0xf4: ['DELEGATECALL', 6, 0, 40],
    0xff: ['SUICIDE', 1, 0, 0],
}
for i in range(1, 33):
    opcodes[0x5f + i] = ['PUSH' + str(i), 0, 1, 3]
for i in range(1, 17):
    opcodes[0x7f + i] = ['DUP' + str(i), i, i + 1, 3]
    opcodes[0x8f + i] = ['SWAP' + str(i), i + 1, i + 1, 3]
```

On top of these OP codes the "Solidity" language was developed as a high-level language that compiles to OP code sequences. Solidity aims to make it easier for people to program "smart contracts". But it is arguable that the Solidity higher-level language, well present in all Ethereum related literature, carries several problems: the shortcomings of its design can be indirectly related to some well known disasters provoked by flaws in published contracts. To quickly summarise some flaws:

- there is no garbage collector nor manual memory management
- floating point numbers are not supported
- there are known security flaws in the compiler
- the syntax of loops and arrays is confusing
- every type is 256bits wide, including bytes
- there is no string manipulation support
- functions can return only statically sized arrays

To overcome the shortcomings and create some shared base of reliable implementations,

programmers using Solidity nowadays adopt sort of “standard” token implementation libraries with basic functions that are proven to be working reliably: known as ERC20, the standard is made for tokens to be supported across different wallets and to be reliable. Yet even with a recent update to a new version (ERC232) the typical code constructs that are known to be working are full of checks (assert calls) to insure the reliability of the calling code. For example typical arithmetic operations need to be implemented in Solidity as:

```
function times(uint a, uint b) constant private returns (uint)
   {
  uint c = a * b;
  assert(a == 0 || c / a == b);
  return c;
}

function minus(uint a, uint b) constant private returns (uint)
   {
  assert(b <= a);
  return a - b;
}

function plus(uint a, uint b) constant private returns (uint) {
  uint c = a + b;
  assert(c>=a);
  return c;
}
```

It must be also noted that the EVM allows calling external contracts that can take over the control flow and make changes to data that the calling function wasn't expecting. This class of bug can take many forms and all of major bugs that led to the DAO's collapse (O'Hara, 2017) were bugs of this sort.

Despite the shortcomings, nowadays Solidity is widely used: it is the most used “blockchain language” supporting “smart-contracts” in the world.

Language Security

This chapter will quickly establish the underpinnings of a smart rule language in DECODE, starting from its most theoretical assumptions, to conclude with specific requirements. Most importantly starting from the recent corpus developed by researches on language-theoretic security" (LangSec). Here below we include a brief explanation condensed from the information material of the LangSec.org project hosted at IEEE, which is informed by the collective experience of the exploit development community, since exploitation is practical exploration of the space of unanticipated state, its prevention or containment.

Threats when developing a language

As one engages the task of developing a language there are four main threats to be identified, well described in LangSec literature:

Ad-hoc notions of input validity

Verification of input handlers is impossible without formal language-theoretic specification of their inputs, whether these inputs are packets, messages, protocol units, or file formats. Therefore, design of an input-handling program must start with such a formal specification. Once specified, the input language should be reduced to the least complex class requiring the least computational power to recognize. Considering the tendency of hand-coded programs to admit extra state and computation paths, computational power susceptible to crafted inputs should be minimized whenever possible. Whenever the input language is allowed to achieve Turing-complete power, input validation becomes undecidable; such situations should be avoided. For example, checking 'benignness' of arbitrary Javascript or even an HTML5+CSS page is a losing proposition.

Parser differentials

Mutual misinterpretation between system components. Verifiable composition is impossible without means of establishing parsing equivalence between different components of a distributed system. Different interpretation of messages or data streams by components breaks any assumptions that components adhere to a shared specification and so introduces inconsistent state and unanticipated computation. In addition, it breaks any security schemes in which equivalent parsing of messages is a formal requirement, such as the contents of a certificate or of a signed message being interpreted identically (e.g., X.509 CSRs as seen by a CA vs. the signed certificates as seen by the clients or signed app package contents as seen by the signature verifier versus the same content as seen by the installer (as in the recent Android Master Key bug). An input language specification stronger than deterministic context-free makes the problem of establishing parser equivalence undecidable. Such input languages and systems whose trustworthiness is predicated on the component parser equivalence should be avoided.

Mixing of input recognition and processing

Mixing of basic input validation ("sanity checks") and logically subsequent processing steps that belong only after the integrity of the entire message has been established makes validation hard or impossible. As a practical consequence, unanticipated reachable state exposed by such premature optimization explodes. This explosion makes principled analysis of the possible computation paths untenable. LangSec-style separation of the recognizer and processor code creates a natural partitioning that allows for simpler specification-based verification and management of code. In such designs, effective elimination of exploit-enabling implicit data flows can be achieved by simple systems memory isolation primitives.

Ungoverned development

Adding New Features / Language Specification Drift. A common practice encouraged by rapid software development is the unconstrained addition of new features to software components and their corresponding reflection in input language specifications. Expressing complex ideas in hastily written code is a hallmark of such development practices. In essence, adding new input feature requirements to an already-underspecified input language compounds the explosion of state and computational paths.

Separation between grammar and process

In a nutshell [...] LangSec is the idea that many security issues can be avoided by applying a standard process to input processing and protocol design: the acceptable input to a program should be well-defined (i.e., via a grammar), as simple as possible (on the Chomsky scale of syntactic complexity), and fully validated before use (by a dedicated parser of appropriate but not excessive power in the Chomsky hierarchy of automata). (Momot et al., 2016)

LangSec is a design and programming philosophy that focuses on formally correct and verifiable input handling throughout all phases of the software development lifecycle. In doing so, it offers a practical method of assurance of software free from broad and currently dominant classes of bugs and vulnerabilities related to incorrect parsing and interpretation of messages between software components (packets, protocol messages, file formats, function parameters, etc.).

This design and programming paradigm begins with a description of valid inputs to a program as a formal language (such as a grammar). The purpose of such a disciplined specification is to cleanly separate the input-handling code and processing code. A LangSec-compliant design properly transforms input-handling code into a recognizer for the input language; this recognizer rejects non-conforming inputs and transforms conforming inputs to structured data (such as an object or a tree structure, ready for type- or value-based pattern matching). The processing code can then access the structured data (but not the raw inputs or parsers temporary data artifacts) under a set of assumptions regarding the accepted inputs that are enforced by the recognizer.

This approach leads to several advantages:

1. produce verifiable recognizers, free of typical classes of ad-hoc parsing bugs
2. produce verifiable, composable implementations of distributed systems that ensure equivalent parsing of messages by all components and eliminate exploitable differences in message interpretation by the elements of a distributed system
3. mitigate the common risks of ungoverned development by explicitly exposing the processing dependencies on the parsed input.

As a design philosophy, LangSec focuses on a particular choice of verification trade-offs: namely, correctness and computational equivalence of input processors.

Smart-rules language

In light of our study of blockchain languages, use-cases and privacy by design guidelines in DECODE, this section lists three functional requirements and three usability requirements influencing the design patterns for our language.

The conclusion of this section is best described adopting once again the DSL terminology and the patterns established by Fowler. The DECODE smart-rule language is an external DSL implemented using a Syntax-Directed Translation. Its Semantic Model leads to coarse-grained tasks to be executed on the network, perhaps following a Dependency Network approach.

A tempting alternative can be that of a Production Rule System, but this way we'd risk to hide too much the internal processes in DECODE, which should be transparent and comprehensible to anyone with a beginner knowledge of programming.

An addition to this approach can be that of equipping the language with tools for constraint programming and even a context of Satisfiability Modulo Theories (SMT) to check satisfying Program Termination Proofs.

Functional requirements

On the basis of the design considerations made in the previous chapters, here are listed the main requirements identified for the implementation of a smart-rule language in DECODE.

Deterministic

This is an important feature common to all blockchain language implementations in use: that the language limits its operations to access only a fully deterministic environment. This means that, in any possible moment in time, any node can join the network and start computing contracts leading to results that are verifiable and confirmed by other nodes.

In other words, the environment accessed by the language is available to all nodes, there aren't variables that are "private" to a single node and may change the result by a change of their value.

The deterministic trait must be common also to the DECODE blockchain language for smart-rules, since it verifies a basic and necessary condition for blockchain based computing: that other nodes can verify and sign the results, reproducing them in their own execution environment. The computation leads to the same results that can be determined in different conditions, because all nodes have access to the same information necessary to the computation.

Trustless

We define as trustless a language (also known as untrusted language) that allows the virtual machine to fence its execution, like in a "sandbox" or isolated execution environment, blocking access to unauthorised parts of the system.

A language that can be run on a "permissionless" (public) blockchain is a language that can be interpreted by any node in any moment a new node may claim the capacity to do so. This means that its parser, semantics and actions on the system must be designed to handle unknowns: any deviance and malevolent code should not affect the system.

Solid

The language grammar and the semantic model adopted by DECODE need to be capable of sandboxing untrusted code and providing security partitioning. Any process of execution should be strictly limited in what it can do. Any function or data passed to a node cannot break the sandbox in ways the participants did not intend.

For sensitive data structures, the use of proxy objects need to be adopted as a security guard, only allowing the sandbox to call pre-approved methods and access pre-approved data.

Usability requirements

Here are listed the requirements emerging from an analysis of priorities about the human-machine interaction scenarios emerging from DECODE.

Simple, graphical representation

A visual programming environment (VPL) facilitates participants to directly re-configure the rules governing their data: this is highly desirable in DECODE, where such code must be transparent and understandable. The event-based BLOCKS graphical metaphor seems the most

desirable for the sort of processing in DECODE: it involves letting participants manipulate a series of graphical elements (blocks) that snap onto one another and that execute sequential programs.

This would mean that someone taking part in a DECODE pilot has the freedom to configure how a DECODE node is operating: change the way data is displayed, enable external services to access the data or remotely operate parts of the DECODE network.

Test environment

A reliable test environment is a fundamental component for a language deployed in mission critical situations, but also for a language dealing with the distribution of its computation and wide adoption by communities of developers in different fields. Languages that improve the developer's experience when writing and testing code directly impact the quality of the code produced.

For DECODE's language implementation is necessary to have a testing environment designed into it and from the start to facilitate its growth at the same pace of the language itself. Also a more advanced framework for testing that goes beyond the simple usage of asserts is desirable: while being very ambitious, the implementation of solid proof of termination mechanisms that are internal to the language should be contemplated on the long term.

First class data

This is a long-term requirement that should take into consideration the trade-off between feasibility, security and convenience. A data type is considered first class in a programming language if instances of that type can be

- the value of a variable
- a member of an aggregate (array, list, etc.)
- an argument (input) to a procedure
- the value returned by a procedure
- used without having a name (being the value of a variable)

For example, numbers are first class in every language. Text strings are first class in many languages, but not in C, in which the relevant first class type is "pointer to a character".

In DECODE it is desirable to establish data structures containing attributes and entitlements as first class data to be seamlessly processed by the language.

Conclusion

This document is a very dense representation of language patterns and requirements to be adopted while implementing DECODE's language. Its feasibility has been verified with an extensive survey on available tools that can be used to implement this execution engine and are compatible with the DECODE licensing model.

This conclusion provides a brief list of components that can be used.

Syntax-Directed Translation

Lua is an interpreted, cross-platform, embeddable, performant and low-footprint language. Lua's popularity is on the rise in the last couple of years. Simple design and efficient usage of resources combined with its performance make it attractive for production web applications even to big

organizations such as Wikipedia, CloudFlare and GitHub. In addition to this, Lua is one of the preferred choices for programming embedded and IoT devices. This context allows to assume a large and growing Lua codebase yet to be assessed. This growing Lua codebase could be potentially driving production servers and extremely large number of devices, some perhaps with mission-critical function for example in automotive or home-automation domains. (Costin, 2017)

Lua solidity has been well tested through a number of public applications including the adoption by the gaming industry for untrusted language processing in "World of Warcraft" scripting. It is ideal for implementing an external DSL using C or Python as a host language.

Lua is also tooled with a working VPL implementation for code visualisation in BLOCKS, allowing the project to jump-start into an early phase of prototyping DECODE smart-rules in a visual way and involving directly pilot participants.

Satisfiability Modulo theories

Satisfiability Modulo theories (SMT) is an area of automated deduction that studies methods for checking the satisfiability of first-order formulas with respect to some logical theory of interest (Barrett et al., 2009). It differs from general automated deduction in that the background theory need not be finitely or even first-order axiomatizable, and specialized inference methods are used for each theory. By being theory-specific and restricting their language to certain classes of formulas (such as, typically but not exclusively, quantifier-free formulas), these specialized methods can be implemented in solvers that are more efficient in practice than general-purpose theorem provers.

While SMT techniques have been traditionally used to support deductive software verification, they are now finding applications in other areas of computer science such as, for instance, planning, model checking and automated test generation. Typical theories of interest in these applications include formalizations of arithmetic, arrays, bit vectors, algebraic datatypes, equality with uninterpreted functions, and various combinations of these.

Constraint-satisfaction is crucial to software and hardware verification and static program analsysis (De Moura and Bjørner, 2011) among the other possible applications.

DECODE will benefit from including SMT capabilities into the design at an early stage: even if not immediately exploited, their inclusion will keep the horizons for language development open while permitting its application in mission critical roles.

Bibliography

Aron, J. (2012) BitCoin software finds new life. *New Scientist*. 213 (2847), 20.

Back, A. et al. (2014) Enabling blockchain innovations with pegged sidechains. *URL: http://www. opensciencereview. com/papers/123/enablingblockchain-innovations-with-pegged-sidechains*.

Barrett, C. W. et al. (2009) Satisfiability modulo theories. *Handbook of satisfiability*. 185825–885.

Bocek, T. & Stiller, B. (2018) 'Smart contracts–Blockchains in the wings', in *Digital marketplaces unleashed*. Springer. pp. 169–184.

Costin, A. (2017) *Lua code: Security overview and practical approaches to static analysis*.

De Moura, L. & Bjørner, N. (2011) Satisfiability modulo theories: Introduction and applications. *Communications of the ACM*. 54 (9), 69–77.

Fowler, M. (2010) *Domain-specific languages*. Pearson Education.

Momot, F. et al. (2016) 'The seven turrets of babel: A taxonomy of langsec errors and how to expunge them', in *IEEE cybersecurity development, secdev 2016, boston, ma, usa, november 3-4, 2016*. [Online]. 2016 IEEE. pp. 45–52. [online]. Available from: https://doi.org/10.1109/SecDev.2016.019.

Nakamoto, S. (2008) Bitcoin: A peer-to-peer electronic cash system. *Consulted*. 12012.

O'Hara, K. (2017) Smart contracts-dumb idea. *IEEE Internet Computing*. 21 (2), 97–101.

Pizka, M. & Rehn, C. (2002) 'Heaps and stacks in distributed shared memory', in *16th international parallel and distributed processing symposium (IPDPS 2002), 15-19 april 2002, fort lauderdale, fl, usa, cd-rom/abstracts proceedings*. [Online]. 2002 IEEE Computer Society. [online]. Available from: https://doi.org/10.1109/IPDPS.2002.1016494.

Roio, D. et al. (2015) *Design of social digital currency*.

Sward, A. et al. (2017) *Data insertion in bitcoin's blockchain*.

Wegner, P. et al. (2012) 'Computational completeness of interaction machines and turing machines', in Andrei Voronkov (ed.) *Turing-100 - the alan turing centenary, manchester, uk, june 22-25, 2012*. EPiC series in computing. 2012 EasyChair. pp. 405–414. [online]. Available from: http://www.easychair.org/publications/paper/106520.

Willett, J. R. (2013) *MasterCoin Complete Specification*. [online]. Available from: {https://github.com/mastercoin-MSC/spec}.

Wood, G. (2014) Ethereum: A secure decentralised generalised transaction ledger. *Ethereum Project Yellow Paper*. 151.