University College London

Department of Computer Science

A thesis submitted in partial fulfilment of the requirements for the degree of Master of Science in Computational Finance/Financial Risk Management, University College London

Smart Confirmation Contracts: An Architecture for ISDA Smart Contracts

Author
Finn Casey Fierro

Academic Supervisor
Professor Christopher Clack
Department of Computer Science
University College London

Industrial Supervisor
Mr Ciarán McGonagle
Legal Department
International Swaps and Derivatives Association

September 11, 2023

Abstract

This thesis presents an in–depth exploration and proposition for a comprehensive platform that accommodates the production, interaction, and usage of legal Smart Contracts for Ethereum Virtual machine–compatible blockchains, with particular attention to financial derivatives. I examine the complex interplay between legal provisions, computational procedures, and financial applications within Over–The–Counter (OTC) derivatives, utilizing the frameworks from the International Swaps and Derivatives Association (ISDA). I advocate for a re–engineering of ISDA's contractual architecture through the deployment of a 'Smart Confirmation Contract System'. This system employs modular 'Type Module' and 'Logic Module' Smart Contracts to facilitate the creation of Non–Deliverable Forwards, as well as Call and Put options. The restructured design is in alignment with the ISDA Common Domain Model's data schema and Digital Asset Definitions.

Acknowledgments

The completion of this thesis would not have been possible without the constant support and guidance of several individuals and institutions. I would like to express my sincere gratitude to my advisor, Professor Christopher Clack, for his expertise, insightful feedback, and patience throughout the process of completion.

I extend my gratitude to the International Swaps and Derivatives Association (ISDA) for their expertise that have significantly enriched this study. Special acknowledgement is due to Ciarán McGonagle, whose extensive guidance and collaboration have been instrumental in improving the rigour, relevance, and depth of this research. Additional thanks go to Mark New and Ian Sloyan for their contributions.

My colleagues and peers have played a key role in the stimulating academic atmosphere. Their constructive criticism, discussions, and encouragement have been integral to my growth. In this regard, I would like to thank Simon Klaus, Louis Kampman and Thekla Ioannou for their help.

Lastly, I wish to acknowledge the enduring support of my family. Their consistent encouragement has been foundational in the completion of this work.

Contents

1 Introduction **1**

2 Background **3**

2.1 Blockchains . . . 3

2.2 Smart Contracts . . . 5

2.3 Ethereum Request for Comment Standards . . . 9

2.3.1 Development and Approval Process . . . 9

2.3.2 Notable ERC Standards . . . 9

2.3.3 Ongoing Innovations . . . 10

2.4 Oracles . . . 10

2.5 The Philosophy of Decentralised Consensus . . . 11

2.6 Legal Status of Smart Contracts . . . 12

2.7 The Internal and External Model Of Smart Legal Contracts . . . 14

2.7.1 External Model . . . 14

2.7.2 Internal Model . . . 14

2.8 Smart Derivative Contracts . . . 15

2.9 International Swaps and Derivatives Association . . . 17

2.9.1 The ISDA Master Agreement . . . 17

2.9.2 ISDA Common Domain Model . . . 19

2.10 ISDA Digital Asset Definitions . . . 20

2.10.1 Smart Contract Templates . . . 21

3 Methodology **22**

3.1 The Smart Confirmation Contract . . . 22

3.1.1 Translating the Common Domain Model into Solidity . . . 25

3.1.2 Modularisation of the CDM . . . 26

3.1.3 Conditioning of the Confirmation Smart Contract with the Common Domain Model . . . 28

3.1.4 The Logic Module: A Singleton Standardised Agnostic Logic Smart Contract . . . 28

3.1.5 Four Section Standardisation of Logic Module Smart Contracts . . . 31

3.1.6 Representing A Derivative Lifecycle As Stacks of Logic Modules . . 33
3.1.7 Building and Bridging CDM Functions to Digital Asset Definition Logic Modules 35
3.2 First and Second Order Periphery 37
3.3 An Ecosystem of Smart Confirmation Contract Derivatives 37

4 Results, Caveats, and Discussion 41
4.1 Problematic CDM translation 41
4.1.1 Enumeration Too Large 41
4.1.2 Recursive Type Structure 42
4.1.3 Addition of Non–Rosetta Types 43
4.1.4 Storage Inefficiency in CDM and its Impact on Smart Derivatives Contracts 43
4.2 Stabilizing Cash Settlement Amounts in EVM Smart Contracts 44
4.3 Architecture of a Comprehensive Decentralised Web Application for Smart Confirmation Contracts 45
4.3.1 User Interface: Contract Template Selection and Customization . . . 45
4.3.2 User Interface: ISDA–Compliant Input Form 46
4.3.3 Smart Contract Compilation and Transaction Initiation 46
4.3.4 Contract Monitoring 46
4.3.5 Contract Interaction 46
4.3.6 Event History and User Interactivity 47
4.3.7 AI–Powered Query Handling 47
4.4 Areas for Enhancement 47
4.4.1 Optimization of Time–series Retrieval 47
4.4.2 Event–based Data Retrieval Constraints 47
4.4.3 Considerations for Natural Language Contract Storage 52
4.4.4 General Improvements and Trustworthiness 52

5 Conclusion 53
5.1 Summary of Key Findings 53
5.2 Theoretical and Practical Implications 53
5.2.1 Theoretical Implications 53
5.2.2 Practical Implications 54
5.3 Limitations and Caveats 54
5.4 Regulatory and Policy Implications 54
5.5 Final Remarks 55

Appendix A code **56**

A.1 Code listing 56

A.2 Project Summary 56

Bibliography **56**

Chapter 1

Introduction

Often enduring over extended periods of time and involving substantial sums, Over–The–Counter (OTC) Smart Derivative Contracts necessitate a multidisciplinary approach that brings together Law, Computer Science, and Finance. This intersection of disciplines is essential for ensuring robust protective measures for all parties involved. Standardized legal documentation and a shared understanding of transactional procedures facilitate this protection. Crucially, achieving consensus at every level — be that technical within the Smart Contract Code — or qualitative in the natural language contract, is indispensable for each agreement.

The International Swaps and Derivatives Association's (ISDA) Master Agreement (MA) [1] plays an instrumental role in standardising natural language derivative contracts. Written in 1985, and updated in 1992 and 2002, It carefully delineates the terms and conditions applicable to all transactions between parties, thereby offering protection to both sides from many aspects of counter–party risk. The MA extends to consider scenarios involving early termination and the ensuing calculations of amounts due in such events. Given the degree of its both flexible and comprehensive nature, and its precedent in managing complex provisions like that of netting obligations into a single obligation — a feature extensively used in the aftermath of the 2008 financial crisis [2]. It is widely regarded as the legal keystone underpinning contemporary OTC derivatives trading.

However, a comprehensive integration and consideration of the recent advancements in blockchain technology remain unaccomplished within financial law [3]. Indeed, blockchain–managed transactions have the potential to reduce transaction fees, automate the settlement of trades, provide direct custodianship and bring transparency to payment flows — improving financial transactions from their initiation to the long–term audits at the end of their life cycle. With consideration of such innovations, the recent implementation of the standard–based representation of events, actions and variable names present in ISDA Common Domain Model [4] provides a strong framework for how market participants should consistently agree upon trade details. Being technology–agnostic, it seeks to make the initial step into synchronizing the technicalities required in financial derivative contracts. It is thus a sensical precursor to a Smart contract–based platform emerging in financial systems.

As I shall explore, accomplishing this integration is a task neither trivial nor straightforward. Aside from how each contract is designed, one must consider the relative benefits to each blockchain, its architecture, the degree to which automation is possible on each (as I find in Chapter 4, the Ethereum Virtual Machine is limited), the current ecosystem present within it, and the ease of writing and the ease of interfacing and reading the code semantic structure. The process of developing a Smart Contract is thereby case–by–case dependent on the machine it runs on, with its context splitting itself from a 'one–size–fits–all' structure. The 'four corners' of the Smart Contract have a great deal of variability that, if desired, could be standardisation at either the Smart Contract Code level or that of a higher controlled natural language level (which would be interpreted and fed through to different Smart contract distributions).

In collaboration with ISDA, the following research consists of proposing a new architecture and a comprehensive platform for the production, interaction and use of Smart Legal Contracts on the Ethereum blockchain. I develop a Non–Deliverable Forward, Call and Put contracts, in alignment with the ISDA Common Domain Model's data structures, and following the logic and events put forth in the ISDA Digital Assets Definitions. I continue to produce a graphical interface of the entire contract process, accessible by each of the contract's parties. The interface displays when events are emitted, alongside live updates of price derived from the relevant price oracle Chainlink.com [5]. No traditional database is utilized, and the web application is only dependent upon the client having access to an RPC provider [6]. By using a contract 'factory' one can access any contract produced in the platform's history, in perpetuity. The user interface aligns with the visual language of traditional financial contracts, to help bridge financial practitioners into using blockchain technology. I propose that Smart Contract templates are constructed by a hub–and–spoke structure, with a 'Confirmation' Smart Contract central to many pre–deployed 'Logic Module' Smart Contracts. Each entire agreement can thus be represented by 'stacks' of Logic Module contracts — wrapped in their own legally enforceable natural language clauses and provisions — with references to overarching natural language contracts.

I further explore the limitations of producing Smart Legal Contracts on the Ethereum blockchain — mainly, the inability for contracts to self–execute, and the need for external agents to 'fix' payoffs upon valuation dates. The lack of financial feasibility of storing full natural language documents, large Smart Contracts, or a large number of nested structures are present issues I find non–ideal solutions for. I explore and propose off–chain solutions, their caveats and improvements. This paper thus aims to produce a full model for how a Web3 Smart Legal Contract ecosystem could exist within the present state of Ethereum.

Chapter 2

Background

The following section explores the context for my research, including a section describing the mechanisms of blockchains, a section describing the current academic zeitgeist surrounding Legal Smart Contracts, and a section exploring the current ISDA agreement framework.

2.1 Blockchains

Blockchains were first invented to produce a digital currency for the Internet. Indeed, the 1980s and 1990s yielded the initial digital currency attempts, with David Chaum's 'DigiCash' [7], being a seminal implementation that used a cryptographic algorithm known as Chaumian Blinding. Contrary to the now–used hashing, Chaumian Blinding involved hiding the serial number of a digital currency using a cryptographic algorithm called 'blinding' — which involved mixing a serial number with a fixed amount of random data. The user would send such a 'blinded' coin to the bank, which would, without being able to see the serial number, digitally sign it. After signing, the user removed the random data, 'unblinding' and leaving them with a digital coin that had a unique serial number with the bank's signature.

The process ensured anonymity as the bank could not link the signed coin to a particular user. When the coin was later spent and deposited back into the bank by a merchant, the bank could verify its signature and check the serial number to ensure that the coin had not been spent more than once. Such 'double spending' prevention is the key obstacle to creating a robust digital currency, ensuring all parties understand and agree upon the transaction order of the system. Digicash, although comprehensive, still required a trusted intermediary (a bank) to validate transactions. It therefore was not 'trustless', or natively peer–to–peer.

Blockchains were later designed in 2005 by cryptographer Hal Finney in his 'Reusable Proofs of Work' system [8], which combined Wei Dai's b–money [9] with Adam Back's

Hashcash [10]. Hashcash being an anti–spam mechanism that requires a small 'hash' problem to be solved before sending an email, a feat non–intensive to regular senders, but hugely computationally expensive to spammers – and b–money being a distributed e–cash system that solved hash–puzzles to achieve decentralized consensus. Indeed, Hal used computational puzzles to give 'miners' a token, the 'proof' that the puzzle was solved. Once these tokens were 'mined' they could be sent to other parties as a currency. Hal however also did not achieve trustlessness, with his system still requiring a centralized database. Indeed, the pre–Bitcoin systems lacked the ability to fix the 'Byzantine Problem', to achieve agreement among distributed parties on the state of the data, where malicious actors could exist [11].

In his seminal paper [12], Satoshi Nakamoto introduced a decentralized consensus algorithm that employs a blockchain data structure, updated in 10–minute intervals known as 'blocks', to solve such trustlessness issue. Nodes in the network participate via a 'proof–of–work' (PoW) mechanism, which is the solution to a cryptographic puzzle involving the SHA–256 hash function[1]. Specifically, each block in the blockchain contains a header, which is a compact summary of the block's content and metadata. The header includes the following key elements:

- **Previous Block Hash**: A SHA–256 hash of the previous block's header, ensuring the blocks are cryptographically linked in a chain.
- **Merkle Root**: A hash derived from all the transactions in the block, serving as a fingerprint of the entire transaction set.
- **Timestamp**: A Unix timestamp indicating when the block was created.
- **Difficulty Target**: A value that the SHA–256 hash of the new block's header must be less than or equal to, in order to be added to the blockchain.
- **Nonce**: A 32–bit number that miners can adjust to try different hash values.

A new block is appended to the blockchain only if its header, when hashed twice using SHA–256, produces a value that is less than or equal to the specified difficulty target, represented by a specific number of leading zeros at the start of the hash.

This PoW process is computationally intensive, requiring specialized hardware and consuming a significant amount of electrical energy. The cryptographic chain between blocks — where each block header contains the hash of its predecessor — ensures the immutability of the entire blockchain. Any alteration to a block's data would change its hash, thereby invalidating all previous blocks due to the recursive nature.

[1]SHA–256 (Secure Hash Algorithm 256–bit) is a cryptographic hash function designed by the National Security Agency (NSA). It produces a fixed–size (256–bit) hash value from variable–length input data. It is collision–resistant, meaning it is computationally infeasible to find two different inputs that produce the same output hash [13].

The 10–minute block time in Bitcoin's protocol serves multiple strategic and technical purposes. It allows for adequate time for the new block to propagate through the network, thereby reducing the likelihood of chain forks and the creation of orphan chains. It further provides enough of a temporal window for nodes to detect and counteract anomalous behaviour, such as double–spending attempts or Sybil attacks[2]. Thirdly, the block time increases the computational burden for an attacker aiming to control a majority of the network's hash rate, rendering 51% attacks economically infeasible due to the high costs of maintaining the required computational power over an extended period of time. The 10–minute target is maintained through a dynamic difficulty adjustment algorithm, which recalibrates approximately every 2016 block to account for fluctuations in the network's total hash rate (the rate of hashes ran in a period of time).

By integrating these components — cryptographic hashing, PoW, and a carefully calibrated block time — Nakamoto's Bitcoin not only solves the Byzantine Generals Problem [11] in a decentralized environment, but also protects the network against a variety of potential attacks.

Similar to Ronald M. Lee's conceptualization of electronic legal contracts as state–transition systems [15], the Bitcoin ledger functions under a similar paradigm. In this framework, the 'state' encompasses the complete set of all minted coins and confirmed transactions. Blockchains offer several remarkable properties. They achieve practical immutability, where tampering is theoretically conceivable but becomes pragmatically unattainable as long as the network undergoes active verification by decentralized nodes. This decentralization distributes authority across the network, thereby eliminating single points of failure. Additionally, blockchains are transparent, as they maintain a publicly accessible ledger of all transactions. Furthermore, they employ advanced cryptographic techniques to ensure the robustness and security of the system. Such benefits are core to the recent rise in their appeal.

2.2 Smart Contracts

Upon observing the function and structure of Bitcoin and Blockchains, Vitalik Buterin discerned that the potential for blockchain technologies could surpass simply supporting mediums of exchange. He envisioned a broader future for blockchains, wherein they could function as fully decentralized computational platforms, capable of complex state transitions that aligned with Turing–complete[3] scripting languages. This vision was later developed as Ethereum [16], a blockchain where the ledger did not only register currency

[2]Sybil attacks exploit the vulnerability in decentralized systems where a single adversary can control multiple nodes, often by creating fake identities and to subverting the system's functionality[14]

[3]A Turing Complete language possesses the capability of simulating any single–taped Turing machine. This includes the support of basic control structures such as loops and conditionals. Turing–completeness is critical for general–purpose programming languages, as it ensures the most computational expressiveness possible.

transfers but also a fully decentralized computer system. These interactions are designed as 'Smart Contracts', scripts that lay on top of a blockchain, resistant to change, and observable by every participant on the network.

Despite Vitalik terming DLT–based scripts 'Smart Contracts', the first definition precedes Ethereum, and was presented pre–blockchain–era in 1996 by Nick Szabo in the context of centralized systems. Szabo proposed Smart contracts are a 'computerized transaction protocol that executes the terms of a contract', automated code that satisfies contractual conditions to reduce exceptions both malicious and accidental, and minimize the need for trusted intermediaries [17]. He proposed their main benefit be the shared understanding and execution of transaction semantics, communicated in clear logic — but he also predicted the rise of synthetic assets (digital assets) that combined traditional securities and derivatives to allow for their efficient trading. Szabo's centralized Smart contract model may align further with the current needs of financial institutions — as adjustments upon mistakes, and trusted databases may be desired, and are a less radical step.

Within this paper I refer to Smart contracts within the immutable–natured blockchain context, be that chain controlled with permissioned infrastructure with many trusted nodes, or a permissionless decentralized protocol. Smart Contracts thus inherit the following benefits:

1. Reduced Transaction Risk

   Every transaction undergoes a multi–node verification process. Upon verification, the transaction is irrevocably recorded on the blockchain. This architecture eliminates the possibility of partial or incomplete transactions, as the execution is strictly governed by the pre–defined code that is written. As a result, this significantly mitigates the risk of malicious activities such as financial fraud, as unless the contract is pre–written for fraud, which is unfortunately common [18], it cannot be committed as it would deviate from the written functionality[4]. In a similar manner, the risk of accidental errors or mis–specifications is virtually eliminated. This is because the contract's state is maintained exogenously on the blockchain, and interactions with the Smart contract typically involve straightforward instructions or function calls to execute the next automated step. For instance, in the Ethereum network, an ERC–20 token transaction that uses up more than the entity's balance would be rejected — as it would be inconsistent with the previously verified state stored on the blockchain, failing the multi–node verification process.

2. Lower Administration, Service Costs and Centralised Risk

[4]It is important to note that Smart Contracts do not entirely eliminate financial fraud within transactions, as criminal deception is still prone to happen externally to the execution of the Smart Contract — as is apparent within the rise of 'Pump and Dump' schemes [19].

Centralized systems involve an entity or institution overseeing, validating, and managing transactions. The centralized authority bears the majority of the administration costs, including infrastructure maintenance, manpower, and security. Such centralized structures can also result in bottlenecks in efficiency and are vulnerable to single points of failure. For example, in 2019 Wells Fargo experienced a significant outage that affected its online, mobile and ATM systems [20]. Issues arose in accessing accounts, receiving or sending payments and receiving accurate balances. The pause in service was attributed to a data centre fire. This incident is critical considering the dependence contemporary society has on digital banking.

   As an alternative, blockchains operate on a fully decentralized model. The administrative functions are distributed across the network in the form of a consensus mechanism. If a singular node that held the entire blockchain were to fail as a result of a fire, many other full–blockchain–storing nodes would remain in full operation and service the needs of clients.

3. Improved Process Efficiency

   Settlements are automatically completed in a trusted peer–to–peer manner as soon as the predetermined conditions are met. This automation significantly reduces turnaround time, as no intermediary is required to be waited upon. For example, Smart Contracts on Ethereum can be programmed to allow for the release of funds automatically when certain criteria are satisfied — and can therefore act as escrow services or collateral management tools.

Understanding these benefits and actualizing the potential of this computation–based decentralized blockchain, Buterin developed the programming language 'Solidity' [21] — comprehensive and accessible enough to pave the way for a diverse array of decentralized applications (dApps) and digital assets [22]. The language was primarily designed to be statically typed (requiring the data type to be specified upon the defining of variables and are known at compilation time, reducing runtime errors that would result from mismatches), and supporting inheritance, libraries, and user–defined types. It was written with inspiration from JavaScript, Python, and C++, allowing it to be accessible to current development practices. The platform I build in this thesis utilizes Solidity.

The language is not without its own vulnerabilities. Indeed, in 2016 a bug within a payment withdrawal in a Smart Contract called the 'DAO incident' enabled a user to withdraw 50 million US Dollars to their private account [23]. They did so by exploiting the fact that the contract sent the funds before updating the user's balance to subtract it. Simply setting up another Smart Contract to recursively half–complete the withdrawal function enabled all the Smart Contract funds to be drained. Although not technically against the Smart Contract's code, this event led to the Ethereum blockchain 'forking' into two (the original one being Ethereum Classic), with the newly followed chain reverting to have the user's balances not stolen — violating the immutability of the chain and leading to heavy audits of future Smart Contract protocols.

To enable both fast and secure block propagation, Ethereum imposes a stringent cap on block size, articulated not in bytes but in 'gas' — a measure of computational work. Gas serves dual roles as both a metering unit for computation and a variable cost for storage. Each operation, be it computational or storage–related, consumes a predetermined amount of gas, subject to its own pricing mechanism [21]. The total gas limit for transactions within a single block remains dynamically around 15 million. An illustration of these constraints is the storage efficiency when a block is hypothetically filled exclusively with SSTORE opcodes (storage). Under this scenario, only 0.024MB could be committed to the block[5]. This constrained environment naturally leads to escalated transaction fees whenever demand for blocks outstrips supply. The result is a scalability issue; the chain becomes prohibitively expensive for mass adoption due to both economic and technical bottlenecks — the more users use it, the more expensive it becomes to use. While Layer 2 solutions [24] like rollups have been implemented to help mitigate this limitation by offloading computational tasks off–chain, the scalability conundrum remains an unresolved issue at the protocol's core layer.

In comparison to legal contracts, Ethereum–based Smart Contracts, therefore, have hard limits on their size — and therefore require rigid modularisation to allow for fully comprehensive agreements. Some of the most influential smart–contract protocols like that of the decentralised exchange Uniswap V3 have 8 contracts, with 4 existing in the 'core' and 4 within the 'periphery' [25]. Each is deployed within different transactions, and each referencing to the address of the other — allowing for full functionality of the decentralised application.

Pressed into this small size, a Smart Contract undergoes the following consecutive stages [26]:

1. **Creation Stage:** Within this stage, the involved parties negotiate the obligations, rights and prohibitions of the contract. After discussion, lawyers and counsellors draft the initial contractual agreement, and software developers continue to convert this agreement into logic–based rules. The Smart Contract conversion is then subject to the traditional design, implementation and testing procedures for software engineering, requiring multiple rounds of iterations with all involved stakeholders.

2. **Deployment Stage:** The fully audited and validated contract is deployed upon the relevant blockchain. Any changes to the framework require the creation of a new Smart Contract — with the associated digital assets 'frozen' or transferred from the old.

[5]This approximation derives from an average SSTORE operation cost of 20,000 gas. Given a 15 million gas cap, this facilitates 750 SSTORE instances, each consuming 32 bytes — totalling 24,000 bytes or approximately 0.024MB.

3. **Execution Stage:** After deployment, contract clauses are actively surveilled in relation to real–world or on–chain events. When predetermined conditions are satisfied, the Smart Contract autonomously executes or authorizes a party to manually invoke specific functions.

4. **Completion Stage:** Upon the execution of the contract, and all stages have been passed, the contract can be considered 'complete', with the associated digital assets transferred from one party to the other. The entire contract is stored at this final stage in perpetuity.

2.3 Ethereum Request for Comment Standards

Since the beginning of Ethereum, there has been a pressing need to standardize the development and interoperability of Smart Contracts. To systematically address this exigency, the Ethereum community has formulated a protocol specification framework, known as Ethereum Request for Comment Standards (ERC) [27]. These standards play a key role in shaping the development of Smart Contracts on the Ethereum network.

2.3.1 Development and Approval Process

The ERC development process starts with the creation of technical specifications for the proposed features, complemented by a presentation of the underlying considerations and rationale. These documents are then introduced as Ethereum Improvement Proposals (EIPs), and are subject to rigorous analysis and iterative refinement by the Ethereum community of developers and researchers.

2.3.2 Notable ERC Standards

Among the many ERC standards, certain proposals have gained substantial traction and have become cornerstones of the Ethereum landscape:

- **ERC–20**: This standard defines the rules for the creation and management of fungible tokens on the Ethereum network. By defining a common interface for token interactions, ERC–20 has become the standard for the majority of Ethereum tokens, including well–known stablecoins like USDT, USDC, and digital assets such as Polygon (MATIC). Its robust design and absence of significant bugs have led to its extensive adoption within decentralized token exchanges and protocols. It can be considered the most seminal Smart Contract template to date [27].

- **ERC–721**: Targeting a different mode of digital asset management, ERC–721 is a specification for Non–Fungible Tokens (NFTs). Unlike ERC–20 tokens, each NFT is distinct, representing ownership of an individual asset. ERC–721 provides functions for the creation, transfer, and querying of these unique assets. For example, with the function ‘ownerOf’, users can easily identify the ownership of a specific digital asset in perpetuity. The rise of digital art, collectables, and virtual real estate has provided market–wide confidence in this standard [28].

- **ERC–1155**: This standard is recognized for its multi–token properties, as ERC–1155 enables a single Smart Contract to govern both fungible and non–fungible tokens. Such an approach optimizes efficiency by condensing token types into one contract, reducing redundancy and complexity. This Smart Contract template is used for gaming assets where fungible tokens represent in–game currency and non–fungible tokens represent single items, or broader financial contexts [29].

2.3.3 Ongoing Innovations

The continuous advancements in blockchain technology necessitate innovation in the ERC standardization process. The community–driven approach not only facilitates the establishment of robust standards but also fosters an environment of adaptation. Through the collective work of its contributors, Ethereum, through its improvement propositions, continues to refine existing standards, maintaining its position at the forefront of blockchain technology.

2.4 Oracles

In the realm of Smart Contract applications, oracles stand out with a distinct function [30]. Like their namesake classical prophetic communication with deities, in the blockchain ecosystem, oracles serve a pragmatic function to bridge between a Smart Contract and the outside digital world. They act as intermediaries that fetch and relay data to on–chain Smart Contracts.

Oracles can be categorized into on–chain and off–chain types. On–chain oracles provide data that is native to the blockchain, such as block numbers, transaction metadata, or the state variables of other Smart Contracts. For instance, UniswapV3 functions as an on-chain oracle by offering token price information derived from the reserves stored in its liquidity pools [25]. This data is both immediate and immutable, ensuring a high level of trust and permanent availability.

Conversely, off–chain oracles (what Oracles commonly refer to) offer a broader range of data types. These oracles source information from external entities, which can vary from API services delivering weather forecasts or stock market prices to hardware devices in the Internet of Things (IoT) ecosystem, and even to using complex off–chain computations and pulling that data on–chain. Some implementations employ human–based oracles to supply qualitative data, such as election results or significant societal events.

Off–chain oracles can further be subdivided into centralized and decentralized architectures. Centralized oracles depend on a single data source, introducing a potential point of failure and thus compromising the system's trustworthiness. On the other hand, decentralized oracles aggregate data from multiple sources — diluting the risk associated with a single point of failure and enhancing the integrity of the data, alongside the reliability of the data, as many fallbacks may exist [30].

ChainLink [5] is a leading Oracle network. Its architecture is relatively simple — when a specific type of external data is needed, a 'Requesting Contract' is initiated, containing the type of information that is needed. From this, the 'Service Level Agreement (SLA) Contract' is initiated, logging the request as an 'event' and generating three sub–contracts: Reputation, Order–Matching, and Aggregating, each of which manages the different stages of data retrieval and validation. The Reputation contract manages the selection of the best oracles for each data request, broadcasting the request to off–chain 'ChainLink' nodes who bid for the opportunity to provide the data at remuneration of an amount of 'LINK' tokens. The Order–Matching contract reviews each of the bids and chooses the best nodes on reputation, past performance and cost. Finally, the Aggregating contract gathers all the data, verifies it, and produces a consolidated result that is returned to the original Smart Contract. Chainlink Nodes stake[6] LINK tokens as a security deposit. If a node acts maliciously, with inaccurate data provided sporadically at uneven times, a portion of their staked LINK tokens is forfeited as punishment — ChainLink is, therefore, both the 'Carrot and the Stick' ensuring off–chain data is accurately inputed into the blockchain.

2.5 The Philosophy of Decentralised Consensus

With off–chain Oracles bridging between digital 'worlds', one may wonder which world provides legitimate data. From the sociological perspective of Strong Social Constructionism [31], which states that the nature of reality is a result of our shared conceptual beliefs of its constituents, I argue that blockchains can be seen as the only real digital reality, with Smart Contracts the 'Law' that governs it.

Indeed, if reality is created by beliefs and systems of knowledge, which hinge upon a mutual trust in shared theoretical constructs — and blockchains are a mechanism for creating a fully decentralized and agreed–upon knowledge system — then one can therefore infer that blockchains are another fully controlled reality humankind has made within its own, wherein anyone can participate in its construction from the grassroots level. New blocks are only added when the majority of nodes (synonymously, society) agree upon the consensus of the data. Indeed, unlike the complex interplay of variables within the physical world, the blockchain universe is subject to well–defined algorithmic parameters.

[6]Staking in the contect of blockchains involves locking up a certain amount of cryptocurrency in a smart contract to participate in a consensus–giving activity, earning rewards and contributing to the functionality of the protocol.

Such an idea can be analogized within the context of Ludwig Wittgenstein's language games [32]. For as in the same manner as the 'standard meter stick in Paris' was the reference point for the system of length measurement systems — systems where objects are measured against another object — but the stick itself is a 'blind–spot' of the system it creates (the stick cannot measure itself), a blockchain network establishes similar measurement consensus upon who owns what, which data is authentic, and the state of its decentralized computation.

Like the standard meter stick, a blockchain has no awareness of data outside its own ecosystem — it is the 'blind spot' against the external digital world. Like a meter–ruler is to the Parisian standard meter stick, off–chain 'Oracles' should be seen as instruments of measuring the external digital world to enable the validation of the on–chain data. The language game of a blockchain, with actions and words like transactions, blocks and consensus mechanisms thus interacts with the external language game of real–world asset prices, APIs, events, photos, videos and ownership to 'enrich the form of life' of consensus.

2.6 Legal Status of Smart Contracts

Smart Contracts epitomize what Professor of Law Lawrence Lessig refers to as the architectural 'Law' in the digital domain [33]. Designed to autonomously execute predefined rules, Smart Contracts function as a self–regulating system. Drawing upon Lessig's seminal work, *Code and Other Laws of Cyberspace* [34], the phrase 'Code is Law' proposes the idea that the code, authored by developers, serves as the governing framework for interactions in cyberspace. It embodies value judgments and imposes rules that regulate these interactions. However, he poises that it is crucial to recognize that while the algorithmic law instantiated by Smart Contracts is unambiguous in its execution, it is not devoid of human influence. Human developers set the initial conditions and rules, thereby introducing a layer of subjectivity and potential for error. Consequently, despite their deterministic nature, Smart Contracts operate within a broader socio–technical ecosystem, complicating the simplistic notion that 'Code is Law'.

Indeed, if one thinks of a contract as a payment or promise, in exchange for a payment or promise (with consideration), then Smart Contracts can be built on top as cryptographic 'boxes' [16] that contain values and can only be unlocked (to reveal their payment or promise) once certain conditions are met. From a legal perspective, they are digital 'vending machines', which upon correctly entering currency or pressing the correct button, perform the task of delivering you a product. Indeed, the code of the vending machine is hard–wired to perform this singular task — and can be seen as the machine's internal 'Law' that forbids it from performing any other action.

Indeed, one expects the vending machine to, once paid, provide the product. Upon failure, one rings the manufacturer, who is obligated to provide a refund or completion of the obligation — the provision of the product. In an identical manner, Smart Contracts do not live in a legal vacuum, and despite existing on trustless blockchains, will, in reality, require enforcement by third parties. If an individual attempts to 'game' the Smart Contract by abusing the functionality of the code — as what occurred in the 'DAO Incident' [23] — it is still considered illegal in practice and subject to law enforcement. Again, 'Code is Law' was demonstrated to only be true to the extent that both parties have a mutual understanding of the values behind how the Smart Contract code is expected to perform.

Perhaps stated in a strictly defined manner, Josh Stark [35] states that 'Smart Contracts' are either defined as either the execution code itself — the 'Smart Contract Code', or as binding legal documents, 'Smart Legal Contracts' where the viability of the contract is dependent on the intersection of the technology (code), but also the existing legal framework. In the latter, the legal framework can encapsulate some Smart Contract Code to execute certain obligations or conditions, whereas in the former, the 'mart Contract Code exists outside natural language, and as computer–based operations.

Smart contracts are thus either execution code or a binding legal agreement. Pinpointing a clear definition is of utmost importance, as depending on whether legal conditions are met, a Smart Contract may or may not serve in a practical context as a contract. Liken to Clack et al's paper [35], in my thesis 'Smart Contract' refers to both, as an

> *"automatable and enforceable agreement. Automatable by computer, although some parts may require human input or control. Enforceable either by legal enforcement of rights and obligations or via tamper–proof execution of computer code."* [35]

In this context, automatable refers to partly automated, with some input needed due to Ethereum requiring being externally called for 'gas' — with enforceable meaning that the code should run without error and that from the legal perspective, rights and obligations accrue with the ability for traditional and non–traditional enforcement methods such as arbitration, recourse to the courts of law, or on–chain 'slashing' of remuneration if the underlying code is tampered with [36].

Within my framework, I expand to let Smart Contracts be a constituent part of an entire agreement. That is, a Smart Contract can encapsulate the transactional–level obligations of the contract, and allow for some fallback–based conditional clauses to be automated — but still exist within a written natural language contract. Smart Contracts are thus a 'piece' of a contractual puzzle put together by parties — much like how the numerical parameters or mathematical concepts currently make up derivative agreements today.

2.7 The Internal and External Model Of Smart Legal Contracts

Such a hybrid proposition is not novel. Indeed, Linklaters proposed two distinctive models for the near future of Smart Contracts, based upon how integrated the Smart Contract code is with legal obligations [37].

2.7.1 External Model

The first is an external model, wherein the agreement would remain in a large majority in its natural language form. However, external to the legal contract, some logical elements are placed in the Smart Contract Code for automation upon conditions being satisfied.

Within this model, the Smart Contract Code would be exogenous to the legality of the natural language contract — with the natural language contract taking precedence in any dispute. The code would thus be a 'reflection' of the obligations presented in the human language agreement. If the automation is executed with error, the natural language would 'rescue' the obligation of the parties.

For example, a lease agreement on a residential house may be written in standard legal language, outlining the rental amount, duration of the lease, maintenance responsibilities etc — and be reflected in a Smart Contract agreement that is created to automate lease payments, the depositing collateral, and a maintenance fund. If there were to be any issues with an automated transfer, like that of an incorrect amount or date, both parties would still adhere to the natural language terms.

The external model enables the efficiency, instant settlement and reduction in human error issues that Smart Contracts bring, but also maintains flexibility, human intervention and legal recourse upon issues in the automation process.

2.7.2 Internal Model

Within the internal model, the legal contract would still largely remain in its current form — however, certain automatable conditional logic would be rewritten in a formal representation for conditional logic to be executed automatically.

The written contract would therefore look like an amalgamation of pseudo–code, controlled legal natural language, logical statements, confirmation parameters, and natural language clauses. Each section would reference each other, and all would have legal force. The pseudo–code (the 'formal' logic), may however be presented differently than the code within the final Smart Contract Code, as each programming language may differ in its sequence of commands — and therefore would require translation in a way that holds true to the transactional logic.

Notably, within the Internal Model, there is a closer resemblance to 'Code is Law', meaning practitioners must have a clear and shared understanding of the code that implements the logic. As I present in Chapter 3, such an area would be suitable for legal standards to initially be formed as a combination of natural language contracts and Smart Contracts — as heavy auditing of certain functions could be extrapolated and referenced in different Internal Model agreements.

An example of the Internal Model is present when one considers a manufacturer and supplier entering into an agreement for the supply of raw materials — where the contract would constitute both natural language and code–based logical statements. The code would allow for logic specifying the automated order placements for materials when the manufacturer inventory falls below a pre–specified level. To ensure pricing is correct, both parties could agree on an Oracle's pricing of predefined and external market variables — like that of global commodity prices, exchanged to local currency values.

Although appearing attractive, such a model is prone to cyber risks, in which there is a propensity for malicious actors, glitches, unauthorized alterations, and breaches of contract which can emerge from the code itself. As a result, each party requires an attempt at predicting the likelihood and severity of the risk, assigning a value to it, and considering whether it is worthwhile taking it on in exchange for the aforementioned efficiency gains.

Through rigorous auditing and standardization, the algorithmic components of an Internal agreement can attain a high level of trustworthiness in executing their designated obligations. However, the subsequent complexity inherent in the Internal Model serves as a double–edged sword, producing contracts that are highly dynamic yet contain elevated risk for all parties involved. Additionally, the incorporation of Smart Contract Code inherently restricts the contract's operational flexibility, as actions are strictly defined to only occur in one way — a limitation not present in conventional legal contracts, which offer a broader scope for variability that can be interpreted in many different manners. These factors likely contribute to the limited market adoption of the Internal Model.

2.8 Smart Derivative Contracts

A clear implementation of Smart Contracts within the financial sector could be the automation of derivative contracts. Indeed, DLT can provide the following substantial benefits [38]:

1. ***Real–time Settlement And Reductions in Transaction Fees:*** Current transfer fees are expensive — with the average international fee reaching 7.6 percent of the remitted amount for money transfer operations [39]. Within the context of derivatives, sending payments is a complex process, including multiple bank verification

of the party's accounts (for KYC[7] and AML[8]), followed by transferals of payment instructions that may require intermediary banks, clearing houses, the SWIFT Network, and eventually an end–of–day reconciliation of transaction, with notifications provided at every step. Derivatives add to this complexity, often requiring collateral management, margin calls and many other considerations that intertwine with the payment process. Smart Derivative Contracts simplify the process by allowing the sender's digital identity (secured by their private key) to be sufficient verification to directly send funds to another digital identity — thereby cutting out costly intermediaries, providing instant ownership and passing on the savings to parties. Further, both collateral and currency exchange operations can be fulfilled in a peer–to–peer manner, with predictable and transparent fees.

2. ***Improved Asset Rehypothecation:*** Asset rehypothecation is a common practice wherein financial institutions use collateral posted by their borrowers to cover their own trades. Current risks exist where institutions confuse the ownership of these assets — producing uncertainties for rehypothecated asset valuations. A clear and immutable transaction history would allow for clear knowledge of chained transactions. Indeed, a tokenized solution for representing collateral could be traded and would enable regulators to monitor each chain and enforce a regulatory rehypothecation limit to halt or reduce trading near that limit. In effect, one could heavily reduce the likelihood of systemic failure in the derivatives market.

3. ***Automation of Compliance:*** To remain in operation, financial institutions need to be audited, report their tax, stress test, and submit routine filing to many appropriate financial regulatory authorities. On average, a study found that auditing amounted to a cost of 8.1 million dollars a year [40]. DLT would allow auditors to instantly access the required information they need, produce automated reports, and submit these reports to officials for review. The new process would drastically cut investigatory labour hours, lending to safer and more efficient markets.

4. ***Lowering Costs of Entry:*** Smart contracts can significantly lower the barriers to entry for both institutional and retail participants in the Over–The–Counter (OTC) derivatives market. Traditional avenues necessitate a complex and expensive legal framework, characterized by prolonged negotiations, exhaustive legal reviews, and multiple intermediaries. In contrast, Smart Contracts that leverage blockchain technology have the potential to automate these processes[9], thereby reducing transactional costs and accelerating execution times.

[7]Know Your Customer (KYC) is a legal framework employed in the financial services industry to authenticate the identity of clients. The process involves the verification of personal information, such as government–issued identification and proof of address, to assess the risk profile of an individual or entity.

[8]Anti–Money Laundering (AML) refers to a set of laws and regulations designed to prevent the illegal gaining of income through actions like money laundering and fraud.

[9]It should be emphasized that the acceleration in framework construction is contingent upon the specific

> Economically, the automation afforded by Smart Contracts has profound implications for market dynamics. By lowering transactional and legal complexities, Smart Contracts facilitate greater market participation, intensifying competition. This increased competition, in turn, exerts downward pressure on the price of derivatives, making them more accessible. Moreover, it fosters a more efficient allocation of resources in financial markets, better optimising capital distribution for the benefit of all. This culminates in a more liquid, efficient, and inclusive financial ecosystem.

Such a shift in the process towards digitization would therefore drastically cut investigatory labour hours, and lend to safer and more efficient markets.

2.9 International Swaps and Derivatives Association

A champion of such concerns as efficiency, standardization, and market safety, the International Swaps and Derivatives Association (ISDA), is providing both technology and law–based solutions in alignment with the advancements in DLT. Indeed, they have recognized the potential power of the standardization of economic terms, alongside their automation could be used within aspects of their widely accepted ISDA Master Agreement, and have created the Common Domain Model for the former, and the Digital Assets Definitions for the latter. Referenced succinctly by Clack and McGonagle [41], ISDA's perspective is that:

"Focusing exclusively on the economic terms of an individual transaction may ignore much of the external complexity that can affect a party's ability to perform its obligations (or assert its rights) in relation to that transaction."

That is, rather than the external model, ISDA believes Smart Contract agreements should be internal, with implementation heavily dependent upon the external complexity of natural language. It therefore follows that a marriage of computer scientists, lawyers, banking practitioners, and policymakers is required to steer financial law toward the standardization and production of Smart Contracts.

This section describes the primary constituents of the Master Agreements, before exploring recent Smart Contract advancements.

2.9.1 The ISDA Master Agreement

The ISDA master agreement is a standardized contract used for Over–The–Counter derivatives. The agreement itself is not one singular document — and is commonly split into the following sections:

implementation of Distributed Ledger Technology (DLT) legal framework. Delays may arise if parties engage in protracted deliberations over the interpretation of pseudo–code.

1. ***Master Agreement***: A standalone document setting forth the standard terms and conditions between the two parties. Complex interactions like payment netting and default 'close–out' netting are provided within this document. It further includes miscellaneous legal provisions like waivers amendments and calculations.

2. ***Schedule***: The amendments or exceptions to the general terms are provided: the customization of the master agreement. Adjustments to default or termination events are common, depending on the commercial context.

3. ***Product Definitions***: The key terms, phrases and concepts are defined for the agreement. It ensures both parties understand the language used within the documents — for the minimization of legal disputes.

4. ***Confirmations***: A written confirmation outlining the terms of each transaction under the master agreement. The confirmation document enables the reproduction of a singular agreement with different parameters — and it is where the majority of the variability in the framework lies.

5. ***Credit Support Annex***: for collateral–based derivatives, outlines the collateral agreements, requirements, and mechanisms for both posting and retrieving collateral.

The totality of these documents creates a cohesive and flexible framework for derivatives trading. The nature of the Master agreement document is presented below in Figure 2.1, with the relationship and transactions documents separated appropriately.

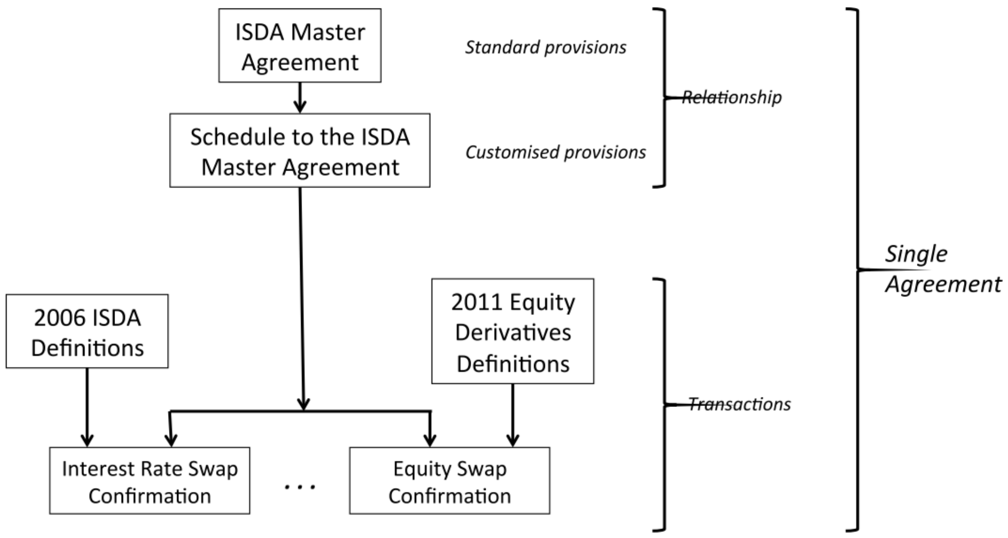


Figure 2.1: Clack and McGonagle and Clack, 2019. The current process of creating an ISDA agreement is separated into both transactional and relationship–based sections.

In the creation of a legally enforceable Smart Contract, the provision structures within the Master Agreement, as characterized by McGonagle [42], inherently possess a forward–looking orientation that hinges on variables unpredictable at the contract's inception. This contrasts with computational code, which is deterministic and operates on immediate conditions. McGonagle considers that a separation of contract clauses into automatable and non-automatable elements offers an intermediary solution. However, a challenge arises from 'severability issues', where operational clauses are intricately interwoven with other semantic components, constraining the real feasibility of automation.

To remediate the aforementioned severability issue, the International Swaps and Derivatives Association (ISDA) has conceived a Legal Agreement and Clause Taxonomy [43]. This Taxonomy deconstructs contracts, attributing semantics to varying obligations and encapsulated events. Significantly, the Taxonomy encourages an approach grounded in substance, aiming to codify the contractual outcomes as opposed to the lexical arrangements that articulate them. ISDA has also created a Clause Library [43], rooted in the Taxonomy. Available digitally via the ISDA Create platform [44], the library gives standardized drafting alternatives corresponding to the outcomes given in the Taxonomy, thereby facilitating a shift away from bespoke drafting — improving efficiency.

In summary, the Taxonomy serves as an effective framework for isolating the substance of clauses, thereby enabling their conversion into machine–readable legal agreement data — like that of the Common Domain Model.

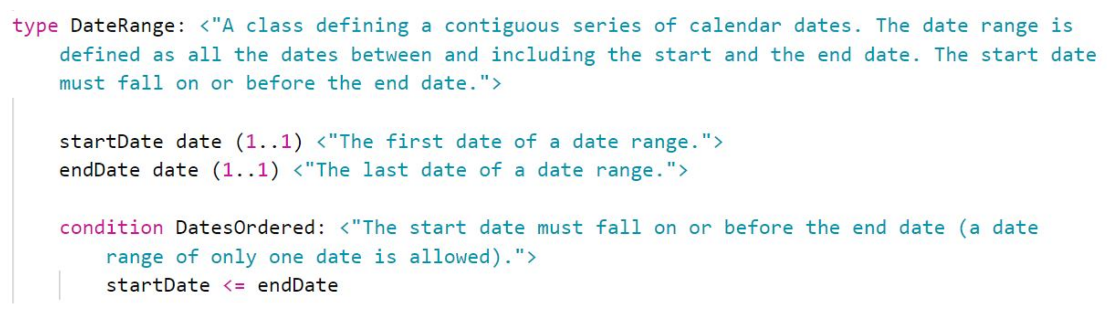
2.9.2 ISDA Common Domain Model

A more recent product released by ISDA to standardize each derivative's data and process representation among derivatives is the Common Domain Model (CDM) [45].

The model itself is technology agnostic, to enable a wide amount of implementation options for the user. Indeed, written in Rosetta [46] the CDM has distributions in Typescript, Python, Java, and many other languages at both the low and high level. Such a design philosophy thus facilitates both low–level innovation within Smart Contracts and high–level innovation transaction representation — a "plug and play" [47] infrastructure. It currently resides as a 'middle–layer' implementation, with type specifications for data and function definitions for logic — the former has been focused on to a greater extent.

The CDM can be further understood as a state transition system [47], with events and operations altering each state. Indeed, within a contract 'operations' can consist of a sequence of "Before" and "After" events, with an event being an action that changes the state (contract information) at a given time. Such an event may describe a transfer of value or a significant change in economic description. The CDM creates a reusable pattern for describing such actions and transitions with careful consideration — as the semantics of whether a "Before" or "After" event is actioned in the operation or requires having been actioned is critical information.

The CDM is open source, allowing any participant to contribute and benefit. With an example type presented in Figure 2.2 below, It is an ambitious tool, that is open to emerging technologies — and in many ways may be a vital bridge connecting traditional finance to cutting–edge innovations.



```
type DateRange: <"A class defining a contiguous series of calendar dates. The date range is
    defined as all the dates between and including the start and the end date. The start date
    must fall on or before the end date.">

    startDate date (1..1) <"The first date of a date range.">
    endDate date (1..1) <"The last date of a date range.">

    condition DatesOrdered: <"The start date must fall on or before the end date (a date
        range of only one date is allowed).">
        startDate <= endDate
```

Figure 2.2: Illustration of the ‘DateRange‘ data type in Rosetta’s implementation of the Common Domain Model (CDM). The figure elucidates the data structure and associated metadata.

Source: `https://ui.rosetta-technology.io/#/workspaces/read-only-COMMON-DOMAIN-MODEL`

2.10 ISDA Digital Asset Definitions

A new product definition specific to digital assets was released in 2023 as The Digital Asset Definitions [48]. This is ISDA’s first standardized legal terms and product definitions for digital asset derivative trades. Partly written in a controlled language structure for the facilitation of the CDM, automation, and Smart Contracts, the definitions provide for cash–settled forwards, calls, and put options.

The primary innovation within the document is the classification of three Disruption Events and Fallbacks specific to blockchain infrastructure:

1. ***Price–source Disruptions***: where the price source (which could exist as an on–chain Oracle), is either unavailable or discontinued — creating a ‘Price Source Disruption Event’ that obligates a notice provision to a ‘Material Change Determining Party’ — and thus creates a waterfall of applicable events, like that of termination, calculation agent—intervention, or switching to a fallback settlement price source.

2. ***Fork Disruptions***: Where as a result of a blockchain protocol change that produces two successor digital assets on distinctive chains, an obligation to raise a ‘Fork Disruption Event’ is made in which the ‘Fork Determining Party’ refers to a pre–determined calculation agent that amends the terms of transactions.

3. ***Change in Law Disruptions***: Where it becomes illegal for one or more of the parties to partake in the agreement, like for example the banning of blockchains

within one's legal jurisdiction. One of the two parties gives a termination notice to the other — to commence an early ceasing of the contract.

The ultimate goal of digital asset definitions is the promotion of an efficient derivatives market for digital assets. Indeed, despite there being recent rapid growth in digital assets, failures in the cryptocurrency market such as FTX's bankruptcy [49], and Celcius's collapse [50], necessitate a contractual framework like that of the one put forward.

2.10.1 Smart Contract Templates

ISDA, UCL and Barclays have collaborated to envision Smart Contract Templates as a method of facilitating code development that aligns with the ISDA documentation architecture [35]. These templates are aimed at early–stage verification, validation, and debugging, with an impetus to create a technology–agnostic template at the earliest stage of code development.

As a preliminary structure, they suggest that prior to negotiation the parties should agree upon a Master Agreement Smart Contract template, and then agree upon the Product Definition code templates, one for each derivatives product type used, and shared for all relevant transactions. Upon completion, copies of the Smart Contract templates will be made, with the code modified with desired changes to the legal provisions and functionality. The resulting modified code will be prepared as a template for propagation with the correct parameter values, visible within the logic and data presented in the Confirmation document. The final version will be run on a centralized or distributed ledger — at the choosing of the parties. The faithfulness of the final contract code to the legal agreement would require stringent agreement between lawyers and computer scientists.

Chapter 3

Methodology

The following sections detail the design and implementation of Smart Contract templates for derivative contracts — taking heed of the ISDA Master Agreement, Digital Asset Definitions, and Common Domain Model. I aim to propose a succinct methodology for implementing modular and reproducible Smart Contract templates — implementing a 'focal point' Smart Contract to wrap the entire agreement.

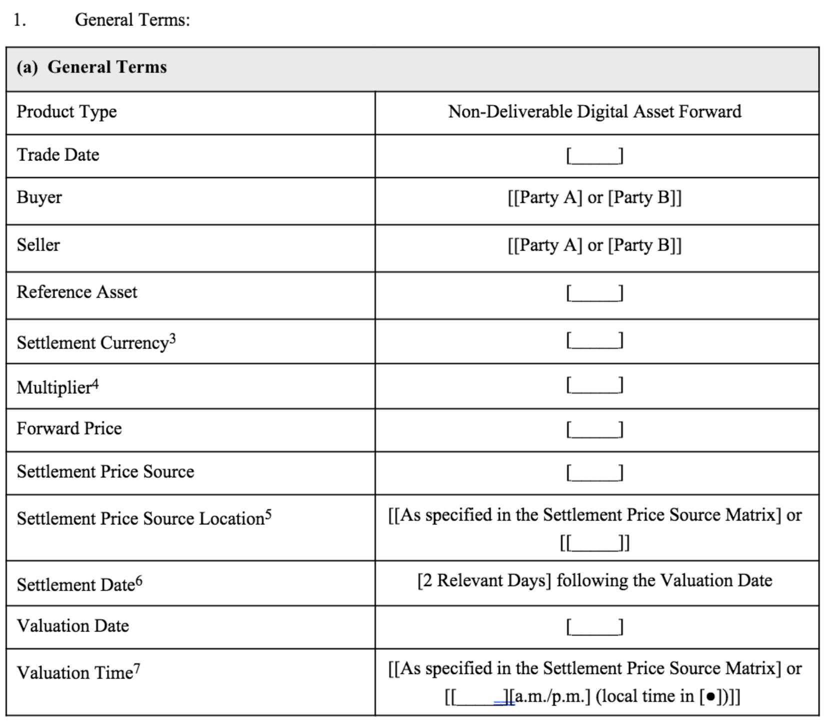
3.1 The Smart Confirmation Contract

An ISDA confirmation contract states its purpose as being "evidence (to) a complete and binding agreement between you and us to enter into the Transaction on the terms set forth below". It is thus the central point in asserting that both parties have reached a full and legally enforceable agreement on certain transactions.

Within the Master Agreement, the confirmations delineate the 'General Terms' of the contract, including who the buyer and seller are, the forward price agreed, and the relevant parameters for the completion of the contract. It further specifies electable disruption events, like that in the Digital Asset Definitions of a 'fork–event' (where the underlying chain splits in two) or a hedging event, wherein a party is unable to practically hedge against their position. These parameterized terms are broken into natural language tables, like that shown below in Figure 3.1.

Upon the full entry of parameters in a series of these tables, the confirmation document is signed, marking the conclusion of the agreement creation process. Containing the majority of the adjustable variability by simply modifying fields such as 'forward price' or price source, and referencing the Master Agreement and Schedule, the Confirmation contract can be equated to a wrapper for the entire agreement framework around it.

I propose a paradigm shift in how the Confirmation document is perceived within the Master Agreement workflow of Smart Contracts. Rather than being the 'end' of the agreement process, it should be recognized as the centre or nucleus of it. Viewed as a *hub*,


1. General Terms:

(a) General Terms	
Product Type	Non-Deliverable Digital Asset Forward
Trade Date	[____]
Buyer	[[Party A] or [Party B]]
Seller	[[Party A] or [Party B]]
Reference Asset	[____]
Settlement Currency[3]	[____]
Multiplier[4]	[____]
Forward Price	[____]
Settlement Price Source	[____]
Settlement Price Source Location[5]	[[As specified in the Settlement Price Source Matrix] or [[____]]
Settlement Date[6]	[2 Relevant Days] following the Valuation Date
Valuation Date	[____]
Valuation Time[7]	[[As specified in the Settlement Price Source Matrix] or [[____][a.m./p.m.] (local time in [●])]]

Figure 3.1: Example of the General Terms of a derivative agreement confirmation. Parameters are held as cells in the table.

all other documents, Smart Contracts, and relevant information can be appended as *spokes*, with the confirmation contract explicitly specifying the precedence of one document over another. The appended spokes can therefore be predeployed upon a blockchain once, and appended at will in the template construction process. Amendments, being permanently recorded and immutable, would easily be implemented by swapping out a spoke for another or overwriting a stored parameter. To find all the information relevant to the contract, one may refer to this hub, creating a more organized and interconnected system of information and dependence.

Taking inspiration from such a central 'agreement' structure, I start by formalizing a Smart Confirmation Contract. Like its natural language inspiration, this Smart Contract would hold the previous transactional parameters of the contract, yet would also hold a reference to the natural language contract (Master Agreement and Schedule), in the fashion of a stored hash or the file stored on–chain itself. Further added are modular 'logic' Smart Contracts, which are strictly given permission to alter the state of the agreement in the confirmation contract. A Smart Confirmation Contract (presented in Figure 3.2) should encapsulate both the written agreement and its operational completion, and must therefore have at minimum the following constituents:

1. **The Natural Language Document References (MA, Schedule etc.)**

   In the form of a hash, or the document itself (in encrypted or decrypted form).

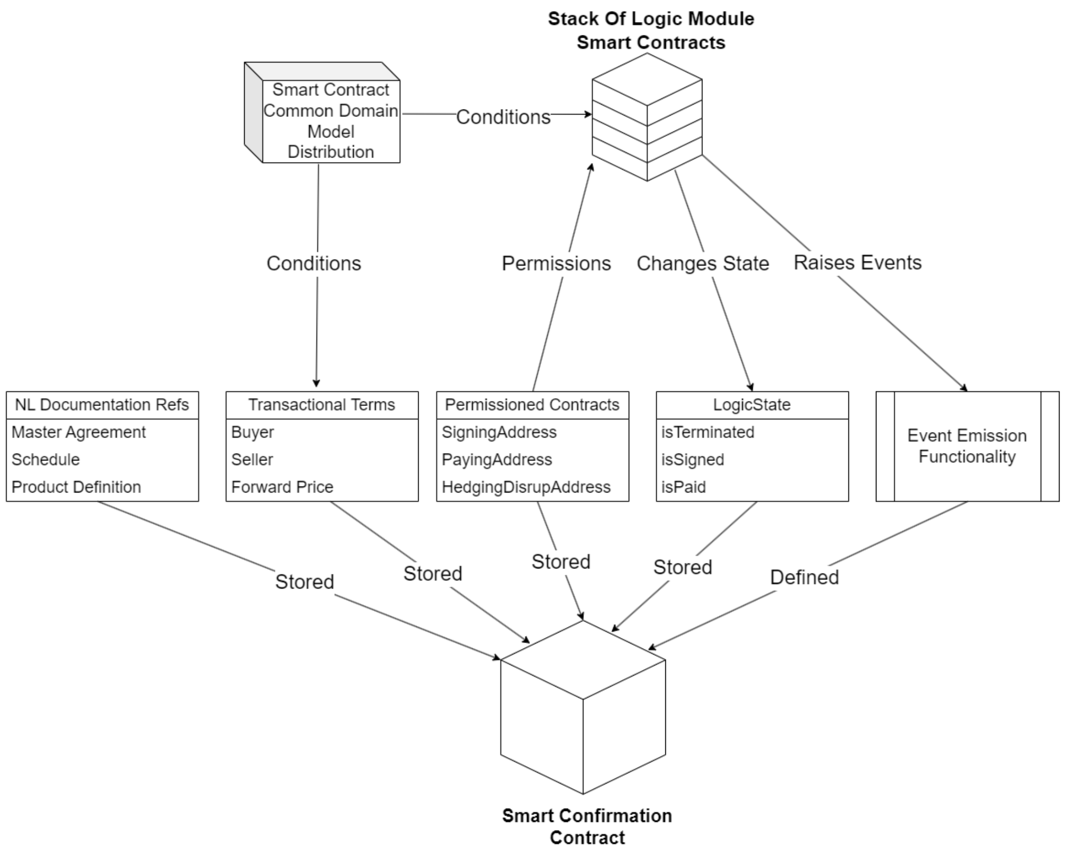


Figure 3.2: A display of the four different data structures that are used in the composition of a Smart Confirmation Contract. Further, how the Common Domain model conditions the data types in the Logic Module stack, and the transaction terms and parameters.

2. **The Transactional Terms (General Terms, Disruption Applicable Events)**

   Such terms can be conditioned with standardized representations, like that of the Common Domain Model.

3. **The Logical State of the Agreement ('Logic State')**

   Represented by a structure of boolean values that create a directed tree structure for the encapsulation of the logical flow in the agreement process. Nodes in this tree correspond to specific actions or states within the agreement, with directed edges indicating the permissible transitions between these states. Boolean values are associated with each node, denoting whether a given action has been completed or is yet to be executed. 'Logic Module Smart Contracts[1]' utilize this directed tree to verify their ability to alter the confirmation contract state, ensuring compliance with the predefined logical sequence of the agreement.

[1]Logic Module Smart Contracts, expanded upon in Section 3.1.4, are standardised singleton Smart Contracts that are responsible for a singular functionality in the derivatives life cycle, like that of the signing process.

4. **A Permissioned Contracts List**

   A list of references to operational 'Logic Module Smart Contracts'. This list, alongside the Logic State of the agreement, defines the transactional lifecycle of the contract.

5. **Event Emission Functionality**

   This entails a mechanism embedded within the confirmation Smart Contract to emit specific events that correspond to various stages or actions within the agreement. At a minimum, this functionality must include one definitive function to emit an event, but typically it would comprise a collection of specialized functions tailored to the unique events of the agreement. These events may be invoked either directly from the confirmation contract or from the associated logic contracts that the confirmation contract permissions.

For each derivative in this system, the majority of variability is now derived from both the transactional terms and the chosen Logic State structure — with subsequently chosen Logic Modules that are chosen to implement them.

Moreover, the hub–and–spoke structure facilitates a centralized point for implementing the Common Domain Model (CDM), eliminating the need to repeatedly inherit or redefine it across various interconnected Smart Contracts. Specifically, the CDM is only required to 'condition' the data structures and functions associated with the transactional terms and the functions within the Logic Modules. By doing so, the Logic Modules that rely on these values can directly reference them without having to re–establish or inherit these structures. Hereafter, I will be detailing the creation of a Smart Confirmation Contract and a platform to prove its conceptual validity, basing the creation on Ethereum Virtual Machine compatible Smart Contracts.

3.1.1 Translating the Common Domain Model into Solidity

I start by translating the Common Domain Model, going from its Typescript implementation to a full Solidity one. I do so by going through each line of the types, Enumerations and Metatypes files, changing the 'interface' definitions to solidity 'structs' and switching each field from being typed with "name: [type]" to "[type] name", in line with Solidity's syntax. I preserve comments, that share the same syntax.

I then wrap the 'types.ts' declarations into a single CDM.sol contract and import the corresponding Enums.sol and Metatypes.sol for inheritance of their types. This creates a direct hierarchy, I remove breaking declarations and circular dependencies, (discussed in detail within the discussion section) before compiling the code to verify that it is operational. I am left with one CDM.sol Smart Contract of considerable size, shown in Figure 3.3.

```
contract CDM is ISDA_CDM_ENUMS, ISDA_CDM_METATYPES {

    /**
     * This file is auto-generated from the ISDA Common Domain Model, do not edit.
        4.0.0-dev.5 * Version;
    */
    /**
     * A class to specify an account as an account number alongside, optionally. an account nam
     */
    struct Account {
    /**
     * A reference to the party beneficiary of the account.
     */
        ReferenceWithMeta accountBeneficiary;
    /**
     * The name by which the account is known.
     */
        FieldWithMeta accountName;
    /**
     * The account number.
     */
        FieldWithMeta accountNumber;
    /**
     * The type of account, e.g. client, house.
     */
        FieldWithMeta accountType;
        MetaFields meta;
    /**
     * A reference to the party to which the account refers to.
     */
        ReferenceWithMeta partyReference;
    /**
     * The reference to the legal entity that services the account, i.e. in the books of which
     */
        ReferenceWithMeta servicingParty;
    }

    struct AcctOwnr {
        Id id;
    }
```

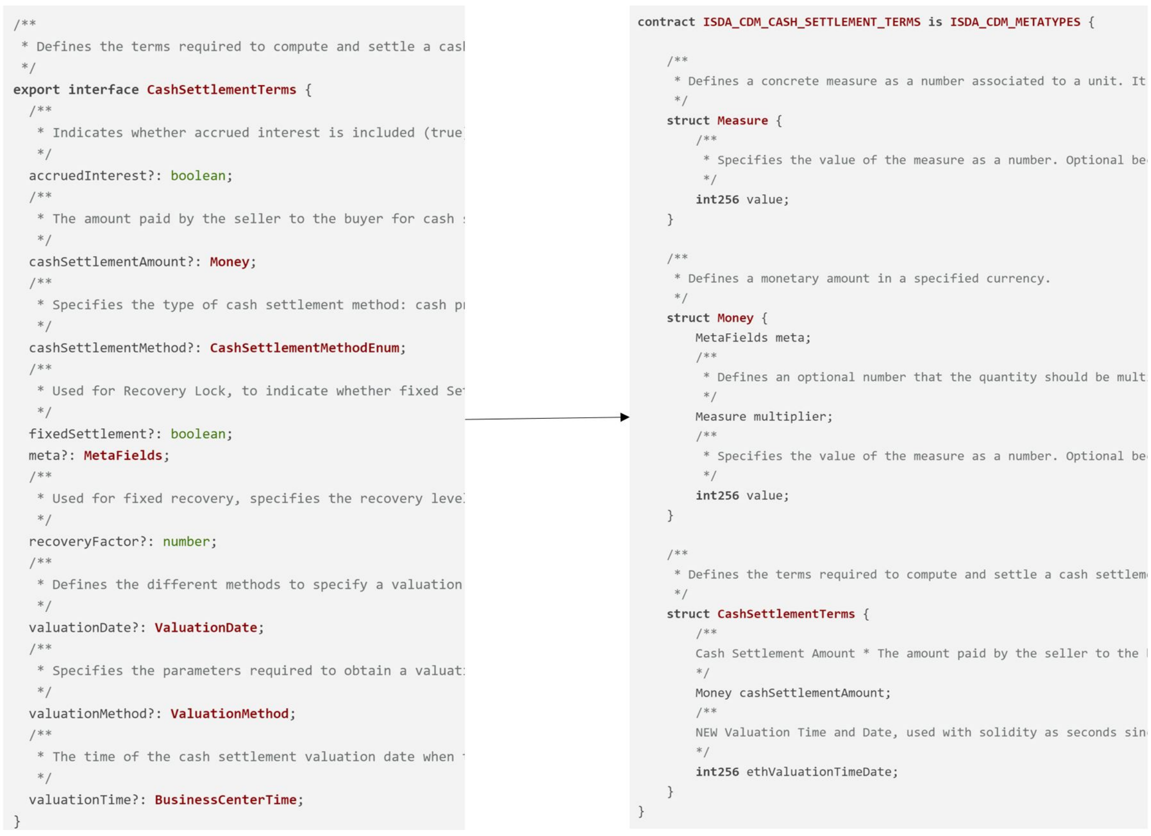
Figure 3.3: The start of the CDM Smart Contract, written in solidity. Visible is the inheritance of the Metatypes and Enumerations Smart Contracts. The entirety of the shown Smart Contract is data type declarations.

3.1.2 Modularisation of the CDM

The CDM.sol I constructed has approximately 2500 fields and over 700 data structures, with the bytecode of a Solidity–based CDM implementation being too large to be deployable on any public EVM chain. As such, I require further modularization of the types into manageable type–oriented Smart Contracts.

For the purpose of the Non–deliverable forward, Call and Put contracts I conjure the Smart Contracts: CDMParticipants.sol, CDMPricesource.sol, CDMPrice.sol, CDMCashSettlementTerms.sol, and CDMSpecifiedCurrency.sol. Each of which contains the relevant sub–types for the overarching type.

In Figure 3.4, the hierarchical structure of participant types within the Smart Contract



```
/**
 * Defines the terms required to compute and settle a cas
 */
export interface CashSettlementTerms {
  /**
   * Indicates whether accrued interest is included (true
   */
  accruedInterest?: boolean;
  /**
   * The amount paid by the seller to the buyer for cash
   */
  cashSettlementAmount?: Money;
  /**
   * Specifies the type of cash settlement method: cash p
   */
  cashSettlementMethod?: CashSettlementMethodEnum;
  /**
   * Used for Recovery Lock, to indicate whether fixed Se
   */
  fixedSettlement?: boolean;
  meta?: MetaFields;
  /**
   * Used for fixed recovery, specifies the recovery leve
   */
  recoveryFactor?: number;
  /**
   * Defines the different methods to specify a valuation
   */
  valuationDate?: ValuationDate;
  /**
   * Specifies the parameters required to obtain a valuat
   */
  valuationMethod?: ValuationMethod;
  /**
   * The time of the cash settlement valuation date when
   */
  valuationTime?: BusinessCenterTime;
}
```

```
contract ISDA_CDM_CASH_SETTLEMENT_TERMS is ISDA_CDM_METATYPES {

    /**
     * Defines a concrete measure as a number associated to a unit. It
     */
    struct Measure {
        /**
         * Specifies the value of the measure as a number. Optional be
         */
        int256 value;
    }

    /**
     * Defines a monetary amount in a specified currency.
     */
    struct Money {
        MetaFields meta;
        /**
         * Defines an optional number that the quantity should be mult
         */
        Measure multiplier;
        /**
         * Specifies the value of the measure as a number. Optional be
         */
        int256 value;
    }

    /**
     * Defines the terms required to compute and settle a cash settlem
     */
    struct CashSettlementTerms {
        /**
        Cash Settlement Amount * The amount paid by the seller to the
        */
        Money cashSettlementAmount;
        /**
        NEW Valuation Time and Date, used with solidity as seconds sin
        */
        int256 ethValuationTimeDate;
    }
}
```

Figure 3.4: I take the Typescript CashSettlementTerms type, exported and translate it to modular EVM–based Smart Contract that implements the relevant CashSettlementTerms fields — with relevant dependencies. Further visible is the new field 'ethValuationTimeDate', which can be used natively within Ethereum Smart Contracts. Notably the unused optional fields are omitted from the Smart Contract to preserve space.

is displayed after translation on the right image. Specifically, the Buyr and Sellr types rely on the AcctOwnr type, which is in turn dependent on the Id type. The latter solely hinges on the native address type inherent to the Ethereum platform.

To harmonize this type of hierarchy with the practical requirements of an operational Smart Contract, additional fields are incorporated to govern the contract's logic. For instance, the signed field is introduced, holding a boolean value to signify whether the contract has been duly signed.

In the context of the Common Domain Model (CDM), the Id type incorporates a LEI[2] (Legal Entity Identifier). However, given the decentralized nature of blockchain, an Ethereum address aptly replaces the LEI. Consequently, redundant fields that consume Ethereum Virtual Machine (EVM) storage slots are pruned from the contract.

It is important to note that Solidity does not offer native support for optional fields. Fields in Solidity structures, even when null, occupy storage slots. To optimize for stor-

[2]The Legal Entity Identifier (LEI) is a 20–character alphanumeric code that uniquely identifies legally distinct entities participating in financial transactions. Within the CDM, it is stored as a string.

age space, unused structures are excised from the contract. What remains are modular contracts stripped down to their essential elements, efficiently representing the required types while conserving EVM resources.

This streamlined approach results in lean, storage–efficient modularised CDM contracts, equipped with only the necessary types and logic, thereby making it optimally compatible with the constraints and capabilities of the Ethereum blockchain.

3.1.3 Conditioning of the Confirmation Smart Contract with the Common Domain Model

With the transactional terms of the natural language Confirmation contract encapsulated as stored variables, I then manually condition their composition to align with the Common Domain Model (CDM) modules of Section 3.2. This alignment is achieved by importing a list of contextually relevant CDM Smart Contract modules as defined in the previous section and initiating them at the time of contract construction.

These specialized modules enable variables such as 'forward price' to be typed as 'price', thereby facilitating a seamless transition between the CDM's off–chain and on–chain representations. Such uniformity ensures that consistency is maintained across different platforms, enhancing the accessibility of the current derivatives market infrastructure. Figure 3.5 illustrates the comprehensive process of transforming from the Rosetta CDM to modular CDM–type Smart Contracts.

Updates to the CDM are reflected within the Rosetta implementation and subsequently permeate through to the Smart Confirmation Contract and its selected Logic Modules. This link ensures that legacy smart confirmation contracts remain aligned with previous CDM implementations, while newer contracts adhere to the latest version. Therefore, it is important that the CDM data structures are kept standardized. This standardization not only preserves compatibility across various implementations but also underpins the overall integrity of the derivatives trading ecosystem.

3.1.4 The Logic Module: A Singleton Standardised Agnostic Logic Smart Contract

To implement the logical state outlined within the Smart Confirmation Contracts, I propose a modular paradigm comprising 'Logic' contracts that seamlessly integrate with the central Smart Contract. This integration is facilitated by standardized 'Logic Module Contracts', designed to extract readable data from the confirmation document and equipped with explicit permissions to modify the state of the Smart Confirmation Contract.

Each Logic Module Contract is architecturally unique, tailored to perform a specific task, and encapsulated within a 'singleton' contract. For instance, the cryptographic 'signing' of the contract, including changes to the 'BuyerSigned' or 'SellerSigned' boolean val-

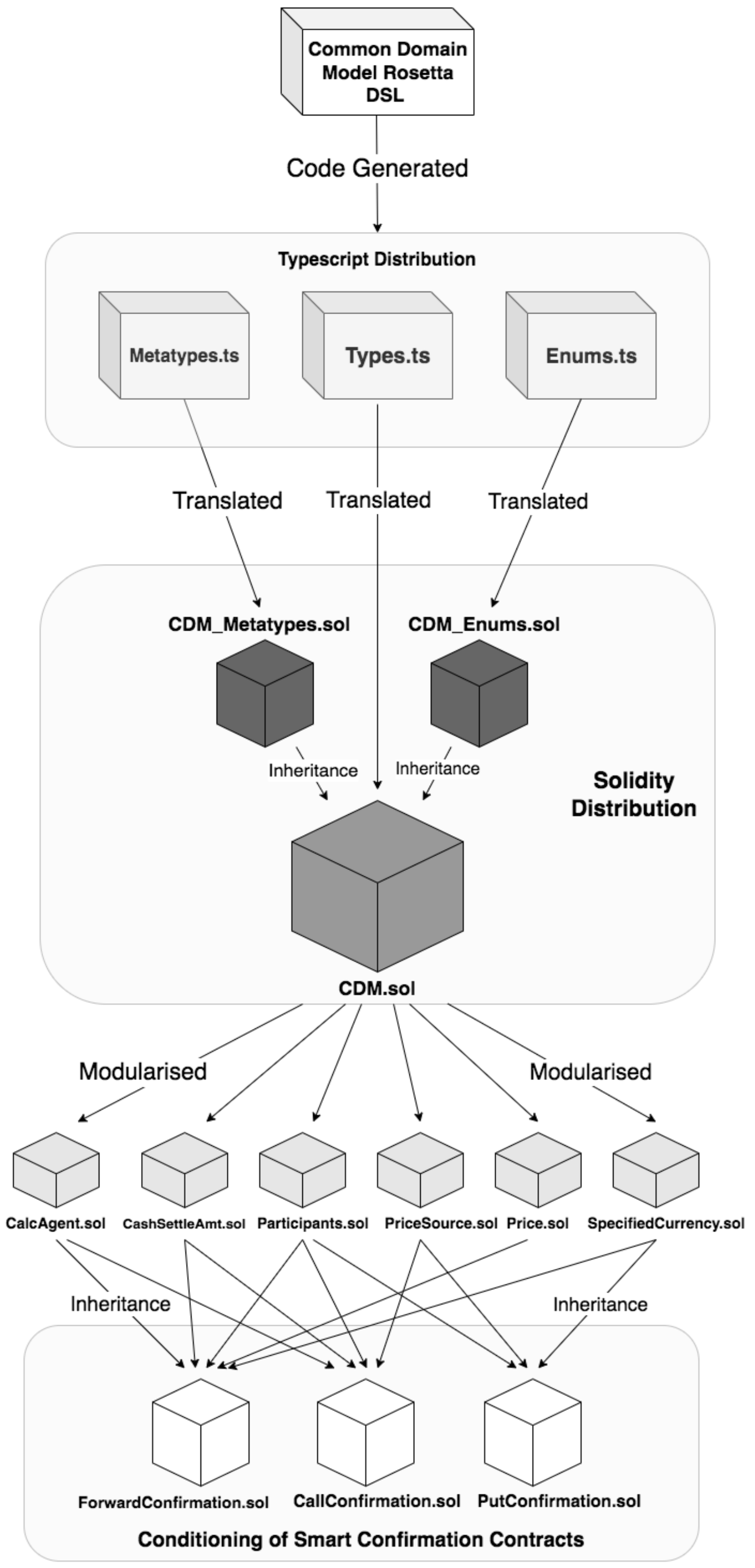


Figure 3.5: The process of going from the CDM that is written in Rosetta, to a code-generated CDM in Typescript, to a CDM written in Solidity, to CDM–type Smart Contract modules.

ues in the smart confirmation contracts logic state, would reside solely within a dedicated 'BuyerSellerSigned.sol' contract. The Logic Module Contract could further alter the state of other contracts on–chain, be that transferring ERC–20 tokens, collateral management contracts or Oracles.

In an effort to foster readability and structural coherence, every function is crafted with an autonomous logic. This design choice eliminates the need for globally hard–coded variables, thereby preserving the sequentiality in the code. Sequentiality in this context ensures that code execution progresses in a line–by–line fashion, allowing for straightforward traversal from one step to the next. Such an approach further benefits legal experts, who can read the operational logic of each function simply by examining its printout. Indeed, they would not have to navigate between disparate functions and classes.

The ramifications of this architecture extend beyond mere code structure. Allowing a single logic module contract to be deployed, it paves the way for flexible utilization by any authorized Smart Confirmation Contract. As a result, each module becomes a 'standard' that can be customized by legal authorities. An example might be the International Swaps and Derivatives Association (ISDA) creating a 'PriceSourceDisruption.sol' logic contract, delineating selectable events, defining terms, outlining prerequisites, and culminating with a pseudocode function to manage fallback effects or terminations.

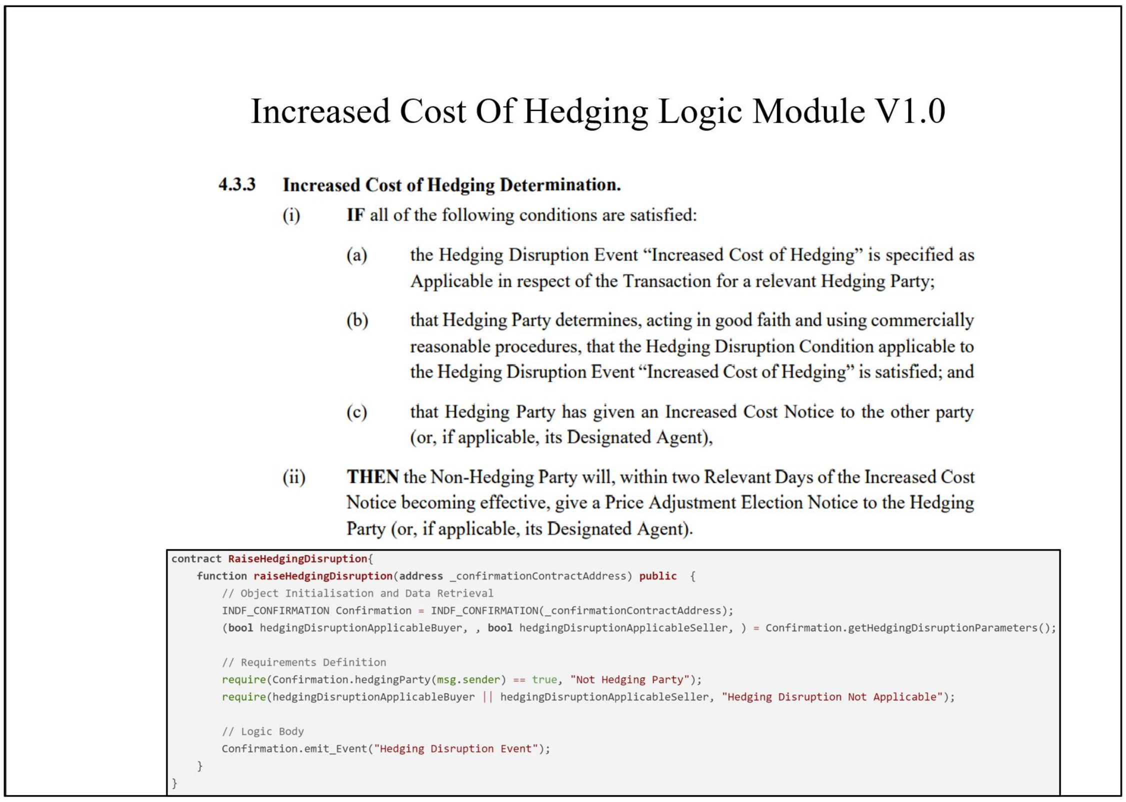
Each Logic Module can be encapsulated within a natural language contract, akin to the Internal Model discussed in Subsection 2.7.2 and shown in Figure 3.6. These terms would clarify both the module's functionality and its role within the overarching architecture of the agreement. Take, for instance, a module responsible for cryptographic signing: the contract could specify the cryptographic algorithm employed, as well as delineate contingency clauses to be activated in scenarios such as algorithmic failure or obsolescence due to cryptographic advancements like quantum computing.

This integration of natural language not only enhances comprehensibility but fosters alignment between technical implementation and legal interpretation. By providing a detailed description of each logic module's function, parameters, conditions, and outcomes, the natural language contract serves as both a guide and a legal reference — making each Logic Module its own Internal Smart Contract.

For legal professionals, this combination provides a bridge between the technical complexities of Smart Contract Code and traditional legal framework. It ensures that the legal context, stipulations, and objectives are readily understood, without the necessity to decipher complex coding structures. For technologists, the natural language explanations offer insights into the legal constraints and considerations that underpin the coded logic, facilitating the development of legally compliant and context–aware Smart Contracts.

Furthermore, the availability of natural language contracts enables a broader audience, including regulators, auditors, and non–technical stakeholders, to engage with and assess the confirmation contract process. The standardized approach to combining legal language with technical code also aids in maintaining consistency and uniformity across different agreements and jurisdictions.

This approach gives the Smart Confirmation Contract the ability to bridge the divide between conventional legal documents and their Smart Contract analogues, amplifying the adaptability, agility and flexibility of the financial contractual ecosystem.



Increased Cost Of Hedging Logic Module V1.0

4.3.3 Increased Cost of Hedging Determination.

(i) **IF** all of the following conditions are satisfied:

(a) the Hedging Disruption Event "Increased Cost of Hedging" is specified as Applicable in respect of the Transaction for a relevant Hedging Party;

(b) that Hedging Party determines, acting in good faith and using commercially reasonable procedures, that the Hedging Disruption Condition applicable to the Hedging Disruption Event "Increased Cost of Hedging" is satisfied; and

(c) that Hedging Party has given an Increased Cost Notice to the other party (or, if applicable, its Designated Agent),

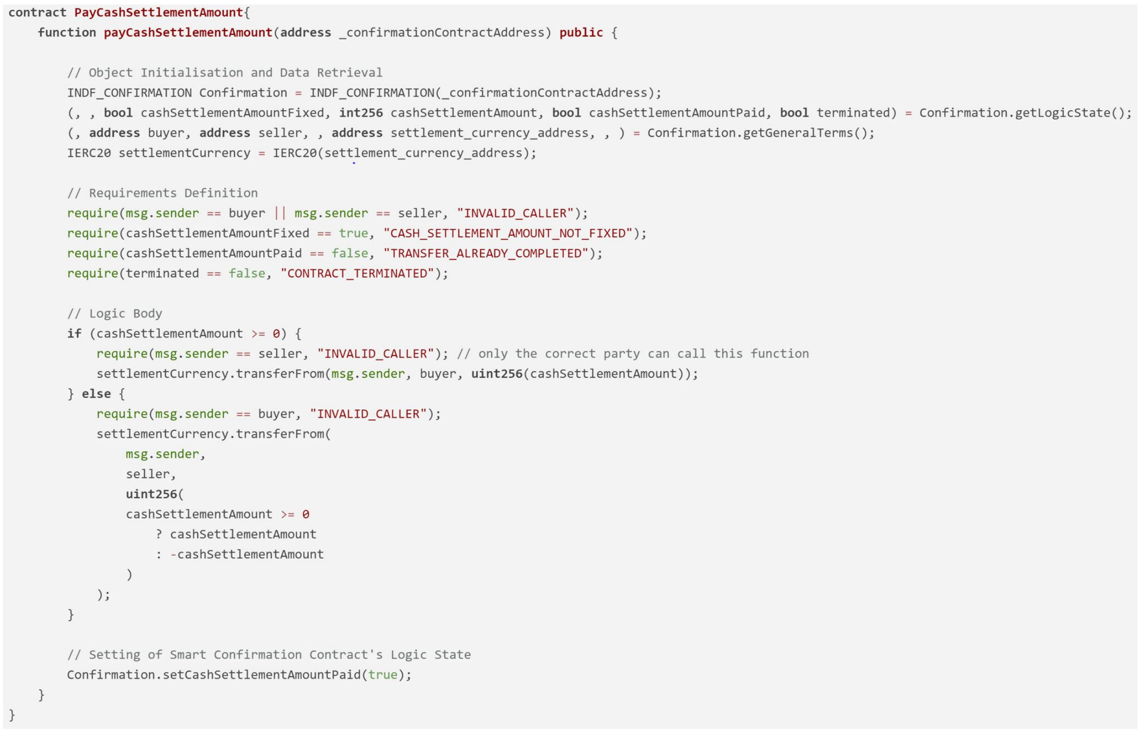
(ii) **THEN** the Non-Hedging Party will, within two Relevant Days of the Increased Cost Notice becoming effective, give a Price Adjustment Election Notice to the Hedging Party (or, if applicable, its Designated Agent).

```
contract RaiseHedgingDisruption{
    function raiseHedgingDisruption(address _confirmationContractAddress) public  {
        // Object Initialisation and Data Retrieval
        INDF_CONFIRMATION Confirmation = INDF_CONFIRMATION(_confirmationContractAddress);
        (bool hedgingDisruptionApplicableBuyer, , bool hedgingDisruptionApplicableSeller, ) = Confirmation.getHedgingDisruptionParameters();

        // Requirements Definition
        require(Confirmation.hedgingParty(msg.sender) == true, "Not Hedging Party");
        require(hedgingDisruptionApplicableBuyer || hedgingDisruptionApplicableSeller, "Hedging Disruption Not Applicable");

        // Logic Body
        Confirmation.emit_Event("Hedging Disruption Event");
    }
}
```

Figure 3.6: An example of how a Logic Module contract for the 'Increased Cost of Hedging' event in the Digital Asset Definitions may appear. Visually, one may have controlled natural language and the relevant code for its implementation. The code would follow the four–section standardised structure. The image is shown with the permission of ISDA.

3.1.5 Four Section Standardisation of Logic Module Smart Contracts

In proposing an architecture for singleton logic modules, it is important to include an internal standardization for how functions should be written. Indeed, as is shown in Figure ???3.7 the Smart Confirmation Contract's address is the first parameter for each Logic Module. This configuration imbues the versatility of individual Logic Modules, allowing them to modify multiple Smart Confirmation Contracts while being individually deployed once. To enhance code legibility, I advise that there is further partitioning of each logic function into clearly delineated sections, separated either by comments or strategic



```
contract PayCashSettlementAmount{
    function payCashSettlementAmount(address _confirmationContractAddress) public {

        // Object Initialisation and Data Retrieval
        INDF_CONFIRMATION Confirmation = INDF_CONFIRMATION(_confirmationContractAddress);
        (, , bool cashSettlementAmountFixed, int256 cashSettlementAmount, bool cashSettlementAmountPaid, bool terminated) = Confirmation.getLogicState();
        (, address buyer, address seller, , address settlement_currency_address, , ) = Confirmation.getGeneralTerms();
        IERC20 settlementCurrency = IERC20(settlement_currency_address);

        // Requirements Definition
        require(msg.sender == buyer || msg.sender == seller, "INVALID_CALLER");
        require(cashSettlementAmountFixed == true, "CASH_SETTLEMENT_AMOUNT_NOT_FIXED");
        require(cashSettlementAmountPaid == false, "TRANSFER_ALREADY_COMPLETED");
        require(terminated == false, "CONTRACT_TERMINATED");

        // Logic Body
        if (cashSettlementAmount >= 0) {
            require(msg.sender == seller, "INVALID_CALLER"); // only the correct party can call this function
            settlementCurrency.transferFrom(msg.sender, buyer, uint256(cashSettlementAmount));
        } else {
            require(msg.sender == buyer, "INVALID_CALLER");
            settlementCurrency.transferFrom(
                msg.sender,
                seller,
                uint256(
                cashSettlementAmount >= 0
                    ? cashSettlementAmount
                    : -cashSettlementAmount
                )
            );
        }

        // Setting of Smart Confirmation Contract's Logic State
        Confirmation.setCashSettlementAmountPaid(true);
    }
}
```

Figure 3.7: Singleton contract for paying cash settlement amounts. As is visible, the different sections of the functions are written and separated, for readability in a printed format.

spacing:

1. **Object Initializations and Data Retrieval**

   In this initial phase, third–party contracts that are pertinent to the function are initiated. Using the confirmation Smart Contract address, the confirmation Smart Contract itself is also initialized through its interface. This segment is dedicated to fetching the required data for the function's operation, deriving it from various sources, whether they are other third–party contracts, Oracles, or stored variables from a trustworthy entity.

2. **Requirements Definition**

   This section delineates the preconditions necessary for accessing the agreement's logic. It essentially establishes the 'gates' that control access, reflecting the contractual conditions, constraints, or prerequisites. Examples of requirements might include validation that the caller is either the 'buyer' or 'seller' or a check that the confirmation contract is not in the 'terminated' state. Thus, these requirements act as essential filters and need meticulous understanding to ensure that they correctly correspond to the logical state within the smart confirmation contract.

3. **Logic Body**

Building on the requirements, this part of the contract encapsulates the core logic or ‘substance’ of the function. It’s here that the function’s primary actions are executed, aligning with the natural language description provided in the definitions document. Whether it involves interacting with third–party contracts, conducting financial transactions, or setting or modifying variables, this section operationalizes the intended logic, executing the necessary steps. The Logic Body can be in alignment with the standardized representations of functions put forth in the Common Domain Model.

4. **The Setting of The Smart Confirmation Contract’s Logic State**

   The final section is committed to recording or updating the state of the Smart Confirmation Contract in accordance with the logic executed. Unlike the earlier mention of this section, this is where the outcomes of the function’s actions are formalized within the contract’s logic state. It may include setting flags, updating status variables, or any other alterations reflecting the conclusion of the logic module’s operation.

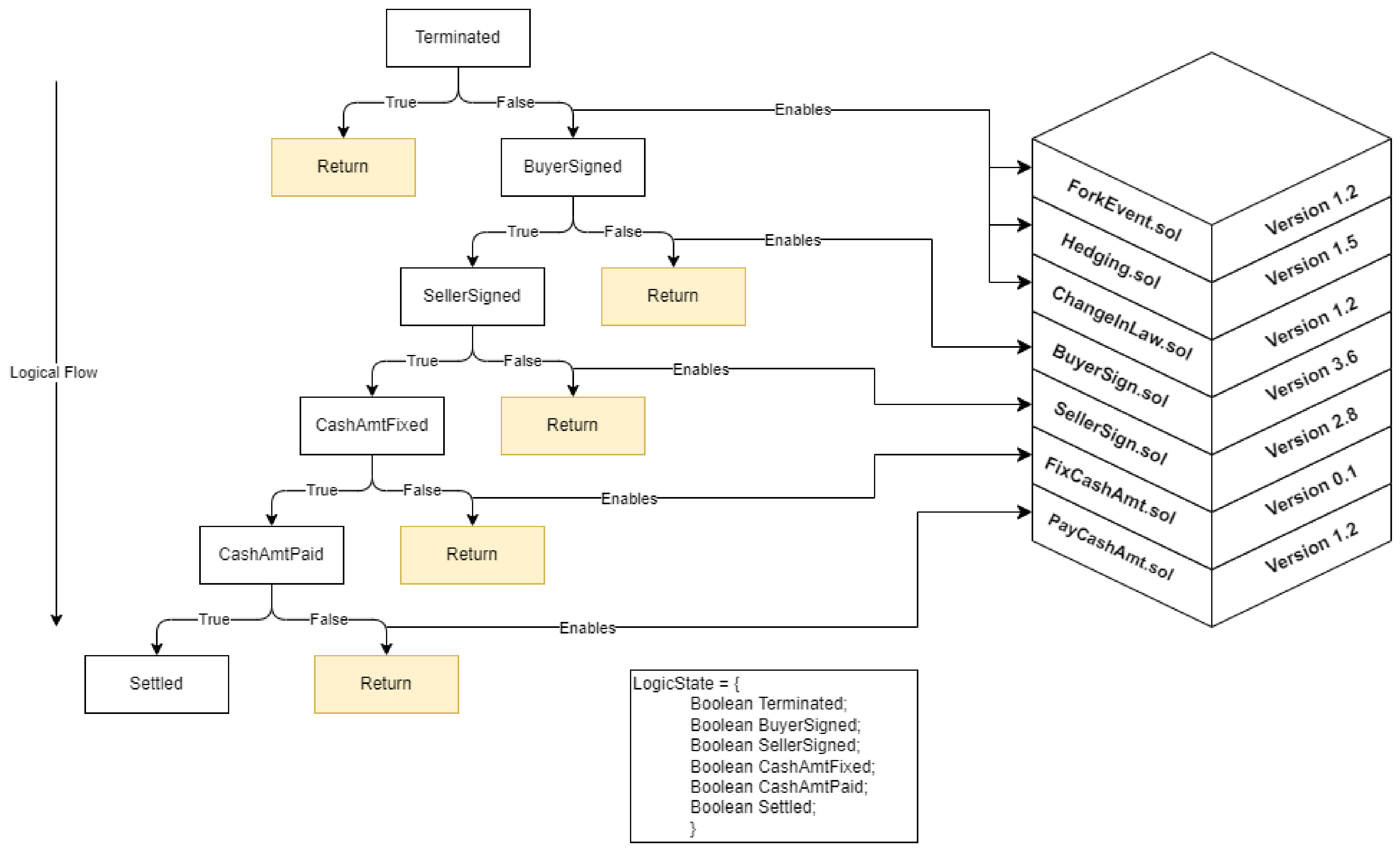
Further, standardization can follow through from the aforementioned Digital Asset Definitions. Indeed, one can create Logic Modules for each of the disruption clauses: Forking, Hedging, Change in Law, and Price Source Disruption. Within the price source disruption, I implement a price–source fallback mechanism, where the price–source determining party can raise a disruption and then change the price source to the next one in sequence — in practice changing the Chainlink address to a new price feed. Such a document is defined below in the Linklaters Internal Model fashion.

3.1.6 Representing A Derivative Lifecycle As Stacks of Logic Modules

In the proposed architecture, the lifecycle of a derivative contract is represented as a stack of logic modules, each governed by the overarching ‘Logic State’.

For instance, one party may opt for a specialized signing logic module that leverages Oracle services to invoke specific APIs, thereby generating off–chain fallback events. Conversely, another party may prefer the traditional route of using cryptographic signatures via their private key to confirm the contract.

Additionally, the system allows for customization based on particular product specifications. Should a party wish to have the contractual option to invoke termination due to a change–in–law event, they can incorporate a Logic Module termed ‘ChangeInLaw.sol’ in this context that would alter the state variable ‘terminated’. Parties can select from a range of available implementations, contingent on their understanding of the nuanced differences each option presents, and available for bespoke customisation (like that of the Schedules purpose) by legal and computer science experts.



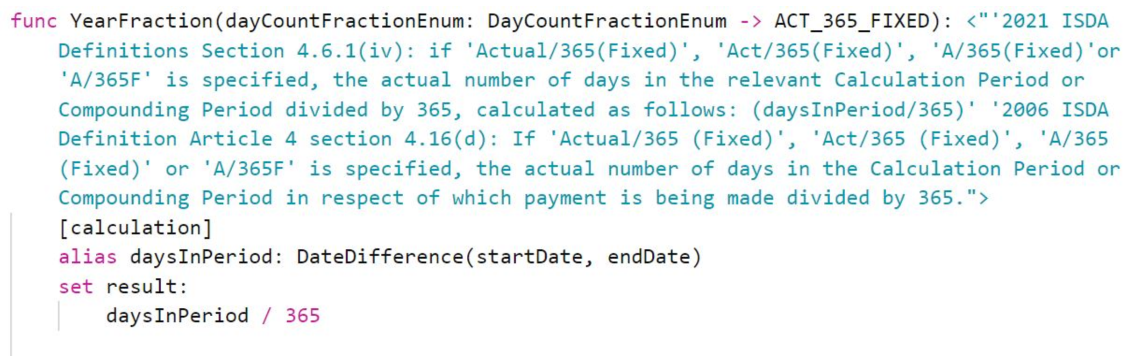
```
LogicState = {
        Boolean Terminated;
        Boolean BuyerSigned;
        Boolean SellerSigned;
        Boolean CashAmtFixed;
        Boolean CashAmtPaid;
        Boolean Settled;
        }
```

Figure 3.8: The Smart Contract Logic State enables different logic modules, based upon which boolean values of the Logic State are enabled or disabled — the lifecycle. The LogicState type is stored within the Smart Confirmation contract. Each Logic Module can be 'mixed and matched' at the discretion of the parties. Over time, the stack is traversed from top to bottom.

As illustrated in Figure 3.8, the contract's logic flow is sequential and hierarchical. Initially, the system verifies the state variable 'terminated'. If it returns as false, the next tier of logic modules related to disruption events is enabled. Furthermore, when either the buyer or the seller is signed, their corresponding logic modules are activated in succession. Hence, the logic flow within the Smart Derivative Contract is inherently ordered and determined by the status of the Logic State stored in the Smart Confirmation Contract.

Upon initiating a Smart Confirmation Contract, the architect chooses the foundational LogicState schema. This schema serves as a skeletal framework to which specific modules can be appended. While some implementations may require only simple functionalities, such as cryptographic signatures, others may necessitate a more comprehensive set of operations, encompassing not only signature verification but also asset transfer and notice delivery.

3.1.7 Building and Bridging CDM Functions to Digital Asset Definition Logic Modules

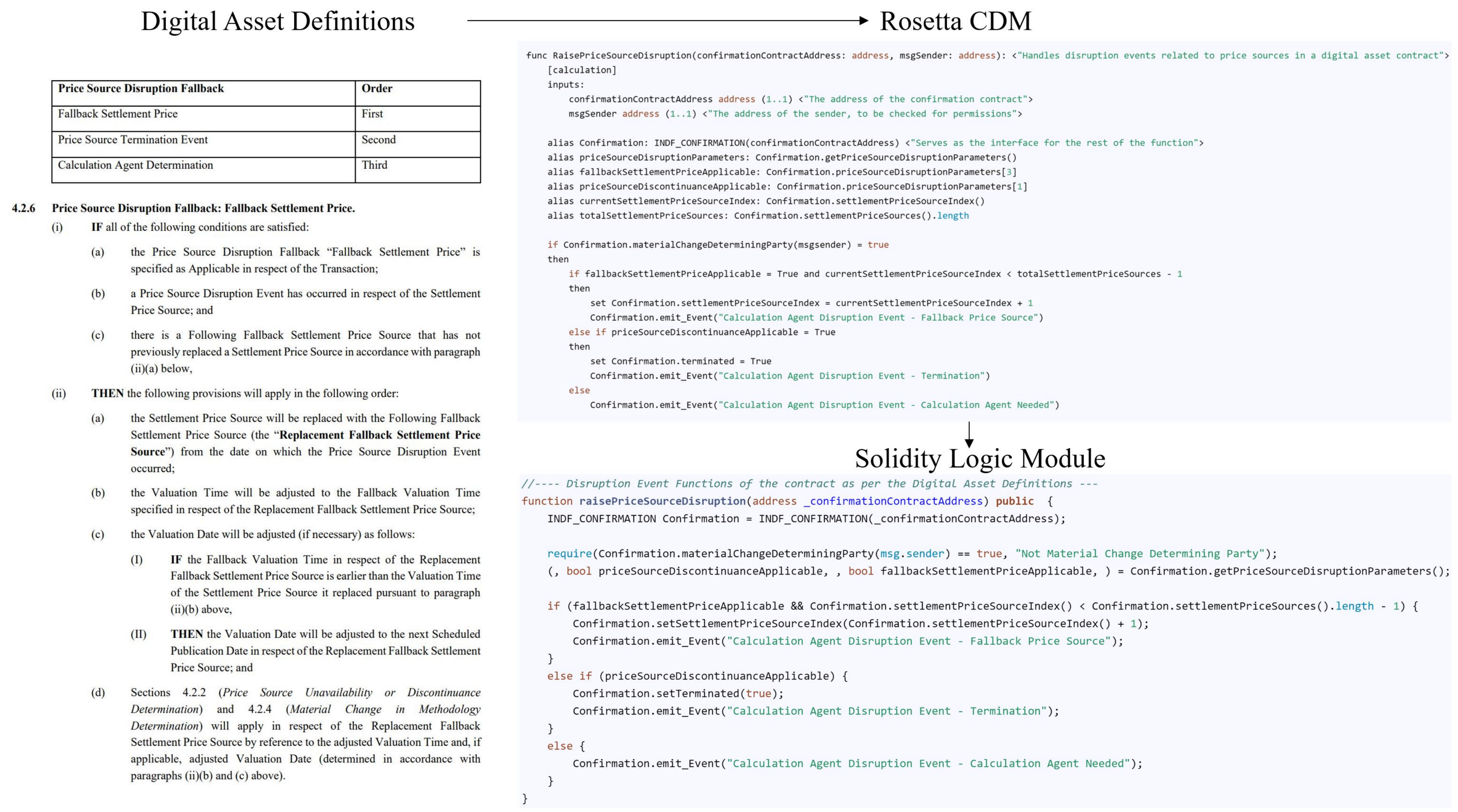


```
func YearFraction(dayCountFractionEnum: DayCountFractionEnum -> ACT_365_FIXED): <"'2021 ISDA
    Definitions Section 4.6.1(iv): if 'Actual/365(Fixed)', 'Act/365(Fixed)', 'A/365(Fixed)'or
    'A/365F' is specified, the actual number of days in the relevant Calculation Period or
    Compounding Period divided by 365, calculated as follows: (daysInPeriod/365)' '2006 ISDA
    Definition Article 4 section 4.16(d): If 'Actual/365 (Fixed)', 'Act/365 (Fixed)', 'A/365
    (Fixed)' or 'A/365F' is specified, the actual number of days in the Calculation Period or
    Compounding Period in respect of which payment is being made divided by 365.">
    [calculation]
    alias daysInPeriod: DateDifference(startDate, endDate)
    set result:
        daysInPeriod / 365
```

Figure 3.9: A CDM type called 'YearFraction', in which calculates the proportion of the year that has passed, based on the specified period standard (derived from the enumeration).

The CDM is in the vast majority defined in data type structures. However, there is a set of functions used for the standardization of calculations. For example, Figure 3.9 displays a Rosetta function used to calculate the fraction of the year that has passed — a task useful in pricing and swap calculations.

Taking such functional standardization further, one can implement a CDM function for implementing Smart Contract logic — specifically, the logic put forth in any ISDA definitions product. I take for example the aforementioned Digital Asset Definitions — a contextually appropriate definition document — which delineates how fallbacks should occur upon a price source disruption being raised. Indeed, if elected and raised the price source disruption changes to a settlement price source, and upon not finding one goes to


Digital Asset Definitions
Rosetta CDM
Solidity Logic Module

Price Source Disruption Fallback	Order
Fallback Settlement Price	First
Price Source Termination Event	Second
Calculation Agent Determination	Third

4.2.6 Price Source Disruption Fallback: Fallback Settlement Price.

(i) **IF** all of the following conditions are satisfied:

(a) the Price Source Disruption Fallback "Fallback Settlement Price" is specified as Applicable in respect of the Transaction;

(b) a Price Source Disruption Event has occurred in respect of the Settlement Price Source; and

(c) there is a Following Fallback Settlement Price Source that has not previously replaced a Settlement Price Source in accordance with paragraph (ii)(a) below,

(ii) **THEN** the following provisions will apply in the following order:

(a) the Settlement Price Source will be replaced with the Following Fallback Settlement Price Source (the "**Replacement Fallback Settlement Price Source**") from the date on which the Price Source Disruption Event occurred;

(b) the Valuation Time will be adjusted to the Fallback Valuation Time specified in respect of the Replacement Fallback Settlement Price Source;

(c) the Valuation Date will be adjusted (if necessary) as follows:

(I) **IF** the Fallback Valuation Time in respect of the Replacement Fallback Settlement Price Source is earlier than the Valuation Time of the Settlement Price Source it replaced pursuant to paragraph (ii)(b) above,

(II) **THEN** the Valuation Date will be adjusted to the next Scheduled Publication Date in respect of the Replacement Fallback Settlement Price Source; and

(d) Sections 4.2.2 (*Price Source Unavailability or Discontinuance Determination*) and 4.2.4 (*Material Change in Methodology Determination*) will apply in respect of the Replacement Fallback Settlement Price Source by reference to the adjusted Valuation Time and, if applicable, adjusted Valuation Date (determined in accordance with paragraphs (ii)(b) and (c) above).

```
func RaisePriceSourceDisruption(confirmationContractAddress: address, msgSender: address): <"Handles disruption events related to price sources in a digital asset contract">
    [calculation]
    inputs:
        confirmationContractAddress address (1..1) <"The address of the confirmation contract">
        msgSender address (1..1) <"The address of the sender, to be checked for permissions">

    alias Confirmation: INDF_CONFIRMATION(confirmationContractAddress) <"Serves as the interface for the rest of the function">
    alias priceSourceDisruptionParameters: Confirmation.getPriceSourceDisruptionParameters()
    alias fallbackSettlementPriceApplicable: Confirmation.priceSourceDisruptionParameters[3]
    alias priceSourceDiscontinuanceApplicable: Confirmation.priceSourceDisruptionParameters[1]
    alias currentSettlementPriceSourceIndex: Confirmation.settlementPriceSourceIndex()
    alias totalSettlementPriceSources: Confirmation.settlementPriceSources().length

    if Confirmation.materialChangeDeterminingParty(msgsender) = true
    then
        if fallbackSettlementPriceApplicable = True and currentSettlementPriceSourceIndex < totalSettlementPriceSources - 1
        then
            set Confirmation.settlementPriceSourceIndex = currentSettlementPriceSourceIndex + 1
            Confirmation.emit_Event("Calculation Agent Disruption Event - Fallback Price Source")
        else if priceSourceDiscontinuanceApplicable = True
        then
            set Confirmation.terminated = True
            Confirmation.emit_Event("Calculation Agent Disruption Event - Termination")
        else
            Confirmation.emit_Event("Calculation Agent Disruption Event - Calculation Agent Needed")
```

```
//---- Disruption Event Functions of the contract as per the Digital Asset Definitions ---
function raisePriceSourceDisruption(address _confirmationContractAddress) public  {
    INDF_CONFIRMATION Confirmation = INDF_CONFIRMATION(_confirmationContractAddress);

    require(Confirmation.materialChangeDeterminingParty(msg.sender) == true, "Not Material Change Determining Party");
    (, bool priceSourceDiscontinuanceApplicable, , bool fallbackSettlementPriceApplicable, ) = Confirmation.getPriceSourceDisruptionParameters();

    if (fallbackSettlementPriceApplicable && Confirmation.settlementPriceSourceIndex() < Confirmation.settlementPriceSources().length - 1) {
        Confirmation.setSettlementPriceSourceIndex(Confirmation.settlementPriceSourceIndex() + 1);
        Confirmation.emit_Event("Calculation Agent Disruption Event - Fallback Price Source");
    }
    else if (priceSourceDiscontinuanceApplicable) {
        Confirmation.setTerminated(true);
        Confirmation.emit_Event("Calculation Agent Disruption Event - Termination");
    }
    else {
        Confirmation.emit_Event("Calculation Agent Disruption Event - Calculation Agent Needed");
    }
}
```

Figure 3.10: This figure displays the logic explained within the digital asset definitions, compared against its interpreted Rosetta CDM Logic Module contract and the subsequent Solidity Logic Module.

termination. If not terminated the Calculation Agent is signaled to become involved, to change the financial parameters to fairly represent the change.

Figure 3.10 shows the simplified reflection of the Digital Assets disruption clause in Rosetta, before the subsequent translation to a Solidity Logic Module. Such a process proves the ability of lawyers to start the Smart Contract production pipeline, and subsequently for computer scientists to interpret and produce the relevant Smart Contract Code for execution.

3.2 First and Second Order Periphery

In the proposed architecture, a standardizing body like ISDA would deploy a set of Logic Modules along with the Master Agreement onto a blockchain platform. Parties engaging in OTC derivatives trading would converse with ISDA to formulate a Smart Confirmation Contract, which would be conditioned by the Common Domain Model (CDM). This contract specifies the necessary Logic Modules and the Logic State governing the transactional lifecycle and its trade. For enhanced customization, parties can append a natural language schedule that supersedes specific terms in the Master Agreement. One further cites the product definitions pertinent to the particular trade. The Smart Confirmation Contract, Logic Modules, natural language contracts, and directly implicated data, would constitute what is termed the 'first–order periphery' (presented in Figure 3.11).

Beyond the bounds of this first–order periphery lies a 'second–order periphery', which comprises Smart Contracts, Oracles and data not immediately stored or referenced in the Smart Confirmation Contract. This could include, for instance, ChainLink Smart Contracts that serve as Oracles, ERC–20 tokens for collateral or payments, and external collateral management contracts for risk mitigation.

Apart from indirectly influencing them through the selection of first–order Logic Modules, parties do not exercise direct authority over the second–order periphery. This domain is under the control of a third–party entity. Both cyber risk and the potential for transaction failure are risks that these parties entrust to these third–party — and should be carefully overviewed, with many fallbacks in place for their malfunctions.

3.3 An Ecosystem of Smart Confirmation Contract Derivatives

In an illustrative example shown in Figure 3.12, two distinct Smart Confirmation Contracts — a Non–Deliverable Call and a Non–Deliverable Forward — share the Master Agreement and select logic modules. Additionally, some of these logic modules interact with elements in the second–order periphery, illustrating the possibility of a complex interplay between centralized standardization and decentralized financial instruments.

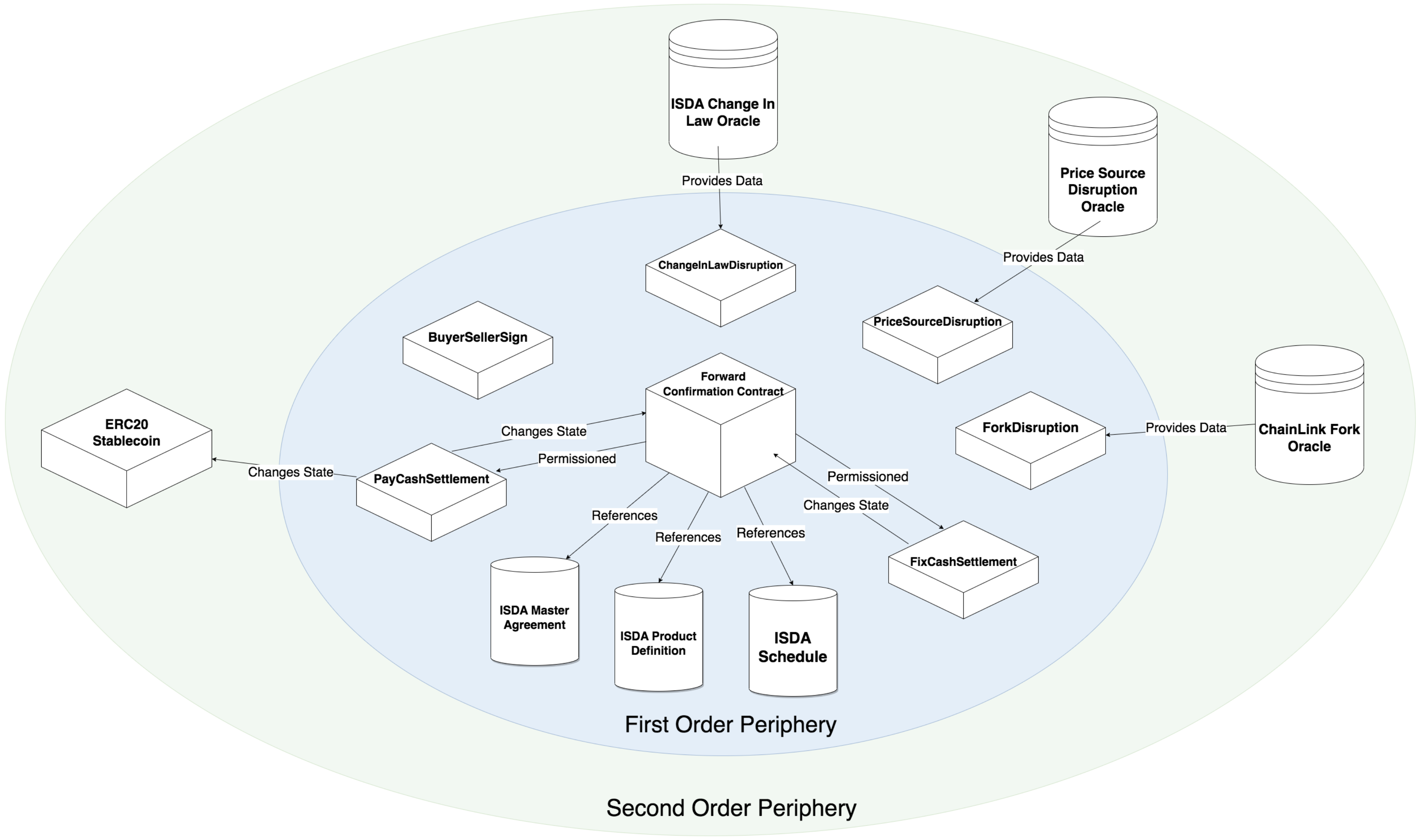


Figure 3.11: Semantically, The flat cuboids represent Smart Logic Contracts. The taller flat cuboid represents an ERC–20 stablecoin like USDC. The blank cylinder represents a Natural Language document. The double–striped cylinder represents Oracle Smart Contracts. The Smart Confirmation Contract (Cube) lies in the middle, surrounded by its Logic Modules and the Natural Language Contracts it references. First–order periphery are contracts able to alter the Smart Confirmation Contract. Second-order periphery are the contracts unable to alter the confirmation contract and interact with through the first–order periphery — like that of external ERC–20 tokens.

In the transition to a blockchain–based ecosystem, a complete overhaul is not obligatory. Participants may opt for an incremental integration, initially choosing to focus on specific elements of the Logic State process. For example, they could commence by cryptographically signing natural language documents to ensure data integrity — such an application would only require a Smart Confirmation Contract, a reference to the natural language contract, and a single Logic Module responsible for signing. Alternatively, one may desire the emission of event notifications on a Smart Contract, thereby achieving immutability and transparency, akin to an improved notice–delivery process. Moreover, the modular architecture allows parties to select a 'transfer' module that orchestrates off–chain transactions. This mitigates the cyber risk associated with the most security–sensitive components of a derivative.

In summary, this proposal serves as a pivotal stepping stone for integrating the Common Domain Model into an ecosystem of legally compliant Smart Contract building blocks. These contracts are verified by legal experts and constructed by computer scientists. Featuring amendable parameters, immutable records, and transparent processes, this new framework constitutes a Pareto improvement over today's settlement and contract management systems.

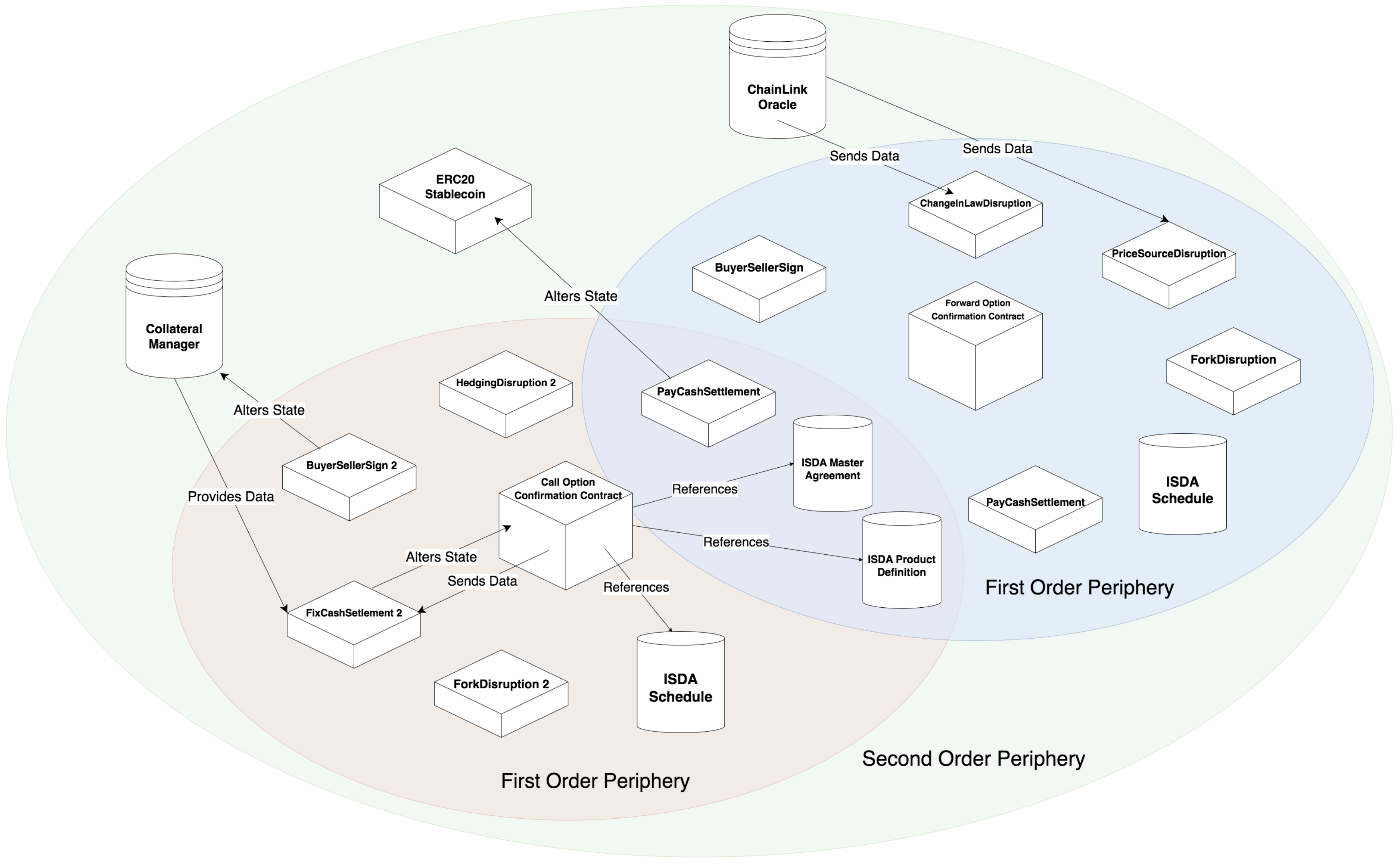


Figure 3.12: Semantically, The flat cuboids represent Smart Logic Contracts. The taller flat cuboid represents an ERC–20 stablecoin like USDC. The blank cylinder represents a Natural Language document. The double–striped cylinder represents Oracle Smart Contracts. The Smart Confirmation Contracts (Cubes) lie in the middle, surrounded by their Logic Modules and the Natural Language contracts they reference. Different Smart Confirmation Contracts can interact with the same Logic Modules and can reference the same natural language contracts. Each Smart Confirmation Contract will by proxy share the same second–order periphery.

Chapter 4

Results, Caveats, and Discussion

In this chapter, I elaborate on the process of constructing the Modular CDM Smart Contracts, Logic Module Smart Contracts, and Smart Confirmation Contracts. These components form the architectural backbone for the ensuing implementation. A decentralized application (dApp) is then developed[1], utilizing Digital Asset Definitions to instantiate financial derivatives such as non–deliverable forwards, calls, and puts. This chapter serves to scrutinize the various considerations and challenges and concludes with an evaluative discussion of the methodology's overall feasibility.

4.1 Problematic CDM translation

Although the CDM–type modules were produced and operational, they are in many ways problematic in their construction.

4.1.1 Enumeration Too Large

In the translation of the TypeScript implementation of the Common Domain Model (CDM) to Solidity, a bottleneck arises due to the FloatingRateIndexEnum enumeration exceeding Solidity's limitation of 256 elements. Two remedies are available:

[1]The dApp development employs a web application stack:

1. **NextJS**: A React-based framework optimized for server-rendering, static site generation, and performant client-side rendering.
2. **Wagmi**: A specialized Web3 utility library designed to simplify interactions with blockchain assets.
3. **Bootstrap**: A front-end framework for rapid UI development, providing pre-built, responsive design components.
4. **ChatGPT's API**: Utilized for natural language processing capabilities within the application's user interface.
5. **ethers.js**: A lightweight pure JavaScript library to interact with the Ethereum blockchain.
6. **web3.js**: A collection of libraries that enable interaction with a local or remote Ethereum node using HTTP, IPC, or WebSocket.

1. Refactor the Rosetta CDM to introduce a bifurcated structure, creating FloatingRateIndexEnum1 and FloatingRateIndexEnum2, each containing up to 256 fields. Upon more than 512 fields being needed, one would create FloatingRateIndexEnum3.
2. Employ a pruning strategy that retains only the 256 most commonly used fields within the existing FloatingRateIndexEnum, necessitating periodic updates for deprecated or less–used fields.

The first strategy, despite introducing a significant alteration in the existing CDM, offers a more stable and extensible solution and is therefore selected for implementation. This approach avoids the dynamic nature of periodically updating the enum list, which could induce additional complexities. Importantly, this modification has no adverse impact on the current CDM modules in use, thus mitigating transitional friction.

4.1.2 Recursive Type Structure

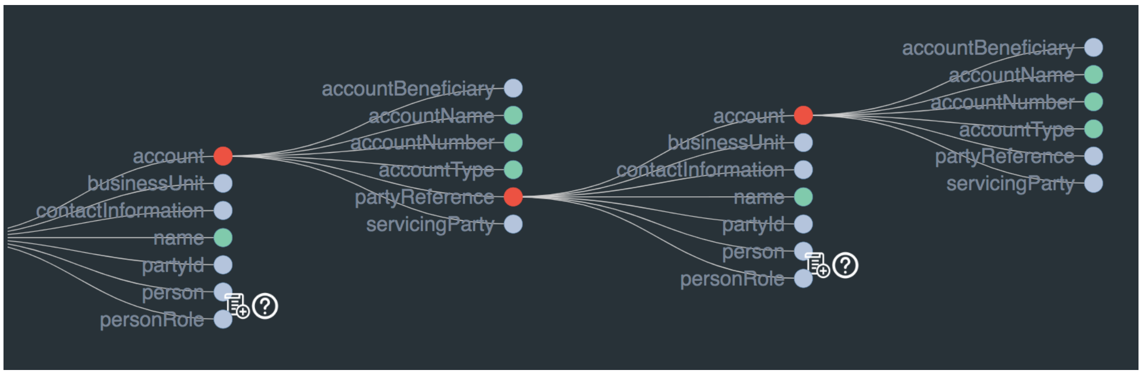


Figure 4.1: A directed tree structure displaying how the CDM data types loop back on themself in a recursive manner. The account has the sub–type of party reference, which has the sub–type of account. This must be dealt with when considering Smart Contract creation.

In the process of translating the Common Domain Model (CDM) to Solidity, a specific issue arises due to the recursive nature of some CDM types (shown in Figure 4.1). While this design is permissible in TypeScript, it becomes problematic in Solidity, which requires the initialization of all optional fields, even if to null values. This theoretically demands infinite storage space and an unbounded amount of Ethereum gas for initialization.

Several solutions exist to address this issue. One option is to flatten the recursive structures, and linearize the data model, although this would deviate from the original CDM architecture and make the CDM implementation redundant. Another option involves implementing a pointer–based system that tracks initialized instances of recursive types, thus eliminating the need for redundant storage but at a great cost of added complexity.

The best strategy entails inclusion of only those fields that are required for the downstream CDM modules. I remove types that do not contribute to the functionality of the specific implementation, this approach sidesteps the recursive issue without inflating storage or gas costs. For example, inside the Account type, if the accountNumber field is redundant from implementation, I cut it from the created module. This method further preserves the core integrity of the CDM while aligning with Solidity's constraints, making it a compromise between structure and computational efficiency.

4.1.3 Addition of Non–Rosetta Types

Solidity, unlike general–purpose programming languages such as Java and Python, is specialized for blockchain–specific use cases due to its Ethereum Virtual Machine compatibility. This limits its type system, necessitating unique types like 'address' which don't have direct analogues in standard programming languages. Consequently, types that would be optimally denoted as 'address' — such as 'LEI' — must be typed as 'string' to maintain compatibility.

To accommodate Ethereum–specific scenarios, fields like 'ethAddress' are introduced to specify entities like Buyers and Sellers within the Smart Contract environment. Additionally, cryptographic signing fields labeled 'signed' are incorporated into the types 'Sellr' and 'Buyr'. Although these fields could add complexity to the general CDM, their inclusion enhances Smart Contract compatibility.

Moreover, the term 'reference' — a commonly used type within CDM distributions — conflicts with a reserved keyword in Solidity. To resolve this, the type is renamed to 'ref'.

Operational enhancements are also integrated to bridge the gap between Solidity and the Rosetta CDM. Fields such as 'ethValuationTimeDate' are added to rectify the mismatch between Solidity's milliseconds–since–epoch time format and Rosetta CDM's string–represented dates. Additional operational fields, like 'priceSourceEthAddress' and 'currencyEthAddress', are introduced to facilitate functionalities such as Chainlink Oracle integration and base currency specification via Ethereum addresses.

4.1.4 Storage Inefficiency in CDM and its Impact on Smart Derivatives Contracts

The use of the Common Domain Model (CDM) introduces notable computational overhead due to its data type architecture. Specifically, when one looks at the definition of a Multiplier in a conventional Smart Contract, it is typically represented simply as an int256. This occupies a single storage slot on the blockchain, optimizing storage and computational resources.

Contrastingly, the CDM complicates this straightforward representation by requiring a hierarchical nesting of data types. In the CDM, a Multiplier isn't just an int256 — it becomes part of a broader construct referred to as CashSettlementAmount. This variable is, in turn, nested within another data structure called CashSettlementTerms. CashSettlementTerms relies on yet another data structure, Money, which itself is contingent on Measure, which has the desired field value. Only then do we reach a point where an int256 variable is finally declared and utilized for storing the multiplier's value.

Each of these nested data types requires its own storage slot on the blockchain, cumulatively summing up to five separate storage slots just to represent what could otherwise be stored in a single slot. This design choice exacerbates storage costs and imposes a computational burden when navigating through these layers of abstraction to access the multiplier value.

In the realm of smart derivatives contracts, where readability is a paramount concern, the structural inefficiencies inherent in the CDM's design could be detrimental. As these contracts escalate in complexity and nuance, the accumulated storage and computational costs associated with this intricate data structuring could impede both performance and affordability. Specifically, the escalating expenses related to contract deployment may become prohibitively high, undermining the feasibility of using blockchain technologies for these financial instruments.

Consequently, for the CDM to sustain long–term viability in the sphere of decentralized finance, two paths present themselves: either a reevaluation of its current architecture is undertaken, or a streamlined version must be adopted to mitigate such inefficiencies.

4.2 Stabilizing Cash Settlement Amounts in EVM Smart Contracts

In the Ethereum Virtual Machine (EVM), Smart Contracts are unable to self–execute or defer state changes. This constraint is attributable to the volatility of future gas expenditure and the dynamic nature of the blockchain. As such, the automation of on–chain digital asset transfers upon reaching a predefined valuation date necessitates third–party involvement. As an alternative, one has one of the two alternative design choices, that either:

1. One of the counterparties takes responsibility for initiating the agreed–upon cash settlement upon reaching the valuation date.

2. The architectural framework includes a provision that permits either party to 'lock in' or 'freeze' the cash settlement amount at any point after the valuation date.

Mathematically, one can let t_v be the valuation date, and T be the current time such that $T > t_v$ implies the valuation date has passed and $T < t_v$ indicates the valuation date lies in the future. Letting the Chainlink Price $P(t)$ be a function of time t, and the Forward Price F be a fixed value, then the Cash Settlement Amount $A(t)$ can be expressed as:

$$A(t) = P(t) - F, \quad t \geq t_v$$

I introduce a binary indicator variable α such that $\alpha = 1$ if the Cash Settlement Amount has been 'locked in' or 'frozen', and $\alpha = 0$ otherwise. This leads to the following piecewise function for the Forward Cash Settlement Amount $A^*(t)$:

$$A^*(t) = \begin{cases} P(t) - F & \text{if } \alpha = 0 \text{ and } t \geq t_v \\ A(t_{\text{freeze}}) & \text{if } \alpha = 1 \end{cases}$$

Where t_{freeze} is the time when α transitions from 0 to 1, indicating that the Cash Settlement Amount has been 'locked in' or 'frozen'.

This situation provides a dilemma. If a party encounters difficulties or neglects to 'freeze' the cash settlement amount, the Forward Cash Settlement Amount remains vulnerable to variations over time. Such instability introduces elements of risk and unpredictability that counter the foundational aims of financial derivatives structure in Smart Contracts. One may be able to outsource this risk to a third party agent, who is given permission to make the transfer or freeze at t_v or can take the risk on themself.

From a different perspective, this unpredictability is an inevitable feature, given that, within the current architecture of peer–to–peer and trustless Smart Contracts[2], no viable alternatives exist to avoid this limitation.

4.3 Architecture of a Comprehensive Decentralised Web Application for Smart Confirmation Contracts

To facilitate interaction with Smart Confirmation Contracts, I engineer a decentralized web application on the Sepolia EVM testnet blockchain. This platform's operation is only reliant on its connectivity to an Ethereum Virtual Machine–compatible blockchain. It supports the creation of a diverse range of financial derivatives, including but not limited to Put, Call, and Forward options. Users have the flexibility to define disruption parameters, aligning with the Digital Asset Definitions.

4.3.1 User Interface: Contract Template Selection and Customization

The web application's initiation page, shown in Figure 4.2, provides a user interface that is divided into two sections. The left pane serves as a repository of pre–configured Smart

[2]Upon loosening the requirement of trustlessness, one could invite an oracle to message the Smart Contract upon the valuation date — freezing at the correct time. Both parties must entirely trust this third party, and the additional risks and legal complications of the third party being ineffective in their task must be considered.

Contract templates. A future improvement would involve the incorporation of a modular assembly feature, extending upon the ISDA Create web application to enable users to construct custom contracts through the combination of different Logic Modules and natural language contracts. Once a template is chosen, the platform directs the user to an International Swaps and Derivatives Association (ISDA)–aligned interface for the customization of the Smart Confirmation Contract.

4.3.2 User Interface: ISDA–Compliant Input Form

The right pane of the application serves as an intuitive input interface. The layout is deliberately modelled to resemble traditional ISDA confirmation contracts, aimed at simplifying the onboarding process for legal professionals who may lack blockchain familiarity, reducing barriers to adoption.

4.3.3 Smart Contract Compilation and Transaction Initiation

Upon clicking the 'Create Contract' button, the application triggers two sequential operations: the compilation of the Smart Contract's bytecode within the Smart Confirmation Contracts factory[3], and the initiation of a corresponding blockchain transaction. This operation assumes the existence of a browser extension capable of Web3 interaction. The integration of a specialized Software Development Kit (SDK) is under consideration to further optimize this process — and allow for external scripts to interact with the protocol.

4.3.4 Contract Monitoring

For each Smart Confirmation Contract, a unique web page is dynamically generated and indexed via its blockchain address. For example, the contract written to the address *'0xBdC703C9459c3680f777335E72e05DbAb4718eB2'* would be found under */contract/0xBdC703C9459c3680f777335E72e05DbAb4718eB2*. Such a decision also enables users to access each contract via a QR code, which, if printed onto a natural language document allows for instant access to the smart confirmation contract that underpins it.

4.3.5 Contract Interaction

With a Web3 wallet [51], users can fully interact with the Logic Modules, to fulfill the transactional obligations in the agreement. Indeed, they can call any function, emit any event, and make any transaction.

[3] A Smart Contract Factory is a 'wrapper' contract around a Smart Contract, that when messaged produces a pre–specified Smart Contract from a template.

4.3.6 Event History and User Interactivity

The application provides a comprehensive history log that captures all emitted contract events, offering users insights into contract creation, signing, and any subsequent disruption events. If given permission in the General Terms, the participants can choose to raise the applicable events specified in the Logic Modules. These events are stored on the blockchain permanently, allowing for transparency for auditors who desire to see the transaction lifecycle of the contract.

4.3.7 AI–Powered Query Handling

The platform offers query capabilities by saving contract history, natural language contract texts, Smart Contract code, and transaction parameters in memory. This data reservoir enables the integration of large language models (LLM) to execute precise queries and offer descriptions of the financial derivatives in question. If the LLM is trained upon legal data, it can further provide AI–based legal advice for each participant, tailored completely with an entire knowledge of the ecosystem and the contract in question.

However, Language Models like LLMs are susceptible to generating hallucinatory information, and their utility in giving legal or financial advice remains limited due to their lack of credibility. Nevertheless, for rudimentary queries, such models can offer valuable insights.

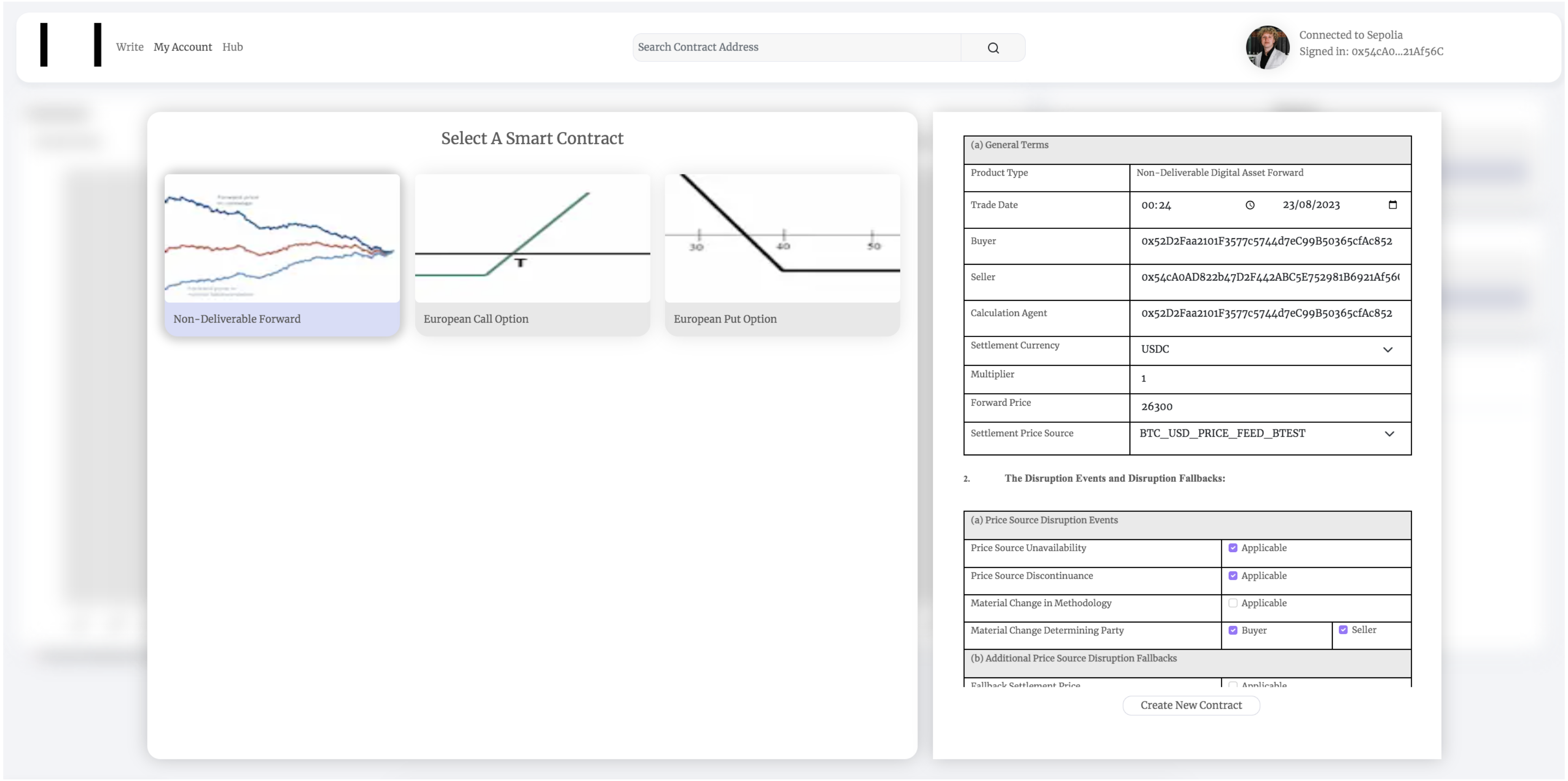
4.4 Areas for Enhancement

4.4.1 Optimization of Time–series Retrieval

Chainlink typically updates price feeds at 30–minute intervals, generating a substantial volume of data over extended periods. The retrieval of 200 rounds of such data necessitates a considerable number of HTTP requests to the associated Web3 RPC. This intensity, both computational and bandwidth–related, presents an opportunity for optimization. Algorithms to fetch and store data in a more efficient manner could be investigated to alleviate this bottleneck.

4.4.2 Event–based Data Retrieval Constraints

Direct user interaction with the blockchain for data retrieval poses specific challenges. Notably, events emitted by Smart Contracts are not stored within the contracts themselves. Instead, a burdensome process ensues, requiring traversal and filtering through each block to identify emitted events that correlate with the specified Smart Contract. This operation is computationally expensive and time–consuming. Employing an archive node could

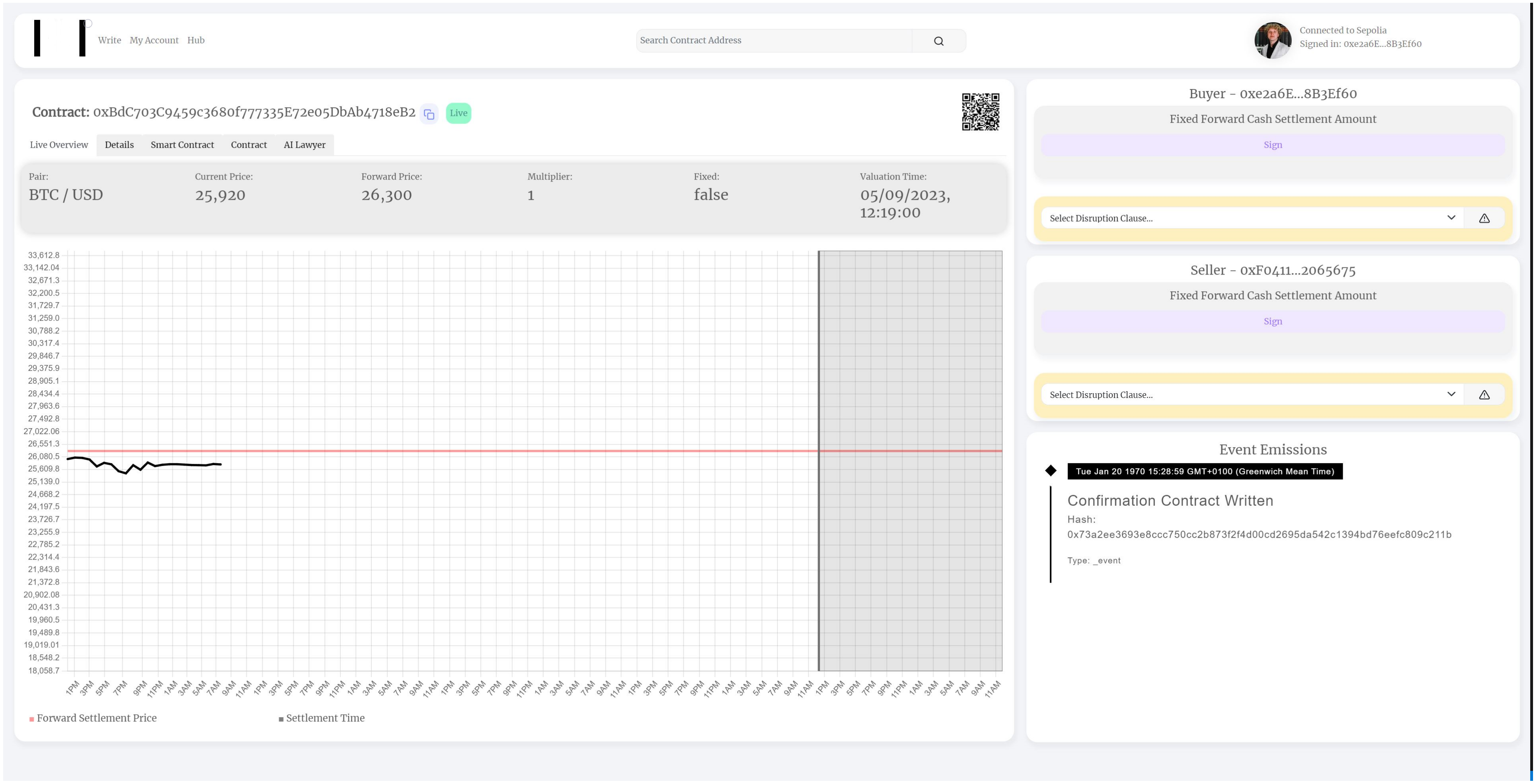


(a) General Terms	
Product Type	Non-Deliverable Digital Asset Forward
Trade Date	00:24 23/08/2023
Buyer	0x52D2Faa2101F3577c5744d7eC99B50365cfAc852
Seller	0x54cA0AD822b47D2F442ABC5E752981B6921Af56
Calculation Agent	0x52D2Faa2101F3577c5744d7eC99B50365cfAc852
Settlement Currency	USDC
Multiplier	1
Forward Price	26300
Settlement Price Source	BTC_USD_PRICE_FEED_BTEST

2. The Disruption Events and Disruption Fallbacks:

(a) Price Source Disruption Events		
Price Source Unavailability	✓ Applicable	
Price Source Discontinuance	✓ Applicable	
Material Change in Methodology	☐ Applicable	
Material Change Determining Party	✓ Buyer	✓ Seller
(b) Additional Price Source Disruption Fallbacks		
Fallback Settlement Price	☐ Applicable	

Figure 4.2: Smart Confirmation Contract creation webpage. On the left displays a panel to choose the Smart Confirmation Contract template that is desired. On the right displays the confirmation general terms and parameters, relevant to the creation of the Smart Confirmation Contract.


Write My Account Hub

Search Contract Address

Connected to Sepolia
Signed in: 0xe2a6E...8B3Ef60

Contract: 0xBdC703C9459c3680f777335E72e05DbAb4718eB2 Live

Live Overview | Details | Smart Contract | Contract | AI Lawyer

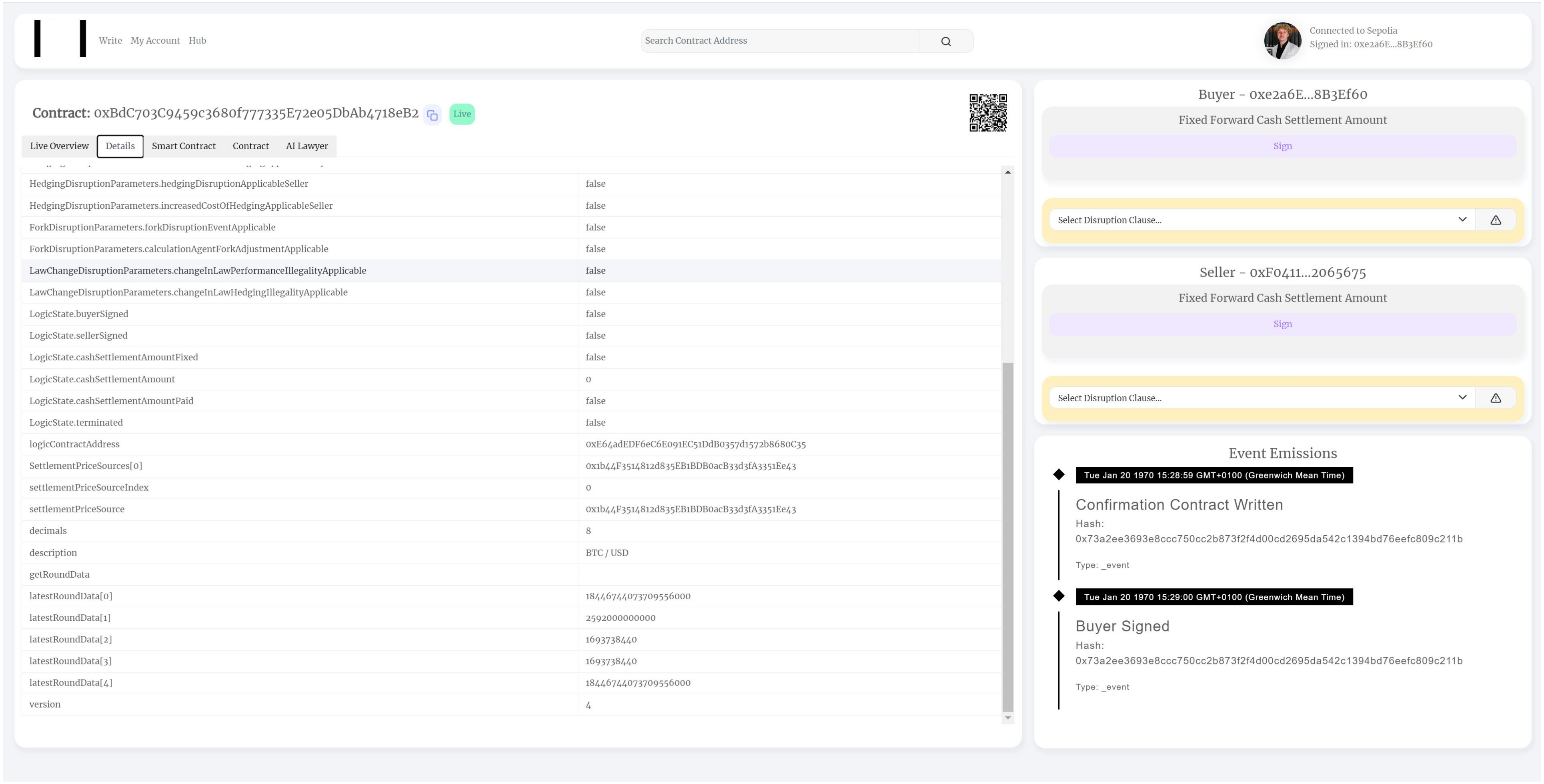
Pair:	Current Price:	Forward Price:	Multiplier:	Fixed:	Valuation Time:
BTC / USD	25,920	26,300	1	false	05/09/2023, 12:19:00

Buyer - 0xe2a6E...8B3Ef60
Fixed Forward Cash Settlement Amount
Sign
Select Disruption Clause...

Seller - 0xF0411...2065675
Fixed Forward Cash Settlement Amount
Sign
Select Disruption Clause...

Event Emissions
Tue Jan 20 1970 15:28:59 GMT+0100 (Greenwich Mean Time)
Confirmation Contract Written
Hash:
0x73a2ee3693e8ccc750cc2b873f2f4d00cd2695da542c1394bd76eefc809c211b
Type: _event

Figure 4.3: The Smart Confirmation Contract webpage presents real–time data and interactive functionalities for the contract's operation. The interface features a graphical representation of price time–series data, alongside actionable options that allow users to sign, freeze, and transfer assets. Additionally, a historical log of emitted events is displayed for comprehensive contract auditing. To uniquely identify and access this Smart Confirmation Contract, a QR code is strategically positioned in the upper right corner.



Contract: 0xBdC703C9459c3680f777335E72e05DbAb4718eB2

HedgingDisruptionParameters.hedgingDisruptionApplicableSeller	false
HedgingDisruptionParameters.increasedCostOfHedgingApplicableSeller	false
ForkDisruptionParameters.forkDisruptionEventApplicable	false
ForkDisruptionParameters.calculationAgentForkAdjustmentApplicable	false
LawChangeDisruptionParameters.changeInLawPerformanceIllegalityApplicable	false
LawChangeDisruptionParameters.changeInLawHedgingIllegalityApplicable	false
LogicState.buyerSigned	false
LogicState.sellerSigned	false
LogicState.cashSettlementAmountFixed	false
LogicState.cashSettlementAmount	0
LogicState.cashSettlementAmountPaid	false
LogicState.terminated	false
logicContractAddress	0xE64adEDF6eC6E091EC51DdB0357d1572b8680C35
SettlementPriceSources[0]	0x1b44F3514812d835EB1BDB0acB33d3fA3351Ee43
settlementPriceSourceIndex	0
settlementPriceSource	0x1b44F3514812d835EB1BDB0acB33d3fA3351Ee43
decimals	8
description	BTC / USD
getRoundData	
latestRoundData[0]	18446744073709556000
latestRoundData[1]	2592000000000
latestRoundData[2]	1693738440
latestRoundData[3]	1693738440
latestRoundData[4]	18446744073709556000
version	4

Buyer - 0xe2a6E...8B3Ef60

Fixed Forward Cash Settlement Amount

Sign

Select Disruption Clause...

Seller - 0xF0411...2065675

Fixed Forward Cash Settlement Amount

Sign

Select Disruption Clause...

Event Emissions

Tue Jan 20 1970 15:28:59 GMT+0100 (Greenwich Mean Time)

Confirmation Contract Written

Hash:
0x73a2ee3693e8ccc750cc2b873f2f4d00cd2695da542c1394bd76eefc809c211b

Type: _event

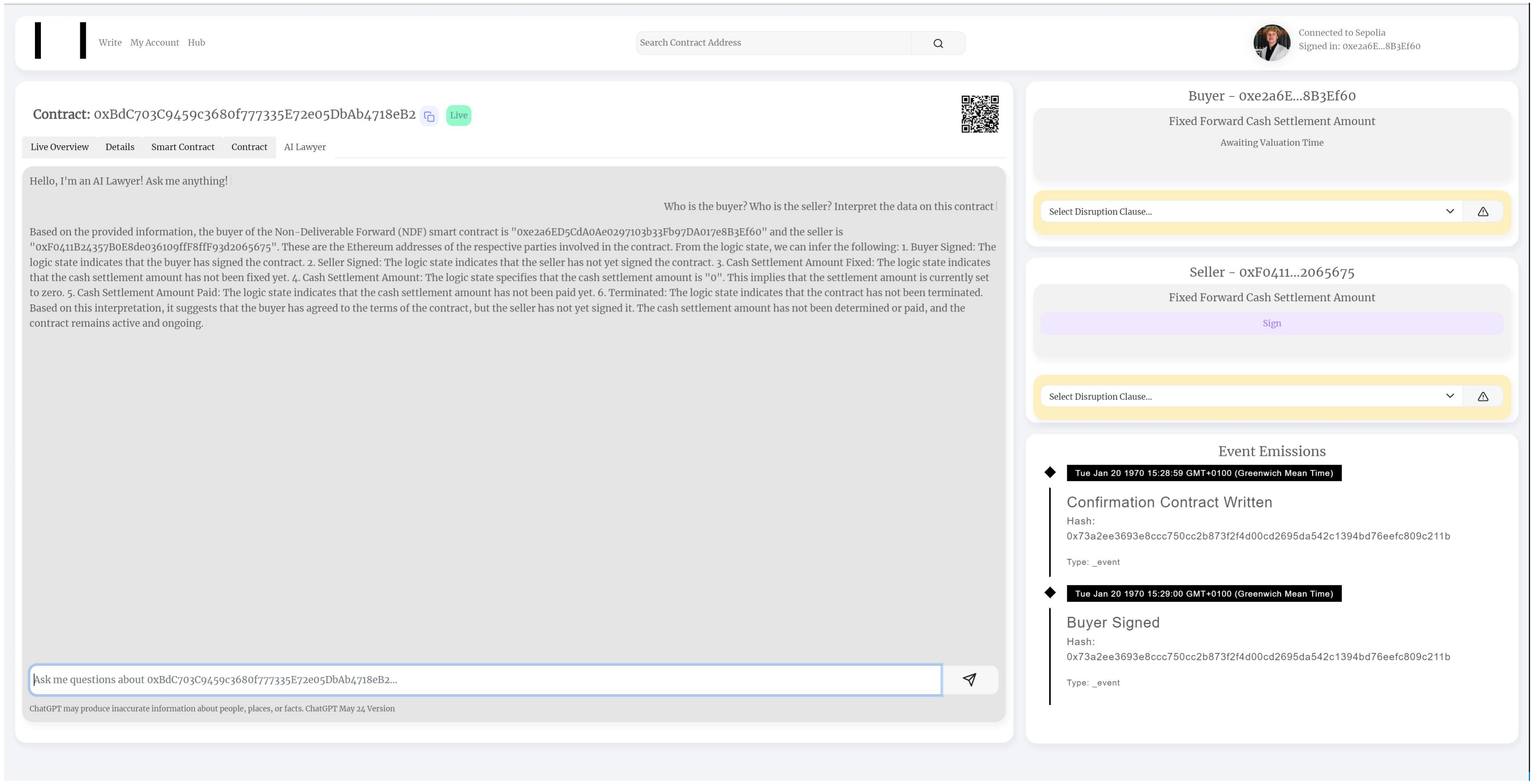
Tue Jan 20 1970 15:29:00 GMT+0100 (Greenwich Mean Time)

Buyer Signed

Hash:
0x73a2ee3693e8ccc750cc2b873f2f4d00cd2695da542c1394bd76eefc809c211b

Type: _event

Figure 4.4: This interface provides users with real–time insights into both the general terms and the logical state of the agreement. By invoking the blockchain's read methods at each block interval, the platform guarantees that the displayed values remain consistently updated and accurate.


Write My Account Hub
Search Contract Address
Connected to Sepolia
Signed in: 0xe2a6E...8B3Ef60
Live
Live Overview
Details
Smart Contract
Contract
AI Lawyer
Buyer - 0xe2a6E...8B3Ef60
Fixed Forward Cash Settlement Amount
Awaiting Valuation Time
Select Disruption Clause...
Seller - 0xF0411...2065675
Fixed Forward Cash Settlement Amount
Sign
Select Disruption Clause...
Event Emissions

Contract: 0xBdC703C9459c3680f777335E72e05DbAb4718eB2

Hello, I'm an AI Lawyer! Ask me anything!

Who is the buyer? Who is the seller? Interpret the data on this contract

Based on the provided information, the buyer of the Non-Deliverable Forward (NDF) smart contract is "0xe2a6ED5CdA0Ae0297103b33Fb97DA017e8B3Ef60" and the seller is "0xF0411B24357B0E8de036109fF8ffF93d2065675". These are the Ethereum addresses of the respective parties involved in the contract. From the logic state, we can infer the following: 1. Buyer Signed: The logic state indicates that the buyer has signed the contract. 2. Seller Signed: The logic state indicates that the seller has not yet signed the contract. 3. Cash Settlement Amount Fixed: The logic state indicates that the cash settlement amount has not been fixed yet. 4. Cash Settlement Amount: The logic state specifies that the cash settlement amount is "0". This implies that the settlement amount is currently set to zero. 5. Cash Settlement Amount Paid: The logic state indicates that the cash settlement amount has not been paid yet. 6. Terminated: The logic state indicates that the contract has not been terminated. Based on this interpretation, it suggests that the buyer has agreed to the terms of the contract, but the seller has not yet signed it. The cash settlement amount has not been determined or paid, and the contract remains active and ongoing.

Ask me questions about 0xBdC703C9459c3680f777335E72e05DbAb4718eB2...

ChatGPT may produce inaccurate information about people, places, or facts. ChatGPT May 24 Version

Tue Jan 20 1970 15:28:59 GMT+0100 (Greenwich Mean Time)

Confirmation Contract Written

Hash:
0x73a2ee3693e8ccc750cc2b873f2f4d00cd2695da542c1394bd76eefc809c211b

Type: _event

Tue Jan 20 1970 15:29:00 GMT+0100 (Greenwich Mean Time)

Buyer Signed

Hash:
0x73a2ee3693e8ccc750cc2b873f2f4d00cd2695da542c1394bd76eefc809c211b

Type: _event

Figure 4.5: This interface serves for real–time data queries to the Smart Confirmation Contract. Live data is extracted and transmitted to the AI chatbot's API, ensuring immediate and accurate information is explained.

expedite this process considerably, albeit at the cost of integrating an additional API into the production pipeline.

Moreover, it is feasible to architect an off–chain Oracle designed to autonomously trigger upon reaching a predetermined valuation date and fetch the corresponding price — though the design and implementation of such an Oracle are outside the scope of this dissertation.

4.4.3 Considerations for Natural Language Contract Storage

The current architecture of the web application elects to store natural language contracts on the server for client–side viewing. An alternative approach would involve storing these contracts directly on the Ethereum blockchain as byte data, allowing for client–side construction. However, Ethereum's current storage constraints — limiting each block to approximately 0.02 MB of data — render this solution financially and operationally impracticable. Future expansions in Ethereum's block size could mitigate this constraint.

4.4.4 General Improvements and Trustworthiness

Addressing the bottlenecks and limitations can create enhancements in the web application's performance, scalability, and user experience. Beyond these technical improvements, the implementation of auditing protocols for both the Smart Contracts and the web application's codebase is extremely important. Such audits not only identify vulnerabilities and inefficiencies but also contribute to the platform's credibility. The totality of these improvements would position the platform as a robust, trusted, and market–ready solution.

Chapter 5

Conclusion

5.1 Summary of Key Findings

This thesis presents a comprehensive architecture for Smart Confirmation Contracts, aimed at reengineering the contractual frameworks established by the International Swaps and Derivatives Association. The architecture is implemented as a fully functional Web3 decentralized application. It incorporates key ISDA components such as the Digital Asset Definitions, Common Domain Model, and the Master Agreement framework. The research also delves into the standardization of data types, functions, and the incorporation of 'internal' natural language contracts, thereby offering a robust and flexible solution for Smart Legal Contracts in the derivatives market.

5.2 Theoretical and Practical Implications

5.2.1 Theoretical Implications

The research raises theoretical questions, particularly concerning the hierarchical relationships within the Smart Confirmation Contract. For example, it prompts an inquiry into the precedence of Logic Module natural language explanations over those found in the Schedule or Product Definitions. Such an inquiry opens up a discourse on the necessity and sufficiency of natural language explanations within the Internal Model Logic Modules. Could these explanations alone provide the required contractual nuance for contractual obligations, thereby eliminating the need for external natural language documents altogether? One may question if the Master Agreement could be replaced in its entirety by Logic Modules.

5.2.2 Practical Implications

From a practical standpoint, the architecture facilitates a gradual transition to Smart Legal Contracts. It offers a modular approach, allowing market participants to selectively adopt Logic Modules based on their comfort level and the extent of standardization. This phased adoption approach minimizes the risks inherent in a complete transition to Smart Contracts. It promotes participation and fosters trust within the financial ecosystem. This strategy also allows for the observational learning of market participants, as late adopters can witness the successes and improvements experienced by their market peers before making the transition themself. In fact, it is plausible that small to medium-sized enterprises (SMEs) may be the early adopters of Smart Confirmation Contracts, preceding larger financial institutions in adoption.

Both the theoretical and practical implications of this research contribute to the ongoing discourse in computational finance, smart contracts, and decentralized technologies. They offer avenues for further exploration and refinement, setting the stage for future research endeavors.

5.3 Limitations and Caveats

The existing implementation has its shortcomings, notably the need to truncate the Common Domain Model to comply with the Ethereum Virtual Machine's storage constraints. An optimal solution would involve the development of a bespoke data type standard tailored for Smart Contracts, perhaps a CDM–Light, which would align with existing ERC standards. This could entail the standardization of even rudimentary types like price and buyer, restricting them to a single level of nesting for storage efficiency and readability.

In terms of the application itself, there are areas for enhancement, particularly in the retrieval of historical event emissions. Additionally, the architecture could benefit from a more robust pipeline that enables comprehensive customization of Logic Module stacks. Rigorous testing and typing of all NextJS components are required, with the required quality assurance standards for contemporary web applications.

5.4 Regulatory and Policy Implications

As of the current analysis, a notable recent policy recommendation comes from the DeFi Working Group within the International Organization of Securities Commissions (IOSCO), spearheaded by personnel from the U.S. Securities and Exchange Commission. The group proposes that in the foreseeable future,

> *"Further moves toward standardization across DeFi data sets and codebases could assist regulators in understanding and assessing DeFi arrangements and*

activities."[1][52].

Should such standardization materialize, Logic Modules emerge as a viable solution due to their self-contained architecture and print-friendly readability, enabling regulators to swiftly scrutinize the intrinsic functionalities of financial derivatives, accompanied by their textual clarifications to mitigate vagueness in their interpretation. Moreover, the adoption of a uniform data type model such as the Common Domain Model would ensure that DeFi datasets adhere to consistent variable naming conventions, streamlining regulatory oversight.

5.5 Final Remarks

In conclusion, this thesis has presented the groundwork for a transformative approach to financial derivatives contracts through the development and implementation of Smart Confirmation Contracts. By leveraging Web3 technologies and integrating key ISDA frameworks, the research has not only demonstrated the feasibility of such contracts but also addressed issues related to standardization, modularity, and regulatory compliance.

[1]Decentralized Finance (DeFi) refers to a blockchain-enabled financial infrastructure that obviates the need for intermediaries by employing smart contracts for peer-to-peer transactional activities.

Appendix A

CODE

A.1 Code listing

The code for this thesis is extensive — covering thousands of lines over multiple interconnected files. For this reason, I have attached two .zip compressed folders. The first, called ISDAPROTOCOL includes the Smart Confirmation Contract inside the NF folder — and further includes the logic modules inside the LOGICMODULES folder. The translation scripts of the CDM exist withing the cdm/typescript folder.

I have further attached a .zip file for the Web Application, called ISDAWEBAPP. The associated hooks, pages, and assets align with the NextJS standard. On request I can provide GitHub repository access to any code required.

A.2 Project Summary

- **Project Title:** Smart Confirmation Contracts: An Architecture for ISDA Smart Contracts
- **Project Topic:** ISDA Smart Contracts
- **Industrial Supervisor:** Ciaran McGonagle
- **Team Member:** Finn Casey Fierro, ucabfc2@ucl.ac.uk
- **Description:** Implementation of a Smart Contract Architecture for ISDA
- **Own Contribution:** Entirety of work

Bibliography

[1] International Swaps and Derivatives Association, “User’s guide to the 2002 isda master agreement,” 2003. [Online]. Available: https://www.rbccm.com/assets/rbccm/docs/legal/doddfrank/Documents/ISDALibrary/Users%20Guide%20to%20the%202002%20ISDA%20Master%20Agreement.pdf

[2] C. M. McNamara and A. Metrick, “The lehman brothers bankruptcy f: Introduction to the isda master agreement,” *Journal of Financial Crises*, vol. 1, no. 1, pp. 137–150, 2019. [Online]. Available: https://elischolar.library.yale.edu/journal-of-financial-crises/vol1/iss1/7

[3] “Blockchain technology in financial services: a comprehensive review of...” *Journal of Governance and Sustainability*, 2020. [Online]. Available: https://www.emerald.com/insight/content/doi/10.1108/JGOSS-07-2020-0039/full/html

[4] C. D. Clack, “Design discussion on the isda common domain model,” *arXiv preprint arXiv:1711.10964*, 2017, subjects: Software Engineering (cs.SE). [Online]. Available: https://doi.org/10.48550/arXiv.1711.10964

[5] “Chainlink 2.0: Next steps in the evolution of decentralized oracle networks,” 2020. [Online]. Available: https://research.chain.link/whitepaper-v2.pdf

[6] “Web3 building blocks: A primer for assembling a web3 toolkit,” 2021, discusses the role of Web3 providers like MetaMask and JSON RPC providers in connecting to the Ethereum network. [Online]. Available: https://www.hobbsco.de/blog/web3/web3-providers-explained

[7] D. Chaum, “Blind signatures for untraceable payments,” *Advances in Cryptology*, 1983. [Online]. Available: http://www.hit.bme.hu/~buttyan/courses/BMEVIHIM219/2009/Chaum.BlindSigForPayment.1982.PDF

[8] H. Finney. (2004) Reusable proofs of work. [Online]. Available: https://cryptome.org/rpow.htm

[9] W. Dai. (1998) B-money. [Online]. Available: http://www.weidai.com/bmoney.txt

[10] A. Back. (1997, March 28) hash cash postage implementation. Cypherpunks Mailing List. [Online]. Available: http://www.hashcash.org/papers/announce.txt

[11] R. S. Leslie Lamport and M. Pease, "The byzantine generals problem," *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 4, no. 3, pp. 228–234, 1982.

[12] S. Nakamoto, "Bitcoin: A peer-to-peer electronic cash system," *Working Paper*, 2008. [Online]. Available: https://bitcoinwhitepaper.co/

[13] N. I. of Standards and Technology, "Fips pub 180-4: Secure hash standard (shs)," National Institute of Standards and Technology, Tech. Rep., 2015. [Online]. Available: https://nvlpubs.nist.gov/nistpubs/FIPS/NIST.FIPS.180-4.pdf

[14] J. R. Douceur, "The sybil attack," in *International Workshop on Peer-to-Peer Systems.* Springer, 2002, pp. 251–260. [Online]. Available: https://www.microsoft.com/en-us/research/publication/the-sybil-attack/?from=http%3A%2F%2Fresearch.microsoft.com%2Fapps%2Fpubs%2Fdefault.aspx%3Fid%3D74220

[15] R. M. Lee, "A logic model for electronic contracting," *Decision Support Systems*, vol. 4, no. 1, pp. 27–44, 1988. [Online]. Available: https://www.sciencedirect.com/science/article/pii/0167923688900966

[16] V. Buterin. (2013) Ethereum: A next-generation smart contract and decentralized application platform. [Online]. Available: https://ethereum.org/en/whitepaper/

[17] N. Szabo. (1997) The idea of smart contracts. [Online]. Available: https://www.fon.hum.uva.nl/rob/Courses/InformationInSpeech/CDROM/Literature/LOTwinterschool2006/szabo.best.vwh.net/idea.html

[18] H. Hu and Y. Xu, "Scsguard: Deep scam detection for ethereum smart contracts," *arXiv preprint arXiv:2105.10426*, 2021. [Online]. Available: https://arxiv.org/abs/2105.10426

[19] J. Xu and B. Livshits, "The anatomy of a cryptocurrency Pump-and-Dump scheme," in *28th USENIX Security Symposium (USENIX Security 19).* Santa Clara, CA: USENIX Association, aug 2019, pp. 1609–1625. [Online]. Available: https://www.usenix.org/conference/usenixsecurity19/presentation/xu-jiahua

[20] CNBC. (2019) Wells fargo says working to fully restore system as outage spills into day 2. Accessed: 2023-09-07. [Online]. Available: https://www.cnbc.com/2019/02/08/wells-fargo-says-working-to-fully-restore-system-as-outage-spills-into-day-2.html

[21] G. Wood and N. Savers. (2021) Ethereum yellow paper. Accessed: 2023-09-07. [Online]. Available: https://github.com/ethereum/yellowpaper

[22] S. N. Khan, F. Loukil, C. Ghedira-Guegan, E. Benkhelifa, and A. Bani-Hani, "Blockchain smart contracts: Applications, challenges, and future trends," *Peer-to-Peer Networking and Applications*, vol. 14, pp. 2901–2925, 2021.

[23] R. Morrison, N. C. H. L. Mazey, and S. C. Wingreen, "The dao controversy: The case for a new species of corporate governance?" *Front. Blockchain*, vol. 3, 2020.

[24] T. M. Research. (2021) Arbitrum: Layer-2 scalability for defi protocols. Accessed: 2023-09-11. [Online]. Available: https://research.tokenmetrics.com/arbitrum-layer-2-scalability-for-defi-protocols-crypto-deep-dive/

[25] U. Team. (2021) Uniswap v3 whitepaper. Accessed: 2023-09-07. [Online]. Available: https://uniswap.org/whitepaper-v3.pdf

[26] Z. Zheng, S. Xie, H.-N. Dai, W. Chen, X. Chen, J. Weng, and M. Imran, "An overview on smart contracts: Challenges, advances and platforms," *Future Generation Computer Systems*, 2019. [Online]. Available: https://doi.org/10.48550/arXiv.1912.10370

[27] F. Vogelsteller and V. Buterin. (2015) Eip-20: Erc-20 token standard. Accessed: 2023-09-07. [Online]. Available: https://eips.ethereum.org/EIPS/eip-20

[28] W. Entriken, D. Shirley, J. Evans, and N. Sachs. (2018) Eip-721: Erc-721 non-fungible token standard. Accessed: 2023-09-09. [Online]. Available: https://eips.ethereum.org/EIPS/eip-721

[29] W. Radomski, A. Cooke, P. Castonguay, J. Therien, E. Binet, and R. Sandford. (2019) Eip-1155: Erc-1155 multi token standard. Accessed: 2023-09-09. [Online]. Available: https://eips.ethereum.org/EIPS/eip-1155

[30] A. Beniiche, "A study of blockchain oracles," *arXiv preprint arXiv:2004.07140*, 2020. [Online]. Available: https://doi.org/10.48550/arXiv.2004.07140

[31] I. Hacking, *The Social Construction of What?* Harvard University Press, 1999. [Online]. Available: http://www.jstor.org/stable/j.ctv1bzfp1z

[32] D. Avital, "The standard metre in paris," *Philosophical Investigations*, vol. 31, no. 4, pp. 318–339, 2008.

[33] L. Lessig, *The Laws of Cyberspace*, 1998, draft 3, Harvard Law School. [Online]. Available: https://cyber.harvard.edu/works/lessig/laws_cyberspace.pdf

[34] ——, *Code and Other Laws of Cyberspace.* Basic Books, 1999.

[35] C. D. Clack, V. A. Bakshi, and L. Braine, "Smart contract templates: foundations, design landscape and research directions," 2017.

[36] Ethereum Foundation. (2023) The merge. Accessed: 2023-09-11. [Online]. Available: https://ethereum.org/en/roadmap/merge/

[37] International Swaps and Derivatives Association and Linklaters, "Smart contracts and distributed ledger – a legal perspective," 2017. [Online]. Available: https://www.isda.org/a/6EKDE/smart-contracts-and-distributed-ledger-a-legal-perspective.pdf

[38] Deloitte and W. E. Forum, "Over the horizon: Blockchain and the future of financial infrastructure," 2023, contact: Leif Boegelein, lboegelein@deloitte.ch, +41 58 279 7340. [Online]. Available: https://www2.deloitte.com/ch/en/pages/risk/articles/over-the-horizon-blockchain-and-the-future-of-financial-infrastructure.html

[39] W. Bank, "Remittance prices worldwide: Issue n. 19, september 2016," September 2016, accessed: 2023-09-11. [Online]. Available: http://remittanceprices.worldbank.org

[40] International Federation of Accoutants, "2015 audit fee report," FEI Publication, Tech. Rep., 2015, survey results from 76 publicly-held companies, 92 U.S. privately-held companies, and 57 nonprofit organizations. FEI Publication Code: 2015-018. [Online]. Available: https://www.ifac.org/knowledge-gateway/contributing-global-economy/publications/audit-fees-survey-2022

[41] C. D. Clack and C. McGonagle, "Smart derivatives contracts: the isda master agreement and the automation of payments and deliveries," 2019.

[42] C. McGonagle, "Translations: creating legally effective smart derivatives contracts," *Butterworths Journal of International Banking and Financial Law*, vol. 36, no. 8, p. Page Range, 2023, accessed: 2023-09-07. [Online]. Available: https://www.lexisnexis.co.uk/blog/docs/default-source/loan-ranger-documents/jibfl_2021_vol36_issue08_sep_pp540-543.pdf?sfvrsn=652ef6df_2

[43] International Swaps and Derivatives Association, "Isda clause library – credit support documentation," https://www.isda.org/book/isda-clause-library-credit-support-documentation/, 2023.

[44] ——. (2021) Isda create overview 2021. Accessed: 2023-09-09. [Online]. Available: https://www.isda.org/a/8nVgE/ISDA-Create-Overview-2021.pdf

[45] ——, "What is the isda cdm?" 2019, accessed: 2023-09-08. [Online]. Available: https://assets.isda.org/media/c2565206/0b805a0d-pdf/

[46] International Capital Market Association. (2021, August) Cdm for repo and bonds: Factsheet for implementation. Accessed: 2023-09-08. [Online]. Available: https://www.icmagroup.org/assets/documents/Regulatory/FinTech/CDM-for-repo-and-bonds-factsheet-23-August-2021-2.pdf

[47] C. D. Clack, "Design discussion on the isda common domain model," 2018, accessed: 2023-09-07. [Online]. Available: https://arxiv.org/abs/1711.10964

[48] International Swaps and Derivatives Association. (2023) Isda digital asset derivatives definitions. Accessed: 2023-09-07. [Online]. Available: https://www.isda.org/book/isda-digital-asset-derivatives-definitions/#:~:text=The%20ISDA%20Digital%20Asset%20Derivatives,)%20or%20Ether%20(ETH)

[49] I. Allison, "Divisions in sam bankman-fried's crypto empire blur on his trading titan alameda's balance sheet," *CoinDesk*, 11 2022. [Online]. Available: https://www.coindesk.com/business/2022/11/02/divisions-in-sam-bankman-frieds-crypto-empire-blur-on-his-trading-titan-alamedas-balance-shee

[50] E. Napolitano, "The fall of celsius network: A timeline of the crypto lender's descent into insolvency," *CoinDesk*, July 2022. [Online]. Available: https://www.coindesk.com/markets/2022/07/15/the-fall-of-celsius-network-a-timeline-of-the-crypto-lenders-descent-into-insolvency/

[51] D. W. Allen and J. Potts, "Web3 toolkits: A user innovation theory of crypto development," *Journal of Open Innovation: Technology, Market, and Complexity*, vol. 9, no. 2, p. 100050, 2023. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S219985312300152X

[52] The Board of the International Organization of Securities Commissions, "Policy recommendations for decentralized finance (defi) consultation report," International Organization of Securities Commissions, Consultation Report, September 2023. [Online]. Available: https://www.iosco.org