decode

Smart Rules implementation, Evaluation of Prototypes and integration

This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement no. 732546

Project no. 732546

DECODE

DEcentralised Citizens Owned Data Ecosystem

D3.6. Smart Rules implementation, Evaluation of Prototypes and integration

Version Number: V1.0

Lead beneficiary: Dyne.org

Due Date: December 31st, 2018

Author(s): Denis Roio, Puria Nafisi Azizi (Dyne.org)

Editors and reviewers: Francesca Bria, Oleguer Sagarra (IMI), Marco Ciurcina (NEXA), Alberto Sonnino (UCL)

Dissemination level:		
PU	Public	**X**
PP	Restricted to other programme participants (including the Commission Services)	
RE	Restricted to a group specified by the consortium (including the Commission Services)	
CO	Confidential, only for members of the consortium (including the Commission Services)	

Approved by: Francesca Bria (Chief Technology and Digital Innovation Officer, Barcelona City Hall)
Date: 31/01/2019

This report is currently awaiting approval from the EC and cannot be not considered to be a final version.

Table of contents

1. Introduction

This deliverable consists of the implementation of smart-rules effectively executing cryptographic operation and data transformations using a human readable language modeled according to a taxonomy of subjects and predicates found in the pilot applications. It delivers a technology that brings together expression and execution into utterances based on translatable human language phrases. This technology is a simple, non-touring complete natural language interpreter (Zencode) based on a domain specific language (DSL) that can run and execute inside a very portable virtual machine (Zenroom) capable of cryptographic transformations.

Since DECODE project's inception, reaching this point of development has been an extremely motivating ambition, as it also serves an important solution for the techno-political implications illustrated by the AlgoSov.eu observatory and the recently published PhD thesis "Algorithmic Sovereignty" (Roio, 2018).

1.1. For the awareness of algorithms

The goal of this task is ultimately that of realizing a simple, non-technical, human-readable language for smart-rules that are actually executed in a verifiable and provable manner within the Zenroom controlled execution environment.

To articulate the importance of this quest and the relevance of the results presented, which I believe to be unique in the landscape of blockchain smart-contract languages, is important to remind us of the condition in which most people find themselves when participating in the regime of truth that is built by algorithms.

As the demand and production of well-connected vessels for the digital dimension has boomed, machine-readable code today functions as a literature informing the architecture in which human interactions happens and decisions are taken. The telematic condition is realised by an integrated datawork continuously engaging the observer as a participant. Such a "Gesamtdatenwerk" (Ascott, 1990) may seem an abstract architecture, yet it can be deeply binding under legal, ethical and moral circumstances.

The comprehension of algorithms, the awareness of the way decisions are formulated, the implications of their execution, is not just a technical condition, but a political one, for which access to information cannot be just considered a feature, but a civil right (Pelizza and Kuhlmann, 2017). It is important to understand this in relation to the "classical" application of algorithms executed in a centralized manner, but even more in relation to distributed computing scenarios posed by blockchain technologies, which theorize a future in which rules and contracts are executed irrevocably and without requiring any human agency.

The legal implications with regards to standing rights and liabilities are out of the scope here, while the focus is on ways humans, even when lacking technical literacy, can be made aware of what an algorithm does. Is it possible to establish the ground for a shared language that informs digital architects about their choices and inhabitants about the digital territory? Going past assumptions about the strong role algorithms have in governance and accountability (Diakopoulos, 2016), how can we inform digital citizens about their condition?

When describing the virtualisation of economic activity in the global context, Saskia Sassen describes the need we are observing as that of an analytical vocabulary:

> *The third component in the new geography of power is the growing importance of electronic space. There is much to be said on this issue. Here, I can isolate one particular matter: the distinctive challenge that the virtualization of a growing number of economic activities presents not only to the existing state regulatory apparatus, but also to private-sector institutions increasingly dependent on the new technologies. Taken to its extreme, this may signal a control crisis in the making, one for which we lack an analytical vocabulary.(Sassen, 1996)*

The analysis of legal texts and regulations here shifts into an entirely new domain; it has to refer to conditions that only algorithms can help build or destroy. Thus, referring to this theoretical framework, the research and development of a free and open source language that is intelligible to humans becomes of crucial importance and, from an ethical standing point, DECODE as many other projects in the same space cannot be exempted from addressing it.

When we consider algorithms as contracts regulating relationships (between humans, between humans and nature and, nowadays more increasingly, between different contexts of nature itself) then we should adopt a representation that is close to how the human mind works and that is directly connected to the language adopted. Since algorithms are the systemic product of complex relationships between contracts and relevant choices made by standing actors (Monico, 2014), the ability to verify which algorithms are in place for a certain result to be visualised becomes very important and should be embedded in every application: to understand and communicate what algorithms and to describe and experiment their repercussions on reality.

2. Implementation

This section describes the salient implementation details of the Zencode DSL, the smart-rule language for DECODE, tailored on its use-cases and based on the Zenroom controlled execution environment (VM). Implementation details refer only to Zencode and not to how Zenroom is implemented, since the latter is already covered in other documents.

The implementation section contains three parts explaining:

- the language model inherited by Behaviour Driven Development
- the data validation model based on Schema Validation
- the implementation of implicit certificates

2.1. Behaviour Driven Development

In Behaviour Driven Development (BDD), the important role of software integration and unit tests is extended to serve both the purposes of designing the human-machine interaction flow (user journey in UX terms) and of laying down a common ground for interaction between designers and stakeholders. In this Agile software development methodology the software testing suite is based on natural language units that grant a common understanding for all participants and observers.

To implement BDD the first step is that of mapping a series of interconnected cascading sentences to actual source code; this implementation is usually done manually by programmers that have knowledge of the higher level application protocol interface (API) that grants communication between the backend and the frontend of a software application. The BDD implementation can then be seen as an alternative frontend whose purpose is that of lowering the distance between expression and execution by means of utterances expressed in human language.

Far from giving an exhaustive description of BDD implementations and characteristics, this brief chapter intends to summarise the features of this approach where they specifically apply to the development goals of Zencode (previously stated) and the solution provided.

Referring to the Cucumber implementation of BDD, arguably the most popular in use by the industry to day and factual standard (Wynne, 2012), the grammar of utterances is very simple and definable as a "cascading" flow indeed, since the fixed sequence of lines can follow only one fixed order:

Given .. and* .. When .. and* .. Then print ..

This sequence is fixed and in simple terms consists of:

1. an extendable initialisation of states "Given (and)"
2. followed by an extendable transformation of states "When (and)"
3. concluded by returning the final states "Then print".

The Zenroom implementation is kept simple at this stage and does not takes any "fuzzy" approach to the parsing, but simply defines fixed sequences of strings and variables that are expected to occur within them: the variables are what is ultimately possible to change by users and are marked by a repeating sequence of two adjacent single quotes (' ').

The underlying parser acts upon a positive, unique and so far non-flexible match of the whole phrase minus the variables, then executes a function that takes as many arguments as the variables present in the lines across the utterance. As a result, every single non-repeating line of the utterance has a declared function that interacts with the underlying implementation of Zenroom, whose actions are defined in its LUA subset language.

Brief examples of this implementation follow:

```
Given("I introduce myself as ''", function(name) whoami = name end)
Given("I am known as '',        function(name) whoami = name end)
```

The above definition of two lines possibly occurring within the utterances in Zencode are demonstrating how a state "who am I" basically my own name can be set using two different phrases, leading to the execution of the same function which basically operates a simple assignment to the variable *whoami*. This simple demonstration is a hint to the fact that multiple patterns can be defined also in different ways, making the Zencode DSL implementation very easy to translate across different spoken languages as well contextualised within specific idiolects adopted by humans.

Furthermore, another example of implementation:

```
Given("that '' declares to be ''",function(who, decl)
        -- declaration
        if not declared then declared = decl
        else declared = declared .." and ".. decl end
        whois = who
end)
Given("declares also to be ''", function(decl)
        ZEN.assert(who ~= "", "The subject making the declaration is unknown")
        -- declaration
        if not declared then declared = decl
        else declared = declared .." and ".. decl end
end)
```

Shows how is possible to accept multiple variables and process them through more complex transformations that also contemplate the concatenation of contents to previous states. States are in fact permanent within the scope of the execution of a single utterance and will be modified in the same deterministic order by which they are expressed across lines. What is also visible within this example implementation, which we intend to facilitate by customisation made by people who have a simple knowledge of Zenroom's API and LUA scripting, is that the 'ZEN.' namespace makes available a number of utility functions to easily check states (asserts) and propagate meaningful error messages that are then part of a backtrace output given to the calling application (host) on occurrence of an error.

The full implementation of Zencode available at the time of publishing this document is inside the source-code files 'zenroom/src/lua/zencode_*' and is relatively easy to maintain for the pilots analysed in our project, as well easy to extend to more use-cases. The current implementation addresses specific schemes that useful to the pilots in DECODE, while contemplating future extension:

- Simple symmetric encryption of ciphertext by means of a PIN and KDF transformations (pilot: Amsterdam Register)

- Diffie-Hellman asymmetric key encryption (AES-GCM) (pilot: Making Sense IoT)
- Blind-sign credentials for unlinkable selective attribute revelations[1] (pilot: DECIDIM and Gebied Online)
- In addition there is also the implementation of an "implicit certificate" crypto scheme (Qu-Vanstone, ECQV) that is limited to first order curve transformations, which may apply to pilots requiring simple certification schemes[2].

All the implementations are illustrated in more detail in the following chapters.

2.2. Declarative Schema Validation

In order to make the processing of Zencode more robust, all data used as input and output for its computations is validated according to predefined schemas. This makes the Zencode DSL a declarative language in which data recognition is operated before processing.

The data schemas are added on a per-usecase basis: they refer to specific cryptographic implementations as they are added in Zencode. Careful evaluation regarding their addition is made to realise if old schemas can be extended to include new requirements.

Schemas are expressed in a simple format using Lua scripting syntax, for example:

```
-- zencode_keypair
keypair = S.record {
   schema = S.Optional(S.string),
   private = S.Optional(S.hex),
   public = S.ecp
}
```

The schema above is the smallest and most commonly used one, composed by one required field and two optional ones, used to validate the input and output of public/private keypairs to be used in transformations.

The only required field in the schema is the 'public' key which is validated using the 'ECP' type ('S.' is an abbreviation for the 'SCHEMA.' namespace). The validation of 'S.ECP' is an actual cryptographic validation: Zenroom will check that the big integer number represented by the field corresponds to a valid point on the curve. In case the validation is not passed, the execution of the Zencode script will not take place and Zenroom will return a meaningful error message indicating the wrong field.

The other optional field is the 'private' key which can correspond to any sequence of values, therefore no cryptographic validation is possible for it; in this case then the validation used is one that refers to the encoding of the field: 'S.hex' is verifying that the value is encoded with a sequence of characters that express only hexadecimal numbers

[1] This implementation refers to work on the Coconut credential system (Sonnino et. al, 2018) designed after specific needs in DECODE's pilots. It does not implement, however, the threshold issuance part, which is only required in the scenario of a fully open blockchain implementation, which is still work in progress.

[2] It is important to note that while the ECQV scheme was not examined by other partners in our project, it has been choosen for its stable role in the industry and for its augmented complexity within an approachable implementation, complexity which could better inform the Zencode implementation.

(that is, 0..9 numbers and case-insensitive letters from A to Z). Other encoding tests are also available, for instance 'S.base64' if that is the encoding used in the specific implementation.

Another more complex example follows:

```
-- packets encoded with AES GCM
AES-GCM = S.record {
   checksum = S.hex,
   iv = S.hex,
   schema = S.Optional(S.string),
   text = S.hex,
   zenroom = S.Optional(S.string),
   encoding = S.string,
   curve = S.string,
   pubkey = S.ecp
}
```

In this example no new validations are being used and in fact it just adds fields compared to the previous: it defines a portable packet of ciphertext data that is returned as output of AES-GCM asymmetric encryption as well is accepted as input to AES-GCM decryption. A similarity between these two examples is evident: the presence of the 'schema' field. This field is a sort of "introspective" indication matching the data structure to its schema specification. If this field is not present (as it is always optional) then no validation on the data structure will take place, meaning the Zencode implementation leaves the risk (and hopefully the validation task) to the host.

This chapter ends with the current implementation of schema validation data types that are currently implemented for symmetric and asymmetric encryption of ciphertexts as well for implicit certificates. The schema implementation for Zencode is maintained into the sourcecode within the source file 'src/lua/zencode_schemas.lua' and can be accessed by the function 'ZEN.validate(data,'schema','error')' which is a wrapper of 'ZEN.assert(validate(data,schemas['schema']),'error')'.

```
_G['schemas'] = {

  -- packets encoded with AES GCM
  AES-GCM = S.record {
     checksum = S.hex,
     iv = S.hex,
     schema = S.Optional(S.string),
     text = S.hex,
     zenroom = S.Optional(S.string),
     encoding = S.string,
     curve = S.string,
     pubkey = S.ecp
  },

  -- zencode_keypair
  keypair = S.record {
     schema = S.Optional(S.string),
     private = S.Optional(S.hex),
     public = S.ecp
  },

  -- zencode_ecqv
  certificate = S.record {
     schema = S.Optional(S.string),
     private = S.Optional(S.big),
     public = S.ecp,
     hash = S.big,
     from = S.string,
     authkey = S.ecp
  },

  certificate_hash = S.Record {
     schema = S.Optional(S.string),
     public = S.ecp,
     requester = S.string,
     statement = S.string,
     certifier = S.string
  },takes

  declaration = S.record {
     schema = S.Optional(S.string),
     from = S.string,
     to = S.string,
     statement = S.string,
     public = S.ecp
  },

  declaration_keypair = S.record {
     schema = S.Optional(S.string),
     requester = S.string,
     statement = S.string,
     public = S.ecp,
     private = S.hex
  }

}
```

2.3. Implicit Certificates

This section will illustrate a Zencode implementation of the Elliptic Curve Qu-Vanstone implicit certificate scheme (ECQV) as described by the Standards for Efficient Cryptography 4 (SEC4, 2014).

The ECQV implicit certificate scheme is intended as a general purpose certificate scheme for applications within computer and communications systems. It is particularly well suited for application environments where resources such as bandwidth, computing power and storage are limited. ECQV provides a more efficient alternative to traditional certificates.

The ECQV is identifiable as a simple yet important building block within DECODE, as it permits the efficient creation of certificates that contain only the public reconstruction data instead of the subject's public key and the CA's signature, also resulting into a smaller payload than traditional certificates.

ECQV relates well to those DECODE pilots in need to authenticate participants according to signed credentials, where the issuance of a public key is subject to the verification of certain conditions by a Certificate Authority (CA) capable of verifying and signing those conditions. This scenarios applies well to the pilot experimentations ongoing in Amsterdam for the DECODE project, where a certificate (and a keypair) is issued based on attributes that are certified by the municipal register and then used for authentication procedures operated by third parties and based on those attributes.

2.3.1. *Differences with traditional certificates*

To justify the implementation and adoption of ECQV in place of traditional certificates, here are quickly listed three salient characteristics, closely referring to the documentation offered by the SEC4-1.0 document.

With traditional certificates, when an entity U requests a traditional certificate for a public key, U should prove to the CA it knows the corresponding private key. This is to prevent U from choosing an arbitrary public key, that may already belong to another user, and have it certified. This situation is clearly undesirable (and may even lead to security problems). With implicit certificates this proof is unnecessary, as there is no public key before the certificate is issued. Further, U has no control over the final value of his public key, due to the CA's contribution, making it impossible for U to cause the confusion described above.

Unlike traditional certificates, an implicit certificate does not contain a digital signature. In fact, one could simply choose an arbitrary identity I and a random value to form a certificate. Together with the public key of a CA, this generates a public key for the entity identified by I. However, if one constructs an implicit certificate in such a way, i.e., without interacting with the CA, it is infeasible to compute the private key that corresponds to the public key generated by the certificate.

Another difference between traditional certificates and implicit certificates is that when presented with a valid traditional certificate, one knows that the certificate belongs to someone. A valid certificate containing the certificate data string IU is a proof that the CA signed this certificate for U , and also that U knows the private key corresponding to the public key included in the certificate. One does not have this guarantee with implicit certificates, satisfying certain privacy conditions made evident by the GDPR.

2.3.2. *Zencode Implementation*

This section will demonstrate the Zencode implementation in four steps, covering all the transformations into a human-readable language from the mathematical formula to the implementation capable of being executed in the Zenroom VM without any external dependency.

The first step is the mathematical formula for ECQV as explained in the SEC4 document.

$$\begin{array}{lcl}
\underline{\qquad\qquad U \qquad\qquad} & & \underline{\qquad\qquad CA \qquad\qquad} \\
k_U \in_R [1, \dots, n-1] & & \\
R_U := k_U G & & \\
 & \xrightarrow{U,\ R_U} & \\
 & & k \in_R [1, \dots, n-1] \\
 & & P_U := R_U + kG \\
 & & Cert_U := Encode(P_U, U, *) \\
 & & e := H_n(Cert_U) \\
 & & r := ek + d_{CA} \pmod n \\
 & \xleftarrow{r,\ Cert_U} & \\
e := H_n(Cert_U) & & \\
d_U := ek_U + r \pmod n & & \\
Q_U := eP_U + Q_{CA} & &
\end{array}$$

The second step is the implementation of this formula into the machine language executed by the Zenroom VM (a dialect of LUA).

```
-- Zenroom 0.8.0
-- setup
random = RNG.new()
order = ECP.order()
G = ECP.generator()
-- make a request for certification
ku = INT.new(random, order)
Ru = G * ku
-- keypair for CA
dCA = INT.new(random, order) -- private
QCA = G * dCA       -- public (known to Alice)
-- from here the CA has received the request
k = INT.new(random, order)
kG = G * k
-- public key reconstruction data
Pu = Ru + kG
declaration = { public = Pu:octet(),
                requester = str("Alice"),
                statement = str("I am stuck in Wonderland.") }
declhash = sha256(OCTET.serialize(declaration))
hash = INT.new(declhash, order)
-- private key reconstruction data
r = (hash * k + dCA) % order
-- verified by the requester, receiving r,Certu
du = (r + hash * ku) % order
Qu = Pu * hash + QCA
assert(Qu == G * du)
```

The third step is the improvement of the previous implementation using meaningful variable and function names.

```
-- Zenroom 0.8.1
-- setup
random = RNG.new()
order = ECP.order()
G = ECP.generator()
-- typical EC key generation on G1
function keygen(rng,modulo)
   local key = INT.new(rng,modulo)
   return { private = key,
            public = key * G }
end
-- generate the certification request
certreq = keygen(random,order)
-- certreq.private is preserved in a safe place
-- certreq.public is sent to the CA along with a declaration
declaration = { requester = str("Alice"),
                statement = str("I am stuck in Wonderland") }
-- Requester sends to CA -->
-- ... once upon a time ...
-- --> CA receives from Requester
-- keypair for CA (known to everyone as the Mad Hatter)
CA = keygen(random,order)
-- from here the CA has received the request
certkey = keygen(random,order)
-- certkey.private is sent to requester
-- certkey.public is broadcasted
-- public key reconstruction data
```

```
certpub = certreq.public + certkey.public
-- the certification is serialized (could use ASN-1 or X509)
certification = { public = certpub,
                  requester = declaration.requester,
                  statement = declaration.statement,
                  certifier = str("Mad Hatter") }
CERT = sha256(OCTET.serialize(certification))
CERThash = INT.new(CERT, order)
-- private key reconstruction data
certpriv = (CERThash * certkey.private + CA.private) % order
-- CA sends to Requester certpriv and CERThash
-- eventually CA broadcasts certpub and CERThash
-- ... on the other side of the mirror ...
-- Alice has received from the CA the certpriv and CERT
-- which can be used to create a new CERTprivate key
CERTprivate = (certpriv + CERThash * certreq.private) % order
-- Anyone may receive the certpub and CERThash and, knowing the CA
-- public key, can recover the same CERTpublic key from them
CERTpublic  = certpub * CERThash + CA.public
-- As a proof here we generate the public key in a standard way,
-- multiplying it by the curve generator point, then check equality
assert(CERTpublic == G * CERTprivate)
print "Certified keypair:"
I.print({ private = CERTprivate:octet():base64(),
          public  =  CERTpublic:octet():base64()    })
```

At last, the implementation in Zencode follows, clearly showing the simplification made possible by Zenroom for the ECQV implicit certificate cryptographic scheme. Each of the following "scenarios" are blocks of code that can be executed independently from one another, taking validated input and output data structures.

```
-- Zenroom 0.9

Scenario 'keygen': $scenario
   Given that I am known as 'MadHatter'
   When I create my new keypair
   Then print my keyring

Scenario 'request': Make my declaration and request certificate
   Given that I introduce myself as 'Alice'
   and I have the 'public' key 'MadHatter' in keyring
   When I declare to 'MadHatter' that I am 'lost in Wonderland'
   and I issue my implicit certificate request 'declaration'
   Then print all data

Scenario 'keygen': $scenario
   Given that I am known as 'Alice'
   and I have a 'declaration_public' 'from' 'Alice'
   Then print data 'declaration_public'

Scenario 'keygen': $scenario
   Given that I am known as 'Alice'
   and I have a 'declaration_keypair'
   Then print data 'declaration_keypair'

Scenario 'issue': Receive a declaration request and issue a certificate
   Given that I am known as 'MadHatter'
   and I have a 'declaration_public' 'from' 'Alice'
   and I have my 'private' key in keyring
   When I issue an implicit certificate for 'declaration_public'
   Then print all data
```

```
Scenario 'split': Print the public section of the certificate
   Given I have a 'certificate_public' 'from' 'MadHatter'
   When possible
   Then print data 'certificate_public'

Scenario 'split': Print the private section of the certificate
   Given I have a 'certificate_private'
   When possible
   Then print data 'certificate_private'

Scenario 'save': Receive a certificate of a declaration and save it
   Given I have a 'certificate_private' 'from' 'MadHatter'
   and I have the 'private' key 'declaration_keypair' in keyring
   When I verify the implicit certificate 'certificate_private'
   Then I print data 'declaration'

Scenario 'keygen': $scenario
   Given that I am known as 'Bob'
   When I create my new keypair
   Then print my keyring

Scenario 'challenge': Receive a certificate of a declaration and use it to encrypt a message
   Given that I am known as 'Bob'
   and I have my 'private' key in keyring
   and that 'Alice' declares to be 'lost in Wonderland'
   and I have a 'certificate' 'from' 'MadHatter'
   When I draft the text 'Hey Alice! can you read me?'
   and I use 'certificate' key to encrypt the text into 'ciphertext'
   Then I print data 'ciphertext'

Scenario 'respond': Alice receives an encrypted message, decrypts it and sends an encrypted answer back to sender
   Given that I am known as 'Alice'
   and I have my 'private' key in keyring
   When I decrypt the 'ciphertext' to 'decoded'
   and I use 'certificate' key to encrypt 'decoded' into 'answer'
   Then I print data 'answer'
```

The Zencode language is a DSL enforcing a strong declarative behavior underneath and all base data structures are checked against a validation scheme upon input and output. The checks are also of cryptographic nature, for instance public keys are checked to make sure they are actual points on the elliptic curve in use. Here below the data validation schemes so far in use:

```
_G['schemas'] = {

  -- packets encoded with AES GCM
  AES-GCM = S.record {
    checksum = S.hex,
    iv = S.hex,
    schema = S.Optional(S.string),
    text = S.hex,
    zenroom = S.Optional(S.string),
    encoding = S.string,
    curve = S.string,
    pubkey = S.ecp
  },
  -- zencode_keypair
  keypair = S.record {
    schema = S.Optional(S.string),
    private = S.Optional(S.hex),
    public = S.ecp
  },
  -- zencode_ecqv
  certificate = S.record {
    schema = S.Optional(S.string),
    private = S.Optional(S.big),
    public = S.ecp,
    hash = S.big,
    from = S.string,
    authkey = S.ecp
  },
  certificate_hash = S.Record {
    schema = S.Optional(S.string),
    public = S.ecp,
    requester = S.string,
    statement = S.string,
    certifier = S.string
  },
  declaration = S.record {
    schema = S.Optional(S.string),
    from = S.string,
    to = S.string,
    statement = S.string,
    public = S.ecp
  },
  declaration_keypair = S.record {
    schema = S.Optional(S.string),
    requester = S.string,
    statement = S.string,
    public = S.ecp,
    private = S.hex
  }
}
```

2.3.3. Blind-signed attribute credentials

The ECQV Zencode implementation described in the previous chapter has offered an important occasion to refine our language by modeling it to serve a well tested and fairly complex cryptographic sceme. It has however strong limits for the work envisioned in DECODE pilots and especially with regards to the "Privacy by Design" (Colesky et al., 2016; Danezis et al., 2015; Hoepman, 2014) recommendations we are ought to follow. To summarize ECQV limits:

- The use of certifications is traceable as crypto-materials aren't blinded and can be individuated across communication logs (or a ledger in case of adoption of DLTs)
- Two-way communication needs to take place for every single step: between the requester and the issuer, as well between the verifier and the requester.
- Especially when executed in a remotely networked situation, the certification scheme is prone to man-in-the-middle attacks (Adrian et al., 2015)

To overcome these and other limits of cryptographic implementations typically based on Diffie-Hellman keypairs, this document moves forward with the implementation of a "Threshold Issuance Selective Disclosure Credentials" system named Coconut (Sonnino et al., 2018) and developed by colleagues at UCL to specifically address the challenges posed by the development of a open blockchain in the scenarios outlined by DECODE's pilots.

Coconut offers several advantages for our use-cases:

- it allows for multiple certificate authorities to sign credentials.
- it provides blind-signature verifications for both issued and proven credentials, ready for use on a DLT.
- it relatively small sized keys and credentials, even when several authorities are involved.
- it provides optional support for threshold based credential validation which will be especially useful when DECODE is deployed on an open blockchain.

2.3.4. Coconut Implementation

The implementation of Coconut requires PAIR EC crypto operations (and in particular the "Miller Loop" on twisted curve space) for which we specifically adopt the BLS383[3] curve proposed by Milagro's developers for these kinds of operations. Other PAIRING capable curves will work as well, but have not been tested.

```
local g1 = ECP.generator()
local g2 = ECP2.generator()
local o  = ECP.order()

-- stateful challenge hardcoded string

local hs = ECP.hashtopoint(str([[
Developed for the DECODE project
]] .. coco._LICENSE))

local challenge = g1:octet() .. g2:octet() .. hs:octet()

-- random generator init

local random = RNG.new()
local function rand() return INT.new(random,o) end

-- El-Gamal cryptosystem

function coco.elgamal_keygen()
  local d = rand()
  local gamma = d * g1
  return d, gamma
end

function coco.elgamal_enc(gamma, m, h)
  local k = rand()
  local a = k * g1
  local b = gamma * k + h * m
  return a, b, k
end

function coco.elgamal_dec(d, a, b)
  return b - a * d
end

-- local zero-knowledge proof verifications
```

[3] There is no academic documentation on the BLS383 curve yet, its integrity is tested empirically across the various implementations of the Milagro crypto library.

```
local function to_challenge(list)
  return INT.new( sha256( challenge .. OCTET.serialize(list)))
end

local function make_pi_s(gamma, cm, k, r, m)
  local h = ECP.hashtopoint(cm)
  local wk = rand()
  local wm = rand()
  local wr = rand()
  local Aw = g1 * wk
  local Bw = gamma * wk + h * wm
  local Cw = g1 * wr + hs * wm
  local c = to_challenge({ cm, h, Aw, Bw, Cw })
  local rk = wk:modsub(c * k, o)
  local rm = wm:modsub(c * m, o)
  local rr = wr:modsub(c * r, o)
  return {          c  = c,
                    rk = rk,
                    rm = rm,
                    rr = rr }
end

function coco.verify_pi_s(gamma, ciphertext, cm, proof)
  local h = ECP.hashtopoint(cm)
  local a = ciphertext.a
  local b = ciphertext.b
  local c = proof.c
  local rk = proof.rk
  local rm = proof.rm
  local rr = proof.rr
  local Aw = a * c + g1 * rk
  local Bw = b * c + gamma * rk + h * rm
  local Cw = cm * c + g1 * rr + hs * rm
  return c == to_challenge({ cm, h, Aw, Bw, Cw })
end

local function make_pi_v(vk, sigma_prime, m, r)
  local wm = rand()
  local wr = rand()
  local Aw = g2 * wr + vk.alpha + vk.beta * wm
  local Bw = sigma_prime.h_prime * wr
  local c = to_challenge({ vk.alpha, vk.beta, Aw, Bw })
  local rm = wm:modsub(m * c, o)
  local rr = wr:modsub(r * c, o)
  return { c = c, rm = rm, rr = rr }
end

local function verify_pi_v(vk, kappa, nu, sigma_prime, proof)
  local c = proof.c
  local rm = proof.rm
  local rr = proof.rr
  local Aw = kappa * c + g2 * rr + vk.alpha * INT.new(1):modsub(c,o) + vk.beta * rm
  local Bw = nu * c + sigma_prime.h_prime * rr
  return c == to_challenge({ vk.alpha, vk.beta, Aw, Bw })
end

-- Public Coconut API
function coco.ca_keygen()
  local x = rand()
  local y = rand()
  local sk = { x = x,
           y = y  }
  local vk = { g2 = g2,
              alpha = g2 * x,
              beta  = g2 * y  }
  -- return keypair
  return { sign = sk,
         verify = vk }
end

function coco.cred_keygen()
  local d, gamma = ELGAMAL.keygen()
  return { private = d,
                             public  = gamma }
end

function coco.prepare_blind_sign(gamma, secret)
  local m = INT.new(sha256(str(secret)))
  local r = rand()
  local cm = g1 * r + hs * m
  local h = ECP.hashtopoint(cm)
  local a, b, k = ELGAMAL.encrypt(gamma, m, h)
  local c = {a = a, b = b}
  local pi_s = make_pi_s(gamma, cm, k, r, m)
  -- return Lambda
  return { cm   = cm,
         c    = c,
         pi_s = pi_s }
end

function coco.blind_sign(sk, gamma, Lambda)
         local ret = coco.verify_pi_s(gamma, Lambda.c, Lambda.cm, Lambda.pi_s)
         assert(ret == true, 'Proof pi_s does not verify')
         local h = ECP.hashtopoint(Lambda.cm)
         local a_tilde = Lambda.c.a * sk.y
         local b_tilde = h * sk.x + Lambda.c.b * sk.y
         return { h = h,
        a_tilde = a_tilde,
       b_tilde = b_tilde  }
end

function coco.aggregate_creds(d, sigma_tilde)
  local agg_s = ELGAMAL.decrypt(d, sigma_tilde[1].a_tilde, sigma_tilde[1].b_tilde)
  if #sigma_tilde > 1 then
    for i = 2, #sigma_tilde do
      agg_s = agg_s + ELGAMAL.decrypt(d, sigma_tilde[i].a_tilde, sigma_tilde[i].b_tilde)
    end
  end
  return { h = sigma_tilde[1].h,
         s = agg_s }
end

function coco.prove_creds(vk, sigma, secret)
  local m = INT.new(sha256(str(secret)))
  local r = rand()
  local r_prime = rand()
  local sigma_prime = { h_prime = sigma.h * r_prime,
                 s_prime = sigma.s * r_prime  }
  local kappa = vk.alpha + vk.beta * m + vk.g2 * r
  local nu = sigma_prime.h_prime * r
  local pi_v = make_pi_v(vk, sigma_prime, m, r)
  -- return Theta
  local Theta = {
    kappa = kappa,
    nu = nu,
    sigma_prime = sigma_prime,
    pi_v = pi_v }
  return Theta
end

function coco.verify_creds(vk, Theta)
  local ret = verify_pi_v(vk, Theta.kappa, Theta.nu, Theta.sigma_prime, Theta.pi_v)
  assert(ret == true, 'Proof pi_v does not verify') -- verify zero knowledge proof
  local ret1 = not Theta.sigma_prime.h_prime:isinf()
  local ret2 = ECP2.miller(Theta.kappa, Theta.sigma_prime.h_prime)
           == ECP2.miller(vk.g2, Theta.sigma_prime.s_prime + Theta.nu)
  return ret1 and ret2
end
```

The data formats used in Coconut are validated by Zencode (not by this Lua underlying implementation) and defined using the same names used in the Coconut paper as follows:

```
  coconut_ca_vk = S.record {
    g2 = S.hex,
    alpha = S.hex,
    beta = S.hex
  },

  coconut_ca_sk = S.record {
    x = S.int,
    y = S.int
  },

  coconut_ca_keypair = S.record {
    schema = S.Optional(S.string),
    version = S.Optional(S.string),
    verify = S.table,
    sign = S.table
  },

  coconut_req_keypair = S.record {
    schema = S.Optional(S.string),
    version = S.Optional(S.string),
    public = S.ecp,
    private = S.hex
  },

  coconut_pi_s = S.record {
           rr = S.int,
           rm = S.int,
           rk = S.int,
           c = S.int
  },

  coconut_sigmatilde = S.record {
    schema = S.Optional(S.string),
    version = S.Optional(S.string),
    h = S.ecp,
    b_tilde = S.ecp,
    a_tilde = S.ecp
  },

  coconut_aggsigma = S.record {
           schema = S.Optional(S.string),
           version = S.Optional(S.string),
           h = S.ecp,
           s = S.ecp
  }
}
```

This implementation is fully covered by tests and following lab-tests has been proven to work reliably. It is probably the most advanced implementation of a cryptographic scheme in Zenroom and as such has been taken as an important reference to define the the Zencode language, which is illustrated in the following chapter.

3. Evaluation of prototypes

In order to better explain the potential offered by the Zencode Domain Specific Language (DSL) to DECODE's prototypes its important to understand the versatility of its usage. Approaches may change on a domain-specific basis and its possible to tailor and simplify usage on the specific context it applies to.

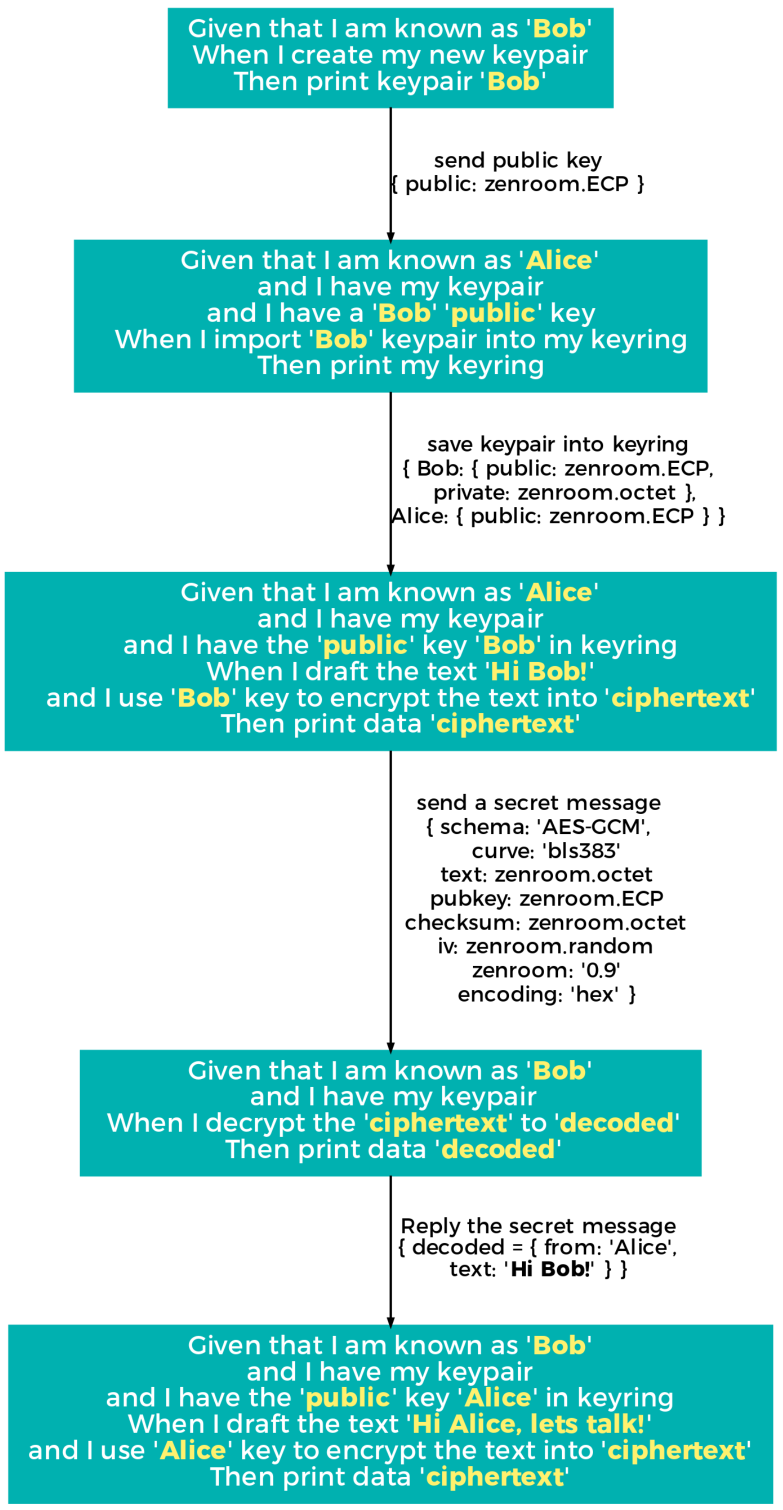
As we are on the quest to merge the description of an algorithm with its executive expression we get close to the concept of a speech act that refers to a specific context and adopts a limited taxonomy which may or may not be inscribed in a larger ontology.

It is very important to understand that the boxes in the flow diagrams shown contain **actual Zencode** meaning that is not just a description, but is source-code that is interpreted and executed by the Zenroom VM to accomplish the tasks described. It is then the main way to faithfully describe what the prototype does internally with the data: each of the prototypes built in DECODE can simply visualize the Zencode that is running to inform any operator of what is going on.

This solution has been realized after trying many different approaches involving visual programming and block programming, which were perhaps richer visually, but less integrated and in general consisting of a way to represent code rather than code itself. The final Zencode solution is also simplier to implement for prototyped host applications.

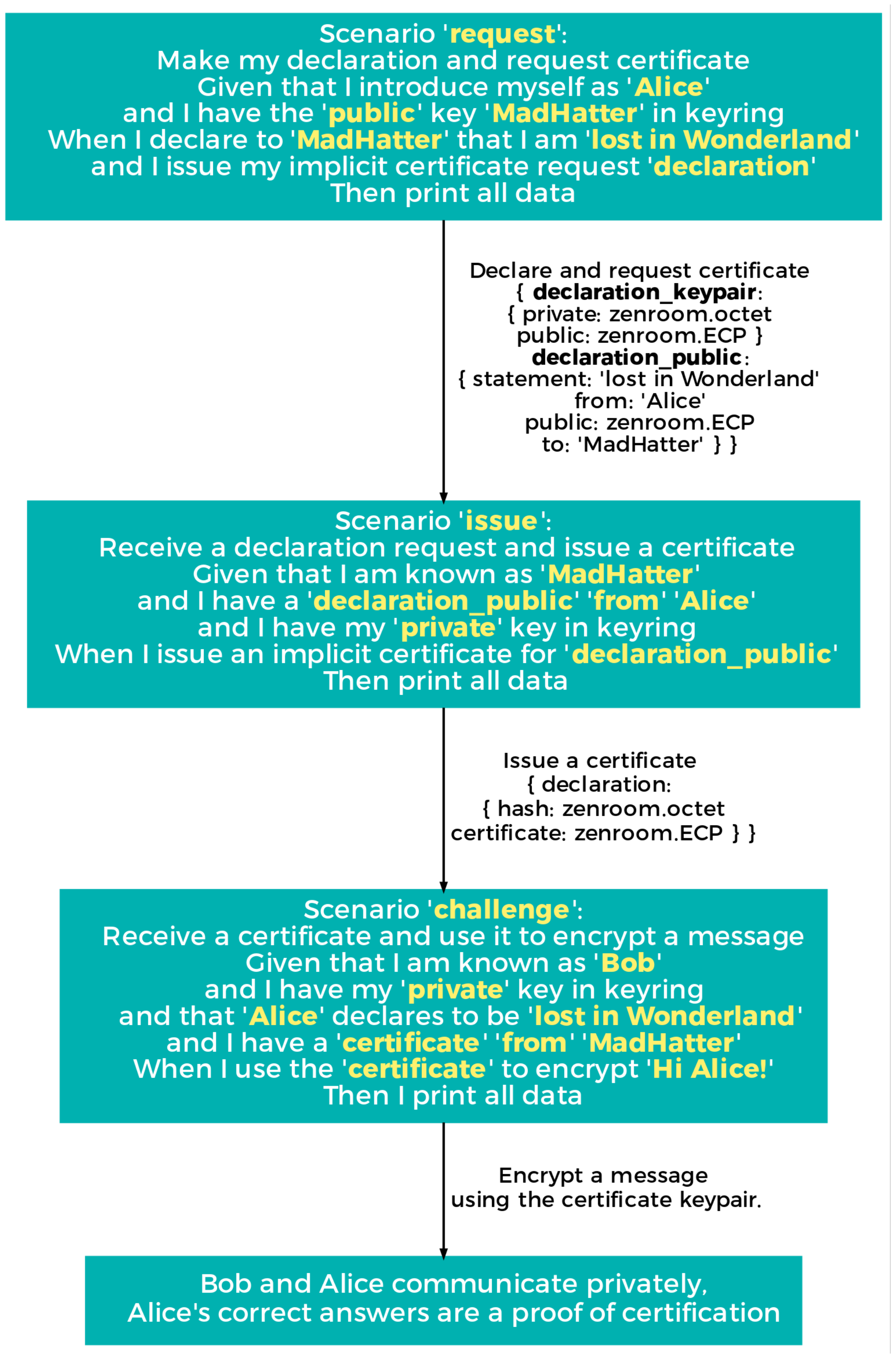
At the time of writing our explanation can be based on an extended experimentation of in-vitro usage (lab tests) and a limited experimentation of in-vivo usage mostly bound to the conceptualization of use-cases in the IoT pilot and the Amsterdam's register pilot. In order to extend the coverage of Zencode to more pilots, we need to have a completed implementation of the underlying cryptographic contract, in this case the petition.

What follows is a brief visualisation of what is realised so far. In particular the first visualisation below refers to the implementation of an asymmetric cryptographic exchange in the fashion of the PGP implementation, based on an exchange of pulic/private keys and their collection into a keyring:

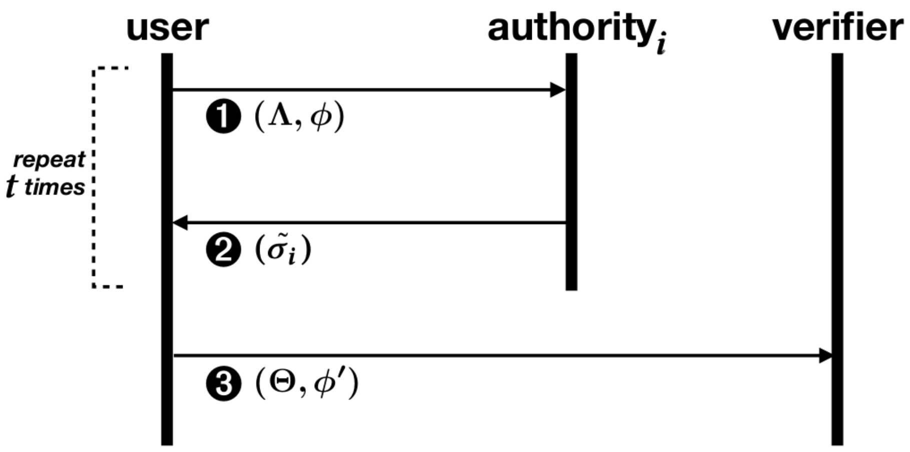

Given that I am known as 'Bob'
When I create my new keypair
Then print keypair 'Bob'
send public key
{ public: zenroom.ECP }
Given that I am known as 'Alice'
and I have my keypair
and I have a 'Bob' 'public' key
When I import 'Bob' keypair into my keyring
Then print my keyring
save keypair into keyring
{ Bob: { public: zenroom.ECP,
private: zenroom.octet },
Alice: { public: zenroom.ECP } }
Given that I am known as 'Alice'
and I have my keypair
and I have the 'public' key 'Bob' in keyring
When I draft the text 'Hi Bob!'
and I use 'Bob' key to encrypt the text into 'ciphertext'
Then print data 'ciphertext'
send a secret message
{ schema: 'AES-GCM',
curve: 'bls383'
text: zenroom.octet
pubkey: zenroom.ECP
checksum: zenroom.octet
iv: zenroom.random
zenroom: '0.9'
encoding: 'hex' }
Given that I am known as 'Bob'
and I have my keypair
When I decrypt the 'ciphertext' to 'decoded'
Then print data 'decoded'
Reply the secret message
{ decoded = { from: 'Alice',
text: 'Hi Bob!' } }
Given that I am known as 'Bob'
and I have my keypair
and I have the 'public' key 'Alice' in keyring
When I draft the text 'Hi Alice, lets talk!'
and I use 'Alice' key to encrypt the text into 'ciphertext'
Then print data 'ciphertext'

This simplified flow diagram shows **actual Zencode** that can be executed, higlighting variables that are normally just surrounded by single quotes. Between each code block, which is executed asynchronously as required and at different times, there is a schema which indicates the shape of data in output.

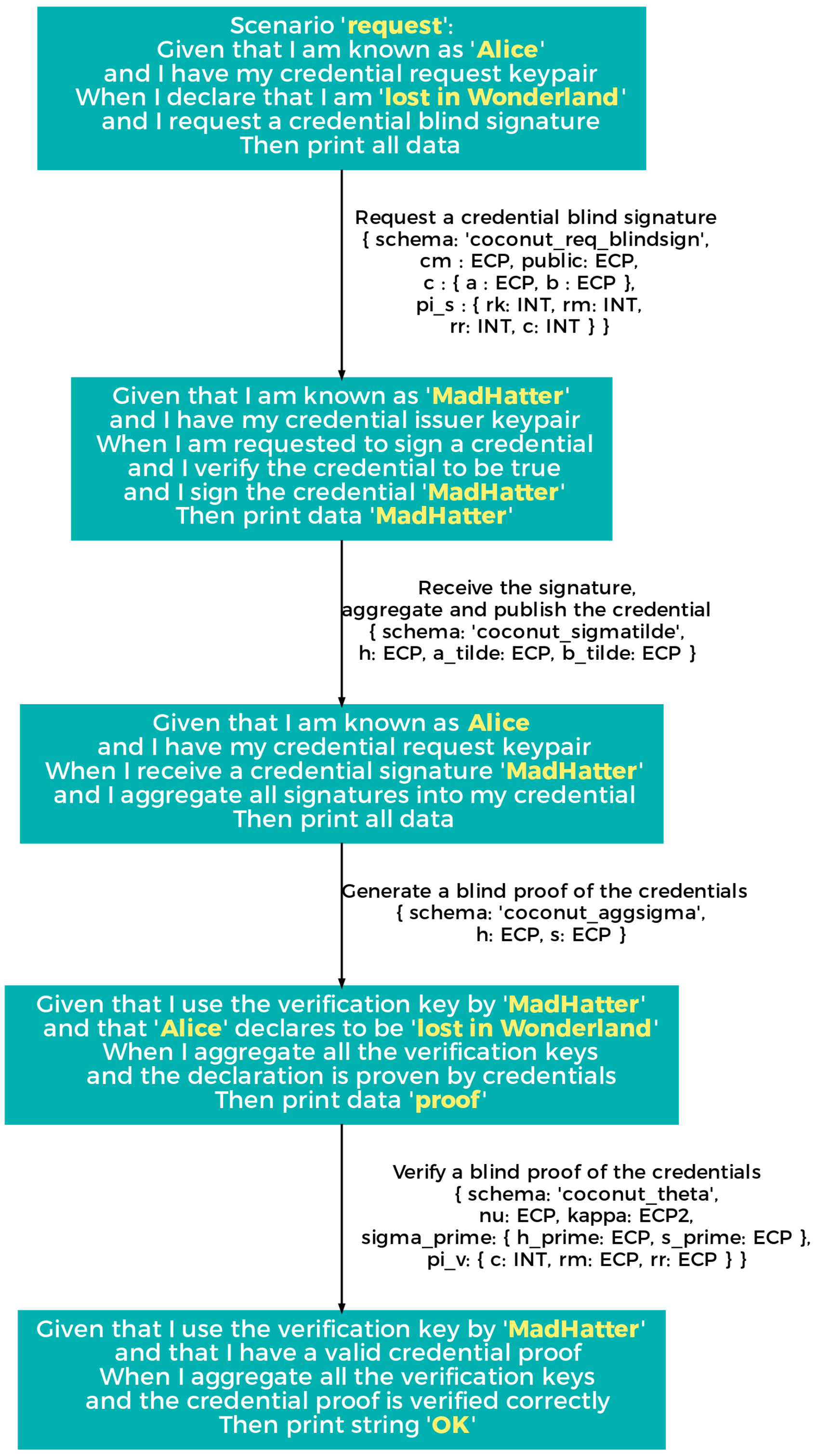
What follows is another flow diagram leading to data outputs that can be reused into the above: is the use of ECQV implicit certificates via Zencode, which leads to obtaining public/private keypairs that are compatible with asymmetric encryption.


Scenario 'request':
Make my declaration and request certificate
Given that I introduce myself as 'Alice'
and I have the 'public' key 'MadHatter' in keyring
When I declare to 'MadHatter' that I am 'lost in Wonderland'
and I issue my implicit certificate request 'declaration'
Then print all data
Declare and request certificate
{ declaration_keypair:
{ private: zenroom.octet
public: zenroom.ECP }
declaration_public:
{ statement: 'lost in Wonderland'
from: 'Alice'
public: zenroom.ECP
to: 'MadHatter' } }
Scenario 'issue':
Receive a declaration request and issue a certificate
Given that I am known as 'MadHatter'
and I have a 'declaration_public' 'from' 'Alice'
and I have my 'private' key in keyring
When I issue an implicit certificate for 'declaration_public'
Then print all data
Issue a certificate
{ declaration:
{ hash: zenroom.octet
certificate: zenroom.ECP } }
Scenario 'challenge':
Receive a certificate and use it to encrypt a message
Given that I am known as 'Bob'
and I have my 'private' key in keyring
and that 'Alice' declares to be 'lost in Wonderland'
and I have a 'certificate' 'from' 'MadHatter'
When I use the 'certificate' to encrypt 'Hi Alice!'
Then I print all data
Encrypt a message
using the certificate keypair.
Bob and Alice communicate privately,
Alice's correct answers are a proof of certification

At last, below is a diagram showing again the code and the data-structures of the credential authentication mechanism implemented following the Coconut paper (Sonnino et al., 2018) and illustrating the flow of request, issue and publication of credentials outlined in this graph:



And realised in Zencode language format as illustrated by the following figure:



```
Scenario 'request':
Given that I am known as 'Alice'
and I have my credential request keypair
When I declare that I am 'lost in Wonderland'
and I request a credential blind signature
Then print all data
```

Request a credential blind signature
{ schema: 'coconut_req_blindsign',
cm : ECP, public: ECP,
c : { a : ECP, b : ECP },
pi_s : { rk: INT, rm: INT,
rr: INT, c: INT } }

```
Given that I am known as 'MadHatter'
and I have my credential issuer keypair
When I am requested to sign a credential
and I verify the credential to be true
and I sign the credential 'MadHatter'
Then print data 'MadHatter'
```

Receive the signature,
aggregate and publish the credential
{ schema: 'coconut_sigmatilde',
h: ECP, a_tilde: ECP, b_tilde: ECP }

```
Given that I am known as Alice
and I have my credential request keypair
When I receive a credential signature 'MadHatter'
and I aggregate all signatures into my credential
Then print all data
```

Generate a blind proof of the credentials
{ schema: 'coconut_aggsigma',
h: ECP, s: ECP }

```
Given that I use the verification key by 'MadHatter'
and that 'Alice' declares to be 'lost in Wonderland'
When I aggregate all the verification keys
and the declaration is proven by credentials
Then print data 'proof'
```

Verify a blind proof of the credentials
{ schema: 'coconut_theta',
nu: ECP, kappa: ECP2,
sigma_prime: { h_prime: ECP, s_prime: ECP },
pi_v: { c: INT, rm: ECP, rr: ECP } }

```
Given that I use the verification key by 'MadHatter'
and that I have a valid credential proof
When I aggregate all the verification keys
and the credential proof is verified correctly
Then print string 'OK'
```

3.1. Implementations

At the time of writing all functional prototypes in DECODE are embedding Zenroom and can therefore seamlessly implement Zencode without adding more work to implementors, but simply substituting the current Lua based Zenroom scripts to their Zencode implementation. Below a list ofsoftware prototypes also visible at https://github.com/decodeproject

- Mobile app (Zenroom embedded as a react-native javascript component, soon to be converted to native https://github.com/DECODEproject/wallet
- IoT encoder (Zenroom embedded via Go bindings) https://github.com/DECODEproject/iotencoder
- Chainspace (Zenroom binary executed separately) https://chainspace.io

All DECODE pilots benefit from this development which is successfully integrated through these components. The DECIDIM pilot still needs a working cryptographic implementation of its petition contract in order to be translated to Zencode; the IoT based pilots can all immediately benefit from the Zencode implementation of DH asymmetric encryption based on AES-GCM secure standard; the Amsterdam register pilot can immediately benefit from the Zencode implementation of ECQV implicit certificates.

Future horizons of development of Zencode include further implementations supporting interoperable and extensible crypto schemes on the same EC curve that can still work with the above implementations, as well further refinement of the parser and extension of the schema validation. From this point onwards Zencode must be informed by piloting, while it will be also refined in cooperation with legal experts to match the smart-rule statements so far identified to express consensual data processing conditions.

4. Integration

The integration of Zencode is so far relying on the same integration schemes present for Zenroom, with the addition of a minimal layer of boilerplate code for its execution. This is so to facilitate flexibility in piloting, but will be later changed to lock down to the sole execution of Zencode via new specific API calls.

Therefore, for now, in addition to the C call that we have exported to Java, Go, Python and Javascript languages along with utility wrappers:

```
int zenroom_exec(char *script, char *conf, char *keys,
             char *data, int verbosity);
```

We also have the boilerplate internal to the 'script' buffer:

```
verbosity_level = 1

ZEN:begin(verbosity_level)

ZEN:parse([[
-- your zencode here
]])

ZEN:run()
```

The execution of actual Zencode lines happens sequentially at the time of the 'ZEN:run()' call. Each line as part of the whole statement block (utterance) makes use of data types which may or may be validated and should be present in the KEYS and DATA buffers. A list of Zenroom/Zencode integrated implementations follow: they have been developed in relation to each pilot software implementation as needed, covering several languages.

- Go language bindings https://github.com/DECODEproject/zenroom-go
- Python language bindings https://github.com/DECODEproject/zenroom-py
- Java (JNI) and SWIG (universal) language bindings are inside Zenroom's source repository https://github.com/DECODEproject/zenroom

Also notable the presence of the 'zenroom' module inside the NodeJS Package Manager collection (NPM) and of course its extremely portable WebAssembly optimized build (universal binary) see: https://www.npmjs.com/package/zenroom As well the packaging of a Docker container: https://hub.docker.com/r/dyne/zenroom

Even considering the work ahead to integrate needs of pilots into cryptographic contracts that need to be translated to Zenroom and then wrapped into Zencode, it is evident that the way we engineered the Zenroom VM and the Zencode DSL will make it easy to integrate it in new applications.

5. Bibliography

Adrian, D., Bhargavan, K., Durumeric, Z., Gaudry, P., Green, M., Halderman, J.A., Heninger, N., Springall, D., Thomé, E., Valenta, L., VanderSloot, B., Wustrow, E., Zanella-Béguelin, S., Zimmermann, P., 2015. Imperfect Forward Secrecy: How Diffie-Hellman Fails in Practice. Presented at the Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security, ACM, pp. 5–17. https://doi.org/10.1145/2810103.2813707

Ascott, R., 1990. Is There Love in the Telematic Embrace? Art J. 49, 241.

Colesky, M., Hoepman, J.-H., Hillen, C., 2016. A critical analysis of privacy design strategies, in: Security and Privacy Workshops (SPW), 2016 IEEE. IEEE, pp. 33–40.

Danezis, G., Domingo-Ferrer, J., Hansen, M., Hoepman, J.-H., Metayer, D.L., Tirtea, R., Schiffner, S., 2015. Privacy and Data Protection by Design-from policy to engineering. ArXiv Prepr. ArXiv150103726.

Diakopoulos, N., 2016. Accountability in algorithmic decision making. Commun ACM 59, 56–62.

Hoepman, J.-H., 2014. Privacy design strategies, in: IFIP International Information Security Conference. Springer, pp. 446–459.

Monico, F., 2014. Premesse per una costituzione ibrida.: la macchina, la bambina automatica e il bosco. AutAut Condizione Postumana.

Pelizza, A., Kuhlmann, S., 2017. Mining Governance Mechanisms. Innovation policy, practice and theory facing algorithmic decision-making. Handb. Cyber-Dev. Cyber-Democr. Cyber-Def.

Roio, D., 2018. Algorithmic Sovereignty (PhD Thesis). University of Plymouth.

Sassen, S., 1996. Losing Control? Sovereignty in an Age of Globalization. Columbia University Press.

Sonnino, A., Al-Bassam, M., Bano, S., Danezis, G., 2018. Coconut: Threshold Issuance Selective Disclosure Credentials with Applications to Distributed Ledgers. ArXiv Prepr. ArXiv180207344.

Wynne, A., 2012. The Cucumber Book: Behavior-Driven Development for Testers and Developers.