Red/C3 Research and Development

Vladimir Vasilyev

> *Someone has to get this information to the people!*

Contents

Introduction

This document is a preliminary research and domain analysis of Ethereum blockchain, smart-contract languages (SCLs), and everything in-between. It is written in order to help Red team during the upcoming development of Red/C3 project, and, as such, consists of:

- collected body of annotated literature, that absorbed in itself more than 50 research papers, numerous technical reports, blog posts and presentations;
- closer inspection of exiting problems and open opportunities in Ethereum ecosystem;
- outline of do's and don'ts — various aspects that deserve closer attention, as you deem fit;
- general TL;DRs with pros and cons of most (if not all) existing approaches to SCL development.

In the course of this research I ineviteably imposed déformation professionnelle on myself, and cannot provide a broader perspective on the research topic at hand. Instead of ignoring collected biases and newly formed point of view, I decided to share my own perspective on C3 design, and provide a general outline of one possible solution, which, I believe, closely follows Red's design and philosophy. Thus, the title is a bit misleading — it is more "Design speculations" than "Development". This also implies the subjectivity of all the opinions expressed here, and makes them subject to a collective discussion.

The work on this research spanned 2 months, and report itself is for Red team's eyes only (at least initially). In the following pages, formal `rigor` is sacrificed in favor of flexibility and ease of expression — I give you a friendly advice, not the ultimate scientific truth. Besides, written communication is not worth it if you can't spare a joke with a pun or two :^)

Last but not least — all the murky technical details of blockchain and Ethereum are ommited. I confidently assume that readers are familiar with most of them; if not — an extensive bibliography should serve as a refresher.

Without a further ado, let's release the Kraken! But let us do so tersely — for time is of the essence.

What's in a name?

What does *"Smart Contract Language"* actually mean? A simple `split "Smart Contract Language" space` may shed the light on this question:

Smart [contract]

An autonomous actor on decentralized network.

Contract

An agreement or binding between two or more parties.

- **Legally**, a contract is commonly defined as an agreement between parties which is protected by law.
- A contract in the world of **financial** trading is a different concept: a promise of payment between parties, whose value changes over time, and which can be traded as an asset.
- Finally, contract might be an overly stricted term for **informal** agreement.

[Formal] language

A symbolic system for expressing of and reasoning about a class of particular objects or concepts.

That is, summing it all together:

Smart contract language

A symbolic system for reasoning about legal, financial and informal agreements or bindings between two or more parties, and for expressing them in an algorithmic form for autonomous actor(s) to execute and for consensus algorithm(s) of the underlying decentralized network(s) to enforce.

The task of designing a smart contract language (SCL), then, is a threefold combination of the terms above:

(1) Contract

Find the adequate model for expressing contractual terms and their properties.

(2) Smart contract

Examine issues and open challenges of blockchain technology, hard constraints of its execution model, and soft constraints of development ecosystem that surrounds them (i.e. user requirements). We will consider Ethereum only, as it's the most promising and prevalent platform, and the first chosen compilation target for C3.

(3) Domain-specific language (DSL) with its associated toolchain

Choose a formalism that can adequately fit (1) into (2) with reasonable trade-offs, and design an ensemble of tools that alleviate major pain points and address existing issues.

Red/C3, being a DSL (dialect) for smart contract development, must merge *non-operational* (1) and *operational* (2) aspects together. If designed well, it will *transcend* the blockchain bubble, becoming a platform-independent contract notation.

Blockchains come and go, and operational aspects will change over time (e.g. introduction of NEO backend or eWASM compilation target, support of Bitcoin). Merkelized bandwagon may very well turn turtle on the next downhill of ETH/USD exchange rate, which already looks like an existentially burdened roller coaster. But *contract* itself, as an agreement or binding between humans, won't go anythere.[1]

DO

Strive for clear separation of “contract” (as in agreement) and “smart” (as in blockchain). Both are orthogonal. While potential of blockchain is disruptive, it's nature is brittle. If “smart” dies, “contract” must remain.

As such, the list above outlines the remainder of my report and serves as a general overview of all the chapters: we will start with contract models and formalisms (1), continue with blockchain and EVM quirks (2), and end up with design considerations of yours truly (3).

[1] Unless humanity itself will cease to exist in the near future.

1 Contract

1.1 Requirements

What constitutes an *adequate* contract formalism? To rephrase: what requirements given formalism must fulfill in order to capture contract model fully and naturally? Let me sketch a general outline, based on the definition of a contract I gave in the previous chapter. That is, contract formalism must be able to express the following aspects:

1. Participants (Bob and Alice).
2. Commitments (obligations, permissions, prohibitions, i.e. *deontic modalities*).
3. Constraints (deadlines and real-world factors).
4. Reparations (what comes in effect when commitment is not fulfilled or constraint is violated).
5. Actions (what actions participants commited to [not] perform).

Surprisingly, this intuitively given definition precisely captures contractual terms. *Institutional grammar*, introduced by Crawford et al. [4] in 1995, was designed to express the rules, norms and shared strategies of human interactions in institutions. Grammar components are reffered as ADICO (see Table 1):

Component	Aspect
Attributes	Participants
Deontic	Commitments
A**i**m	Actions
Conditions	Constraints
Or else	Reparations

Table 1: *components of institutional grammar and their mappings to our definition.*

In contrast to such coarse-grained definition, Tom Hvitved [1] gives more fine-grained and stricter list of requirements that contract formalisation must fulfill:

1. **Contract model, contract language, and a formal semantics** — semantic, mathematical model and syntactical representation. Ideally, formal model coexist with formal language in the form of language's semantics.
2. **Contract participants** — self-nominating requirement.
3. **(Conditional) commitments** — self-nominating requirement. Commitments may be conditional on what happens during the execution of a contract.
4. **Absolute temporal constraints** — absolute deadlines, e.g. *return 10$ before 1-Jan-2019*.
5. **Relative temporal constraints** — relative deadlines, e.g. *pay 45$ 90 days after delivery*.
6. **Reparation clauses** — what comes in effect then another clause is not fulfilled.
7. **Instantaneous and continuous actions** — e.g. *pay 10$ instantly*, or

stretch the payment over the course of two weeks.

8. **Potentially infinite and repetitive contracts** — the lease agreement is an example of a potentially infinite, repetitive contract, since the contract is renewed by default every month, unless the tenant or the landlord decide to terminate the agreement. *Potentially infinite* means that a latest time of termination is not known in advance when the contract is signed. The contract can, in principle, run forever.
9. **Time-varying, external dependencies (observables)** — objective, but perhaps time-varying, quantities, that are to be measured on particular date. *"Objective"* means that at any particular time the observable has a value that all parties of a contract will agree upon. Observables can form conditions that cannot be controlled by contract participants. Say, in contract *"pay amount of USD equal to the noon centigrade temperature in LA"*, *"noon centigrade temperature in LA"* is an observable.
10. **History-sensitive commitments** — commitments may depend on what has previously happened during contract execution. History sensitivity differs from conditional commitments in that the latter refers to commitments that may or may not become active, whereas the former refers to commitments where the exact terms depend on previous events.
11. **In-place expressions** — the possibility to write expressions in contracts (arithmetic, binding statements, observable queries &c).

Ideally, formalisation must also support:

12. **Parametrized contracts** — templates, to put it simply.
13. **Isomorphic encoding** — the *isomorphism principle* refers to formal encodings that are in one-to-one correspondence with the informal paper contracts. That is, one component (paragraph) in the paper contract corresponds to one component in the formalization, and dependencies between components in the paper contract are present between the corresponding components in the formalization.
14. **Run-time monitoring** — the ability to monitor the execution of a contract in real-time, that is, to check whether the contract has been breached and to report upcoming commitments in order to avoid future breaches of contract.
15. **Blame assignment** — in the case where a contract is breached, the monitor should not only report a breach of contract, but also who among the contract participants is responsible.
16. **Amenability to (compositional) analysis** — examples of analyses that are relevant for contracts include:

    - **satisfiability** — whether a contract can be fullfilled;
    - **satisfiability with respect to a particular party** — whether a party can avoid breaching a contract in which it is involved;
    - **contract valuation** — what is the expected value of a contract for a given party;
    - **contract entailment** — whether fulfilling a contract entails the fullfilment of another contract.

Compositionality means that analyses can be defined universally for all possible contracts that can be expressed in the formalism. That is, analyses are not defined *ad hoc*, such as valuation of one particular type of instalment sale.

Clack et al. [3] in their treatment of smart contract templates make an important observation:

> Smart contract is a Ricardian triple of legal prose (*legalese*), parameters and code.

With all of the above (and C3 initial design outline [5] in our hands), we can draw some useful insights. I formulate them as provocative questions:

- Contracts have temporal aspects. How can they be modeled in the

context of blockchain, where only block timestamps are present, and transaction order is imposed by miners?
- How contract participants can be modeled, if all what we have is a set of 160-bit addresses, which can be dynamically extended at runtime (i.e. smart contract may call / create other smart contracts and recieve calls from multiple actors)?
- Contracts have reparation clauses. How to handle them at exception level?
- Contracts have external dependencies (observables). Blockchain handles them via *Oracles* — certified services that provide authenthic data from the external providers. How can we tackle that?
- Execution of a smart contract is tied to gas fees. Should high-level contract definition somehow mirror that fact? What aspect of a contract can be modeled in terms of gas consumption?
- Contracts may have history-sensitive commitments. While EVM allows to do log entries, they are expensive (gas-wise). Should history of a contract be part of its execution? Or should it be modeled exclusively by testing framework?
- C3 testing framework should support run-time monitor and blame-assignment. Well, should it? Or maybe the open nature of blockchain is enough? Also — compositional analysis. What model can handle that **and** history sensitivity of contracts?
- Can we model all actions as Ether transactions? Why? Why not?
- How top layer of C3 handles isomorphism principle? Can it conform to ADICO grammar and support templates?
- Contract may be potentially infinite. This is what, in some sence, Ethereum account model guarantees — once deployed, contract cannot be changed, and its code may be executed on-demand by anyone on the network. But what if contract is a buggy one, and needs to be updated or destroyed?

But, let us not dwell on it too much (yet), for we have more important things to do — Hvitved, in the first part of his Ph.D. thesis [1], also provided an extensive overview of all existing (2012) contract formalisms and compared them with a list of requirements above. Why not to examine it?

> **DO**
>
> Read the first part of Hvitved's thesis if you need an objective, birds-eye overview of contractual formalisms.

Curious reader is highly advised to read his treatise fully. As for me — I'll provide a summary of the most interesting (and powerful) formalisms. Why? Because only they fulfill the most important requirement (1), and some are already used in blockchain setting[2], which only proves their adequacy.

[2] However, most haven't yet escaped from academical cages into the wild.

1.2 Formalisms

All existing contract formalisms, according to Hvitved, fall roughly into 3 major categories:

- With focus on declarative specification and (meta) reasoning:
  - **Logic based**
- With focus mainly on contract execution:
  - **Event-condition-action (ECA) based**[3]
  - **Trace based**[4]

The ideal formalism, of course, should take the best from both worlds:

- developer should be able to *reason* about his code and to express the intended contract declaratively.

- code, written by developer, should execute correctly, and within reasonable time/space limits.

That is, smart contract written in the ideal formalism should be amenable to:

- **Validation** — checking that the expressed meaning of a plain-text contract is preserved in code and vise versa.
- **Verification** — making sure that code execution is correct and conforms to the intended computation.
- **Optimization** — tweaking execution for real-world scenarios.

We can now formulate another set of thought-provoking questions:

- What kind of logic enables one to clearly reason about contracts? What kind of mental framework fascilitates the correct expression of smart contracts?
- Of what kind of atomic units contract is composed at execution level? What are the basic blocks to build from? Can ADICO aspects be naturally expressed in terms of such blocks / atoms?
- How all of the above fits into blockchain setting and EVM execution model?

In light of the above, let me now present my own taxonomy, which, I believe, answers the posed questions more-or-less straightforwardly. I propose that all of the existing contractual formalisms, at their core, can be categorized as: **compositional**, **network-based**, or **object-oriented**.

[3] If event E occurs and condition C holds, then action A is taken.

[4] More specifically, trace-based formalisms are nothing but a process algebra. More on this later.

1.2.1 Compositional

Compositional formalisms, as evident from their name, offer a great degree of flexibility and granularity when it comes to contract specification. Compositional SCLs provide a small set of built-in primitives and rules for combining such primitives together. Add to that formal properties and clean algebraic semantics, and you'll get an ideal intermediate representation (IR) to which more high-level contractual definition can be compiled.

1.2.1.1 Combinators

It will come with no surprise that the most influential approach to compositional SCL specification is a **functional** one. Back in 00's, Peyton Jones et al. [6] introduced a *combinatorial* Haskell library (DSL) that defined a set of primitive combinators (high-level functions) for specificaion of bilatheral financial contracts. Peyton Jones himself is a major figure in Haskell world, and his work on this DSL resulted in a corporate solution, offered by LexiFi. As a consequence, majority of nowaday smart-contract language prototypes are just a variation on this theme, that is, a Haskell library with Coq proof.

Combinators themselves offer conjunctional, disjunctional and sequential composition; acqusition, transmission and scaling of currency, and management of contract's temporal properties: it's expiry date (*horizion*) and acqusition date. A sketch of Reduced combinatorial DSL is presented below:

```
ZCB: contract [
    "zero coupon bond"
    time amount currency
][
    scale amount get truncate time one currency
]

; receieve £100 on 1 January 2001
contract1: ZCB 1-Jan-2001 100 GBP

perhaps: european: contract [
    "European option"
    time contract
][
    get truncate time contract or zero
]

; the right to choose, at a given time, whenever or not to ful
perhaps 20-Feb-2000 contract1

american: contract [
    "American option"
    start end contract
][
    option: anytime perhaps end contract
    get truncate start option then option
]

; acquire an underlying contract at any time between the two c
american 1-Jan-2000 1-Jan-2001 contract1
```

The key strength of combinatorial approach (compositionality), however, does not outshine its weaknesses. In particular, PJ's seminal works is limited to *financial* contracts between *two* parties, and its operational semantics are deeply rooted in stochastic processes and game theory — these things are anything but user-friendly. Execution model is absent. Moreover, semantic is defined only in terms of *valuation*, which determines how much value given party will get out of fulfilled contract, thus bypassing all blockchain-related aspects[5] that we deem important.

What is interesting (and important) though, is that valuation process is *symbolic* — contract definition, composed of multitude of combinators, gradually residues down to a primitive combinator `zero`, which denotes a fulfillment of a contract. This clearly separates non-operational and operational aspects, as no execution takes place in the process of residuation.

DO

Consider a compositional, symbolic language as a target IR for compilation.

PJ's et al. formalism also fails to model temoral aspects and contractual parties at a larger scale. Yours truly highly doubts that Purity in its naked form has it's place in SCL domain at all — by banishing time and state it doesn't make contract programming any easier, and obscures the essential contractual properties behind the lopsided paradigm.

DON'T

Functional languages may provide a high degree of compositionality, but their support of temporal aspects and observables is highly questionable, and, more often than not, boils down to multi-layered abstractions over state management (e.g. monads).

Let's see how can all of this be applied in blockchain setting. Biryukov et al., in their work on Findel SCL [8], implemented combinatorial library as an Ethereum contract, with each combinator having an associated gas cost and execution semantic. They make an interesting point about ERC-20 token support[6], and observe the limits of Ethereum platform:

- lack of precise clock
- imperative paradigm
- underdeveloped type system

While I agree that blockchain time model is extremely tricky to handle, two other remarks sound like typical “FP must rule over the world” rebukes.

Simplicity [9] is the most interesting project based on this approach (no wonder it was (is?) considered as an IR for C3 compiler [5]). Its execution semantics is defined in terms of (virtual) Bit Machine; as such, Simplicity code can be effectively measured in terms of time/space complexity and optimized, if necessary. AST of Simplicity is represented as DAG, and can be statically analyzed and merkelized.

All in all, functional programming and combinatorial approach has its place as an intermediate representation, but not as a high-layer language. Their algebraic and strongly typed nature is a double-edged sword: it makes verification and optimization possible at the expence of validation and informal reasoning by an individual. Absence of clear deontic modalities and model of contract participants stifles high-level reasoning even further, and casts a shadow on applicability of combinator-based approaches in blockchain setting. Functional paradigm, at its core, also makes state and time modeling more challenging than they should be. Wrapping it all together:

- ADICO:
  - ☐ Participants
  - ☐ Commitments
  - ☐ Constraints
  - ☑ Reparations
  - ☑ Actions
- R16 (Hvitved's list of requirements):
  - 9/16
- Amenability to:
  - ☐ Validation
  - ☑ Verification
  - ☑ Optimization *(Arguable. Checked only because of Simplicity Bit Machine)*

I suspect that all future academical work[7] will stem from that direction, given that statical analysis, sound type systems and formal verification are all considered to be the cornerstones of modern SCL development. FP can do just that. But, as I said, it fails to model important contractual aspects, and, in its naked form, is extremely hard to reason in. For instance, American option contract took Peyton et al. 3 attempts to write correctly.

The question when arises: where can we go from that? Can we do better? Well...

[5] No surprise — blockchain didn't even existed back then.

[6] Thus, we could say that an ideal Ethereum-based formalism should support economical abstraction over Ether.

[7] Business startups are catching up too. For instance, Firmo ~~technical paper~~ fluff piece [10] is a 1-to-1 copy of Peyton et al. seminal work and formalism.

1.2.1.2 Traces

... let's take the functional approach and list all the things missing:

- Deontic modalities
- Temporal aspects
- Multiple parties
- Constraints, in-place expressions
- User-friendly representation
- Clearly defined execution model and operational semantics

Now, if you'd devised a new SCL that addresses all these issues, while preserving the high degree of flexibility and algebraicity that combinators offer, you'd naturally arrived at **trace-based** formalisms — a second member in the compositional category; an interesting, and, dare I say, a damn powerful one.

And the crux of it is a bit surprising. At the very core, as I said earlier, trace-based formalisms are a process algebra, highly inspired by Tony Hoare's communicating sequential processes (CSP) [11]. Now, this demands some explanation.

Process can be essentially described as a set of *traces*. Trace itself is just a timestamped sequence of events that a given process might be observed to perform. To rephrase this in an important observation: contract is a timestamped sequence of events that conforms to a set of constraints, expressed as deontic modalities, conditional expressions and observable queries. To put it even more bluntly:

> **DO**
>
> Contract is inherently a reactive process.

All cool and dandy, but, this sounds like a crazy talk and speculative hand-waving. Do processes *really* have *anything* in common with contracts? Andersen et al. [12], in what I believe is a seminal paper on the subject (and an extremely well-writen one!), observed the following fundamental patterns of (de)composing contract out of subcontracts and atomic commitments:

- **Atomic contract** — a commitment stipulates the transfer of a resource or set of resources between two parties (that is, a *transaction* (sic!))
- A contract may require **sequential** execution of subcontracts
- A contract may require **concurrent** execution of subcontracts, that is, execution of all sub-contracts, where individual commitments may be interleaved in arbitrary order
- A contract may require execution of one of a number of **alternative** subcontracts
- A contract may require **repeated** execution of a subcontract

> **DO**
>
> Seek for a formalism in which contract can be specified compositionally and reflect the above mentioned patterns. Above that, strive to separate contract composition (contract language) from the definition of atomic commitments (base language), including their temporal properties, set of parties and transacted values.

Striking, isn't it? Description of these basic properties highly ressembles CSP primitives! (see Table 2 below)

Contractual primitives	CSP operations
Atomic transaction	`STOP` and `SKIP`, → (prefixing)
Seqential composition	`;` (sequential composition)
Concurrent composition	`|||` (interleaving), `|[{..}]|` (interface parallel)
Conditional alternatives	`if .. then .. else ..`, □ (external choice)[8]
Repetition	recursion, `:` (choice of)

Table 2: *relation between contractual properties and CSP primitives.*

Not only that, but the big influence of combinatorial approach is highly noticable. No surprise here — Bahr et al. [7] extended seminal work of Peyton Jones with more rigid time model, static type system, and observables, and ended up with a trace-based formalism. Moreover, Hoar himself [11] initially modelled processes as Lisp *functions* that accept symbols (events) and other functions (processes) as their arguments.

> ⓘ **DO**
>
> Contract itself should be a first-class citizen. That is, contracts, as entities, should be able to pass and exchange other contracts between each other.

We can alsow draw the clear parallels with Hvitved's R16 list of requirements:

- Since contract is characterized by its traces, and traces are effectively a log of events that may happen (or already happened), history-sensitivy comes at no cost.
- And since trace is a *timestamped* sequence, temporal constraints with instantaneous and continious actions can be modeled with ease.
- Potentially infinite and repetetive contracts rely on recursion and "splicing" of events with processes via `:` operator.[9]
- Conditional commitments and observables are covered by processes' reactions to events and alternations.

What's misssing, however, are deontic modalities and some model of contract participants, but, most importantly, a clear execution model and operational semantics. Fear not, for Hvitved comes to the rescue again, with Bahr as his sidecick [13]. Together they devised a (formaly proved) Contract Specification Language (CSL) with *run-time monitor*. Not only this means that operational semantics is specified in terms of symbolic residuation (as it was with combinators), but that breach of contract can be detected and reported. Thus, this is, by far, *the only* formalism that fulfills the requirement of blame assignment.

As an example, let's take the following contract:

§ 1 — Sale of goods

> Seller shall sell and deliver to buyer (description of goods) no later than (date).

§ 2 — Consideration

> In consideration hereof, buyer shall pay (amount in Ether) on delivery at the place where the goods are received by buyer.

§ 3 — Right of inspection

> Buyer shall have the right to inspect the goods on arrival and, within (days) business days after delivery, buyer must give notice (detailed-

claim) to seller of any claim for damages on goods.

And Reduce it down to a CSL-like specification (based on Andersen et al.):

```
Red/C3 [
    Legalese template with holes to fill in, as above.
    ...
    Contract participants:
    Harald   <account-address>
    Vladimir <account-address>
    ...
]

spec: [seller buyer goods payment days date notice]

nonconforming: contract spec [
    if all [
        (delivery: now) < date + days
        goods are not 'hot before date
    ][
        buyer sends seller notice
        then if now < delivery + days [
            seller sends buyer (payment / 2)
        ]
    ]
]

sale: contract spec [
    if now < date [
        seller sends buyer goods
        then if now < date [
            buyer sends seller payment
        ]
    ]
    otherwise nonconforming seller buyer goods days now notice
]

sale @Harald @Vladimir (unpack stickers) 100$ETH 14-Sep-2018 8
```

As you can see, contract is described as a set of inter-recursive template definitions, each containing all the necessities, such as in-place expressions, conditionals, observable queries, atomic transactions and reparations.

It is important to note that CSL was developed as a part of POETS (Process-Oriented Event-driven Transaction Systems) framework [15], specifically designed to represent ontological and domain knowledge of resource accounting in enterprise setting. Thus, one may argue, that trace-based formalisms, being rooted in neckbearded academical theory and having >20 years of CSP development behind their back, may have their merit in a typical business setting.

The only question remained is relation to blockchain and possibility of execution by Ethereum VM. Here I can only point out to RChain Cooperative [16], as its Rholang language is rooted in π-calculus (process algebra). Sales pitches are: blazingly fast transactions, infinite scalability, guaranteed security backed up by formal verification. There are also a couple of works from Atzei [17] and Bartoletti [18] concerned with Bitcoin smart contracts, and they highly resemble process calculus too. But whenever all of this can fit into EVM is an unanswered question.

If we trace (pun intended) back a bit, then we find out that Occam language, designed for parallel computation from the ground up, is nothing but a CSP. In turn, transputer microprocessors, on which Occam runs, are stack-based, so as EVM. If one will muse over this relation between highly parallel and stack-based architectures, he'll ineviteably recall that Chuck Moore, Forth's (stacks!) father, was behind Green Array chips and UltraTechnology.

I know, I know, I speculate too much. Point is, as good as trace-based formalisms are:

DON'T

I'm not quite sure that trace-based and CSP-inspired formalisms can be *efficently* compiled and executed on Ethereum virtual machine, and wasn't able to find any academic work or FOSS project in this regard. As such — not bulletproofed, caveat emptor.

But, this is just my gut speaking. Given that traces are a natural extension of combinators (which already were battle-tested on EVM), it will be quite safe to assume that their mapping on execution model is achievable. What's left to be desired though, is the following:

- Traces, just like combinators, tend to sacrifice informal reasoning in favor of formal rigor, albeit only on semantical level. While trace-based formalisms syntax is quite readable (see CSL example above), the CSP theory behind them requires one to grow a bushy beard and to weave a Ph.D. thesis from it.

- Deontic modalities are veiled behind event-driven conditionals and inter-process communications. Contract participants are not presented as concrete entities and objects, but rather are smeared over traces as transaction *between* actors. A natural extension, that would address this issues, is required. For example, a conditional construct not driven by reactions to events, and a catch-throw exception mechanism for reparation clauses and unexpected events. Some syntax sugar in form of lambda expressions and participants-as-objects, with minimal core library, would be nice to have.

- If we accept “contract as a first-class citizen” mechanic, then it is imperative to have a high-level predicate language that will impose a set of constraints on all subcontracts and set of operands that compose a given contract. Now, throw mutually recursive descriptions of contractual templates into the mix, and you'll get yourself a bit of a

headache.

- Recursion. There's hardly any practical contract which is infinite. Even those that are supposed to run for a very-very long time are periodically renewed by all parties, that is, a new contract takes place, identical or a partially evaluated one. It is also obvious that potentially infinite looping has no place in gas-bounded computations. As such, there's a need for a limited recursion (see footnotes).

- Events as interned strings or even `word!`s do not carry enough semantic information to dispatch upon. Some information need to be directly encoded in them, and the basic set of built-in event types should be provided.

- Time model should be extended to encompass not only discrete timepoints, but also continious intervals, both absolute and relative.

With all that said, it is worth to mention that second part of Hvitved's thesis is about trace-based formalism mixed with dynamic logic for representation of deontic modalities [14]. CSL language, which we briefly reviewed earlied, is an extension of his work and that of Andersen et al [12].

Traces, in their bare form, only slightly extend combinators by adding an adequate temporal model. I believe that dynamic, highly reactive nature of contracts, as posed by trace-based formalisms, is more realistic than a static functional one, which would require some sort of FRP framework to play with traces on equal terms. Coupling with dynamic logic only contributes to this advantage, and CSP-based semantics further solidifies it in a formaly verified, industry-proven cement.

DO

Traces inherit combinatoral approach and are strictly more powerful than purely functional formalisms.

Inherently concurrent and reactive view of contracts, however, comes at a price. Not only they are hard to reason about for an average programmer, but some features, such as mutual recursion, symbolic residuation and event constraints, are hard to model and tricky to get right on a larger scale. Absence of EVM-based projects, related to trace-based formalisms, casts a shadow of uncertainty on the practicality of the subject. On the other hand, enterprise systems, such as abovementioned POETS, and even blockchain architectures (RChain) are based on traces and associated properties, which gives this approach a much deserved credibility.

I also suspect that optimization can be achieved only on architectural level, and/or inside compiler's backend. User cannot tweak the efficency of his contract as he writes it, unless there exists an escape-hatch to a much lower, LLL-like language for an extreme, to-the-metal code tuning. This remark is related not only to traces, but to all discussed formalisms — what can we offer to control freaks and “I'll do it myself” types?

- ADICO:
  - ☐ Participants
  - ☑ Commitments
  - ☑ Constraints
  - ☑ Reparations
  - ☑ Actions
- R16:
  - 13/16
- Amenability to:
  - ☑ Validation *(The most readable one among all other formalisms. More on this later)*
  - ☑ Verification
  - ☐ Optimization

[8] I wonder if⊓ (internal choice) can be adapted to represent nondetermenistic aspects of blockchain? E.g. choice based on a transaction order. I doubt that we should encourage such bad practice though.

[9] What is crucial, however, that recursion and loops, especially potentially infinite, have no place in EVM execution model. This can be ruled out with symbolic unfolding and gradual constraining of a loop, a technique which, in crypto parlance, is called a *covenant*.

1.2.2 Network-based

Network-based formalisms model a contract as a set of nodes, which represent current state of affairs, and a set of edges, which denote transitions between states and reactions to external events. Combined together, nodes connected by edges results in a graph relation, or, more specifically, a network of interleaving elements. They an be conviniently drawn or converted to XML-like hierarchical format, and extensively analyzed by well-established means.

1.2.2.1 Petri nets

I'll be a bit historical and start with the oldest formalism in this category. Are you ready? *Petri nets*. Yeah, you heard that. But don't cast a “shush, madman!” on me, it actually makes quite a lot of sense, especially in light of our past observations.

During our treatise of compositional approach to SCL development, we found out that contracts are, at their core, reactive processes — a concurrent system of relations between participants, that require synchronization over time and events, and also sequential and conditional transactions of shared and distributed resources.

If we wish to model our contracts, then, we need a formalism that will catch the essence of concurrent behavior and distributed systems. Fortunately, Petri nets are all about that. Moreso, by their nature, Petri nets are nothing but direct bigraphs, that is — a network!

Now, as promised, a bit of a history lesson. *The very first* attempt at e-contract formalisation was done in 80's by Lee, in his seminal paper [19] on logic model of electronic contracts. This work, being the first of a kind, is of a grave importance, for it shows us modeling of contractual aspects uninfluenced by any other approach. That is, we can see reasoning about contract model that does not dovetails trends of research community — a pure insight in and of itself.

So, what do we see? Lee embraces *temporal*, *deontic* and *performative* aspects of contracts, and uses a predicate logic over Petri net state transitions to model them, in the form:

```
transit Precondition Postcondition [Party Deadline Action]
```

Which reads as: “if Precondition (predicate on the current active states) holds, and event (Action is taken by Party fulfilling some Deadline) occurs, then Postcondition (defining the new active states) holds”.

Lee further extends predicate language with Von Wright's T calculus, to model *sequential* and *concurrent* state transitions, and with temporal logic system of Rescher and Urquhart (RU calculus) for modeling of *absolute* and *relative* time constraints (deadlines). Deontic modalities are represented as transition paths that gradually lead either to contract fulfillment or violation.

Being a pioneer in e-contract formalisation, Lee correctly stressed out the importance of temporal and deontic aspects of contracts, and, by modeling them as Petri nets, emphasized their reactive, concurrent nature.

According to Hvitved's survey, this is one of the most powerful formalisms (on a par with traces), which manages to fulfill most of the requirements, except

for the most important one — presence of formal semantics and execution model. Instead, Lee presents a BNF grammar and outlines the possibility of implementing contractual semantics in Prolog, and also sketches the natural language interface (isomorphism principle). Whenever or not it is possible to do so — no one knows for sure. The absent (and questionable) features are:

- Repetition. In the original paper this was sorted out by special states that act as objects with inner counter. I'm not sure that introduction of ad hoc constructs has its place here though, as it plays against compositionality. The question about synchronization of such counters immidiately arises, and also about presence of high-level operations over them.
- In-place expressions and constraints with commitments separated from reactions to events.

One important feature of Petri nets goes against the initially proposed C3 design: they are much easier to express graphically than textually. Consider an example contract (adapted from Hvitved):

§ 1

> Seller agrees to transfer and deliver to Buyer, on or before 2011-01-01, the goods: 1 laser printer.

§ 2

> Buyer agrees to accept the goods and to pay a total of 200 Euro for them according to the terms further set out below.

§ 3

> Buyer agrees to pay for the goods half upon receipt, with the remainder due within 30 days of delivery.

§ 4

> If Buyer fails to pay the second half within 30 days, an additional fine of 10% has to be paid within 14 days.

§ 5

> Upon receipt, Buyer has 14 days to return the goods to Seller in original, unopened packaging. Within 7 days thereafter, Seller has to repay the total amount to Buyer.

Once compiled down to Lee's formalism, it will look like this (Reduced version):

```
; § 1
transit state/1 [state/1 state/3 state/4] [:seller before 1-Ja
transit state/1 state/(default :seller) [:seller before 1-Jan-
; § 2
transit state/2 state/5 [:buyer instantly pays :seller 100$EUR
transit state/2 state/(default :buyer) [:buyer instantly not p
; § 3
transit state/3 state/6 [:buyer within 30 days pays :seller 10
transit state/3 state/7 [:buyer within 30 days not pays :selle
; § 4
transit state/7 state/8 [:buyer within 14 days pays seller 110
transit state/7 state/(default :buyer) [:buyer within 14 days
; § 5
transit state/4 state/9 [:buyer within 14 days returns :seller
transit state/9 state/10 [:seller within 14 days pays :buyer 1
transit state/9 state/(default :seller) [:seller within 7 days
```

Which points out an important quirk: not only this textual representation is too low-level and inadequate for direct contract expression, but each trasition, more often than not, should be accompanied by a twin path, which represents

the violation and non-conformance to a contract in case of non-presence of an event. This makes extension of existing Petri net specification cumbersome and error-prone, and questions the scalability and compositionality of this approach.

The coolest feature, however, stems from the fact that this formalism is implemented in Prolog. User can formulate *logical queries* as questions about a given contract. “What will happen if such-and-such sequence of events (*scenario*) happens?”:

```
at Time what if Scenario
```

Add to that the natural language interface, and you'll get a perfect environment for *manual* testing! Developer can play around with his contract, checking his understanding by probing Petri net with such queries, discovering bugs and refining contract definition on-the-fly. Such apporach cannot serve as an exhaustive testing, but, I would argue, should be a part of the testing framework.

> **DO**
>
> *Manual* (probing and poking) and *automatic* (exhaustive generation of scenarios) forms of testing should be presented. The former enables informal reasoning and helps developer build up his confidence, while the latter illuminates the blind paths which developer couldn't possibly foreseen.

In blockchain setting, Petri nets are used quite fully for dataflow analysis of the network itself. For example, Pinna et al. [20] used them for analysis of Bitcion cashflow traffic, and used gathered data to map addresses to entities behind them[10]. García-Bañuelos et al. [21] mapped Business Process Model and Notation (BPMN) to Petri nets (with further compilation to Solidity), thus encoding collaborative business processes (*choreography*) into a smart contract. This work brings to the surface three important facts:

- Petri nets are amenable to optimization (fusion of transitions, step reduction, factoring large nets into equivalent smaller ones) and verification (checking for deadlock states and infinite loops).
- Scenarios of Petri net execution (and of any timed networks for that matter) can be expressed as a set of conforming and non-conforming *traces*.

> **DO**
>
> Petri nets and timed automata can be expressed in trace semantics, thus bridging the gap between the two most powerful formalisms.

- Last but not least: network-based formalisms can be expressed graphically, and can, to some extent, be used in a business settings.

To develop the last idea: big corps and fintech are quite used to BPMN, UML and other graphical notation, and blockchain technology is a life-saver for them, as it eliminates most major pain points related to expression of business processes. However, as we all well know, business analysts are anything but skilled coders, and they'll use text-based SCL only as a last resort, usually outsorcing the task to some techies. Now, I have something to say, and I wanna say it really loud:

> **DO**
>
> If C3 want to be The Big (with a capital B), swiggy-swaggy player in Byzantium (pun intended) empire, it absolutely, positively should provide a graphical, network-based notation for smart contract development (think UML and BPMN).

If we could achieve that, then the world is our oyster. Just imagine grabbing milky golden cows by their udders, squeezing hard and whispering in the ear "we can compile your *existing* business process notation down to EVM and deploy it on the mainnet. *Right here*. *Right now*. Just give us some money[11] :^)".

Think about that. What cow would resist? No one. Ain't no one.

However, one crucial detail escaped from us, which should be stressed out. Gas cost. Can Petri net transitions emulate gas bounded execution by being bounded to some fee? Well, they *already* are bound!

Petri net transition from one *place* to another consumes an amount of *tokens*[12] and transmits them from one place to another. With formally and strictly defined execution policy, such transmisions can effectively model flow of Ethereum (and other crypto assets) between parties.

> **DO**
>
> Not only Petri net enable dataflow analysis, but they also can represent gas bounded execution and Ether cashflow.

Work of Bartoletti [22] on *lending* Petri nets (amount of tokens can go down below zero) suggest that Petri nets can also model game-theoretical nature of blockchains, where one player enters lending relationship with the other.

All in all, Petri nets, without a doubt, is a strong (if not the strongest) formalism, with the only downside of being a bit quirky in some aspects and not having an adequate textual representation (though, with a bit of effort, this limitation can be bypassed). They are ideal for analysis of a deployed ensemble of contracts, or of the blockchain network as a whole, which can be nicely visualized as animated flow of tokens from places to places. As such, I do not think that they should be a high-level graphical model, but rather a run-time representation of running contract(s), displayed by a testing framework.

- ADICO:
  - ☑ Participants
  - ☑ Commitments
  - ☐ Constraints
  - ☑ Reparations
  - ☑ Actions
- R16:
  - 12/16
- Amenability to:
  - ☑ Validation
  - ☑ Verification *(add to that theory behind the Petri nets themselves, and trace semantics)*
  - ☑ Optimization *(same as above)*

[10] Anonymity, eh?

[11] Jokes aside, I'm serious about that. Given that Fullstack seeks financial security and stability, such paid functionality / service is a must.

[12] Petri net tokens, not ERC-20 ones.

1.2.2.2 Automatas

Automatas (Finite-state machines, to be specific) closely follow Lee's approach, in a sence that they model contracts as a state-transition system. However, they do so at a different level of detail, and more explicitly — while Petri nets denote a global description, modeling with FSMs involves one automata per contract party; that is, contracts expressed as finite-state machines are projection of global contract descriptions to each of the contract

parties.

This was an approach taken by Molina-Jimenez et al. in their seminal paper [23] on X-contracts (execution contracts). Unfortunately, while being one of the universal models of computation, FSMs provide zero value when it comes to contractual specification:

- They lack any timing aspects, and cannot handle synchronious execution of contracts without introducing extreme bloat. For instance, in a case when *two* contracts are active simultaneously, we need to construct a product automaton[13], which yields $n \times m$ states when the sub automata have n and m states, respectively, making whole construct incomprehensible to depict. As a consequence, FSMs tend to grow in size rather quickly, and do not scale well.
- Their graphical representation is rather informal (at least as it is presented in Molina-Jimenez's work) and hard to reason about; their textual representation is quirky and error prone. For instance, in one of the surveys [24] conducted by Ethereum researchers, logical errors in finite-state machine encodings were considered as one of the major sources of security bugs.
- Automatas concentrate solely on execution model, and avoid defining formal language and associated semantics. It is not clear how contract participants can be explicitly modeled in this formalism, and whenever or not composition of automatas can naturally represent contract refinement.

This suggests that automatas, in their naked form (and esp. in comparison with Petri nets), cannot qualify neither as a low-level contractual model, nor as a top-level formalism, and poses a question about possible evolutions on this approach.

DON'T

Finite-state machine to Petri net is what combinator is to trace — a younger naive brother, who lives in an imaginary world unencumbered by reality.

Stegmaier et al. [25] propose compact and readable syntax constructs for specification of abstract state machines control flow: sequential and parallel execution, repetition, arbitrary choice and different steps semantics. After reading this list of features, re-examine Table 2 once again. Having déjà vu?[14]

Another important work in this area are contract automata (CA) from Azzopardi et al. [26] . Essentialy, contract automatas are multi-action automatons tagged with deontic clauses, which represent legal binding between two parties. Compositions of contract automatas is achieved with synhronisation over alphabet of actions, which again reminds us of CSP inteface parallel operator `||{..}||`. Moreso, researched defined semantics of contract automata in terms of trace-based formalism, and even devised a bi-directional mapping between the two — this suggest that reactive networks and traces can coexist and supplement each other.

DO

Contract automata can be partially expressed as traces and vice versa. That way two formalisms (networks and traces) can complement one another and provide different levels of abstractions.

A fly in the ointment: contract automata can model only two-party systems, and do not scale beyond them. There's also an interesting quirk — in CA formalism, obligation on a party to perform an action automatically induces a weak obligation on the other party to satisfy their obligation. In other words, for party A to perform an action with regard to party B, party B must *allow*

this action to happen, i.e. there should be a “handshake” synchronization over every interaction between contractual parties.

Awkward, bloaty, yes, but this “handshake” part made me realize that contracts can be viewed as *protocols* — a set of rules that endpoints of communication follow. And protocols, in a broader sense, open up a new mode of thinking about contractual formalisms, for they relate to media, politics, diplomacy and, most importantly, to object-oriented (OO) *programming* and computer *networks* (recall that we're still discussing *network*-based formalisms for contractual domain-speicifc *programming* language)!

> **DO**
>
> Contract is a protocol that governs exchange of resources between transactional endpoints.

Since OO and design by contract (DbC) is the closest we can get to protocols from programming perspective, I postpone their possible discussion until the next chapter (see section 1.2.3), and instead concentrate on relation between contract automata and traces.

As was said above, contractual automata can be expressed as a set of deontic constraints over execution trace of a contract. To rephrase — semantic of CA is defined in a subset of trace-based formalism (called *Contract Language* or CL), which is basically all the same stuff we've discussed in section 1.2.1.2. This mapping is bi-directional, but limited — some aspects of CA cannot be expressed in CL and vice versa. For example, CA has inherited notion of parties that is not in CL, but CL is more flexible and rigorious then it comes to composition, whereas CA relies on crude hierarchies of automata with reparations.

CL and CA address different types of systems — the former takes a logic-based approach to describe systems in which the parties behave autonomously, while the latter takes an operational approach for systems in which the parties interact and actions may be synchronised.

A similar, and, at the same time, quite different perspective shares John Camilleri in his PhD thesis [2] on formal modeling and analysis for normative natural language. As evident from the title, in his work he addresses translation of legalese, expressed in controlled natural language, to a formalized execution model, for further enforcement and analysis vis-a-vis external world on internal model(s).

> **DO**
>
> Testing framework should provide *static* (vis-a-vis law and standard practice) and *dynamic* (vis-a-vis some enclosing structures, e.g. traces or network of automatas) checks.

To achieve these goals, Camilleri proposes a framework architecture which, at its core, incorporates traces, *timed automata* (basically FSMs with global and local tickers slapped on top) and C-O diagrams (will be discussed in the next section).

> **DON'T**
>
> Conventional FSMs are not worth it. The more feature-complete automatas are, the more they resemble traces and other reactive network formalisms, e.g. Petri nets and networks of contract / timed automatas.

He also shows the possibility of translation between the three models (that is, C-O diagrams, networks of timed automata (NTA) and traces) in any direction, with NTA acting as an intermediary, which opens up impressive

possibilites of random testing, model checking, formulation of queries (e.g. Lee's scenarios) and integration with existing tools, like CLAN and UPAAL. Traces come with process calculus heritage, while NTA stand on the shoulders of automata theory. This and Azzopardi's work [26] again leads us to an important observation, which I want to restate here one more time:

> ⓘ **DO**
>
> Timed automata and the like, coupled with trace semantics and deontic logic, result in **the** ultimate merger. Mixed together, they `address` both non-operational (logic, validation — traces) and `operational` (execution model, verification and optimization — automata) aspects of contract specification.

Let's see how automatas stay afloat in blockchain ocean. Yoichi Hirai's Bamboo language is probably on everyone's lips, as it claims to mitigate common smart contract programming mistakes and nip security patterns (e.g. re-entrancy) in the bud — yeah, just like that! The core idea of Yoichi's approach (he's one of the main Ethereum researchers, by the way) is an explicit state management: contract execution is modeled as a two-player game (N.B. game theory) between external world and contract itself.

> ⓘ **DO**
>
> SCL should provide some means for exploration (and exploitation) of game-theorietical (as in microeconomics) aspects of blockchain.

Each “player” can do only specific “move” during the game. During this bi-latheral interaction, contract changes its state and signature. That is, during execution, contract reacts to transaction from externals worlds and “morphs” into specific versions of itself, changing smart contract's interface. *What can be called depends on which state you are in*. This is not an automata per se, but closely follows their state-dependent nature.

Bamboo compiler is written in OCaml, and emits EVM bytecode binary together with application binary interface (ABI) specification in JSON format. Language itself demands a particular mindset and style of programming: looping constructs are explicitly forbidden, so as assignments into storage `variables` (except for array elements), which forces user to rely on continuations.

Another less known (though no less interesting) project, which develops upon Bamboo's ideas, is Obsidian DSL. Its design is user-centered (i.e. developers conduct formative user studies to evaluate various language features and design decisions, and build DSL iteratively, reflecting on the recieved feedback) and is rooted in the insight that smart contracts are typically state-oriented.[15] In some sense, this approach is the exact opposite of functional programming and combinators, which strive to banish the state completely.

Let's list core Obsidian features:

- Similar to Bamboo, contract is modeled as a set of transitions between states, and each states governs what contract can and cannot do — this feature is called a *typestate*. Apparently, state itself is a first-class citizen in Obsidian.
- Each contract is modeled as an object (this means that Obsidian can also belong to section 1.2.3), which provides static typestate guarantees; but a given object can have multiple references (aliases), and not every one of them can provide such guarantees. Usage of *permissions* allows to specify which references provide which static guarantess and allow which operations.
- In each state contract can *own* a particular resource, e.g. token or cryptocurrency. *Linear resources* (linear types) make sure that such

resources cannot be used more than once, but must be used before leaving the current scope (thus ensuring that a resource is not lost accidentally).

- Each value in Obsidian may have type members, e.g. treasury can mint an unique kind of money value, dependent upon a specific value of treasury object. This means that contract can provide types instantiated by the contract object itself — this is what *path-dependent types* are.

And let's throw some stones! Even though this features sound interesting and are backed up by user feedback, I think that conducted user studies [27] [28] are too limited and do not provide enough evidence. They include the limited set of participants (mostly students), short duration (1-2 hours) and artificial tasks (e.g. rock-paper-scissors and lottery), extraction of features to a different language (backport to Java), and influence of documentation (each participant was informed about the task in advance, and language features under study were explained).

Another important point: Obsidian static features turned out to be unnatural to those who are used to more dynamic languages, and, in such cases, typically lead to desperate fights with compiler. Also, scoping rules are quite confusing: variables become available and instantiated once some state is reached, even though *lexically* we're still executing code in a particular block. This forces one first to search for an entered state, and then to jump back and forth between it and current lexical block to recall what functions and values are now available.

To wrap it up: finite-state machines cannot represent contract model adequately. Networks of timed automata and contract automata, on the other hand, are more suitable for this task, but are still limited to two-party systems with explicit synchronisation over actions. I'd say that automatas are ideal for monitoring and enforcement, and can fit well in IR role, but they should not be used as a top-level formalism.

FSMs, without a surprise, are widely used in smart-contract coding, and some SCLs seems to draw inspiration for them. However, as I said earlier, coding of finite-state machines is error-prone, and is proved to be one of the major source of SC bugs. Specific care should be taken if Red team wish to integrate FSM dialect as a part of C3.

Finally, I won't provide any mock-ups of possible C3 syntax, and instead refer reader to material on Obsidian, Bamboo, and multitudes of existing Rebol/Red FSM implementations.

- ADICO:
  - ☑ Participants
  - ☑ Commitments
  - ☐ Constraints
  - ☑ Reparations
  - ☑ Actions
- R16:
  - 7/16
- Amenability to:
  - ☐ Validation
  - ☑ Verification
  - ☑ Optimization

[13] Classical FSM is characterized as a 5-tuple $(S, \Sigma, \delta, I, F)$. Product of two such tuples (automatas), A_1 and A_2, is called product automaton, that is, $A_p = A_1 \times A_2 = (S_1 \times S_2, \Sigma_1 \times \Sigma_2, \delta_1 \times \delta_2, I_1 \times I_2, F_1 \times F_2)$.

[14] K-kansei dorifto?!

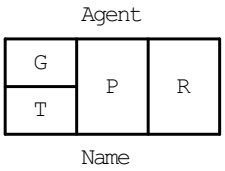
[15] It's worth to remind the reader that Ethereum account-based model is highly mutable and stateful. Though, it is unclear if making state management explicit is a net-positive or a net-negative, and if “state as a first-class citizen” language feature actually plays on Ethereum's strengths and compensates for its weaknesses.

1.2.2.3 Contract-oriented diagrams

Contract-Oriented diagrams (or C-O diagrams for short), is a visual model for compositional e-contract specification proposed by Martinez et al. [29] . It's worth mentioning that this is the first and *the only* visual formalism on our list, targeted specifically at business process developers and the like. Core design principles of C-O diagrams are:

- representation must be usable and understandable for non-expert users;
- the logic behind this representation must provide reasoning technique (*validation*);
- the internal machine-codification must be easy manipulated by programmer and allow runtime monitoring (*verification*).

C-O diagrams are capable of expressing conditional behavior and real-time constraints, including deontic modalities. Model itself represents contract as a set of trees (*forest*), where each node is a 5-element "box", corresponding to one of the contract clauses:



Elements themselves are defined as follows (see Table 3):

Box element	Description	ADICO
Agent	Indicates who is the performer of the action (contractual party)	**A**ttributes
Propositions	Normative aspects that are applied over actions, and/or the actions themselves	**D**eontic / A**i**m
Guard	A boolean expression, telling us under which conditions the clause must be taken into account	**C**onditions
Time	An associated time restriction, expressed as a period of time in which the clauses must be satisfied	**C**onditions
Reparation	Another contract that must be satisfied in case the main norm is not satisfied	**O**r else
Name	An unique box identifier, used for reference and description	—

Table 3: *relation between C-O diagram box elements and ADICO aspects*

Propositions and reparations can either specify links to other trees of contract clauses (node), or to embed atomic actions (leaf). Furthermore, propositions, either atomic or composite, can be *refined* with 3 basic compound actions (see Table 4)[16]; repetition is achieved by refering child's subclause back to parent clause.

Refinement	CSP operations
AND	||| (concurrent composition, interlevaing)
OR	□ (external choice)
SEQ	; (sequential composition)

Table 4: *refinements and their similarity to trace primitives*

classes of box fields dictate what kind of expression they can contain — static analysis, walking the tree

Permission does not have reparation

> **DO**
>
> C-O diagrams suggest the structure of abstract / concrete syntax tree (AST / CST) and point out possible relations between *multiple* contracts.

[16] This relation holds only for refinements over primitive actions, but not over composite clauses.

>>>

> ⚠ Everything below is under construction, with some breadcrumbs scattered around.

ADICO

aimed at business process developers and non-experts (recall my remark about graphical notation and BPMN) — coupled with Draw and View

service-oriented architectures and component-based systems

argue that visual representation is more intutive and user friendly — agreed

graphical language with tanks for Ether..?

Oblidge, Permit, Forbid

reparations contain either primitive actions or other C-O tree

time restrictions by means of intervals (absolute and relative) — not enough granularity (paper on timing aspects?)

no execution model — translation; AST, IR

contract can also be specified as a table

```
| Clause no. | Agent | Modality | Action | Reparation |
```

evaluation of visual model Diaz [29]

formal semantics defined in terms of timed automata Martinez [30] + Camilerri [2] (TA + traces)

subset of Timed Computational Tree Logic (TCTL) and UPAAL model checker for verification.

NTA extended with normative clauses

DbC — Eiffel, Hoare's notion of pre- and post-conditions with invariants

contract is an invariant?

To further close the gap between contracts and its representation, we consider that three criteria must be met:

- the representation must be usable and understandable for non-expert users,
- the logic behind this representation must provide reasoning techniques and
- the internal machine-codification must be easy manipulated by programmer and allow runtime monitoring.

We envision two possible ways to accomplish this:

1. the development of suitable techniques to get a proper translation from contracts written in natural language into formal languages, or
2. the development of a graphical representation (and tools) to manipulate contracts at a high level, with formal semantics supporting automatic translation into the formal language. In this paper we take the second approach.

Camilerri [2] translation of timed automata and traces to C-O diagrams

Martinez [29]

Diaz [30]

- ADICO:
  - ☑ Participants
  - ☑ Commitments
  - ☑ Constraints
  - ☑ Reparations
  - ☑ Actions
- R16:
  - N/A
- Amenability to:
  - ☑ Validation
  - ☑ Verification
  - ☐ Optimization

> **DO**
>
> Netwok-based contract models may suggest the structure of abstract / concrete syntax tree.

> **DO**
>
> Networks of contracts can point out the relation between multiple deployed smart contracts.

[2]

- Network-based
  - Automata (contractual / timed automata), Finite state machines (Bamboo, Obsidian, Linden)
  - Logic programming (Lee Petri Nets, more suitable for analysis of deployed contracts)
  - C-O diagrams

1.2.3 Object-oriented

https://languageengineering.io/a-smart-contract-development-stack-54533a3a503a https://languageengineering.io/a-smart-contract-development-stack-part-ii-game-theoretical-aspects-ca7a9d2e548d https://github.com/ethereum/langlab/issues

- Object-oriented
  - Event-condition-action (as a set of policies with encapsulated fields?)
  - Normative statements (closely follow ECA, but with little to no detail. Exclude form the list?)
  - Business contract language (Set of roles and polices)
  - Solidity
  - Voelter

SMALLTALK GODDAMIT!

1. Defeasible reasoning

1.3 Summary

Residuation, partial evaluation, symbolic execution, refinement (“X must not Y” AND “X may Z”).

Operational aspects: events, each has start and end dates (horizon, acquiry date).

Composition and description of atomic transactions.

Red should benefit on technical level (concurrency model, some symbolic extensions, reactivity) and on community level (..).

Seek to leverage Nenad's experience with R#, reactive framework and ownership model.

Deontic modalities:

- Obligations
- Permissions
- Prohibitions
- more fine-grained variations of the above (Hohfeldian modalities)

Time (Clack temporal aspects):

- Discrete
- Continious
- Absolute
- Relative
- Interval

Contrary-to-duty, contrary-to-prohibition

- Participant - address
- Commitment - ...
- Action - transaction
- Condition - ...
- Reparation - ...

Man (validation) Language (verification) Machine

o - o, **o - o**, **o - o**

Execution (intermediate representation, model) and logic (language, reasoning)

2 Ethereum

3 Design

```
nADICO (VID declarative description) -> C-O (View tree of face
                                                |
                                                v
                                      CL (traces) & CA (co
                                                |
                                                v
                                    testing framework / ru
```

Bibliography

[1] Hvitved, “Contract Formalisation and Modular Implementation of Domain-Specific Languages”. https://pdfs.semanticscholar.org/33db/bb29ed7e5b7c58dc651a8c3223bc9711a863.pdf

[**2**] Camilleri, “Contracts and Computation — Formal modelling and analysis for normative natural language”.
https://gupea.ub.gu.se/bitstream/2077/53815/1/gupea_2077_53815_1.pdf
[**3**] Clack et al., “Smart Contract Templates: foundations, design landscape and research directions”.
https://arxiv.org/pdf/1608.00771.pdf
[**4**] Crawford et al., “A Grammar of Institutions”.
http://wtf.tw/ref/crawford_ostrom_1995.pdf
[**5**] :^)
https://ico.red-lang.org/RED-whitepaper.pdf
[**6**] Peyton Jones et al., “Composing contracts: an adventure in financial engineering”.
https://www.microsoft.com/en-us/research/wp-content/uploads/2016/07/contracts-icfp.pdf
[**7**] Bahr et al., “Certified Symbolic Management of Financial Multi-party Contracts”.
http://hiperfit.dk/pdf/icfp15-contracts-final.pdf
[**8**] Biryukov et al., “Findel: Secure Derivative Contracts for Ethereum”.
http://orbilu.uni.lu/bitstream/10993/30975/1/Findel_2017-03-08-CR.pdf
[**9**] Russell O’Connor, “Simplicity: A New Language for Blockchains”.
https://blockstream.com/simplicity.pdf
[**10**] Firmo Technical Paper.
https://www.firmo.network/resources/FIRMO_TechnicalPaper.pdf
[**11**] Hoar, “Communicating Sequential Processes”.
http://www.usingcsp.com/cspbook.pdf
[**12**] Andersen et al., “Compositional Specification of Commercial Contracts”.
http://hjemmesider.diku.dk/~simonsen/papers/j6.pdf
[**13**] Hvitved, Bahr et al., “Domain-Specific Languages for Enterprise Systems”.
http://bahr.io/pubs/files/hvitved11rep-report.pdf
[**14**] Hvitved et al., “A Trace-based Model for Multiparty Contracts”.
http://www2.in.tum.de/~zalinescu/papers/hkz_jlap11_contracts.pdf
[**15**] Simonsen et al., “POETS: Process-Oriented Event-driven Transaction Systems”.
http://hjemmesider.diku.dk/~simonsen/papers/j7.pdf
[**16**] RChain whitepaper.
https://docs.google.com/gview?url=https://github.com/rchain/reference/raw/master/docs/RChainWhitepaper.pdf
[**17**] Atzei et al., “SoK: unraveling Bitcoin smart contracts”.
https://eprint.iacr.org/2018/192.pdf
[**18**] Bartoletti et al., “BitML: A Calculus for Bitcoin Smart Contracts”.
https://eprint.iacr.org/2018/122.pdf
[**19**] Lee, “A Logic Model for Electronic Contracting”.
http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.924.8107&rep=rep1&type=pdf
[**20**] Pinna et al., “A Petri Nets Model for Blockchain Analysis”.
https://arxiv.org/pdf/1709.07790.pdf
[**21**] García-Bañuelos et al., “Optimized Execution of Business Processes on Blockchain”.
https://arxiv.org/pdf/1612.03152.pdf
[**22**] Bartoletti et al., “Lending Petri Nets and Contracts”.
https://core.ac.uk/download/pdf/54606752.pdf
[**23**] Molina-Jimenez et al., “Run-time Monitoring and Enforcement of Electronic Contracts”.
http://www.cs.ncl.ac.uk/research/pubs/articles/papers/636.pdf
[**24**] Luu et al., “Making Smart Contracts Smarter”.
https://eprint.iacr.org/2016/633.pdf
[**25**] Stegmaier et al., “A Universal Control Construct for Abstract State Machines”.
https://www.uni-ulm.de/fileadmin/website_uni_ulm/iui.inst.170/home/stegmaier/ABZ2016_Stegmaier_UCC.p
[**26**] Azzopardi et al., “Contract Automata An Operational View of Contracts Between Interactive Parties”.

http://www.cse.chalmers.se/~gersch/arti2016.pdf
[**27**] Coblenz et al., “User-Centered Design of Permissions, Typestate, and Ownership in the Obsidian Blockchain Language”. https://www.hciforblockchain.org/wp-content/uploads/sites/25/2018/04/Coblenz.pdf
[**28**] Coblenz et al., “A User Study to Inform the Design of the Obsidian Blockchain DSL”. http://www.cs.cmu.edu/~NatProg/papers/barnaby17%20-%20Obsidian-plateau.pdf
[**29**] Martinez et al., “A Model for Visual Specification of e-Contracts”. https://folk.uio.no/gerardo/scc2010.pdf
[**30**] Diaz et al., “Specification and Verification of Normative Texts using C-O Diagrams”. http://www.cse.chalmers.se/~gersch/tse2013.pdf
[—] Itemis and Ethereum collaboration https://docs.google.com/document/d/1iL51Dj13qukxgCXcu8l6O2ViBPvL0-MomYLQfi1U1gA/edit?ts=59d5f128#heading=h.fwukdp6q0y9
[—] Reitwießner, “Babbage – a Mechanical Smart Contract Language”. https://chriseth.github.io/notes/articles/babbage/babbage.pdf
[—] Saunders, “A formal analysis of Hohfeldian relations”. https://www.uakron.edu/dotAsset/886214d9-cde6-46c2-8541-2dd25306fa95.pdf
[—] Pace et al., “Types of Rights in Two-Party Systems: A Formal Analysis”. http://www.cs.um.edu.mt/gordon.pace/Research/Papers/jurix2012.pdf