

TOWARDS DECLARATIVE SMART CONTRACTS

by

Kevin Purnell

A THESIS SUBMITTED TO MACQUARIE UNIVERSITY

FOR THE DEGREE OF

MASTER OF RESEARCH

DEPARTMENT OF COMPUTING

OCTOBER 2019



MACQUARIE
University
SYDNEY • AUSTRALIA

Examiner's Copy

Declaration

I certify that the work in this thesis entitled TOWARDS DECLARATIVE SMART CONTRACTS has not previously been submitted for a degree nor has it been submitted as part of the requirements for a degree to any other university or institution other than Macquarie University. I also certify that the thesis is an original piece of research and it has been written by me. Any help and assistance that I have received in my research work and the preparation of the thesis itself have been appropriately acknowledged. In addition, I certify that all information sources and literature used are indicated in the thesis.

Kevin Purnell

Acknowledgements

I would like to thank my supervisor, Dr. Rolf Schwitter, for initial direction and access to his prior research, the many brainstorming discussions that helped clarify this problem and possible solutions, and his guidance and encouragement during thesis preparation.

I also appreciate Macquarie University allowing me to return and complete an honors year approved more than a decade ago which I had to turn down because of work pressures.

Further, I appreciate and thank the HDR coaches for guidance while writing this thesis.

Finally, I thank my partner Jill for her patience and support, and my dog Albert who kept dragging me from my study for regular walkies.

Abstract

With the exception of some well-funded industries, legal documents remain difficult and expensive to use, and prone to ambiguities. Emerging blockchain technologies hold the promise of changing this, however the tools for coding these ‘smart contracts’ require programmers and are prone to fraud. To fully realise the benefits of smart contracts, widespread adoption is required, which depends on improving security and replacing programmers with tools that lawyers, business-people and the general public can use. Our objective is to investigate improved approaches to the creation, testing and deployment of smart contracts by demonstrating that pure declarative languages can be used, and that these facilitate achieving improved utility in smart contracts. Our investigation implemented a ‘Will and Testament’ as a smart contract on a custom simulator, and demonstrated improved utility by auto-generating a smart contract from a status-quo user interface with an untrained user. We found a number of small benefits to using a declarative language like simplification, ease of code auto-generation and ease of testing. We have identified an approach to smart contract creation supportive of adoption because conversion starts with current legal contacts, is tolerant of varying levels of automation, and allows human-in-the-loop interaction. Smart contracts are seen as game changing by many, and should issues with cost, usability and security be solved, the economic impact is likely to be large.

Contents

Declaration	ii
Acknowledgements	iii
Abstract	iv
Contents	v
Terms and Abbreviations	vii
Chapter 1 Introduction	1
1.1 Study Context	1
1.2 Study Objective	2
1.3 Study Overview	2
Chapter 2 Background	3
2.1 Introduction	3
2.2 Legal Contracts	3
2.3 Distributed Shared Ledgers	4
2.4 Blockchains	4
2.5 Smart Contracts	4
2.6 Wallets	5
2.7 Ricardian Contracts	5
2.8 Traditional Electronic Contracts	6
Chapter 3 Literature Review	8
3.1 Overview	8
3.2 Different Programming Languages	8
3.3 Visual Approaches	10
3.4 Auto-generation Approaches	12
3.5 Formal Verification Approaches	13
3.6 Conclusion	14
Chapter 4 Method of Investigation	16
Chapter 5 Design	17
5.1 Introduction	17
5.2 User Scenario	18
5.3 Improving Utility	18
5.4 Translation to Executable Code	20
5.5 Designing a Simulator	21
Chapter 6 Implementation	24
6.1 Constructing the Simulator	24
6.2 Description of the Simulator	26
6.3 Description of Templates	28

6.4	Low Complexity – ‘Will and Testament’ Contract	34
6.5	Comparison with a Solidity Implementation	39
6.6	Mid Complexity – ‘Real Estate Sale’ Contract.....	43
6.7	Complex – ‘CEO Employment’ Contract.....	45
Chapter 7	Results	47
7.1	Coding a Smart Contract with ASP is possible	47
7.2	Extensibility	47
7.3	Using ASP facilitates achieving improved utility in Smart Contracts	48
7.4	Summary	50
Chapter 8	Discussion.....	51
8.1	Key Findings.....	51
8.2	Limitations.....	51
8.3	Implications	51
8.4	Future Research	52
8.5	Conclusion	54
References.....		55
Appendix A	– Simple ‘Will’ – Solidity vs ASP.....	62
Appendix B	– ‘Will and Testament’ Template	63
Appendix C	– Low Complexity – ‘Will and Testament’	66
Appendix D	– Mid Complexity – ‘Real Estate Sale’	72

Terms and Abbreviations

51% attack	blockchain attack where 51% of computing power is with attacker (bitcoin 2019)
ABI	Application Binary Interface, an low level API for Ethereum smart contracts
API	Application Programming Interface, a standardised way to interface to a program
ASP	Answer Set Programming, a successful declarative programming language
blockchain	immutable datastore where address is hash of content, also stored on next block
declarative code	programming code with only logic statements (e.g. ASP)
DApps	Distributed Applications
DSL	Distributed Shared Ledgers
DSLing	Domain Specific Language, a small language written to address a specific need
Entity	a legal person or organisation (the legal, not computing definition)
Ethereum	the blockchain used as the foundation for this study
GUI	Graphical User Interface
imperative code	typical programming code with both logic and control statements (e.g. C++)
NAF	Negation as Failure
performance	legal contract 'performance' is the transfer of the thing of value for consideration
PII	Personally Identifiable Information (NIST 2019)
SCE	Smart Contract Editor
SFLC	Standard Form Legal Contract, a standard contract template only requiring details
SIE	Smart Instantiation Editor
smart contract	a distributed application stored in a blockchain and automatically executed
Solidity	a JavaScript like programming language used in Ethereum to write smart contracts
utility	the usefulness of something, especially in a practical way (Cambridge Dictionary 2019)

Chapter 1 Introduction

1.1 Study Context

Imagine a world where most legal contracts are digital, and thus more useful and automatable. Recent blockchain technologies promise an immutable distributed platform, making the idea of legally enforceable digital contracts (“Smart Contracts”) that execute automatically over time seem more achievable (Szabo 1994; Wood 2017). Blockchains (Nakamoto 2008) are recent innovations in shared databases that promise immutability for data by chaining blocks of data together with addresses derived from hashing the content of each block. As this hashed address is also added to the next block, any change will cause a detectable break and any attempt to modify the next block results in cascading changes. This makes undetectable change difficult, and change detection easy. The first blockchain implemented bitcoin, but there is no reason to limit content to financial transactions (Wood 2017).

‘Smart Contracts’ (hereafter smart contracts) replace financial transfers with code that when executed enforces predefined activities in response to external events. At face value, smart contracts provide considerable opportunity for increased automation; however, many problems require solving before this concept becomes practical. For example; the first generation of smart contracts proposed in 2014 (Buterin 2014) requires programmers to code smart contracts in a JavaScript-like language, an expensive overhead since end-users are lawyers, business-people and the general public. Smart contracts are more than the electronic agreements and automation used for decades by the financial services and e-commerce sectors (Glantz 2013). Smart contracts offer a way to deliver a generic low cost, trustworthy platform to the general public that enables and enforces legal agreements. As such, smart contracts dramatically expand the potential for use of digital agreements, and have gained attention from the legal community (Ryan 2018).

All current methods of encoding legal documents are problematic when weighed against cost, usability and reliability. Paper contracts risk misplacement or modification, are often hard to understand, and suffer ‘difficult to detect’ ambiguities. Further, use of the legal system is notoriously expensive and time-consuming, favouring those with access to greater resources (Randall 2009); a disincentive to use. Current electronic contract implementations in areas like financial markets and e-commerce are invariably private systems, built at great expense by one or more of the parties involved, and not flexible enough to be applied to other business domains at reasonable cost. Further, not being architected from the start with security and resilience to attack in mind, these systems are as vulnerable as any traditional system deployed on private networks (Hernandez 2018). Finally, smart contracts as currently implemented in systems like Ethereum introduce their own unique problems. Smart contracts generally can’t be coded by end users (lawyers, business-people and the general public), are difficult to program, are vulnerable to hacks (Falkon 2017), compound the comprehensibility problem, do not address the ambiguity problem, and remain difficult to understand once deployed. The imperative language used (Solidity) has limited expressive power and requires many lines of code to express the logic of complex legal contracts, which is reflected in costs to create and debug contracts (Swamy 2018).

1.2 Study Objective

This study investigates improved approaches to the creation, testing and ongoing life-cycle of smart contracts, with the goal of demonstrating that encoding of legal logic in a pure declarative language is possible and facilitates improved utility to Ethereum's current implementation.

Ethereum's approach encodes legal logic in procedural 'Solidity' which is compiled to bytecode that executes on the Ethereum Virtual Machine (EVM) (Wood 2017). At the time of writing, the Ethereum blockchain environment is possibly the best benchmark for our project as other possibly more advanced environments provide little publicly available information. We make no attempt to address current problems faced by blockchains like security, privacy, processing and storage demands and optimal partitioning of content; nor do we investigate future potential as exemplified by the Decentralized Autonomous Organisation (Chohan 2017).

We investigate possible solutions to the challenges listed in section 1.1 via a literature review and by experimenting with a custom blockchain simulator. Firstly, we design an approach to smart contract creation that has clear advantages over Solidity when evaluated subjectively from the perspective of the ultimate end-user (lawyers, business-people, general public). These preliminary evaluation criteria cover aspects of utility for these end-users, being; 1) ease of use (desirable that smart contract creation not require training); 2) understandability (builds confidence the contract does what is intended); 3) ease of testing (helps identify and remove bugs); 4) free of security exploits and errors at deployment (self-evident); 5) scalability (can handle large complex contracts); 6) affordability (costs should be minimised). Secondly, we investigate how using a declarative language facilitates implementing this approach by building a blockchain simulator and using it to investigate how best to implement a suitable legal document as a declarative smart contract. The hypothesis we investigate and verify is:

"For the subset of legal documents amenable to implementation as smart contracts, coding with a declarative language is possible and facilitates achieving improved utility for those smart contracts."

To verify this hypothesis we seek: 1) to demonstrate that a legal document can be implemented with declarative code, and 2) evidence that using declarative code facilitates the implementation of a smart contract approach with improved utility over the Solidity approach.

Because of the ubiquity of legal document and contract use in society, even partial realisation of the above aims will likely generate significant economic value (Forbes 2018; McKinsey 2018).

1.3 Study Overview

The rest of this thesis is structured as follows:

Chapter 2 an overview of foundational concepts and technologies of declarative smart contracts.

Chapter 3 describes current research into smart contracts and evaluates alternative approaches.

Chapter 4 describes the method we use to investigate our hypothesis.

Chapter 5 describes the design of our experimental environment and the legal document used.

Chapter 6 details the blockchain simulator developed to investigate and verify our hypothesis.

Chapter 7 presents and evaluates our results and findings.

Chapter 8 discusses these results and findings and lists our insights into factors defining the boundaries of applicability for our approach and promising areas for future research.

Chapter 2 Background

2.1 Introduction

This section provides an overview of the foundational concepts and technologies assumed by our investigation and is provided only because smart contracts are a very recent development.

2.2 Legal Contracts

Legal contracts are the essential mechanism covering exchange of goods and services in our society. They have a long history in western society (Nicholas 1962), having their genesis in the first attempts to regulate behaviour in exchanges of any type. Reflecting this is the recognition by law of verbal agreements as having legal force if they can be proven to exist. This is easier to do than is generally believed, because even the purchase of fast food qualifies as a legal contract.

For a legal contract to exist, four conditions must be satisfied (Treasury 2019). There must be:

1. an 'Offer' of a good or service
2. an 'Acceptance' of that good or service
3. 'Intention' that the above agreement can be enforced by law
4. 'Consideration' or reward given in exchange for the above good or service

Clearly, this definition is applicable to a wide spectrum of exchanges, ranging from trivial to very complex, with their recording being driven by the tension between convenience and complexity. Because such exchanges can occur in both private and public settings, the mechanisms used can be extended to areas like process management within companies; for example, one department may agree a service level agreement with another that provides parts. Not all legal contracts are amenable to conversion to smart contracts; firstly, a contract must use a form of legal logic that is computable with current technologies; secondly, the 'Offer' must be tangible enough to benefit from some form of electronic process (see section 2.8). Further, some legal documents (Wills) do not involve 'Acceptance' or 'Consideration' but have features that can benefit from automation.

The following is a list of types of legal reasoning surfaced to date in literature (Ellsworth 2005; Fruehwald 2011), and is somewhat self-explanatory as to why only deductive reasoning is easily handled by computer: 1) Deductive Reasoning – reasoning over facts with rules, easy to encode; 2) Inductive Reasoning – reasoning from specific cases to formulate an applicable legal principle; 3) Reasoning by Analogy – reasoning based on similarity to previously decided cases (precedent); 4) Abductive Reasoning – seeking the simplest, most likely cause of known facts (hypothesis); 5) Reasoning by Principle—uses policy, custom, principles (common values) to decide outcome. Current computing techniques have mixed success with the last 4 of these 5 types, and consequently are outside the scope of this study. Further, the literature reveals that ongoing research to understand and encode 'legal reasoning' encounters difficulties with areas like obligation and duty (deontic logic) (McNamara 2010), and where circumstances override some agreement (defeasible logic) (Koons 2017).

Restricting application to legal documents that use purely deductive reasoning still greatly increases the types of legal document that can be automated with smart contract technologies, but a taxonomy may be useful when investigating the boundaries of applicability.

2.3 Distributed Shared Ledgers

The common purpose behind ledgers is the desire for a trusted source of truth. The most familiar use of the term ledger is “the principal book in which commercial transactions of a company are recorded” (CED 2014). A variation is the shared ledger, typically used by a community as the source of truth to track ownership of assets, an example being the Torrens Title Register of land ownership (NSW LRS 2019). Another variation is the distributed ledger, a type of shared ledger replicated and synchronised between members of a network (Walport 2016), first used in ancient Rome (Smith 1875), and by Yap islanders to track wealth (Fitzpatrick 2019). Finally, there is the distinction between un-permissioned ledgers and permissioned ledgers with one or more owners.

The wave of innovation known as the third industrial or digital revolution has been applied to ledgers and has provided many improvements like automation, but failed to fully mitigate risks from fraud (Davis 2015). The advent of blockchain technologies provides a potentially immutable datastore and has captured the imagination of the business community, sparking a wave of initiatives like the Australian Securities Exchange’s Clearing House Electronic Subregister System (CHES) replacement, a system that records shareholdings and manages settlements (ASX 2019).

2.4 Blockchains

Blockchain technology has roots in a string of innovations that used cryptography to create digital cash. These started with David Chaum’s secure ‘digital cash’ proposed in 1982 (Chaum 1982). Blockchain as an approach to data integrity was first described in January 1991 (Haber and Stornetta 1991). This paper proposed a way to make changes to document timestamps infeasible, without revealing content. The key ideas are:

- keeping an electronic record (ledger) of documents, with links between documents
- each link matches the previous document hash, so any change breaks this link
- accessible copies of the ledger are distributed across a network
- change is harder than detection, so theoretically defenders have the advantage and truth can be determined from the majority of ledgers

In 1992 Hash Trees (Merkle Trees) were introduced to improve scalability by grouping documents into blocks, with individual document hashes stored in trees (Bayer 1992). Further work showed how to implement SUNDR, a trusted multi-user network file system on untrusted servers by focusing on detection of attacks (Mazières and Shasha 2002), providing the mechanism for blockchains to be implemented on the internet.

These ideas were materialised between 2008 and 2015, spurred by a 2008 white paper (Nakamoto 2008) describing a type of digital currency and funds transfer mechanism (Bitcoin) which operates independently of a central bank (live January 2009). An illustration of the momentum achieved by blockchains is the Australian government’s Australian National Blockchain initiative launched in Aug 2018, a partnership between two large legal firms, IBM and Data61 (Ryan 2018).

2.5 Smart Contracts

The content stored on blockchains need not be restricted to bitcoin or coin balance ledgers; the ledger concept can store any asset record, and even computer code.

Nick Szabo, a legal scholar and computer scientist, proposed the concept of smart contracts in a 1994 paper (Szabo 1994). Szabo’s concept of a smart contract intended to bring the experience of

contract law and practice to electronic commerce protocols. A paper by (Buterin 2014) provided the impetus for introducing an improved scripting language for Ethereum (live July 2015), arguing that improved scripting was ideal for implementing ‘distributed applications’ (DApps). The realization that DApps and Szabo’s smart contracts were essentially the same thing spurred industry interest, and the creation of the Enterprise Ethereum Alliance (EEA) 2017.

Ethereum smart contracts are executable code meant to duplicate the intended tangible functionality of legal documents with the advantages of automated management and processing. To facilitate coding smart contracts in Ethereum, a Java-like imperative language called Solidity is provided. Solidity compiles to ‘Ethereum bytecode’ executed on the Ethereum Virtual Machine by miners financially incentivised to perform this function. These transactions can be ‘Ether’ transfers, smart contracts or transactions automatically generated by smart contracts (Wood 2017).

2.6 Wallets

Wallets allow blockchain users to store cryptocurrencies and interact with smart contracts. As this study proposes modifications to Ethereum, Ethereum wallets are described (Ethereum 2019). Wallets have evolved considerably so that a wide variety of forms and features are available; however, the most useful ideas for this study are ‘Hardware Wallets’ and ‘Hot/Cold Wallets’. A cold wallet is one not connected to the internet, so any attempted hack cannot complete. A hardware wallet is a USB stick type device that can be connected to a computer or phone to authorise interactions. When unattached a hardware wallet is cold, but when attached, a hardware wallet has the advantage of not exposing the owners private key to the internet. Ethereum’s smart wallets offer further security and recovery features over Ethereum wallets; examples being Gnosis Safe, Argent and Authereum (Ethereum SCW 2019).

This study proposes further enhancements to Ethereum smart wallets beyond storage of cryptocurrencies. It considers personal information to have similar value to cryptocurrency and requiring the same level of security. A proposal to store personal information in a bitcoin like wallet was first mentioned by Jäger (Jäger 2013); however, rather than discussing the designs provided by Jäger and subsequent collaborators (Kramer 2015), an idealised design that suits our application is described. There is intuitive appeal to the idea of holding personal information offline and controlling how it is accessed. Connection to a wallet can be as simple as entering a private key written on paper into the appropriate interface; however, this is not practical for manipulating personal information and consequently we focus on hardware wallets for this purpose. Hardware wallets can hold information that supports automatic population of user screens, for example, a contract between natural persons refers to personal information such as name, date of birth and address, commonly called Personally Identifiable Information (PII) (NIST 2019). Hardware wallets have advantages for storing PII and authorising its use in smart contracts via private key.

2.7 Ricardian Contracts

Ricardian Contracts were developed in 1996 for a Bond Trading System by Ian Grigg of Systemics Inc. (Grigg 2004), a London Fintech company, with the motivation being to automate transfer of financial instruments and money between accounts in a high trust environment.

A Ricardian Contract can be defined as a single document that is a) a contract offered by an issuer to holders, b) for a valuable right held by holders, and managed by the issuer, c) easily readable by

people (like a paper contract), d) readable by programs, e) digitally signed, f) carries keys and server information, and g) allied with a unique and secure identifier (Grigg 2004).

Simply, a Ricardian Contract is a document that is both readable by people (formatted as a text file) and parsable by programs that can convert it into internal forms for execution. It includes a special section for each type of contract, such as bond, share, currency, and descriptions in program-parsable terms of usage of decimal points, titles, and symbols.

Ricardian Contracts are designed to operate within a closed system and were originally implemented more than a decade before blockchain ideas surfaced. They inform smart contract development because; 1) they have already been implemented, and have achieved legal status in financial markets, 2) they illustrate what content a blockchain smart contract must store, and identify some principles.

2.8 Traditional Electronic Contracts

Currently many legal documents are electronic but retain a paper based format (Thomson Reuters 2020) with exceptions in a number of domains where the benefits from greater automation are overwhelming; for example, financial markets and e-commerce (B2* hereafter used to denote B2B, B2C, B2G etc. exchanges). These systems have been deployed for decades (Investopedia 2019); for example, online shopping. Reasons why other types of contracts have not been automated is worth analysis, with cost often dominating. Banks and businesses involved in B2* have the resources to create and maintain a central system providing functionality and security for users. Many ordinary legal contracts are between parties with few resources, while more complex contracts are often low volume and more easily handled in a paper based format. Compared to paper contracts, electronic contracts have clear advantages in areas like retrieval and automation, but a major disadvantage has been amplification of risks from fraud. Blockchain technologies potentially provide solutions to all above barriers, because they; 1) deliver a potentially immutable electronic platform, 2) are distributed, so can provide a public resource free of distortion from any controlling party, 3) amplify the resources available for creation of central services in most legal domains.

TABLE 1 – A SAMPLE OF LEGAL CONTRACT 'PERFORMANCE' TYPES BY 'DIFFICULTY TO IMPLEMENT'

Difficulty	Type of 'Performance'	Example(s)
1 simple	Monetary Transfer	Transfer \$x from A to B at 1:00 EST, 11Aug2019
2	Asset Transfer (via electronic ledger)	Transfer ownership of asset X from A to B at 1:00 EST, 11Aug2019
3	External Input (Event)	a) A died at 2pm:EST, 12Aug2019 b) I received my new dishwasher 11am Tuesday 20Aug2019
4	Human in the Loop	Only execute X if A and B agree
5	Dispute Resolution	If A and B do not agree, then C decides
6	Physical Asset Transfer	Change owner of Real Estate on the electronic ledger and perform a physical handover of keys etc.
7	Meta-Clauses	If any clause of this contract is held to be unenforceable, it will remain in effect to the extent that it is not invalid or unenforceable
8 complex	Complex Legal Concepts	A will use reasonable endeavours to perform X

At minimum these developments mean further penetration of electronic contracts into other legal domains but determined by factors that can be studied; like how difficult it is to automate the

‘performance’ of a contract (Australian Contract Law 2010) and what benefits then accrue. A brief survey provides a preliminary list of ‘performance’ types (Table 1).

Transactions within financial markets typically involve money transfer, the transfer of financial assets via electronic ledger and recording currency and interest rate daily changes, types 1 to 3 in Table 1. The simplicity of ‘performance’ required by the contract allows this automation, and consequently these markets have achieved a very high level of automation, with automated accounting systems used since the 1970s, and automated trading systems since the mid-1980s and by 2008 performing more than 50% of trades on American exchanges (Glantz 2013). These systems currently use artificial intelligence techniques to automatically identify and trade opportunities.

Blockchain initiatives have gathered a lot of momentum in financial markets and supply chain circles in the last few years in recognition of advantages like security over traditional systems. Professional service groups like Gartner, Accenture, McKinsey and R3 are popularising the trend (Panetta 2019), with Accenture estimating savings of 30% in Banking (Accenture 2017), McKinsey suggesting blockchains’ strategic value currently is cost reduction (McKinsey 2018), and Fintech companies like R3 (Fortune 2018) creating computing ecosystems for the creation and management of private blockchains, and facilitating consortiums of users who collaborate to create private blockchains that transact a specified range of financial instruments.

Chapter 3 Literature Review

3.1 Overview

Research into improving smart contracts began with the implementation of Ethereum in 2015 and has since intensified. Alternative research directions include different programming languages, visual approaches, automatic code generation, and attempts at formal verification. Each of these directions is reviewed and evaluated against criteria consistent with our objective of improving the utility of smart contracts from the perspective of the ultimate end-user.

This review has two goals; 1) identifying the state-of-the-art regarding the use of pure declarative languages to code smart contracts, 2) identifying smart contract approaches that dramatically improve the utility of smart contracts relative to Solidity. The first search focuses on identifying research gaps and opportunities to investigate the use of pure declarative languages for smart contracts. The second search is used to inform the design of the blockchain simulator used to investigate how to implement these declarative smart contracts.

An example of the clear improvement to utility over Solidity that we seek, is an approach that auto-generates smart contract code from a status-quo user interface. The increased utility stems from the dramatic reduction in costs for end-users, gained by removing programmers.

3.2 Different Programming Languages

3.2.1 Current Research

Ethereum's language for coding smart contracts, Solidity, is rapidly evolving in response to perceived and actual weaknesses evidenced by considerable losses to hacks (Falkon 2017). Analysis suggests the famous DAO hack (on an organisation called 'The DAO') was possible because Solidity was inadequate for writing secure, bug free software (Siner 2016). For example; the DAO hack was due to Solidity allowing implicit recursive calls, a feature avoided by secure language designers.

An alternative approach could be to investigate language theory to identify languages or language paradigms better suited to smart contracts. Early on, Solidity was described as a JavaScript like language (Solidity. 2017), but later releases claim influence from C++ and Python (Solidity 2019). The C language is the outstanding example of language design impacting bug rates and security vulnerabilities in programs (Ray, et al. 2014), with another being complexity (Zatko 2011). Research has shown it is difficult to establish more precise relationships (Ray 2014; Berger 2019), but it still follows that advantages can be found in approaches that yield fewer lines of code for a given result, and that design for correctness and security (Prowell 2005; Bickford 2008) helps. Another useful approach is using pretested code libraries (OpenZeppelin 2019).

Current popular programming languages like Python and functional programming also have associated smart contract research efforts. Vyper is a contract-oriented, pythonic programming language that like Solidity complies to a bytecode that targets the Ethereum Virtual Machine. It strives to be simple, secure and auditable, given most users have little prior programming experience (Vyper 2019); however, it still requires programming, and in this regard does not provide the dramatic improvements to utility over Solidity that we seek.

Idris is a pure functional language with dependent types first released in August 2017 by Edwin Brady (Idris 2019). It is based on Haskell and ML but can be used as a proof assistant like Coq (Inria 2019). It complies to C and JavaScript upon which Solidity is based. Benefits of using Idris accrue from the safety created by its type system, and the reduced cost of formal verification afforded by being able to attempt formal proofs with the actual programming language. It is likely that Idris improves code reliability, however because programmers are still involved it does not provide the dramatic improvements to utility over Solidity that we seek.

Another interesting approach is the development of Ergo, a domain specific language by the Accord Project; a collaborative initiative aimed at developing an ecosystem and tools specifically for smart legal contracts (Accord Project 2019). Ergo is aimed at capturing the execution logic of legal contracts, with the language being developed in Coq, an approach easing eventual formal verification. The documentation discusses legal-tech developers, so a reasonable conclusion is that objectives do not currently include allowing untrained people to write smart contracts. Domain specific language approaches hold the promise of producing a declarative language specifically designed for smart contracts. Activity in the financial domain is summarised in (DSLFIN 2019).

Prestwich discusses features being added to Solidity, like ‘Function Modifiers’, the ‘Checks-Effects-Interactions Pattern’, argues for declaring allowed states using Solidity’s ‘require’ function, and notes that best practices emerging in Solidity are declarative (Prestwich 2018). He even states;

“Ideally, we should create a new declarative language to write these contracts.”

Other activity in the declarative area is apparent, with a Stanford study proposing ‘Contract Definition Language’ (CDL) (Agarwal 2016) described as declarative, with CDL descriptions being “open logic programs”. On closer inspection the language has a Prolog-like syntax, with features very close to ‘Answer Set Programming’ (see section 6.1.2), like negation as failure and the ability to specify a state-transition system. A further similarity to our study is the ability to reason over types. Agarwal presents a proof-of-concept via two case studies that model U.S. Federal Statutes, which illustrate the techniques our study uses to model legal logic. We believe Agarwal’s paper reinforces our focus on declarative languages and note his paper acknowledges the synergistic potential of a user-centric front-end without investigating it. This paper surfaces the question of why invent a new declarative language when there exist languages with over two decades of development and testing behind them, like Answer Set Programming (ASP) .

A recent paper explores links between legal and smart contracts and provides a comparative analysis of imperative and declarative languages for smart contracts, concluding that declarative languages “may simplify” blockchain smart contracts but may require imperative code to handle certain complex functions (G. F. Governatori 2018). Another recent paper explicitly investigates Answer Set Programming (ASP) as being a knowledge representation that also allows legal reasoning (Batsakis 2018), and compares it to other declarative languages for which solvers are available, such as TOAST (ARG-tech 2012) for structured argumentation with ASPIC+, and SPINdle (Data61, CSIRO 2013) for Defeasible Logic. This paper’s findings conclude that legal logic has to be manually encoded because there are no support tools, that SPINdle is the most expressive language for the problem domain but does not support ‘negation as failure’, while ASP may require workarounds for certain legal constructs relative to SPINdle. These last two papers inform about the domain in a theoretical exploration that is useful as a guide, especially the listing of declarative languages with usable solvers, and the value of tools to aid the coding of legal logic.

3.2.2 Research gaps

The trend visible is towards declarative and domain specific approaches, some developed with meta-programming languages like Coq. We see an opportunity to develop the user-centric front-end generating declarative code that Agrawal mentions, while recognising the potential of Ergo which appears to combine a number of threads (functional, declarative, formal verification). When compared against Solidity; 1) languages like Vyper don't have obvious advantages, 2) functional languages like Idris could improve code quality but require higher skill levels from programmers, 3) pretesting code is likely to improve code quality, 4,5) DSL and declarative languages promise further improvements over the advantages of functional languages.

Note, this subjective evaluation is used to identify promising approaches to the creation, testing and deployment of smart contracts that we use later to design a simulator with improved utility over Solidity. It is not used to verify the hypothesis.

TABLE 2 – DIFFERENT PROGRAMMING LANGUAGE APPROACHES TO IMPROVING SMART CONTRACT UTILITY CONTRASTED TO SOLIDITY

This table indicates the degree to which an alternative approach to creating smart contracts either improves upon, or is disadvantaged relative to the current smart contract creation approach (Solidity and bytecode) used by Ethereum;
 “+” indicating improvement and “-” indicating disadvantage.
 Note that a blank means there is no difference to Solidity.

no.	proposed improvements to smart contract creation, testing & deployment						
		ease of use	understandability	ease of testing	free of security exploits and bugs at deployment	scalability	cost
	current solution (Solidity>bytecode)						
1	change imperative language (Vyper, eWASM, Flint)						different programming languages
2	change to functional language paradigm (Idris, Formality)	-	--	+	+		
3	use pretested code libraries (OpenZeppelin)			+	+		
4	use Domain Specific Language (DSL) with formal verification (Ergo)	+	+	++	++		
5	change to declarative language paradigm (CDL, LPS, ASP, TOAST, SPINdle)	+	+	++	++	+	

Legend:

+	slight advantage	-	slight disadvantage
++	advantage	--	disadvantage
+++	large advantage	---	great disadvantage

3.3 Visual Approaches

3.3.1 Current Research

Thinking through the challenge of simplifying human input when creating smart contracts immediately brings to mind graphical interfaces. These were originally seen as a way of improving human-computer interaction and have succeeded by becoming the dominant user interface for personal computers and phones. Graphical interfaces were first imagined (Bush 1945), then prototyped (Sutherland 1963), (Engelbart 1968), (Xerox PARC 1973), and eventually commercialised by the Apple, Android, Windows and other Graphical User Interface (GUI) operating systems.

The primary simplifying mechanism of these systems is metaphor, a technique that uses an abstract graphic to represent something in the real world (HSC CoWorks 2019). This approach allows novice users to navigate a GUI screen as if it is a physical office, and works well where physical equivalents exist, but legal contracts can involve abstract ideas; for example, an employment contract may require the employee's best efforts to raise the employer's profile with a certain demographic. The second problem is that screen space clearly limits the complexity that can be represented, and techniques like zoom, scroll and decomposition mitigate this only to a

limited degree. Metaphor however identifies one path forward, that of exploiting something already understood by the user.

Ideas for simplifying smart contract creation started to surface soon after Ethereum debuted, with the observation that blockchain/smart contract projects were complicated (Buelau 2017), and that it was not realistic for end users (lawyers, business-people, public) to write code (Marks 2018). Marks makes a strong case for reducing construction to simply specifying transaction logic, and suggests that manipulating tokens on a screen can suffice to do this. Marks introduces EtherScripter, the Unreal Editor (a games industry approach), and Hyperledger Composer (since depreciated). Another initiative was the Confideal visual smart contract editor being developed to simplify smart contract creation through pre-coded templates (Buelau 2017; EconoTimes 2017) but this website has recently been depreciated (Confideal 2019).

EtherScripter uses Blockly, a visual programming language generating JavaScript used to teach students how to code, and focused on assisting with correct syntax. The Unreal Editor, allows users to use the Blueprints Visual Scripting system to create smart contract processes (Epic Games 2019), and is capable of scripting the full smart contract lifecycle for a complex contract, but is visually daunting given the target of untrained users. Of these, the Hyperledger Composer approach of using minimalist icons related via arcs was the simplest and most intuitively appealing, but clearly lacking as it has been also depreciated (Hyperledger Composer 2019). The approach to visual smart contract creation used by its successor, 'Hyperledger Framework' is not visible (Hyperledger 2019).

Kowalski is investigating what he describes as a logic based production system language that unifies both forward and backward inference methods into a single framework (Kowalski 2019). While this work envisages coding in a language that looks declarative¹, its novel ideas surface in the testing area. The web site shows the results of executing an LPS program implemented in SWI Prolog² and running on SWISH³ that displays:

“a Gantt chart showing a timeline of facts updated by external events and actions performed by the system” (Kowalski 2019).

A different graphic shows a graph visualising states and state transitions. Visualisations at smart contract creation are beyond the scope of this study but this informs future work (section 8.4) as visualising is a way to build understanding and confidence in the contract just created. Further, because the instantiation step has been completed, the unambiguous executable code produced can be more easily manipulated to display the contract one aspect at a time, as compared to visualisation during contract creation.

Another recent approach is to digitalise current paper format legal documents. This can be done with mark-up and results in a template with variables that have to be instantiated just as paper has to be filled in by hand (OpenLaw 2019). OpenLaw is an initiative by law professor Aaron Wright and others dating from 2017; and is a confirmation that some in the legal profession are comfortable retaining the traditional sequential text format of legal contracts. This approach is built on top of Ethereum, consists of a core written in Scala, and autogenerates JavaScript from instantiated

¹ LPS is framed as a logic based production system language that aims to close the gap between logical and imperative computer languages.

² SWI-Prolog is a versatile implementation of the Prolog language available free at <https://www.swi-prolog.org/>

³ SWISH -SWI-Prolog for SHaring is a web front-end for Prolog. <https://swish.swi-prolog.org/>

OpenLaw Markup Language used to render the contract on screen. It allows free form contracts built from modules.

3.3.2 Research Gaps

Of the visual approaches investigated, the OpenLaw approach appears the most practical, while earlier metaphor based approaches appear to have stumbled. That traditional sequential text is preferred in this instance invites an analysis of the benefits compared to other approaches. OpenLaw however target JavaScript, providing a research opportunity to investigate a pure declarative language as the target.

TABLE 3 – VISUAL APPROACHES TO IMPROVING SMART CONTRACT UTILITY CONTRASTED TO ETHEREUM'S SOLIDITY

This table indicates the degree to which an alternative approach to creating smart contracts either improves upon, or is disadvantaged relative to the current smart contract creation approach (Solidity and bytecode) used by Ethereum;
 "+" indicating improvement and "-" indicating disadvantage.
 Note that a blank means there is no difference to Solidity.

no.	proposed improvements to smart contract creation, testing & deployment							
		ease of use	understandability	ease of testing	free of security exploits and bugs at deployment	scalability	cost	
	current solution (Solidity>bytecode)							
6	visual smart contract editor (Confideal, Hyperledger, Unreal Editor, Blockly)	+	+			-		visual approaches
7	digitize contract with mark-up, instantiation auto-generates code (OpenLaw)	+++	+			++	+	
8	visual testing (LPS)		++	++	+			

Legend:

+	slight advantage	-	slight disadvantage
++	advantage	--	disadvantage
+++	large advantage	---	great disadvantage

3.4 Auto-generation Approaches

3.4.1 Current Research

From our experience auto-generation approaches can be partitioned into 1) translation of legal logic from the original paper legal contract, and 2) instantiating code templates. Translation of the original legal text is clearly more demanding than instantiation, but the cost invites an analysis of the economics as it is common practise for traditional Standard Form Legal Contracts (SFLC) to be developed and issued by a specialised central authority (JCT 2019).

Attempting to translate documents written for a specific audience and industry may encounter difficulties due to the varying reasons particular sentences and words are included. These range from stating precedence, to emphasis, defining terms, and elaborating contract 'performance'. Sometimes wording important to understanding the overall purpose of the contract is missing, implied by the cultural setting; while other text can be ambiguous, causing disputes. All these factors make machine translation from the original legal contract difficult, especially when compared to the low unit costs achievable by repeated reuse of manually translated templates.

Choudhury et. al. investigate the auto-generation of smart contracts for Ethereum using ontologies and semantic rules (Choudhury 2018). This research uses Web Ontology Language (OWL) and Semantic Web Rule Language (SWRL), so can be seen as building on Semantic Web inspired initiatives (W3C 2019; Antoniou 2012). Once an ontology for the domain and rules are in-place, relevant text can be parsed for meaning. This step generates a JSON file which is then used to instantiate an Ethereum smart contract template. A proof-of-concept is achieved by translating eligibility criteria for clinical trials and car rentals. While promising, the method as described

requires an expert to generate a smart contract template, ontology and rules for every use case. Over a longer timescale, reuse is an objective; however, the authors envision using state-of-the-art natural language processing techniques to generate the ontology and rules, so this approach has not yet been fully realised. As it currently stands, this approach requires considerable custom effort by contract type to achieve translation from the original legalese. While this approach is not yet practical, it does illuminate a pathway towards tools that translate from paper contracts.

An augmenting approach is the combination of hand coded rules enhancing the performance of machine learning (Curtotti 2010) in the classification of components of text.

3.4.2 Research Gaps

The auto-generation approach from paper legal contracts discussed above is not yet practical and also beyond the scope of this study, however auto-generation of code at instantiation of templates seems both practical and promising (see Table 4).

3.5 Formal Verification Approaches

3.5.1 Current Research

Formal Verification is defined as ‘proof using formal mathematical methods that a program behaves consistently with its specification’ (derived from Berztiss 1988). Formal verification holds the promise of not only improving smart contract reliability but actually proving the absence of exploitable flaws, unfortunately at a prohibitive cost. This promise means this technique cannot be ignored. That said, all formal verification approaches surfaced by this study, while having potential to improve the reliability problem, suffer by magnifying the usability problem.

As discussed under section 3.2, the cost of formal verification with Idris is much reduced because formal proofs can be attempted within the actual programming language, however it is likely that Idris improves reliability at the expense of usability, given its conceptually difficult nature. Another approach is with Scilla or Smart Contract Intermediate-Level LAnguage (scilla-doc 2019), an intermediate level language being developed within the Coq Proof assistant (Inria 2019). Scilla is designed to be a target for Solidity with the intention of a more rigorous automata-based model of execution that can be proven via Coq. As for Idris, this approach ignores the fact that most users have little prior experience with programming.

A paper by Zheng reports on the development and verification of a novel formal symbolic process virtual machine (FSPVM-E) for verifying the reliability and security of Ethereum smart contracts, completely in the Coq proof assistant (Yang and Lei 2018). This virtual machine simultaneously executes Ethereum smart contracts to verify their reliability and security properties at the time of execution, and so appears to address reliability issues, but not usability or cost issues. An advantage of Zheng’s approach is that it informs a way to achieve on screen formal verification at contract creation similar to Rodin (Event-B.org 2018).

Other initiatives involve proving Ethereum smart contract bytecode in Isabelle/HOL (Amani 2018), and formal verification for ASP programs (Aguado 2015; Harrison 2015).

3.5.2 Research Gaps

Formal verification of code is a complex topic outside the scope of this study, recognised as being difficult and expensive for procedural languages, but holding the promise of proving the absence

of exploitable flaws and bugs in smart contracts. We believe however that declarative languages with their formal mathematical foundations and modelling abilities are likely to have lower formal verification costs, making further investigation worthwhile (section 8.4).

3.6 Conclusion

This review has identified significant research activity in the area of using domain specific and declarative languages for smart contracts, however much of the activity focuses on designing a language rather than testing the suitability of an existing language. Two theoretical papers investigate existing declarative languages and investigate possible advantages, however no evidence of an actual implementation using an existing declarative language has surfaced, opening a research opportunity. Table 4 summarises and highlights current research approaches that improve the utility of smart contracts for the ultimate end-user.

TABLE 4 – ALL ALTERNATIVE APPROACHES TO IMPROVING SMART CONTRACT UTILITY CONTRASTED TO ETHEREUM'S SOLIDITY

This table indicates the degree to which an alternative approach to creating smart contracts either improves upon, or is disadvantaged relative to the current smart contract creation approach (Solidity and bytecode) used by Ethereum;

“+” indicating improvement and “-” indicating disadvantage.

Note that a blank means there is no difference to Solidity.

		<div>ease of use</div> <div>understandability</div> <div>ease of testing</div> <div>free of security exploits and bugs at deployment</div> <div>scalability</div> <div>cost</div>					
no.	proposed improvements to smart contract creation, testing & deployment						
	current solution (Solidity>bytecode)						
1	change imperative language (Vyper, eWASM, Flint)						different programming languages
2	change to functional language paradigm (Idris, Formality)	-	--	+	+		
3	use pretested code libraries (OpenZeppelin)			+	+		
4	use Domain Specific Language (DSL) with formal verification (Ergo)	+	+	++	++		
5	change to declarative language paradigm (CDL, LPS, ASP, TOAST, SPINdle)	+	+	++	++	+	
6	visual smart contract editor (Confideal, Hyperledger, Unreal Editor, Blockly)	+	+			-	visual approaches
7	digitize contract with mark-up, instantiation auto-generates code (OpenLaw)	+++	+			++	
8	visual testing (LPS)		++	++	+		
9	autogeneration from original legal contract (Choudhury)	+					auto-generation
10	formal validation of imperative code (FSPVM-E, Scilla)			--	++		formal verification approaches
11	formal validation of declarative code	+	+	--	++		

Legend:

+	slight advantage	-	slight disadvantage
++	advantage	--	disadvantage
+++	large advantage	---	great disadvantage

The outstanding approach for improving smart contract utility is approach 7 which auto-generates JavaScript smart contract code from a status-quo user interface. This approach provides the clear improvement to smart contract utility over Solidity that we are seek for the design of our simulator. Improved utility for lawyers, business-people and the general public is gained from being able to create smart contracts at a much lower cost because programmers are not needed.

In contrast to the auto-generated JavaScript used by approach 7, we choose to auto-generate an existing pure declarative language (ASP) to encode legal logic, in line with our hypothesis. We also note that Table 4 indicates some advantages for DSL and declarative language approaches when contrasted to Solidity. The combination of approach 7 with the auto-generation of an existing pure declarative language is to our knowledge, unique.

In summary; 1) we identify an approach to smart contract creation, testing and ongoing life-cycle that clearly has greater utility to the ultimate end-user than the Solidity approach, 2) we implement this approach as an experimental simulator within the timeframe allowed this project, 3) we use this simulator to experiment with ways of implementing declarative smart contracts. The hypothesis we investigate and verify is:

“For the subset of legal documents amenable to implementation as smart contracts, coding with a declarative language is possible and facilitates achieving improved utility for those smart contracts.”

To verify this hypothesis we seek: 1) to demonstrate that a legal document can be implemented with declarative code, and 2) evidence that using declarative code facilitates the implementation of a smart contract approach with improved utility over the Solidity approach. The evidence sought ranges from easier auto-encoding and code manipulation to easier testing.

Chapter 4 Method of Investigation

Our methodology verifies the hypothesis with a proof-of-concept following an experimental phase that investigated how best to implement smart contracts with a pure declarative language. This experimental phase was undertaken on a block chain simulator built earlier as part of the experimental setup. The simulator reproduces a code generating user interface identified in the literature review that dramatically improves the utility of smart contracts for the ultimate end-users because it eliminates the need for programmers. As implementation requires construction of a code auto-generation mechanism, this construction can serve as a point of comparison with Solidity. Other points of comparison are ease of testing and ease with which smart contract code can be programmatically read and manipulated after creation, an important feature required when recording smart contract state changes as subsequent blockchain transactions.

The proof-of-concept seeks to verify the two unproven concepts embedded in the hypothesis; 1) that it is possible to encode smart contracts with an existing pure declarative language (ASP), 2) that using a declarative language facilitates delivering improved utility to the end-user relative to Ethereum's current approach (Solidity).

Our investigation involves 3 steps: 1) identify approaches from research that dramatically improve the utility of smart contracts for the ultimate end-user over Solidity, 2) use this information to design and implement an experimental blockchain simulator with improved smart contract utility, allowing investigation of how best to implement smart contracts with a pure declarative language, 3) identify and demonstrate the advantages of using a pure declarative language to code smart contracts, by converting a suitable legal document to a smart contract using this simulator. Step 2 necessarily involves iterative prototyping as a technique for overcoming problems and refining ideas, and provides the freedom to experiment with any part of the design in order to address hurdles, a technique used in user interface research (Bäumer, et al. 1996).

It is intended that evidence surfaced from step 3 verifies the hypothesis stated in section 1.2 and repeated in section 3.6. Demonstration of a legal document implemented as a declarative smart contract via a suite of test cases (see Appendix C) serves as proof-of-concept for the first unproven concept in the hypothesis. The second unproven concept in the hypothesis is investigated by comparison of our ASP implementation with the likely procedural language implementation. Finally, some extensions to more complex legal contract types are investigated in-order to identify the factors that limit applicability. Legal contracts that cannot be, or are not economic to implement, define boundaries.

Chapter 5 Design

5.1 Introduction

The areas holding promise for improving the utility of smart contracts emerging from the literature are; 1) digitizing current legal documents with mark-up, 2) declarative and domain specific languages, 3) visualisation of testing and, 4) formal verification. In this study we use 1) to investigate 2) by creating a smart contract on a purpose built simulator from an example Australian legal contract ('Will and Testament') available online (LawDepot 2019). A 'Will' was chosen because it uses deductive logic, is not too complex and involves a human-in-the-loop (executor).

<p>LAST WILL AND TESTAMENT OF _____</p> <p>I, _____, presently of _____, hereby revoke all former testamentary dispositions made by me and declare this to be my last Will.</p> <p>PRELIMINARY DECLARATIONS</p> <p><u>Prior Wills and Codicils</u></p> <p>1. I revoke all prior Wills and Codicils.</p> <p><u>Marital Status</u></p> <p>2. I am married to _____.</p> <p><u>Children</u></p> <p>3. My living children are _____.</p> <p>EXECUTOR</p> <p><u>Executor</u></p> <p>4. The expression 'my Executor' used throughout this Will includes either the singular or plural number, or the masculine or feminine gender as appropriate wherever the fact or context so requires. The term 'executor' in this Will is synonymous with and includes the term 'executrix'.</p> <p><u>Appointment</u></p> <p>5. I appoint _____ of _____, New South Wales as the sole Executor of this Will.</p>
--

Figure 1 - Opening clauses of the 'Will and Testament' example used for illustration (full listing in Appendix B)

This 'Will and Testament' divides the whole estate according specified percentages, but disqualifies on early death or contest, and has a wipeout (all beneficiaries are dead or disqualified) clause that distributes to a different group of people. We believe the complexity of logic is adequate to serve as an example and demonstrate proof-of-concept. Elaborations like bequeathing different assets to different beneficiaries (e.g. jewellery distributed to females) give no advantage in this instance.

We use a purpose built standalone simulator because the Ethereum test environments (Kovan, Ropsten, Rinkeby) would require inclusion of a declarative language grounder and solver to the underlying Ethereum Virtual Machine.

5.2 User Scenario

For orientation it is useful to envisage an ideal use case which can then serve as a benchmark for current research and a target for any envisioned system. Consider a situation a few years into the future where John decides to write a Will and Testament. John's university friend, now a lawyer, has previously discussed the revolution occurring in legal services around smart contracts. John rings him and agrees to try the new method. John downloads the app (LegalAgreements) from a law society website and sets up a contract writing session as session owner and testator. This involves sending a session link to beneficiaries and other members of his family, the chosen executor (lawyer friend) and witnesses. This session link functions differently depending on the role of the recipient. Beneficiaries and other family members simply authorise access to their wallet and PII by physically attaching their hardware wallets (section 2.6).

Roles such as the executor and witnesses attend an on-line session that allows them to interact with voice, text and video communication via a Discord-like interface (Discord Inc. 2019) (section 8.4), while providing a common real-time view of the contract being written. John works through the form allocating roles and selecting assets via touch from information retrieved from wallets and blockchain hosted shared ledgers. Recently converted shared ledgers such as the NSW Torrens Title Register (NSW LRS 2019) and Motor Vehicle Registry (NSW RMS 2019) greatly enhanced the LegalAgreements app by supplying the full description of these assets without data entry. John adds valuable jewellery and furniture manually, before specifying the percentage distribution to each beneficiary. Finally, after a brief discussion, the witnesses agree to sign the 'Will and Testament' with their private keys. The computer code generated is then packaged and deployed to a blockchain where it sits waiting for events that trigger its actions until it is either fully executed, or cancelled by John. The entire session has spanned about 10 minutes.

The following week, John decides that he would like a hardcopy of the Will which the system generates by reverse engineering the smart contract declarative code, discovering instantiations by comparing code against templates, and using these instantiations to populate the text template.

5.3 Improving Utility

Standard Form Legal Contracts (SFLCs) consist of a template created and published by some recognised authority (JCT 2019), with the actual contract created by filling out details in ink (instantiation). This two-step process can be applied to electronic contracts, and as most people have seen paper SFLCs and been challenged by them, this familiarity can be exploited.

Digitising a SFLC opens up automated assistance that can further simplify use and display; for example, many forms leave 2-3 lines for address whereas an electronic version would indicate this with one special word at the instantiation point, reducing the length of the document and its visual complexity. These special words could embed tab stops, allowing users to tab from instantiation point to instantiation point, jumping the text in between and again reducing visual complexity.

It is possible to use this digital SFLC as a status-quo user interface to generate smart contract code. To understand how, we take inspiration from Ricardian Contracts (section 2.7), and propose that a pair of templates are now created by the central authority; 1) an electronic equivalent of the paper standard form, and 2) its legal logic as computer code, as shown in Figure 2.

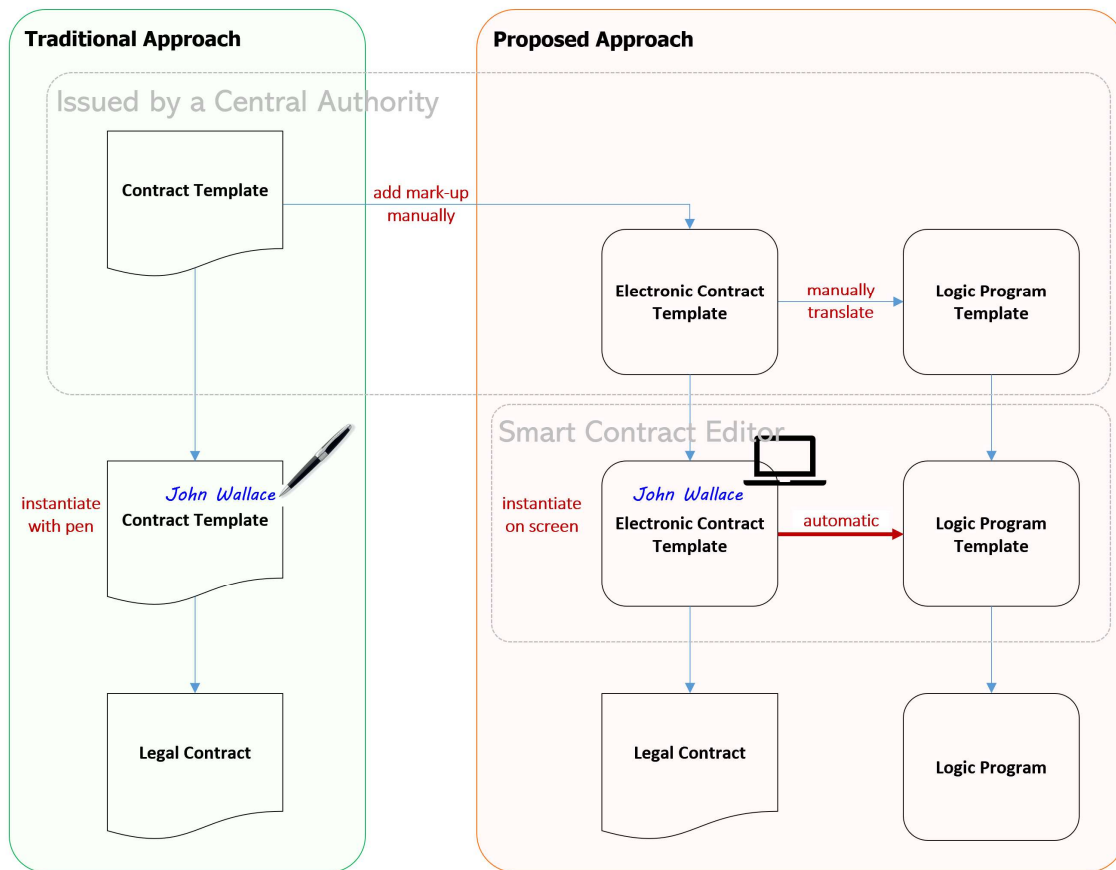


Figure 2 - Traditional Approach vs Proposed Approach to writing Legal Contracts

These two templates are output from a manual first translation step. A second translation step (instantiation) is required when the smart contract is created, and we propose that a smart contract editor be constructed to aid this. Our insight is that information provided by users when instantiating the digital SFLC template can also instantiate computer code templates. Figure 2 shows the traditional process for writing legal contracts on the left side, and the proposed approach on the right side. The proposed approach uses both text and code versions of the contract as pioneered by the Ricardian Contract, but exploits instantiation of the text template to instantiate the code template.

Another simplification can be gained from storing PII in wallets (section 2.6), so once a person is identified, their address is also known and instantiated automatically. This technique and a similar mechanism that retrieves data from DSLs could reduce instantiations by more than half, with most instantiations now simply a screen tap or mouse click, rather than text input. These simple examples belie the potential for other simplifications and a dynamic help feature (Agarwal 2016).

A further significant but possibly overlooked advantage is that traditional paper format handles contracts of arbitrary complexity. Using sequential text to express complex ideas appears culturally embedded (e.g. doctoral dissertations, complex contracts, conference papers etc.), is hard to duplicate with other approaches, and not a feature of most other smart contract research.

Translating a SFLC to an electronic equivalent is straight forward; at minimum simply adding mark-up to existing text at instantiation points. This study requires only a small subset of available mark-ups (presentation, processing, internal referencing), however there is great scope for more sophisticated mark-ups as mark-up capabilities are extended. This consistent trend started with

the first mark-up language (Goldfarb 1992), the first hypertext mark-up language (Berners-Lee 1989), and increasingly more powerful constructs like the powerful referencing construct XLink (W3C. 2010), OWL (Web Ontology Language), SWRL (Semantic Web Rule Language) and RDF (Resource Description Framework), all Semantic Web initiatives (Arroyo, et al. 2004).

After reviewing Choudhury's proposal to auto-generate smart contracts (Choudhury 2018) we recognise that a mark-up approach using Semantic Web technologies is eventually likely to yield a result where legal documents embed machine solvable logic. The OpenLaw approach allowing construction of free form contracts with mark-up is also informative (OpenLaw 2019).

In contrast to these last two proposals, there is a benefit to using a central authority to publish smart contract templates because it matches current practise in some segments; for example, JCT is a collaboration of organisations in the UK construction industry that produce standard contract forms (JCT 2019); and is practical because initial costs can be spread over repeated template usage over time, greatly reducing cost per contract thus making smart contracts much more useful to the end-user. It is envisioned these templates reside in a blockchain hosted DSL and served on request.

5.4 Translation to Executable Code

A typical legal contract such as a 'Will and Testament' (see Figure 1) contains wording with a variety of purposes, the most important for the purpose of automation are clauses defining 'performance' (Australian Contract Law 2010); that is, the actions agreed in exchange for consideration (section 2.2). The purpose of other text ranges from emphasis, to listing precedence, and defining terms. Sometimes wording important to understanding the overall purpose of the contract is missing, implied by the cultural setting; while other text can be ambiguous, often the cause of legal disputes, complexities that make reliable machine translation of contracts difficult (Choudhury 2018).

Using templates manually translated by some central authority (JCT 2019) avoids this difficult machine translation, however the text template and accompanying computer code still have to be instantiated. We aim to auto-generate this code as no programming language is likely to be simple enough for untrained users due to a tension between simplicity and expressive power (Kuhn 2014). Instead the semantic gap has to be bridged by other techniques like decomposition and simplification. We note in section 5.3 that use of a familiar SFLC format simplifies our interface, and also that use of a declarative language simplifies auto-generation of code. Splitting translation into two steps also simplifies, as only instantiations are now required at contract creation. Further, it is possible to group declarative code into three types; facts, logic program and events, and we believe it is possible to code so that only the auto-generation of facts is required at smart contract creation. Further simplification can be achieved by using Smart Instantiation Editors (SIE) and meta-templates. SIEs understand what type of data is required at an instantiation point and where to source it, greatly simplifying the user experience, while meta-templates (instantiated for type before use) can simplify the number of templates required (section 6.3.4).

On reflection, our underlying approach appears to reframe the creation of smart contracts as a translation from traditional legal contracts to computer code, seeking ways to simplify each step. Using a declarative language simplifies the computer code required, simplifying auto-generation, which is further simplified by splitting translation into two steps, and splitting declarative code into facts, logic and events; and as a result only facts need be auto-generated at smart contract creation, which is simplified further by using meta-templates.

5.5 Designing a Simulator

5.5.1 Introduction

The objective of a simulator is to provide a platform that allows investigation of the hypothesis.

5.5.2 Assumptions

As this study investigates future technology, it is reasonable to make the simplifying assumptions. These are; 1) Personally Identifiable Information (PII) data is available from wallets, 2) public shared ledgers like the Torrens Title register (NSW LRS 2019) are implemented as blockchain distributed shared ledgers (DSLs) with owners permissioned for read access.

5.5.3 Architecture

The proposed process for writing a smart contract is:

- users access published templates for the type of legal contract required (see Figure 3)
- users are then presented with that legal contract's template in familiar format on screen
- the system aids users to fill out this document in a fluid intuitive way by exploiting available information like PII data from wallets and asset information from DSLs
- this data instantiates both text and code templates via paired instantiation place holders

This approach allows splitting translation of legal logic into two steps; step 1 translates the legal logic of the contract **type**, step 2 translates the details of the actual contract, essentially instantiation. These steps are identical to the traditional process.

Figure 3 illustrates how users access published templates by supplying a contract key and version via a request to the smart contract template DSL. These templates are the output of translation step 1. Translation step 2 (instantiation) is performed by users with a 'Smart Contract Editor' (hereafter SCE), shown in Figure 3 as extracting PII and other data from wallets and shared ledgers permissioned by the contract writing session to ease data collection. This instantiation step concurrently generates the smart contract in text and code, with code denoted in Figure 3 as **facts**. The other component of smart contract code, denoted as a **logic program** is copied as is from the supplied code template. **Logic** is invariant for all contracts of a certain type, and is coded, published and maintained by a central authority.

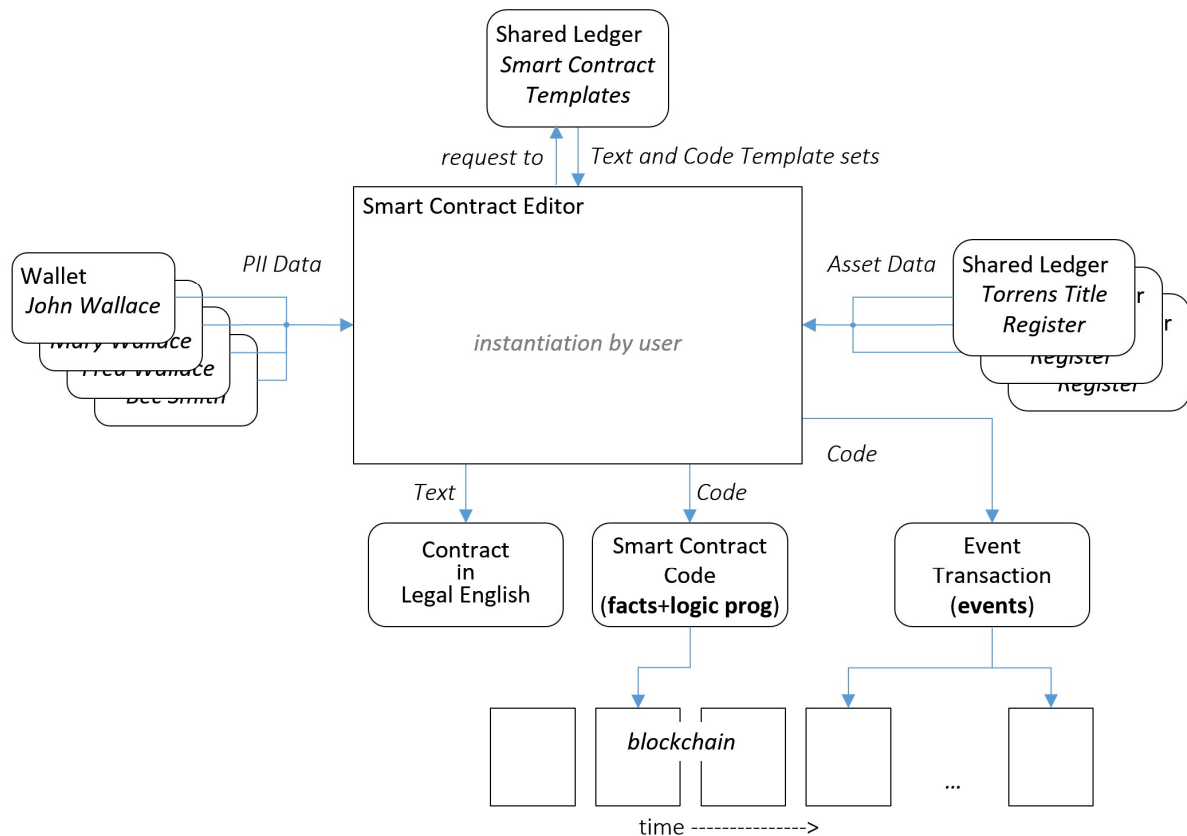


Figure 3 - High level view showing data sources and flows for smart contract creation

The published templates are supplied in a structure consisting of two groups; templates for text and templates for code. Figure 4 shows how the SCE uses supplied text templates to construct an digital equivalent of the paper SFLC on screen. This digital equivalent aids users in its instantiation, and each instantiation automatically generates code (**facts**) from supplied code templates.

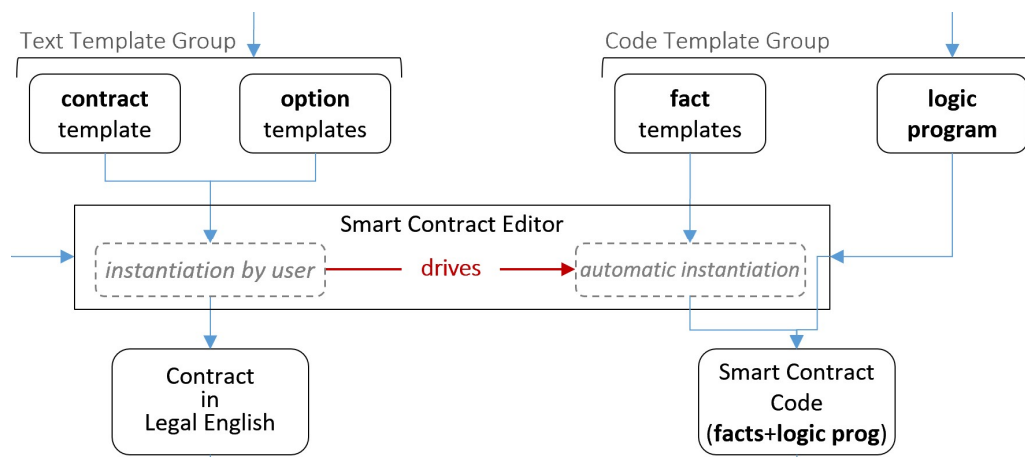


Figure 4 - How users instantiate declarative code

Splitting smart contract declarative code into **facts** and **logic program** has advantages. Translation step 1 that produces the **logic program** only has to be performed once when the contract type is initially created and published, mirroring the current practise of publishing SFLC templates. This makes centralisation of this difficult and expensive step practical, leaving only the much simpler step 2, or instantiation step at contract creation time.

5.5.4 Smart Contract Lifecycle

Legal contracts have lifecycles punctuated by events that drive activities ('performance'). In an analogous way, a smart contract must be able to sense external changes that affect its state and trigger 'performance' as specified. Our approach achieves this with **events**, declarative code that communicates external events to a smart contract, added as a transaction to the blockchain at a subsequent date (see Figure 5). **Events** are simply **facts** that occur after smart contract creation.

Because smart contracts cannot be changed once deployed to a blockchain, the only way state can be changed is via new transactions. Our design requires that the original smart contract and all subsequent transactions related to that smart contract be aggregated before execution. This is possible because of three features of our declarative language; 1) elaboration tolerance (McCarthy 1988), 2) 'Negation as Failure' (NAF) which allows code to function without atoms that are only created on detection of a specific pattern, 3) the way we model objects and situations one-to-one with atoms, allowing our declarative code to be split into three parts.

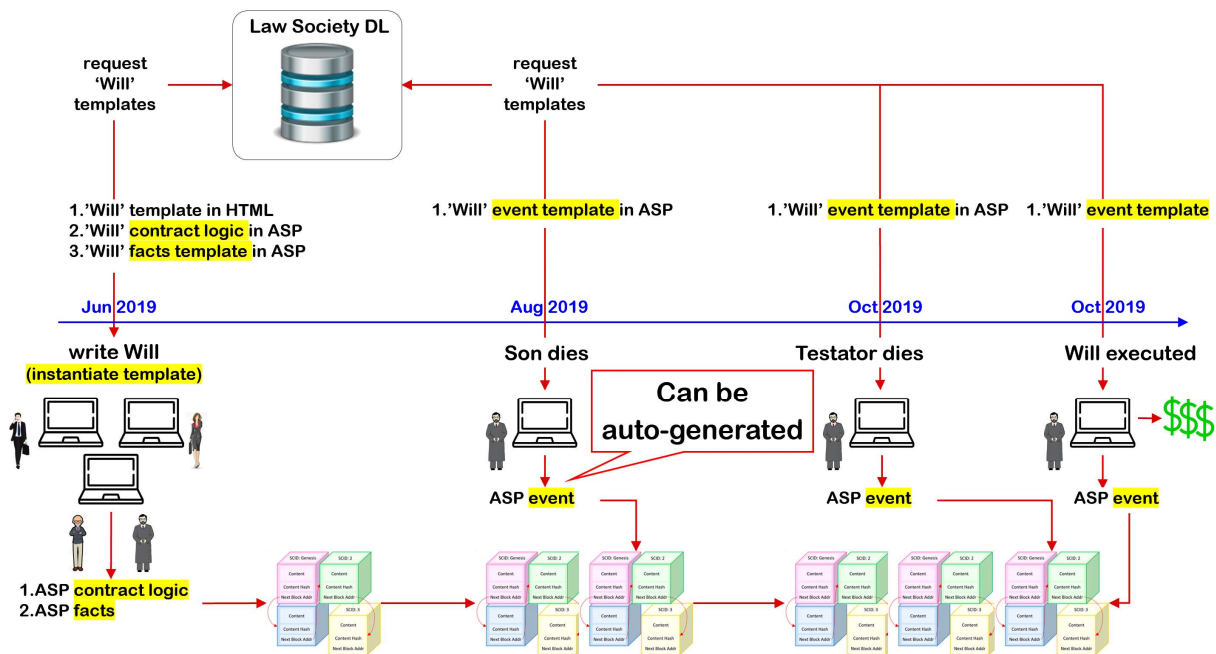


Figure 5 – Lifecycle of the 'Will and Testament' smart contract

5.5.5 Differences to Ethereum Smart Contracts

There are a number of differences between Ethereum and the approach described in this paper. Firstly we propose changing the transaction content deployed to the blockchain from bytecode to declarative code, adding a declarative code solver to the Ethereum Virtual Machine. Secondly, the wallet concept is modified to hold detailed information about the wallet owner (see 2.6 Wallets). Thirdly, there is transparent access to blockchain shared ledgers for owners of those assets, a feature implemented in some emerging blockchains, but requiring explicit mention for this study. Finally, this study proposes 'Human-In-The-Loop' processes, because many legal contracts require human intervention at some point; for example, "Will and Testament" requires an executor. These changes broaden the applicability of smart contracts relative to those built into Ethereum.

Chapter 6 Implementation

6.1 Constructing the Simulator

6.1.1 Development Platform

This study constructs, then uses a blockchain simulator with a smart contract editor (hereafter simulator) to explore and test ideas for improving the usability and reliability of smart contracts. The simulator then serves as a proof-of-concept as we demonstrate how to take a legal contract from paper format, through each translation step to automated execution of a full contract life cycle via test cases. Screens and process designs around creation and testing of smart contracts are close to a possible live implementation, while blockchain features are not discussed and are no longer seen as relevant to the study objective.

The simulator development platform choice was driven by the short time frame allowed this study which mandated use of a familiar toolset, and the nature of this user interface which allows investigation and demonstration in a standalone environment. The simulator is constructed in 64-bit Python v3.6.8 using the PyQt5 v5.9.2 GUI library on a Dell XPS-15 (Intel i7-6700HQ, 16GB RAM, Windows 10 64-bit), using the Spyder IDE v3.3.3. Software used includes HTML as the mark-up language (restricted by PyQt5 (Qt 2017)), and ASP as the declarative language, using Potassco's clingo version 4.5.4 (64-bit).

6.1.2 Answer Set Programming (ASP)

Answer Set Programming (ASP) is a form of declarative programming able to encode knowledge and oriented toward (primarily NP-Hard) search and optimization problems (Lifschitz 2008).

ASP has emerged since 1999 from a number of lines of research, including logic programming (Prolog), knowledge representation and constraint satisfaction (Lifschitz 2008). It combines an expressive language, a model-based problem specification methodology, and efficient solving tools. This study uses ASP-Core-2 standard language syntax (Calimeri 2015).

Brewka et.al. note that close connection to nonmonotonic logics provides ASP with the power to model default negation, deal with incomplete information, encode domain and problem-specific knowledge, defaults, and preferences in an intuitive and natural way (Brewka 2011). ASP also has the important attribute of 'elaboration tolerance' (McCarthy 1988), defined as "the ability of a computer program's representation of a problem to accept changes in problem specifications without need to rewrite an entire program" (Lierler 2017).

Processing ASP typically requires two steps; 1) grounding 2) solving. Grounding involves replacing variables with instances, converting a predicate program (1st order logic) into an equivalent propositional program. Solving involves using methods from satisfiability solving (SAT), Fixed Point Mathematics, and Satisfiability Modulo Theories (SMT) to solve the propositional program (Lifschitz 2008). Solvers return Answer Sets that represent solutions.

The language can be summarised as consisting of three types of clauses; facts, rules and constraints. **Facts** create the solution space, **rules** encode the general problem and generate solutions, and **constraints** filter out unwanted solutions leaving only the answer.

Examples:**Facts:**

`team(1).` means team 1 exists, where ID "1" denotes team 1

Rules:

`pair(T1,T2) :- team(T1),team(T2),T1<T2.` if $T1 = \{1,2,3\}$ and $T2 = \{2,4,6\}$ then
`pair(T1,T2) = { (1,2),(1,4),(1,6),(2,4),(2,6),(3,4),(3,6) }`

Constraints:

`:- pair(T1,T2),T1<2.` delete answers where T1 has a value less than 2. This removes
`(1,2), (1,4), (1,6).` Answer is `{ (2,4), (2,6), (3,4), (3,6) }`

where “:-” means IF, “;” means OR (not shown), “,” means AND

A simplified language syntax is:

Term = either a constant

string starting with lowercase

| “quoted string”

| integer

| variable

string starting with upper case

| arithmetic term

$-(t)$ or $(t \diamond u)$ where $\diamond \in \{+, -, *, /\}$

| functional term

$f(t_1, \dots, t_n)$

Atom = a predicate with arity n that has form

$p(t_1, \dots, t_n)$

Literal = an **Atom** or its negation

Clause = a finite set of **Atoms** a_n of form
 with two parts and three types

$a_1; \dots; a_j \quad :- \quad a_1, \dots, a_m, \text{not } a_1, \dots, \text{not } a_n.$

Head exists if **Body** evaluates to true

Facts:

Head

Rules:

Head $:-$ **Body**

Constraints:

$:-$ **Body**

ASP Program = a finite set of **Clauses**

ASP also allows two forms of negation:

1. Weak negation `not a` true if $a = \text{false}$ or doesn't exist (assumed false).
2. Strong negation `-a` true if $a = \text{false}$ and vice versa.

Note: Weak negation is also called ‘negation as failure’ (NAF).

Aggregation can be achieved with aggregate functions `#sum` and `#count`, while many of the other features available are not listed because they are not used by this study.

6.1.3 Hypertext Mark-up Language (HTML)

The mark-up language used for this study is dictated by the GUI tool used (section 6.1.1). Only a subset of the mark-up functionality provided by HTML is used (presentation and description), with added extensions embedding attributes with each instantiation placeholder (IPH) and providing more control over display scrolling, and internal referencing. The attribute extensions provide variable identifier, type, action, and cardinality information that specifies how routines should process each IPH.

6.2 Description of the Simulator

Figure 6 is a screen shot of our simulator’s SCE; showing the legal document (left) and tools (right).

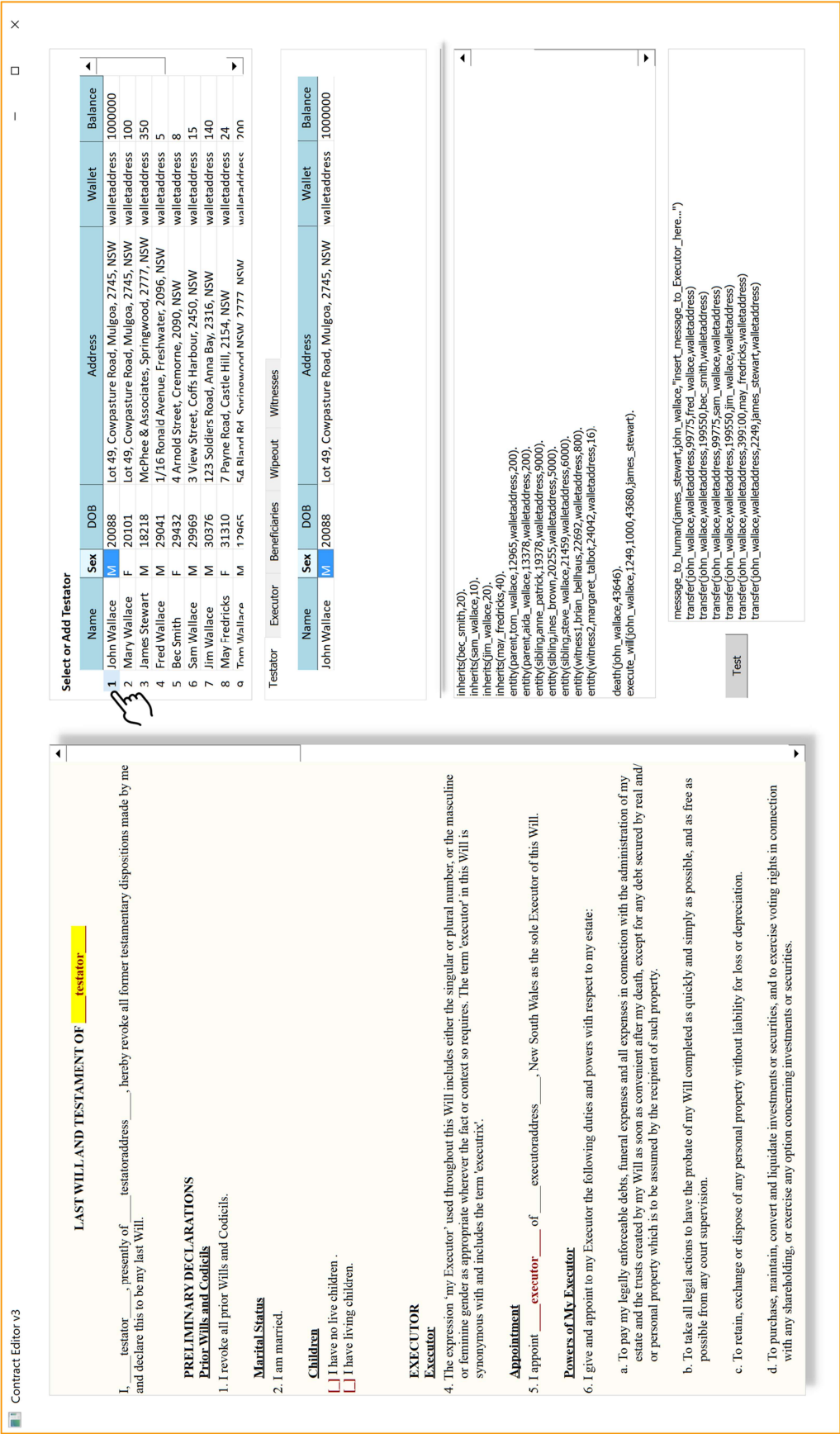
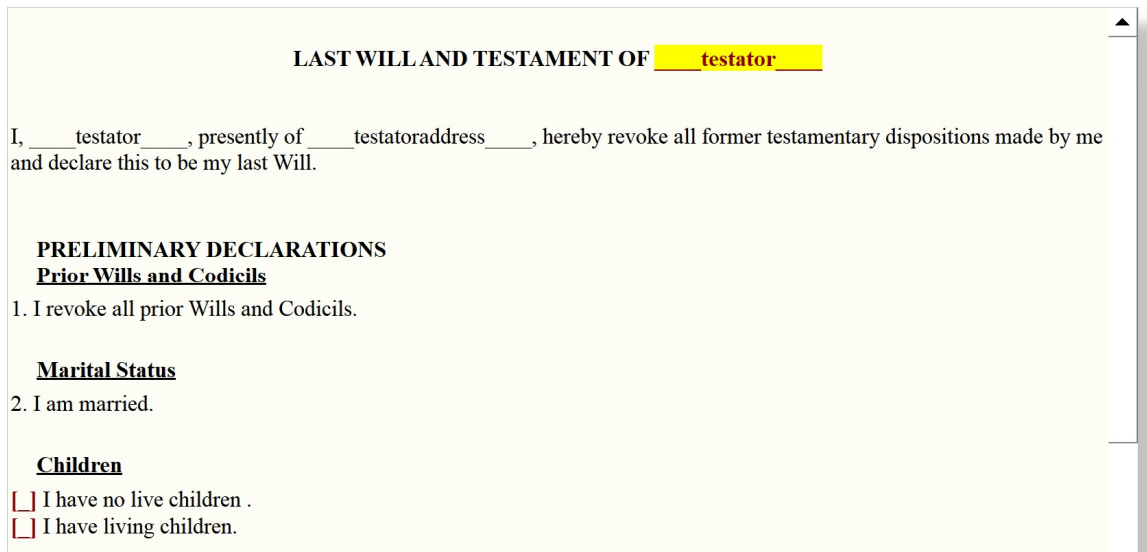


Figure 6 - Simulator's smart contract editor

The simulator's main component is a smart contract editor (Figure 6) split between legal contracts displayed in a familiar format on the left side, and tools that are used to auto-generate, check and test declarative code on the right side. This screen represents the primary mechanism by which specific legal contracts are translated into computer code.

6.2.1 Left side of the Smart Contract Editor

Figure 7 shows our example legal contract displayed from HTML. This contract is navigated by tab key which moves the **highlight** from instantiation place holder (IPH) to IPH. The embedded IPH contains attributes specifying which Smart Instantiation Editor (SIE) to use to capture input.



LAST WILL AND TESTAMENT OF testator

I, testator, presently of testatoraddress, hereby revoke all former testamentary dispositions made by me and declare this to be my last Will.

PRELIMINARY DECLARATIONS
Prior Wills and Codicils
 1. I revoke all prior Wills and Codicils.

Marital Status
 2. I am married.

Children
☐ I have no live children .
☐ I have living children.


Figure 7 - Left side of the Smart Contract Editor displaying the start of 'Will and Testament' from HTML

The entire instantiated 'Will and Testament' as displayed by our SCE is listed in Appendix C.

6.2.2 Right side of the Smart Contract Editor

The right side of the smart contract editor is comprised of four components; 1) a smart instantiation editor (SIE) which changes depending which variable is being instantiated, 2) a summary tool which groups information by entity, 3) a display that shows auto-generated code, and accepts input of events, allowing the system to run test cases, 4) a display showing results.

Component 1 operates by reacting to the type of the highlighted identifier 'testator' (Figure 7). Testator is of type 'entity' specifying that the 'entity' SIE be used as shown in Figure 8.



Select or Add Testator						
	Name	Sex	DOB	Address	Wallet	Balance
1	John Wallace	M	20088	Lot 49, Cowpasture Road, Mulgoa, 2745, NSW	walletaddress	1000000
2	Mary Wallace	F	20101	Lot 49, Cowpasture Road, Mulgoa, 2745, NSW	walletaddress	100
3	James Stewart	M	18218	McPhee & Associates, Springwood, 2777, NSW	walletaddress	350
4	Fred Wallace	M	29041	1/16 Ronald Avenue, Freshwater, 2096, NSW	walletaddress	5
5	Bec Smith	F	29432	4 Arnold Street, Cremorne, 2090, NSW	walletaddress	8
6	Sam Wallace	M	29969	3 View Street, Coffs Harbour, 2450, NSW	walletaddress	15
7	Jim Wallace	M	30376	123 Soldiers Road, Anna Bay, 2316, NSW	walletaddress	140

Figure 8 – Component 1 (top right side) - Smart Instantiation Editor for type 'entity'

All entities known to the session are displayed here, retrieved as PII from wallets invited to participate in the contract writing session. The entry selected by mouse click or touch is instantiated as testator. Note, if no wallet exists, one can be created in session by typing in name, address, etc.

Component 2 summarises instantiated data by group, and is displaying testator.

Testator					
Executor Beneficiaries Wipeout Witnesses					
Name	Sex	DOB	Address	Wallet	Balance
John Wallace	M	20088	Lot 49, Cowpasture Road, Mulgoa, 2745, NSW	walletaddress	1000000

Figure 9 - Component 2 - Entity Summary Tool

Component 3 shows declarative code generated automatically by the instantiation step in real-time. This component also allows manual entry of events, enabling code testing on screen.

```
entity(testator,john_wallace,20088,walletaddress,1000000).
```

Figure 10 – Component 3 - Auto-generated Declarative Code Display

Component 4 displays the results (answer sets) of executing declarative code generated to this point. Three code components are aggregated at this step, automatically instantiated facts as shown in component 3, logic program (listed in Appendix C), and events manually input into component 3. A complete worked example is given in Appendix C.

Test

```
message_to_human(james_stewart,john_wallace,"insert_message_to_Executor_here...")
transfer(john_wallace,walletaddress,99775,fred_wallace,walletaddress)
transfer(john_wallace,walletaddress,199550,hec_smith,walletaddress)
transfer(john_wallace,walletaddress,99775,sam_wallace,walletaddress)
transfer(john_wallace,walletaddress,199550,jim_wallace,walletaddress)
transfer(john_wallace,walletaddress,399100,may_fredricks,walletaddress)
transfer(john_wallace,walletaddress,2249,james_stewart,walletaddress)
```

Figure 11 - Component 4 (bottom right side) – Testing Tool

6.3 Description of Templates

The left side and right side of the smart contract editor are driven by templates served from a smart contract template DSL (Figure 12) in response to a request providing a contract key and version. This DSL is permissioned and managed by an authority whose role is to devise and publish error free templates (section 5.5.2). The triple received from this shared ledger is:

(Key = ContractID + Version, Text Template Group, Code Template Group).

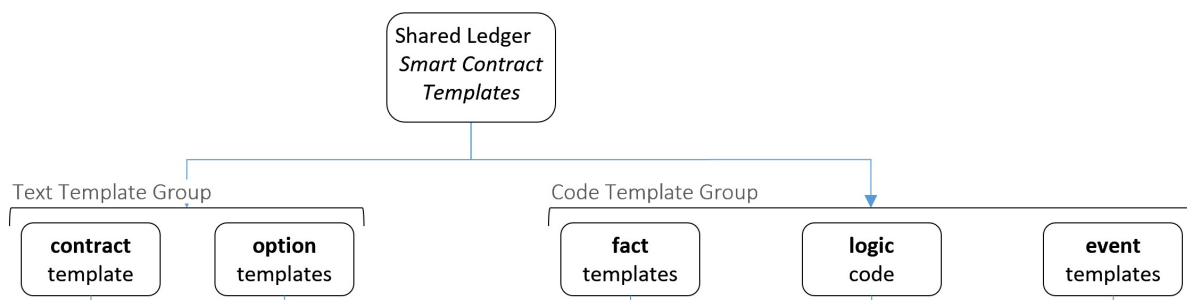


Figure 12 -Templates received from the 'smart contract template' shared ledger

6.3.1 Text Template Group

This group consists of the **Contract** template, and keyed **Option** clause templates (any number).

Contract Template

This template contains the marked-up text displayed on the left side of the smart contract editor as a legal contract. This text has instantiation placeholders (IPH) of form ____identifier____, some

of which stand alone, while others have embedded attributes (active IPHs). Active IPHs contain a Uniform Resource Locator (URL) (highlighted below), which contains a domain name customised to hold variable identifiers and attributes. The first active IPH in 'Will and Testament' (Figure 7) is:

```
<center><b>LAST WILL AND TESTAMENT OF <a name="next" href="http://_u_.iph_testator.es1">____testator____</a>
```

Key components of an active IPH are:

1. name="next" marks a scroll-to position
2. _u_.iph_testator.es1 domain name customised for this study
3. ____testator____ displayed text and instantiation variable identifier

When an active IPH gains focus, a customised attribute mark-up language (see below) located in the domain name is read to determine how to treat this IPH and its instantiation variable.

Option Templates

Options are features allowing a legal contract to have different wording depending on requirements, and appear on screen as "[]" sequences with descriptive text following. A template set can have any number of indexed option clauses. The second IPH group in Figure 7 requests a choice between two options, coded in HTML as:

```
href="http://____.iph_option01.or_">[ ]< I have no live children.  
href="http://____.iph_option02.or_">[ ]< I have living children.
```

Selecting the first rewrites this text with the text snippet at index 01, which does not contain another IPH; while selecting the second rewrites with the text snippet at index 02, which contains an active IPH that requests the names of all children to be identified.

Instantiation Placeholders (IPH)

Having IPHs with form ____identifier____, allows both active and non-active IPHs to be instantiated, often in the same step; for example, the top of Figure 7 has one active IPH and two non-active IPHs. Instantiation of ____testator____ results in all IPH containing "testator" being instantiated, achieved by exploiting the fact that 'entities' have attributes as follows:

name	John Wallace	
sex	male	
DOB	20088	days since 01Jan1900
address	Lot 49, Cowpasture Road, Mulgoa, 2745, NSW	
wallet	h6sf250dhv3y78gasmla	20-digit wallet hash address
balance	\$1,000,000	balance of entity's wallet

This allows the following IPHs to be auto-generated and instantiated:

```
____testatorname____  
____testatorsex____  
____testatorDOB____  
____testatoraddress____  
____testatorwallet____  
____testatorbalance____
```

note: ____testator____ and ____testatorname____
mean the same and both instantiate correctly

This method allows the use of one mouse click to instantiate a large number of fields in a contract.

Custom Attribute Mark-up Language

The customised domain name holds a variable identifier with attached attributes (prefix and suffix):

```
_u_.iph_testator.es1  
| | | | |  
formatting identifier processing
```

domain name customised for this study.


```

formatting= _u
123      position 1 scroll after 'tab' key ('_'=none, 't'=scroll on 'tab' key)
          position 2 text formatting      ('_'=none, 'c'=capitalise, 'u'=all upper case)
          position 3 scroll after process  ('_'=none, 'a'=scroll after update)

identifier = iph_testator
          instantiation place holder with the identifier of the variable to be instantiated.

processing= es1
123      position 1 type                  ('o'=option, 'e'=entity, 'a'=asset, 'm'=memory,
          'l'=list)
          position 2 action                ('c'=copy, 'r'=rewrite, 's'=select, '%'=associate)
          position 3 repetition            ('_'=none, '1'=once, 'n'=repeat, 'a'=all)

```

Figure 13 - Encoding used for attributes held within customised domain names

Customised domain name `_u.iph_testator.es1` specifies that testator is type 'entity', must be selected from a list once only, and formatted in upper case. This causes the 'entity' SIE (Figure 8) to display once. After selection (tap, click or entered if not in the list), the data is used to instantiate both HTML and ASP templates, then redisplay the HTML with instantiated fields highlighted in **green** allowing the user immediate feedback on the last action.

An ASP Fact template will be instantiated concurrently to encode an ASP fact as follows:

```
entity(testator, "John Wallace", 20088, "h6sf250dhv3y78gasmla", 1000000).
```

6.3.2 Code Template Group

Consists of three components; **fact** templates, **logic program**, and **event** templates. Only fact and event templates get instantiated, while the **logic program** is static for all legal contracts of type.

Fact Templates

The smart contract template shared ledger provides four fact templates for 'Will and Testament':

```
entity(iph_type, iph_entityname, iph_entityDOB, iph_entitywallet, iph_entitybalance).
asset(iph_type, iph_assetname, iph_assetdescription, iph_assetaddress).
inherits(iph_beneficiaryname, iph_percentage).
creation(iph_location, iph_date).
```

For example; attributes specify that testator's type is 'entity', so the `entity/5` ASP template is used to auto-generate the ASP fact (see Figure 10) that instantiates the testator. One IPH in this template, `iph_type`, holds special meaning indicating a meta-template; that is, the template must be instantiated for type before becoming a usable template. This is achieved by rewriting "entity" as "testator" and setting `iph_type` to "testator" giving:

```
entity(testator, iph_testatorname, iph_testatorDOB, iph_testatorwallet, iph_testatorbalance)
```

The second instantiation replaces the now recognisable IPHs with data, generating the ASP fact:

```
entity(testator, "John Wallace", 20088, "h6sf250dhv3y78gasmla", 1000000).
```

Derivative IPHs (i.e. `iph_testatorDOB`) are also instantiated, achieved by autogenerating derivative IPHs then instantiating them, similar to the IPHs for HTML:

```
iph_testatorname, iph_testatorsex, iph_testatorDOB, iph_testatoraddress, iph_testatorwallet,
iph_testatorbalance
```

Logic Program

This is ASP code that translates the legal logic of the underlying legal contract. This translation is done by hand by an authorised centralised resource at contract type development, then published to the smart contract template DSL. This code is identical for all smart contracts of a given type.

```

executable(Testator, DOD, Wallet, Residue, ExecuteDate, Executor, Costs) :- %1④
    witnessed,
    death(Testator, DOD),
    execute_will(Testator, Debt, Fees, ExecuteDate, Executor),
    Residue = Estate-Costs,
    Costs = Debt+Fees,
    entity(testator, Testator, _, Wallet, Estate),
    entity(executor, Executor, _, _, _).

```

Figure 14 - An example ASP clause from the ASP logic code for 'Will and Testament'

Figure 14 shows one clause of many that specify contract logic for 'Will and Testament'. This clause specifies that atom `executable/7` is created when all literals in the body evaluate to true, as follows:

<code>witnessed</code>	created if 2 witnesses have electronically signed
<code>death(Testator, DOD)</code>	event , notifies death of the Testator
<code>execute_will(Testator, Debt, Fees, ExecuteDate, Executor)</code>	event , execute command (Executor creates)
<code>Residue = Estate-Costs</code>	internal , calculates Residue as Estate - Costs
<code>Costs = Debt+Fees</code>	internal , calculates Costs as Debt + Fees
<code>entity(testator, Testator, _, Wallet, Estate)</code>	fact , provides the Testators details
<code>entity(executor, Executor, _, _, _)</code>	fact , provides the Executors details

where

```

witnessed :- %1①
    entity(witness1, Witness1, _, _, _),
    entity(witness2, Witness2, _, _, _),
    Witness1 != Witness2.

```

The reason why the **logic program** can be reused across all contracts of the same type is because all literals derive from either **facts** or **events**. Code behaviour varies depending on **facts** at creation and **events** deployed as transactions to the blockchain in the following period.

Event Templates

Templates in the 'event' group are used to record events subsequent to contract writing. When added to the blockchain, event transactions hold the address of the original smart contract, so when mined, code in the smart contract and all its associated events can be aggregated before execution. This mechanism allows state changes to the system over time.

The following 3 event templates are provided for 'Will and Testament', and represent the limited number of event types that can influence distribution.

```

death(iph_entityname, iph_date).
contests(iph_entityname, iph_date).
executewill(iph_testatorname, iph_debt, iph_fees, iph_date, iph_executorname).

```

This simplification is achieved by use of a meta-variable identifier 'entity' in the IPH. While there may be many different parties to a 'Will', they are all of type 'entity', and so death for any one of them can be represented by one atom `death/2`. Events can be added automatically by other smart contracts or by an authorised person (in this example, the executor) in the same way, but both need information about the current state of the system. This is achieved by reading the contract and subsequent events; for example, if a death is to be recorded, reading the `entity/5` atoms in the original smart contract and `death/2` atoms in subsequent events provides the list of candidates.

The meta-variable identifier 'entity' contained in a `death/2` IPH identifies which atom to inspect.

Our design requires an executor to authorise distribution by deploying an `executewill/5` atom.

```

executewill("john wallace", 1249, 1000, 43680, "james stewart").
death("john wallace", 43646).

```

⁴ These line numbers match clauses of the logic program listed in Appendix C.

The two events above in one transaction, triggers estate distribution when mined (section 6.4.3).

6.3.3 Variable Binding

The template designed for humans is provided in HTML, and that for the computer in ASP, with bindings between the two achieved via an IPH of form “___identifier___” for HTML and “iph_identifier” for ASP. These two forms are held together in active IPHs, for example:

```
<a name="next" href="http://_u_.iph_testator.es1">___testator___</a>
```

The instantiation process uses these two IPHs to instantiate both HTML and ASP concurrently, with the following table showing variables, their type and associated IPHs for ‘Will and Testament’.

TABLE 5 - VARIABLE TYPES WITH CORRESPONDING IPHS

variable identifier	type	HTML IPH	ASP IPH	notes
testator	entity	___testator*___	iph_testator*	substitute attribute name for *
option99	option	___option99___	iph_option99	99 has range 01..99
executor	entity	___executor*___	iph_executor*	substitute attribute name for *
beneficiary	entity	___beneficiary*___	iph_beneficiary*	substitute attribute name for *
percentage	association	___percentage___	iph_percentage	associate input with a set in memory
parent	entity	___parent*___	iph_parent*	substitute attribute name for *
sibling	entity	___sibling*___	iph_sibling*	substitute attribute name for *
witness	entity	___witness99*___	iph_witness99*	99 has range 01..02
location	text	___location___	iph_location	
today	date	___today___	iph_today	

6.3.4 Smart Instantiation Editors (SIEs)

Types are the way that functionality is introduced to the smart contract editor, with implementation involving code and creation of a SIE.

The majority of input required by most legal contracts is associated with only a few types:

asset	complex type with attributes, with data typically sourced from DSLs
association	load input against elements of a set held in memory
entity	complex type with attributes, with data typically sourced from PII held in wallets
list	type specifying input from a desktop .csv file with name “identifier.csv”
option	clause identifier used to allow selection of alternate legal clauses
text	simple type string of text
date	simple type days since 1/1/1900

Simple types are instantiated automatically; for example, timestamping the ‘Will and Testament’:

```
creation(“Sydney, Australia”, 43616).
```

Complex types require an SIE and specialised code, but creation is simplified by exploiting variable type attributes to define columns, and number of records (to a limit) for row count. It is possible some SIEs require structures other than tables, but these have not been identified in this study.

TABLE 6 - RELATIONSHIP BETWEEN VARIABLE TYPE AND SIE

variable type	SIE description and default data
asset	auto-generated table of assets sourced from owned assets listed in DSLs
association	auto-generated table that allows input against elements of a set stored in memory
entity	auto-generated table of entities sourced from authorised wallets
list	SIE not generally needed. If required, a table of data sourced from the desktop .csv file
option	simple rewrite system using keyed template HTML, with no SIE


The following are examples of SIEs for each variable type:

*SIE for Asset***Select or Add Asset**

	Description	ID	Type	Address	Valuation	valuationdate	
1	3 Bed House	1	Real Estate	Lot 49, Cowpasture Road, Mulgoa NSW 2745	940000	43603	asse
2	1 Bed Apartment	2	Real Estate	2/10 Sheffield Road, Penrith NSW 2750	350000	43632	asse
3	Mazda 3	3	Motor Ve...	2018 Mazda 3 Astina Reg: MZD-300	22500	43512	k

*SIE for Association***Enter % of Estate Inherited (must add to 100%)**

	Name	%
1	Fred Wallace	10
2	Bec Smith	20
3	Sam Wallace	10
4	Jim Wallace	20
5	May Fredricks	40

*SIE for Entity***Select or Add Testator**


	Name	Sex	DOB	Address	Wallet	Balance
1	John Wallace	M	20088	Lot 49, Cowpasture Road, Mulgoa, 2745, NSW	walletaddress	1000000
2	Mary Wallace	F	20101	Lot 49, Cowpasture Road, Mulgoa, 2745, NSW	walletaddress	100
3	James Stewart	M	18218	McPhee & Associates, Springwood, 2777, NSW	walletaddress	350
4	Fred Wallace	M	29041	1/16 Ronald Avenue, Freshwater, 2096, NSW	walletaddress	5
5	Bec Smith	F	29432	4 Arnold Street, Cremorne, 2090, NSW	walletaddress	8
6	Sam Wallace	M	29969	3 View Street, Coffs Harbour, 2450, NSW	walletaddress	15

Figure 15 - Examples of Smart Instantiation Editors of different types

6.3.5 Meta-Type, Meta-Templates and Meta-Variable Identifiers

The two ASP fact templates shown below have an IPH `iph_type` indicating a meta-template; that is, a template requiring instantiation for type before it can be used as a template.

```
entity(iph_type, iph_entityname, iph_entityDOB, iph_entitywallet, iph_entitybalance).
asset (iph_type, iph_assetdescription, iph_assetid, iph_assetvalue, iph_assetvaluationdate,
      iph_assetaddress).
```

Note that the atom name is the meta-type. When a meta-type is specified in mark-up, the system first instantiates the associated meta-template to provide a template; for example if the active IPH specifies testator is an 'entity', the `entity/5` meta-template is instantiated to provide the template for testator, as follows:

```
entity(testator, iph_testatorname, iph_testatorDOB, iph_testatorwallet, iph_testatorbalance).
```

This approach reduces coding complexity for contract creation and for recording subsequent events; for example, to process the event template for death,

```
death(iph_entityname, iph_date)
```

the system identifies meta-variable identifiers by detecting the pattern 'entity' (the name of a atom known to this contract) in the IPH. The system can then read all entity facts from the original contract, automatically producing a set of entities that can be instantiated in this IPH. This approach greatly reduces complexity; for example, one meta-template replaces seven possible templates (testator, executor, beneficiaries, children, parents, siblings, witnesses).

6.4 Low Complexity – ‘Will and Testament’ Contract

This section demonstrates application to a low complexity real-life example of an Australian legal contract; a ‘Will and Testament’ available online at (LawDepot 2019)

6.4.1 Analysis

	<p style="text-align: center;">LAST WILL AND TESTAMENT OF JOHN WALLACE</p>						
1 Who is testator.	<p>I, John Wallace, presently of Lot 49, Cowpasture Road, Mulgoa, 2745, NSW, hereby revoke all former testamentary dispositions made by me and declare this to be my last Will.</p>						
2 Who is executor.	<p>Appointment</p> <p>5. I appoint James Stewart of McPhee & Associates, Springwood, 2777, NSW, New South Wales as the sole Executor of this Will.</p> <p>Powers of My Executor</p> <p>6. I give and appoint to my Executor the following duties and powers with respect to my estate:</p>						
3 To benefit, death must occur no less than 30 days after testator.	<p>DISPOSITION OF ESTATE</p> <p>Distribution of Residue</p> <p>7. To receive any gift or property under this Will a beneficiary must survive me for thirty (30) days. Beneficiaries of my estate residue will receive and share all of my property and assets not specifically bequeathed or otherwise required for the payment of any debts owed, including but not limited to, expenses associated with the probate of my Will, the payment of taxes, funeral expenses or any other expense resulting from the administration of my Will. The entire estate residue is to be divided between my designated beneficiaries with the beneficiaries receiving a share of the entire estate residue. All property given under this Will is subject to any encumbrances or liens attached to the property.</p>						
4 % to each child. If a death then divide portion. Use same ratios.	<p>8. I direct my Executor to distribute the residue of my estate as follows (“Share Allocations”):</p> <p>a. All the residue of my estate to:</p> <p>10% to Fred Wallace of 1/16 Ronald Avenue, Freshwater, 2096, NSW 20% to Bec Smith of 4 Arnold Street, Cremorne, 2090, NSW 10% to Sam Wallace of 3 View Street, Coffs Harbour, 2450, NSW 20% to Jim Wallace of 123 Soldiers Road, Anna Bay, 2316, NSW 40% to May Fredricks of 7 Payne Road, Castle Hill, 2154, NSW for their own use absolutely.</p>						
5 If all beneficiaries die then equal split between parents and siblings.	<p>Wipeout Provision</p> <p>9. I HEREBY DIRECT that the residue of my estate or the amount remaining be divided into one hundred (100) equal shares and to pay and transfer such shares as follows:</p> <p>a. 100 shares to be divided equally between my parents:</p> <p>Tom Wallace of 54 Bland Rd, Springwood NSW, 2777, NSW Aida Wallace of 54 Bland Rd, Springwood NSW, 2777, NSW and siblings: Anne Patrick of 16 Park Avenue, Cammeray, 2450, NSW Ines Brown of 13 Raymond Road, Neutral Bay, 2089, NSW Steve Wallace of 40 Euroka Street, North Sydney, 2154, NSW or survivors thereof, for their own use absolutely, if all or any of them are then alive.</p>						
6 Any contestor is cut out.	<p>GENERAL PROVISIONS</p> <p>No Contest Provision</p> <p>11. If any beneficiary under this Will contests in any court any of the provisions of this Will, then each and all such persons shall not be entitled to any devises, legacies, bequests, or benefits under this Will or any codicil hereto, and such interest or share in my estate shall be disposed of as if that contesting beneficiary had not survived me.</p> <p>Severability</p> <p>12. If any provisions of this Will are deemed unenforceable, the remaining provisions will remain in full force and effect.</p>						
7 date and place	<p>We declare under penalty of perjury under the laws of the Commonwealth of Australia that the foregoing is true and correct this Thu, 04 Jul 2019, at Sydney, New South Wales.</p>						
8 two witnesses	<p>Signed by John Wallace in our presence and then by us in their presence.</p> <table> <tr> <td>Signature walletaddress</td> <td>Signature walletaddress</td> </tr> <tr> <td>Name Brian Bellhaus</td> <td>Name Margaret Talbot</td> </tr> <tr> <td>Address McPhee & Associates, Bayview, 2104, NSW</td> <td>Address 14 Prince Street, Underdale, 5032, SA</td> </tr> </table>	Signature walletaddress	Signature walletaddress	Name Brian Bellhaus	Name Margaret Talbot	Address McPhee & Associates, Bayview, 2104, NSW	Address 14 Prince Street, Underdale, 5032, SA
Signature walletaddress	Signature walletaddress						
Name Brian Bellhaus	Name Margaret Talbot						
Address McPhee & Associates, Bayview, 2104, NSW	Address 14 Prince Street, Underdale, 5032, SA						

Figure 16 - Analysis of the legal logic in "Will and Testament"

6.4.2 ASP Code for Will and Testament

Encoding Legal Clauses

The ASP encoding for each of the numbered legal clauses in Figure 16 is described below, and identified as fact, logic program or event. Note, not all legal clauses require translation.

1. Who is testator? **fact.** ASP encoded from instantiating `entity/5` at contract creation:

```
entity(testator, john_wallace, 20088, walletaddress, 1000000).
```

2. Who is executor? **fact.** ASP encoded from instantiating `entity/5` at contract creation:

```
entity(executor, james_stewart, 18218, walletaddress, 350).
```

3. To benefit, death... **logic program.** ASP encoded by hand at type creation:

```
executable(Testator, DOD, Wallet, Residue, ExecuteDate, Executor, Costs) :- %1②
    witnessed,
    death(Testator, DOD),
    execute_will(Testator, Debt, Fees, ExecuteDate, Executor),
    Residue = Estate-Costs,
    Costs = Debt+Fees,
    entity(testator, Testator, _, Wallet, Estate),
    entity(executor, Executor, _, _, _).

disqualifying_death(Entity, DaysAfter) :- %1③
    DaysAfter = Date-DOD,
    DaysAfter < 30,
    death(Entity, Date),
    executable(Testator, DOD, _, _, _, _, _).

qualifying_beneficiary(Entity, Wallet) :- %1④
    entity(beneficiary, Entity, _, Wallet, _),
    not disqualifying_death(Entity, _),
    not contests(Entity, _).
```

Clause 1② tests if conditions are right to execute the Will, while 1③ determines if death was within 30 days of testator, and 1④ determines disqualification by death or contesting.

4. % to each child. ... **facts, logic program and events:**

facts. ASP encoded by instantiating `entity/5` and `inherits/2` at contract creation:

```
entity(beneficiary, fred_wallace, 29041, walletaddress, 5).
entity(beneficiary, bec_smith, 29432, walletaddress, 8).
entity(beneficiary, sam_wallace, 29969, walletaddress, 15).
entity(beneficiary, jim_wallace, 30376, walletaddress, 140).
entity(beneficiary, may_fredricks, 31310, walletaddress, 24).
inherits(fred_wallace, 10).           %Fred gets 10% of the estate
inherits(bec_smith, 20).             %Bec gets 20% of the estate
inherits(sam_wallace, 10).           %Sam gets 10% of the estate
inherits(jim_wallace, 20).           %Jim gets 20% of the estate
inherits(may_fredricks, 40).         %May gets 40% of the estate
```

logic program. ASP encoded by hand at type creation:

Algorithm: To calculate the % of estate to pay to each qualifying beneficiary, disqualified beneficiaries must be adjusted for. The adjustment ratio = $\text{sum}(\text{orig}\%)/\text{sum}(\text{qual}\%)$.

TABLE 7 - CALCULATING THE ADJUSTMENT RATIO

beneficiary	orig%	qual%	new%	
fred_wallace	10			fred dies early
bec_smith	20	20	25	
sam_wallace	10			sam dies early
jim_wallace	20	20	25	
may_fredricks	40	40	50	
Totals	100	80	Ratio = 1.25	100

This is achieved with the following code:

```
orig_percent(Entity, Percent) :-                               %2①
    entity(beneficiary, Entity, _, _, _),
    inherits(Entity, Percent).

qual_percent(Entity, Percent) :-                               %2②
    entity(beneficiary, Entity, _, _, _),
    inherits(Entity, Percent),
    qualifying_beneficiary(Entity, _).

sum(original, Sum) :-                                         %2③
    Sum = #sum{ Percent, Entity : orig_percent(Entity, Percent) }.

sum(qualifys, Sum) :-                                         %2④
    Sum = #sum{ Percent, Entity : qual_percent(Entity, Percent) }.

adjust(Ratio) :-                                              %2⑤
    not wipeout,
    Ratio = (OrigSum*1000/QualSum),
    sum(original, OrigSum),
    sum(qualifys, QualSum).
```

Note: The *1000 keeps the calculation in integer space (standard ASP only deals with integers).

events. ASP encoded by instantiating of `death/2` at some time after contract creation:

```
death(fred_wallace, 43675)           %death of Fred on date
death(sam_wallace, 43675)            %death of Sam on date
```

5. If all beneficiaries ... facts, logic program and events:

facts. ASP encoded by instantiating `entity/5` and `inherits/2` at contract creation:

```
entity(parent, tom_wallace, 12965, walletaddress, 200).
entity(parent, aida_wallace, 13378, walletaddress, 200).
entity(sibling, anne_patrick, 19378, walletaddress, 9000).
entity(sibling, ines_brown, 20255, walletaddress, 5000).
entity(sibling, steve_wallace, 21459, walletaddress, 6000).
```

logic program. ASP encoded by hand at type creation:

A situation where all specified beneficiaries have died before the 30-day limit after the Testator’s death is referred to as a “wipeout”. In this instance, the Will specifies equal distribution to the Testator’s surviving parents and siblings. The wipeout condition is detected by the absence of the atom `qualifying_beneficiary/2` in clause 1⑤. The two lines of code ensure the existence of either ‘wipeout’ or ‘-wipeout’ but not both.

```
-wipeout :-                                                    %1⑤
    qualifying_beneficiary(Entity, Wallet).

wipeout :-                                                      %1⑥
    not -wipeout.
```

Distribution on wipeout is “equal portions”, achieved with the following code:

```
qualifies_on_wipeout(Entity, Wallet) :-                        %3①
    wipeout,
    entity(parent, Entity, _, Wallet, _),
    not disqualifying_death(Entity, _),
    not contests(Entity, _).

qualifies_on_wipeout(Entity, Wallet) :-                        %3②
    wipeout,
    entity(sibling, Entity, _, Wallet, _),
    not disqualifying_death(Entity, _),
    not contests(Entity, _).

wipeout_count(Count) :-                                        %3③
    Count = #count{ Entity, Wallet : qualifies_on_wipeout(Entity, Wallet) }.
```

This code creates a set of parents and siblings which qualify, counts them then distributes Residue/Count to each.

events. ASP encoded by instantiating of `death/2` at some time after contract creation:

```
death(fred_wallace, 43675)           %death of Fred on date
death(hec_smith, 43675)             %event, death of Bec on date
death(sam_wallace, 43675)           %event, death of Sam on date
death(jim_wallace, 43675)           %event, death of Jim on date
death(may_fredricks, 43675)         %event, death of May on date
```

6. Any contestor is ... **logic program** and **events**:

logic program. ASP encoded by hand at type creation:

Dealing with contesters is achieved by simply including detection of a `contests/2` atom for a beneficiary when constructing the set of qualifying beneficiaries.

```
qualifying_beneficiary(Entity, Wallet) :-                               %1④
    entity(beneficiary, Entity, _, Wallet, _),
    not disqualifying_death(Entity, _),
    not contests(Entity, _).
```

events. ASP encoded by instantiating of `contests/2` at some time after contract creation:

```
contests(sam_wallace, 43625).           %Sam contests the will, so is disqualified
```

7. date and place ... **facts**:

facts. ASP encoded by instantiating `entity/5` for the two witnesses at contract creation:

```
created(sydney, 43620).                 %location and date (days since 1/1/1900)
```

8. two witnesses ... **facts** and **logic program**:

facts. ASP encoded by instantiating `entity/5` for the two witnesses at contract creation:

```
entity(witness1, brian_bellhaus, 22692, walletaddress, 800).
entity(witness2, margaret_talbot, 24042, walletaddress, 16).
```

logic program. ASP encoded by hand at type creation:

All entities involved have to authorise visibility to the contract writing session by signing in with a private key, a step with the same validation strength as a signature. This allows a mechanism whereby the contract only executes if there are two different entities assigned as witnesses.

```
witnessed() :-                                                         %1①
    entity(witness1, Witness1, _, _, _),
    entity(witness2, Witness2, _, _, _),
    Witness1 != Witness2.
```

Encoding Distribution

Once the above logic is in place, the three encodings of the distribution atom, `transfer/5` can be driven. The three encodings cover the three different types of distribution; 1) normal payout to beneficiaries, 2) payout in case of wipeout, and 3) payout of debt and fees.

Normal payout

```
transfer(Testator, SourceWallet, InheritAmt, Beneficiary, Wallet) :-    %2⑥
    not wipeout,
    executable(Testator, DOD, SourceWallet, Residue, ExecuteDate, Executor, Costs),
    qualifying_beneficiary(Beneficiary, Wallet),
    InheritAmt = ((Residue * Percent/100) * Ratio)/1000,
    inherits(Beneficiary, Percent),
    adjust(Ratio).
```

Wipeout payout

```
transfer(Testator, SourceWallet, InheritAmt, Entity, Wallet) :-         %3④
    wipeout,
    executable(Testator, DOD, SourceWallet, Residue, ExecuteDate, Executor, Costs),
    qualifies_on_wipeout(Entity, Wallet),
    InheritAmt=Residue/Count,
    wipeout_count(Count).
```

Debt and Fees payout

```
transfer(Testator, SourceWallet, Costs, Executor, Wallet) :-           %4①
```



```
executable(Testator, DOD, SourceWallet, Residue, ExecuteDate, Executor, Costs),
entity(executor, Executor, _, Wallet, _).
```

The answer set produced by the above code in response to events, is a list of transfers:

```
transfer(john_wallace, walletaddress, 99775, fred_wallace, walletaddress)
transfer(john_wallace, walletaddress, 199550, bec_smith, walletaddress)
transfer(john_wallace, walletaddress, 99775, sam_wallace, walletaddress)
transfer(john_wallace, walletaddress, 199550, jim_wallace, walletaddress)
transfer(john_wallace, walletaddress, 2249, james_stewart, walletaddress)
```

These transfers are translated by an interface into instructions that can be executed by Ethereum to perform actual funds transfer. A further feature allows e-mails to be generated.

```
message_to_human(james_stewart, john_wallace, "insert messages to the Executor here...")
```

This atom holds the promise of sending specified clauses from the legal document to specific entities in response to specific events, possibly a benefit if long time periods have elapsed between contract creation and a triggering event. For example; on the death of the Testator the system could send clause 6 (Powers of My Executor) of the ‘Will and Testament’ to the Executor. A full code listing is provided in Appendix C – Low Complexity – ‘Will and Testament’.

6.4.3 Test Cases

Testing this code is greatly simplified relative to Solidity because the only consideration is the logic expressed. Further, the combinations of facts and events are of a manageable size because only 6 atoms (effectively 5) are involved, so an exhaustive testing strategy is practicable.

TABLE 8 – SAMPLE TEST CASES FOR THE "WILL AND TESTAMENT" SMART CONTRACT EXAMPLE

Case	Description
0	Testator writes a Will on 2Jun2019 (43616 days from 1/1/1900). No activity generated.
1	Death of Testator recorded on 2Jul2019 (43646 days from 1/1/1900). No activity generated because Executor has not authorised execution of Will.
2	Executor approves distribution on 5Aug2019. All 5 children are alive, so distribution is: Fred 10%, Bec 20%, Sam 10%, Jim 20%, May 40%. Note: Debts (\$1,249) and fees (\$1,000) reduce the residue to \$997,751.
3	Sam contests the Will which is recorded by the Executor. Consequently, Sam is cut out of the Will (clause 11), and his share reallocated to the other children.
4	Children and Mother wiped out in plane crash on way to funeral 29 days after the Testator's death, so no longer qualify as beneficiaries. The Residue will be shared between Parents (Tom) and Siblings (Anne, Ines, Steve).
5	Child Fred Wallace was not on the plane, but dies in a car accident 2 days later (31 days), so still qualifies as a beneficiary (clause 7). The Will rules say Fred's Estate gets the entire Residue. Notes: 43646 = 02Jul2019 death of Testator 43675 = 31Jul2019 29 days wipeout of Mother and 4 of 5 children 43677 = 02Aug2019 31 days death of last child 43680 = 05Aug2019 34 days Will is executed

Code is tested with test cases by entering **event** ASP clauses manually into SCE component 3 (see section 6.2.2). The **event** ASP clauses used in the above test cases are provided in Appendix C along with the answer sets obtained. Appendix C also lists a comprehensive set of test cases, derived by reducing an exhaustive decision table.

Figure 17 shows components 3 and 4 on the right side of the simulator, after test case 3 events have been entered manually into component 3, then executed by clicking the ‘Test’ button.



Figure 17 - Result of test case 3 displayed by the smart contract editor

6.4.4 Observations

A review of the logic and performance clauses in this contract reveal all logic to be deductive and performance to be straightforward. The overall purpose is to distribute an estate at death.

TABLE 9 – AUTOMATABILITY FRAMEWORK APPLIED TO A ‘WILL AND TESTAMENT’ LEGAL CONTRACT

contract	legal reasoning	performance	consideration	automatability
Legal Will and Testament	deductive	money transfer, e-ledger asset transfer, physical transfer, Human-in-the-Loop	money	moderate

We note that some legal contract text is irrelevant to encoding legal logic; for example, under executor (see Appendix B or C), a disambiguation is provided in clause 4 explaining the term “my Executor” can refer to singular or plural, male or female; a distinction not made in the encoding.

Some conditions are best coded grouped with other logic; for example, clause 6 (contester) is best encoded together with other code determining if a beneficiary still qualifies. While undecided and requiring more investigation, this issue tends to support the Ricardian approach of having both text and a code, rather than including declarative code as mark-up within the text document.

6.5 Comparison with a Solidity Implementation

We compare our ASP implementation with a probable Solidity implementation over five programming activities; 1) coding the smart contract type (logic program), 2) testing this smart contract type, 3) auto-generating the instantiation at contract creation, 3) testing the created smart contract, 5) programmatically interacting with the smart contract after deployment.

6.5.1 Coding the Smart Contract Type

Solidity

Coding a smart contract with Ethereum requires building a number of components; 1) coding of the web3.js API using JavaScript specifying the RPC providers and proving access to the smart contract’s functions via object web3.eth.Contract, 2) coding of the application binary interface

(ABI), an object that contains a detailed description of smart contract functions, methods and their arguments, that allows them to be called in bytecode (IBM 2020), 3) code for loading instantiation data, 4) the smart contract in Solidity, then compiled into bytecode, 5) callback and fallback functions, 6) event watching code. It is estimated that more than 100 lines of Solidity would be required to code the 'Will and Testament' example used in this chapter. As this task is to code a contract type not the actual smart contract, some further complexity is expected from packaging and delivering this template code to the node where the actual smart contract is assembled.

ASP

Our approach using ASP also requires the web3.js API which embeds it into the Ethereum environment, however it now simply supplies the smart contract code held as a payload on the blockchain, to an ASP solver (clingo) and translates and executes actions coded in the resulting answer set, removing the need for the ABI. Any custom code required for executing actions required by a Solidity implementation will also be required by the ASP implementation. Callback functions and event watching code are also still required.

The 'Will and Testament' was coded in 20 lines of ASP code (see Appendix C - Logic Program). Program execution involves supplying this code to an ASP solver built into Ethereum.

ASP Advantages

Elimination of the ABI, data upload methods and interaction functions are clear advantages to ASP:

1. ASP does not need an ABI because contract state change is generated by adding lines of code in the form of transactions to the blockchain. This is possible because ASP code is order independent; that is, lines of code can be added in any order without affecting the computation. Further, requests for contract state information is achieved by running queries over the code.
2. Instantiation with ASP is achieved by auto-generating ASP code, eliminating the need for methods that load instantiation data.
3. ASP does not require the custom coding of interaction functions, interaction is achieved with ASP code in the form of events and queries.
4. It is easier to read business logic from ASP, and consequently easier to programmatically read.
5. ASP allows the program specification to be changed programmatically by loading more code as a transaction after deployment, a feature not available with Solidity. This feature of ASP is known as elaboration tolerance (McCarthy 1988) and should ease ongoing maintenance costs.
6. Testing effort for ASP is reduced relative to a procedural language, primarily because control code is not tested, however a further effect is visible from the reduction in lines of code.
7. ASP takes less lines of code and less blockchain space – Appendix A shows a Solidity program with its roughly equivalent ASP program at approximately a quarter the size. Further, Solidity is compiled into bytecode which multiplies its size a number of times. It is possible that ASP uses as little as one tenth the space on blockchain compared to Ethereum bytecode.

6.5.2 Testing the Smart Contract Type

Solidity

Because both logic and control code needs to be tested, the Solidity test suite is by definition larger than the ASP test suite.

ASP

An exhaustive test suite is given in section 6.4.3 and Appendix C.

ASP Advantages

Demonstrably simpler to test (see the Comprehensive Test Case Suite in Appendix C), because control code does not need to be tested. A further consideration is that null cases do not need to be tested because the logic program template is supplied with dummy variables (instantiation place holders), which do not affect results because they are never resolved by the solver.

6.5.3 Auto-generating the Instantiation at Contract Creation**Solidity**

The approach most likely to be used is the auto-generation of data which is then uploaded to the smart contract via custom coded methods using some data interchange format like JSON.

ASP

Only ASP code in the form of a head without a body (fact) is required to be auto-generated. This format is very concise, approaching a minimum solution, and easy to pattern with a template.

ASP Advantages

ASP avoids the need to code data upload methods.

6.5.4 Testing the created Smart Contract**Solidity**

A Solidity implementation that achieves contract creation with untrained users by using templates and code auto-generation, is likely to require some testing before the smart contract is deployed. The best approach may be to download a tested bytecode program that implements the contract type, then upload data to it prior to its deployment on the blockchain. Even with this approach, prudence suggests some testing, a hurdle when unskilled users are used. Possible solutions have drawbacks; 1) formally verifying the contract type bytecode is expensive, 2) reducing the scope and complexity of the contract type would multiply the number of contract types required. Our ASP implementation does not suffer these drawbacks.

ASP

The exhaustive test suite (see section 6.4.3, Appendix C) shows that it is possible for comprehensive testing by the central authority to remove all errors in the templates used for smart contract creation. This enables the testing task to be modified into a simulation that shows users the outcomes of a range of different event scenarios, a task more focused on user understanding. While it may be possible to provide the same feature with a procedural language, the reality is that with current technologies finding and removing bugs is the necessary focus.

ASP Advantages

The area where ASP appears to have its greatest advantage over Solidity, because ASP makes it practical to fully test the smart contract type supplied as a template thus facilitating the achievement of a system that can be used by untrained users. Avoiding testing at this stage with Solidity is more difficult and likely to suffer drawbacks relative to an ASP implementation.

6.5.5 Programmatically interacting with the Smart Contract after Deployment

Solidity

Solidity requires that an ABI, and interaction functions be coded with the smart contract to allow smart contract functions to be called. This functionality cannot be changed once deployed.

ASP

ASP does not require an ABI as state changes are made by adding transactions, or interaction functions. Adding transactions allows state changing events to be recorded, and also allows changes to program specification to be loaded, a feature of ASP called elaboration tolerance (McCarthy 1988). Queries on the state of the smart contract are in the form of queries on the aggregated code.

ASP Advantages

Removing the ABI and avoiding coding of interaction and query functions saves programming effort and reduces complexity, reducing the possibility of undetected security gaps. Being able to change the specification of the code by adding logic code, affords greater flexibility in the ongoing management of smart contracts. Queries on the state of the smart contract are in the form of queries on the aggregated code.

6.6 Mid Complexity – ‘Real Estate Sale’ Contract

This section applies our approach to a representative Australian real estate sale contract available online (FindLegalForms 2019). We estimate this contract represents a mid-point in contract complexity with a length of 21 pages.

Residential Real Estate Sale Contract

PARTIES

This Residential Real Estate Sale Contract (the “Contract”) is made by and between **seller** (“Seller”), and **buyer**, (“Buyer”) (Buyer and Seller may be collectively referred to as the “Parties”).

PROPERTY TO BE SOLD

Seller shall sell and Buyer shall buy the following property and improvements thereon (the "Property") commonly known as: **property**
 propertydescription (Common Description)

☐ The legal description of the property is as follows:
☐ The legal description of the property is given in Attachment 1:

The sale of the Property shall include all buildings and improvements on the property and all right, title and interest of Seller in and to adjacent streets, roads, alleys and rights-of-way. The sale of the Property shall also include the following fixtures and personal property associated with the Property (unless specifically excluded below), all of which (if any) are owned by Seller free and clear of all liens and encumbrances, including: gas heaters; propane tanks (including propane if owned); central heating, ventilation and air conditioning equipment and fixtures; sump pumps; attached TV antennas and cables; lighting and light fixtures; plumbing equipment and fixtures; attached mirrors; linoleum; wall-to-wall carpet; window and porch shades; blinds; storm windows and doors; screens; curtain and drapery rods; awnings; automated garage door openers and remote control units; keys; attached humidifiers; attached outside cooking units; attached fireplace screens and/or glass doors; attic and ceiling fans; built-in kitchen appliances; and: **included** (Included Items)

The following items are specifically excluded from this contract and shall not be transferred to Buyer as a part of the Property:
excluded (Excluded Items)

PURCHASE PRICE

The Purchase Price for the Property is: AUD **price** (the “Purchase Price”).

PAYMENT
(select and fill out the one(s) that apply)

☐ Buyer shall pay an earnest money deposit
☐ Seller agrees to provide financing
☐ Buyer will pay the balance on or before closing

TOTAL
 (Earnest Money + Seller financing + cash balance at Closing)

Figure 18 - First page of a Real Estate Sale Legal Contract displayed by the simulator

This contract manages the process of sale of real estate between two parties, providing details of the two parties, the property, price, inclusions and exclusions, and target dates. The method of payment also specifies the process including deposits (a performance on the buyer) and credit

checks (a performance by the seller), a pattern repeated for a most other deliverables required to validate the contract, from inspection reports and surveys (buyer orders while seller ensures compliance), to inspection of property and equipment, specification about leaving on and paying for utilities, keeping insurances current and warranties (seller performance), which of the parties pays for costs like transfer taxes, stamp duties, repairs, administrative costs, dispute resolution procedure, and even adjustments after closing date due to undetermined tax rates etc. This detail appears to offer opportunities for software that assists managing details, like check-lists, schedule management, automatic management of fees and penalties for missing deadlines etc.

A review of the logic and performance clauses in this contract reveal all logic to be deductive with performance required of both parties. Overall purpose is transfer of ownership for a consideration.

TABLE 10 - AUTOMATABILITY FRAMEWORK APPLIED TO REAL ESTATE SALE CONTRACT

contract	legal reasoning	performance	consideration	automatability
Real Estate Sale	deductive	money transfer, e-ledger asset transfer, physical transfer, Human-in-the-Loop	money	moderate

The preliminary evaluation framework we previously developed indicates this contract can be made into a smart contract with our approach, and marking up the original paper contract took only a few hours with a normal text editor. However in contrast to the ‘Will and Testament’ example, automation of management processes seems to provide the strongest benefit.

For example, on the third page a list of possible inspections, reviews, surveys, approvals and other conditions is listed that require satisfactory completion for the sale contract to hold.

ADDITIONAL CONTINGENCIES

This Contract is also contingent upon satisfactory completion of the following items:
(select those that apply)

- ☐ A contractor's inspection of the property at Buyer's expense resulting in a report satisfactory to Buyer
- ☐ An architect's inspection of the property at Buyer's expense resulting in a report satisfactory to Buyer
- ☐ An environmental inspection of the property at Buyer's expense resulting in findings satisfactory to Buyer
- ☐ A review of use requirements affecting the property at Buyer's expense resulting in findings satisfactory to Buyer
- ☐ A stake survey or survey report at Buyer's expense resulting in findings satisfactory to Buyer
- ☐ Approval of the title insurance commitment by Buyer's lawyer
- ☐ Buyer obtaining and acquiring all the necessary approvals and permits to use the property
- ☐ The sale and closing of Buyer's other property.

Figure 19 - Possible automated management of contract contingencies

The smart contract could be programmed so that it only executes if the selected contingencies are satisfactorily completed. Consider item 1 in Figure 19; it is possible to imagine a system where the contractor's system signals completion of the inspection to the buyer who after review, signals acceptance to this smart contract.

In summary, this mid complexity legal contract is amenable to conversion to a smart contract via our approach and provides a number of automation opportunities. A list of the variables in this contract is provided in Appendix D, highlighting that many other opportunities for automation exist for this contract.

6.7 Complex – ‘CEO Employment’ Contract

This section investigates application of our approach to a representative Australian CEO employment contract available online at (City of Melbourne 2018). We estimate this contract is likely less complex (at 9 pages) than a typical CEO employment contract which could include details of share and option holdings and performance incentives. Nonetheless, this contract illustrates use of legal logic other than the deductive logic used by our other examples. A quick inspection of the ‘Position Role and Responsibilities’ section reveals the challenge.

Contract of Employment

BETWEEN: **nominee**

AND: **council**

DATED: **Thu, 17 Oct 2019**

Recitals

- A. The Council wishes to appoint its Chief Executive Officer for a maximum term in accordance with sections 94 and 95A of the Local Government Act 1989.
- B. This Contract establishes the terms and conditions of employment between the Council and the Chief Executive Officer.
- C. Unless there are any subsequent changes which are agreed in writing between the parties, this Contract contains the entire understanding between the parties as to the employment relationship.
- D. The parties intend this Contract to take effect as a maximum term contract for the purposes of section 95A of the Act.

1. Definitions

1.1. In this Contract:

- 1.1.1. **'the Act'** means the Local Government Act 1989.
- 1.1.2. **'Council Information'** means any trade secret, financial, business, confidential or other information belonging to or concerning the legislative, administrative, operational or business affairs of or relating to the Council or any of its municipal or trading enterprises acquired by virtue of the Chief Executive Officer's employment or any other relationship with the Council, or any of its municipal or trading enterprises.
- 1.1.3. **'KPIs'** mean the key performance indicators agreed and varied from time to time pursuant to Clause 6.
- 1.1.4. **'Performance Criteria'** means the responsibilities identified in Schedule 1, key performance indicators contained in the Council Plan and the Council's Annual Plans and the KPIs.
- 1.1.5. **'TEP'** means the Total Employment Package in Schedule 2.

2. Recitals and Interpretation

- 2.1. The Recitals are accurate and form part of this Contract.
- 2.2. Where required, a reference in this Contract to a Schedule, KPI or performance review process is a reference to that Schedule, KPI or performance review process as varied.

3. Position Role and Responsibilities

3.1. The Chief Executive Officer will:

- 3.1.1. faithfully and diligently serve the interests of Council and be accountable to Council
- 3.1.2. satisfactorily carry out the responsibilities set out in Schedule 1
- 3.1.3. take all reasonable steps to meet:

Figure 20 - CEO Employment Contract showing some ‘Performance’ clauses displayed by the simulator

Clause 3.1. requires that in exchange for Salary and Bonuses, the Chief Executive Officer will:
 “faithfully and diligently serve the interests of Council and be accountable to Council”

To interpret this, we use cultural knowledge to recognise that the CEO must develop a reputation with a majority of senior staff and board members, that he/she is faithfully and diligently serving the interests of Council. The logic used is ‘reasoning by principle’; that is, using cultural norms to determine what behaviour qualifies as “faithfully and diligently serve”.

Clearly no algorithm can determine conformance to ‘faithfully and diligently’, however this could be overcome via a ‘Human-In-The-Loop’ feature that allows a human to enter a rating or even enter data into a tool that produces a rating; for example, 360 degree employee evaluations (Cognology 2019).

TABLE 11 – AUTOMATABILITY FRAMEWORK APPLIED TO A CEO EMPLOYMENT CONTRACT

contract	legal reasoning	performance	consideration	automatability
CEO Employment Contract	deductive, analogy, principle, ...	1. faithfully and diligently serve the ... 2. satisfactorily carry out the ... 3. take all reasonable steps to meet ... 4. diligently exercise delegations as ... 5. promote the best interests of the	salary & bonuses	difficult, not cost effective

But 3.1.1. is one of many such clauses and producing a tool for every clause is impractical. Practicality is also determined by volume, and in this case the employer can be identified as a large city council, meaning the contract is likely to apply to only a few dozen municipalities at most, making automation uneconomic, and defining a boundary.

Automatability though is not the only benefit; for example Figure 20 above displays the first page of the ‘CEO Employment Contract’ in our simulator. This is because it is straight-forward to digitize existing paper legal contracts by adding mark-up, the starting point for our method. Given the low cost of marking up existing documents, there may some benefit from the simpler display, simpler user interaction, and improved storage and retrieval aspects of our approach, and this could be established by future work. As regarding our core focus, that of identifying ways of improving usability and reliability of smart contracts via use of declarative code, legal contracts of this type fall beyond the boundary of what we consider a practical application.

Chapter 7 Results

7.1 Coding a Smart Contract with ASP is possible

This study implements a non-trivial legal document (low complexity ‘Will and Testament’) using an existing pure declarative language (ASP), with a demonstration of functionality provided by a comprehensive test case suite (see Appendix C) serving as proof-of-concept. Key test cases are used as examples (see Section 6.4.3) and fully expanded in Appendix C, because a full listing of all the test files and results (answer sets) is impractical. Testing can be performed with both the simulator (see Section 6.4.3), and a CLI environment also illustrated in Appendix C.

We are not aware of any work that implements smart contracts with the ASP language. We are also not aware of any work that uses marked-up status-quo format legal documents in a user interface that auto-generates smart contracts with ASP code.

7.2 Extensibility

We applied our approach to three real-life Australian legal documents of different types and increasing complexity, and determined that two of these documents were amenable to full translation, while the third contained logic and performance clauses that are beyond the techniques that we use. Of the two amenable contracts, we fully translated the simpler of the two (‘Will and Testament’), and fully marked-up and analysed the more complex (‘Real Estate Sale’ contract) to understand how to code the logic program and determine any benefits.

While attempting to understand what is automatable, we identified dependence on type of legal reasoning (see section 2.2) and type of ‘performance’ (see Table 1, p6), as shown in our preliminary Table 12. While this list may not be exhaustive, the implication is that any legal document that only uses deductive reasoning, has tangible performance and a positive cost/benefit evaluation, is convertible to a smart contract using our method.

TABLE 12 - LEGAL CONTRACT AUTOMATABILITY BY CONTRACT TYPE, ‘LEGAL REASONING’ AND ‘PERFORMANCE’ REQUIRED

contract type	legal reasoning	performance required	consideration	automatability
Financial Contracts	deductive	money transfer, e-ledger asset transfer, rates input	money	easy
B2C e-Commerce Contracts	deductive	e-ledger asset transfer, RFID input, physical transfer, Human-in-the-Loop	money	moderate
Legal Will and Testament	deductive	money transfer, e-ledger asset transfer, physical transfer, Human-in-the-Loop	money	moderate
Real Estate Sale	deductive	money transfer, e-ledger asset transfer, physical transfer, Human-in-the-Loop	money	moderate
CEO Employment Contract	deductive, analogy, principle, etc ...	1. faithfully and diligently serve the ... 2. satisfactorily carry out the ... 3. take all reasonable steps to meet ... 4. diligently exercise delegations as ... 5. promote the best interests of the	salary & bonuses	difficult, possibly not cost effective

A further implication is that any part of a legal document that uses deductive reasoning and has tangible performance, is automatable, opening up the possibility of holding all legal documents in marked-up format with partial automation implemented where possible.

We found that the level of automation and advantages delivered differs by legal document; for example, the key advantage of automating ‘Will and Testament’ is the ability to specify at one time and place all the conditions and calculations required to carry out that ‘Will and Testament’ and then have those specifications executed automatically over time. In contrast, the key advantage of automating ‘Real Estate Sale’ is the assistance the logic program affords in managing the somewhat complex sale process; for example, managing the many reports and inspections required and fees to be paid. We also converted a third contract (a ‘CEO Employment’ contract), and discovered that while very little automation was possible because of the complex legal logic and nature of contract performance required; the ease of text mark-up, improvements to usability, explainability and storage and retrieval provided sufficient benefits for applicability to be reconsidered.

7.3 Using ASP facilitates achieving improved utility in Smart Contracts

In section 6.5 we compare our ASP implementation with a probable Solidity implementation over five programming activities; 1) coding the smart contract type (logic program), 2) testing this smart contract type, 3) auto-generating the instantiation at contract creation, 3) testing the created smart contract, 5) programmatically interacting with the smart contract after deployment and identified the following advantages:

7.3.1 Coding the Smart Contract Type

Elimination of the ABI, data upload methods and interaction functions are clear advantages to ASP:

1. ASP does not need an ABI because contract state change is generated by adding lines of code in the form of transactions to the blockchain. This is possible because ASP code is order independent; that is, lines of code can be added in any order without affecting the computation. Further, requests for contract state information is achieved by running queries over the code.
2. Instantiation with ASP is achieved by auto-generating ASP code, eliminating the need for methods that load instantiation data.
3. ASP does not require the custom coding of interaction functions, interaction is achieved with ASP code in the form of events and queries.
4. It is easier to read business logic from ASP, and consequently easier to programmatically read.
5. ASP allows the program specification to be changed programmatically by loading more code as a transaction after deployment, a feature not available with Solidity. This feature of ASP is known as elaboration tolerance (McCarthy 1988) and should ease ongoing maintenance costs.
6. Testing effort for ASP is reduced relative to a procedural language, primarily because control code is not tested, however a further effect is visible from the reduction in lines of code.
7. ASP takes less lines of code and less blockchain space – Appendix A shows a Solidity program with its roughly equivalent ASP program at approximately a quarter the size. Further, Solidity is compiled into bytecode which multiplies its size a number of times. It is possible that ASP uses as little as one tenth the space on blockchain compared to Ethereum bytecode.

7.3.2 Testing the Smart Contract Type

Demonstrably simpler to test (see the Comprehensive Test Case Suite in Appendix C), because control code does not need to be tested. A further consideration is that null cases do not need to be tested because the logic program template is supplied with dummy variables (instantiation place holders), which do not affect results because they are never resolved by the solver.

7.3.3 Auto-generating the Instantiation at Contract Creation

ASP avoids the need to code data upload methods.

7.3.4 Testing the created Smart Contract

The area where ASP appears to have its greatest advantage over Solidity, because ASP makes it practical to fully test the smart contract type supplied as a template thus facilitating the achievement of a system that can be used by untrained users. Avoiding testing at this stage with Solidity is more difficult and likely to suffer drawbacks relative to an ASP implementation.

An ASP implementation enables an evolution of the testing task into a simulation that shows users the outcomes of the range of different event scenarios allowed, a task more focused on user understanding. While it may be possible to provide the same feature with a procedural language, the reality is that with current technologies finding and removing bugs is the necessary focus.

7.3.5 Programmatically interacting with the Smart Contract after Deployment

Removing the ABI and avoiding coding of interaction and query functions saves programming effort and reduces complexity, reducing the possibility of undetected security gaps. Being able to change the specification of the code by adding logic code, affords greater flexibility in the ongoing management of smart contracts. Queries on the state of the smart contract are in the form of queries on the aggregated code.

7.3.6 Other Advantages

Higher Level of Abstraction

ASP statements manipulate sets and relationships between them, over which it can also reason. This is clearly a much higher level of abstraction than Solidity, but its coding scheme is also closer to natural language, making the code easier to read.

Simpler more Intuitive Syntax

Only two of the three types of ASP statement are used, and these encode both logic and data. Solidity uses many different types of statement including control statements like “if” and “for” statements, and different object types, including data stores, to achieve the same result.

Other Advantages

ASP’s close connection to non-monotonic logics that provides ASP with the power to model default negation, deal with incomplete information, and encode domain knowledge, defaults, and preferences in an intuitive and natural way. ASP is elaboration-tolerant, meaning that the language accepts changes in a problem specification without the need to rewrite the entire program, implements weak and strong negation in order to deal with a local form of the closed world assumption, and is order independent. These features provide the flexibility needed to implement our approach, allowing us to model legal logic in an intuitive way and split code into facts, logic program and events. Splitting into **facts** and **logic program** makes it possible to auto-generate just **facts** at instantiation simplifying code auto-generation. Further, NAF allows **events** to change the state of the program, simplifying this non-monotonic aspect. For legal documents of a given type, the logic program is the same with only the assertional knowledge (facts) differing. This allows translation of legal documents to be split into two stages.

A further simplification is possible because auto-generating ASP with a template system appears to be quite flexible regarding the representation of type hierarchies. For example, ‘Will and Testament’ has seven types; testator, executor, beneficiary, parent, sibling, child, and witness; that are of a common meta-type ‘legal person’ (we use the term ‘entity’). This hierarchy allows replacement of seven template atoms with one meta-template atom `entity/5` modelling the meta-type ‘entity’. This idea allows a reduction of atoms that introduce variables into the system to only 6 types, making exhaustive testing possible in this instance, and leading us to conclude that effective testing in general is much more achievable. Our simulator implemented two meta-types; 1) `entity/5`, 2) `asset/4`. for ‘Will and Testament’ and we found this to be sufficient to also cover ‘Real Estate Sale’, suggesting complex types maybe similar between legal domains. On reflection, legal contracts create relationships between entities, strongly implying ‘entity’ is the main meta-type.

We also found that modelling atom names after objects and concepts in the real world provides further synergies; for example, when an external **event** occurs, the values of variables allowed in the auto-generation can be discovered by inspecting **facts** coded in the original contract, guided by the IPH in atom `death/2` (`iph_entity`), the system knows to query the `entity/5` atom for the entities allowed. This is particularly helpful for generating guidance when the executor records a death event, but is also useful for auto-generation. Advantages like this illustrates that ASP allows an approach to modelling the external world that is intuitive and natural (Brewka 2011). We extend this idea to internal atoms; for example, if the smart contract has been witnessed, a `witness/0` atom is created. If there is a wipeout, a `wipeout/2` atom is created.

The ‘option’ type we use supports assembly of custom smart contracts from building blocks by selecting clauses from libraries (text and code pairs). This idea warrants further research.

Finally, we found our human-in-the loop mechanism provides a way to deal with more difficult logic and performance by allowing handoff to humans. It seems feasible that an assistant could be built that helps humans manage these aspects of a contract, including sending relevant legal clauses from the contract to humans via SMS or e-mail after certain events occur.

7.4 Summary

We have demonstrated that; 1) a non-trivial legal document can be implemented with an existing pure declarative language (ASP), and 2) have provided evidence that using declarative code facilitates the implementation of a smart contract approach with improved utility over the Solidity.

Chapter 8 Discussion

8.1 Key Findings

Our study demonstrates that it is possible to convert legal documents that use deductive logic and have tangible ‘performance’ to smart contracts programmed in ASP, verifying the first unproven concept in our hypothesis. Evidence is provided in the form of rules and worked examples, that show that our approach is extendable to legal documents that use deductive logic and have tangible ‘performance’, were this is cost effective.

Our second key finding is that there are some advantages to coding in a declarative language like ASP. Possibly all advantages can be overcome, but at a cost; for example, it is practical to fully test an ASP ‘smart contract type’ logic program at the central issuing authority, allowing untrained users to create smart contracts without much risk. Following this same approach is risky for Solidity because the smart contract has to be compiled after it is assembled and instantiated, and our approach places an untrained user in control at this point. Together with the other findings listed in section 7.3, verifies the second unproven concept in our hypothesis, that using a declarative language (ASP) facilitates the realisation of improved utility in smart contracts.

8.2 Limitations

The two key limitations on this study were limited timeframe allowed a Master of Research Thesis (9 months), and the fact that a lot of research in this field is done by private companies, and not visible online or via academic channels. The limited time frame forced many compromises, including choice of development environment, the depth to which certain areas are explored, and restricted our investigation to only three Australian legal contracts, with in-depth investigation into only one; the ‘Will and Testament’. Ideally, a more evenly distributed set of legal documents from across the legal spectrum should be evaluated to better understand advantages, limitations, deficiencies and boundaries of applicability. This selection would be greatly aided by the availability of exhaustive taxonomies of legal contract types, legal logic and performance clause types. Finally, our development of evaluation tools was preliminary, limited to a subjective comparison method rather than a more rigorous statistically based method.

8.3 Implications

Smart contracts are seen as game changing by many groups, and should current issues with cost, usability and security be solved, the economic impact is likely to be large. Professional service groups are suggesting significant cost savings (Accenture 2017; McKinsey 2018).

This study has identified an approach to smart contract creation and use that is generally applicable, tolerant of varying levels of automation and human-in-the-loop interaction, and that facilitates achieving higher levels of utility in the smart contracts created, along with lower costs.

Our approach is highly supportive of adoption because most, if not all currently used legal contract templates are held on computer and many are already marked-up. Further, building a customised mark-up tool to assist conversions is straight-forward. This study then, has the potential to have an impact on research into and adoption of smart contracts.

8.4 Future Research

Visualisation of Testing and Formal Verification

This study has demonstrated improved ease of use and to a lesser degree improved testing effectiveness. Ideally, closing the remaining gaps of improved understandability and providing a complete solution to reliability should be the targets of future work. Our intuitions are that visualising the testing task can address understandability, while formally validating all code deployed to the blockchain fully addresses reliability.

TABLE 13 – POTENTIAL ADVANTAGES OF PROPOSED FUTURE WORK RELATIVE TO ETHEREUM'S CURRENT SMART CONTRACTS

This table indicates the degree to which an alternative approach to creating smart contracts either improves upon, or is disadvantaged relative to the current smart contract creation approach (Solidity and bytecode) used by Ethereum;
 “+” indicating improvement and “-” indicating disadvantage.
 Note that a blank means there is no difference to Solidity.

no.	proposed improvements to smart contract creation, testing & deployment						
		ease of use	understandability	ease of testing	free of security exploits and bugs at deployment	scalability	cost
	current solution (Solidity>bytecode)						
3	future work: visualise testing		+++	++	++	?	-
4	future work: formal verification of ASP code			+++	+++	++	-
5	future work: combination of all above approaches	+++	+++	+++	+++	++	-

Legend:

+	slight advantage	-	slight disadvantage
++	advantage	--	disadvantage
+++	large advantage	---	great disadvantage

ASP's mathematically sound foundations gives confidence both visualisation and formal verification can be achieved. Research supporting this view centres around domain specific languages developed with meta programming languages with proof assistant features (Coq), such as Ergo. Given ASP's more than two decades of development and testing; the question is, what are ASPs strengths relative to languages like Ergo, or will developing domain specific languages look more and more like ASP.

Formal Verification is seen as the gold standard of proof that a software program performs as required without coding errors or vulnerabilities, and is seen as necessary for high value, high risk systems like spacecraft, and increasingly for smart contracts if costs can be reduced to an acceptable level. The definition of formal verification is that the executable program behaves identically to its specification, proved using formal methods from mathematics (Berztiss 1988). An ASP program is already a specification, so a naive perspective is that formal verification of ASP requires two proofs; 1) that the grounder and solver are bug free, 2) that the ASP code is equivalent to the 'Answer Sets' produced.

Visualisation requires techniques that convert rigorously defined encodings into the equivalent visual representation. This implies equivalence of different methods of encoding meaning (semantics); for example, 1) mathematical notation, 2) programming notation, 3) visual notation, can all mean the same thing. As humans process visual information many times faster than speech or writing, exploiting visualisation represents an opportunity to improve both the understandability of logic and aid the identification of errors (Tableau 2019).

Markup Languages

This study requires only a small subset of mark-ups available (presentation, processing, internal referencing), and there is scope for exploiting increasingly more powerful constructs like the

powerful referencing construct XLink (W3C. 2010), OWL (Web Ontology Language), SWRL (Semantic Web Rule Language) and RDF (Resource Description Framework), all Semantic Web initiatives (Arroyo, et al. 2004). We also note mark-up's similarity to Knuth's attribute grammar (Knuth 1967). Embedding machine executable mark-up is another promising direction, however initial investigation suggests that this is problematic; for example, in the 'Will and Testament' two events can occur that disqualify a beneficiary; 1) death before or within 30 days of the testator, 2) contesting the Will. These clauses are listed apart, yet are best coded together to generate a 'disqualified' atom. However we see significant potential for mark-up in areas like understandability and of screen assistant behaviours.

Extensions to ASP

Simple extensions to standard clingo are required to allow working with large amounts of currency and real numbers. These extensions would greatly increase the utility of ASP when used for smart contracts, especially financial smart contracts. A simple method for handling currency would be to hold currency in "quoted strings", performing mathematical operations in Python functions. For real numbers two "quoted strings" are required, one for significand and the other for exponent. Further research is required to determine if the above ideas have significant performance downsides, or whether applicable extensions already exist.

Other Declarative Languages

ASP is only one of many declarative languages that can be used to encode legal logic, some of which have more advanced features than ASP. It is intended that other declarative language solvers such as TOAST (ARG-tech 2012) for structured argumentation with ASPIC+, and SPINdle (Data61, CSIRO 2013) for Defeasible Logic etc. be investigated in future work.

Web Smart Contract Editor Application

Our simulator was built with technologies familiar to the author because of study time constraints and because standalone sufficed. A more advanced implementation would require investigation of the best current technologies like React, Angular and KnockoutJS. A commercial implementation of the smart contract editor (hereafter ContractWriter) is envisioned to be a collaborative distributed web app built using current technologies and having many of the features present in Discord (Discord Inc. 2019) like VoIP voice, text chat and video in addition to a shared real-time view of the current smart contract session. Like Discord, ContractWriter would allow people to be invited to join a ContractWriter session via a link sent by SMS, e-mail or messenger service, with digital signing (witnesses) and access to PII via private key.

Instantiation Place Holders

In this study different IPHs are used for HTML ("____identifier____") and ASP ("iph_identifier"). Interestingly, ASP treats "____identifier____" as a constant, meaning that it is possible to use this form of IPH in both text and ASP code templates. This was avoided in this study because of concerns that "____identifier____" does not comply to the ASP-Core-2 standard. If compliance is proven, some simplification of the smart contract editor would result.

Business and Legal Domain

Because of the economic and other benefits of converting traditional legal contracts to smart contracts, a detailed understanding of the factors that make conversion economic would be useful. Some factors have been identified; 1) the type of 'legal reasoning' used, and 2) the type of

‘performance’ required (as per Table 1, p6), are useful indications of the automatability of a contract. Other factors like the complexity of attached schedules, and the interactions with other systems need to be evaluated; for example, how does automated conveyancing (PEXA 2019) impact real estate contracts. Ideally taxonomies (Snowden 2011) of legal contracts, types of performance and legal reasoning are required, along with any other aspects not yet identified. Some work in these areas is currently visible. (Ryan 2018; Tönnissen 2018).

Blockchain Space Savings

Because our declarative code is split into **facts** (with variables) and **logic program** (invariant), blockchain space saving opportunities arise. Only **facts** and a key (to access the template DSL) are required to completely reconstruct both the full text and code of a smart contract. We have evaluated enough implementation options to recognise that there is a space vs compute time spectrum worth further investigation.

8.5 Conclusion

Blockchain technologies promise improvements to legal contracts, yet coding requires programmers and risks fraud, and widespread adoption depends on improving security and removing programmers. We found that using a declarative language facilitated achieving improved utility by implementing a ‘Will and Testament’ as a smart contract on a custom simulator. This simulator auto-generated a smart contract from a status-quo user interface with an untrained user. During this exercise we found a number of small benefits to using a declarative language like simplification, ease of code auto-generation and ease of testing. Our solution supports adoption because it starts with a legal contact, is tolerant of varying levels of automation, and allows human-in-the-loop interaction. Smart contracts are seen as game changing, and should issues with cost, utility and security be solved, the economic impact is likely to be large.

References

- Accenture. 2017. "Blockchain Technology Could Reduce Investment Banks' Infrastructure Costs by 30 Percent, According to Accenture Report." *Accenture*. 17 Jan. Accessed Sep 11, 2019. <https://newsroom.accenture.com/news/blockchain-technology-could-reduce-investment-banks-infrastructure-costs-by-30-percent-according-to-accenture-report.htm>.
- Accord Project. 2019. *Introduction to Ergo*. 11 Sep. Accessed Sep 11, 2019. <https://docs.accordproject.org/docs/0.3.9/ergo.html>.
- Agarwal, Sudhir, Kevin Xu & John Moghtader. 2016. "Towards Machine-Understandable Contracts." *Workshop at the 22nd European Conference on Artificial Intelligence*. The Hague, The Netherlands: Artificial Intelligence for Justice (AI4J). 1-8.
- Aguado, Felicidad, Pablo Ascariz, Pedro Cabalar, Gilberto Perez, Concepcion Vidal. 2015. "Formal Verification for ASP: a Case Study using the PVS Theorem Prover." *Proceedings of the 15th International Conference on Computational and Mathematical Methods in Science and Engineering, CMMSE 2015*. Rota, Spain: CMMSE.
- Amani, Sidney, Myriam Begal, Maksym Bortin, Mark Staples. 2018. "Towards verifying ethereum smart contract bytecode in Isabelle/HOL." *Proceedings of the 7th ACM SIGPLAN International Conference on Certified Programs and Proofs (CPP 2018)*. Los Angeles, USA: ACM. 66-77.
- Antoniou, Grigoris, Paul Groth, Frank van Harmelen, and Rinke Hoekstra. 2012. *A Semantic Web Primer*. Cambridge, Massachusetts: The MIT Press.
- ARG-tech. 2012. *TOAST: An ASPIC+ implementation*. 20 June. Accessed June 10, 2019. <http://www.arg-tech.org/index.php/toast-an-aspic-implementation/>.
- Arroyo, Sinuhé, Ying Ding, Rubén Lara, Michael Stollberg, and Dieter Fensel. 2004. "SEMANTIC WEB LANGUAGES - STRENGTHS AND WEAKNESS." *International Conference in Applied Computing*. Lisbon, Portugal: International Conference in Applied Computing (IADIS04). 23-26.
- ASX. 2019. *CHESS Replacement*. 4 Oct. Accessed Oct 4, 2019. <https://www.asx.com.au/services/chess-replacement.htm>.
- Australian Contract Law. 2010. "Performance and termination." *Australian Contract Law*. Accessed Sep 11, 2019. <https://www.australiancontractlaw.com/law/termination.html>.
- Batsakis, Sotiris, George Baryannis, Guido Governatori, Ilias Tachmazidis. 2018. "Legal Representation and Reasoning in Practice: A Critical Comparison." *Legal Knowledge and Information - JURIX 2018*. Het Kasteel, Groningen, Netherlands: IOS Press. 31-40.
- Bäumer, Dirk, Walter R. Bischofberger, Horst Lichter, and Heinz Züllighoven. 1996. "User Interface Prototyping - Concepts, Tools, and Experience." *Proceedings - International Conference on Software Engineering*. Institute of Electrical and Electronics Engineers. 532-541.
- Bayer, Dave, Stuart Haber, and W. Scott Stornetta. 1992. "Improving the Efficiency and Reliability of Digital Time-Stamping." *Sequences II: Methods in Communication, Security & Computer Science*. New York: Springer. 329-334.
- Berger, Emery. 2019. "On the Impact of Programming Languages on Code Quality." *Computing Research Repository (CoRR)*.
- Berners-Lee, Tim. 1989. *"Information Management: A Proposal."*. Memo, Geneva: CERN.
- Bertziss, Alfs, Mark Ardis. 1988. "Formal Verification of Programs." In *SEI Curriculum*, by Carnegie Mellon University Software Engineering Institute. Pittsburgh, USA: Carnegie Mellon University Software Engineering Institute.

Reference List and Appendices

- Bickford, Mark, Constable, Robert. 2008. "Formal Foundations of Computer Security." *Science for Peace and Security Series D: Information and Communication Security Vol.14*. Brussels, Belgium: Nato. 29-52.
- bitcoin. 2019. "51% Attack, Majority Hash Rate Attack." *bitcoin*. 11 Sep. Accessed Sep 11, 2019. <https://bitcoin.org/en/glossary/51-percent-attack>.
- Brewka. 2011. "Answer Set Programming at a Glance." *Communications of the ACM* 54 (12): 93-103.
- Buelau, Alex. 2017. "Confideal – The Visual Smart Contract Editor." *CoinSchedule*. 29 Oct. Accessed Sep 11, 2019. <https://www.coinschedule.com/blog/confideal-visual-smart-contract-editor/>.
- Bush, Vannevar. 1945. "As We May Think." *The Atlantic* 10.
- Buterin, Vitalik. 2014. "Ethereum White Paper." *We Use Coins*. Accessed June 10, 2019. https://www.weusecoins.com/assets/pdf/library/Ethereum_white_paper-a_next_generation_smart_contract_and_decentralized_application_platform-vitalik-buterin.pdf.
- Calimeri, Wolfgang Faber, Martin Gebser, Giovambattista Ianni, Roland Kaminski, Thomas Krennwallner, Nicola Leone, Francesco Ricca, Torsten Schaub. 2015. *ASP-Core-2 Input Language Format*. Technical Report, ASP Standardization Working Group.
- Cambridge Dictionary. 2019. *Cambridge Dictionary*. 31 12. Accessed 12 31, 2019. <https://dictionary.cambridge.org/dictionary/english/utility>.
- CED. 2014. *Collins English Dictionary – Complete and Unabridged*. London: HarperCollins.
- Chaum, David. 1982. "Blind Signatures for Untraceable Payments." *Advances in Cryptology*. Boston: Springer. 199-203.
- Chohan, Usman. 2017. "The Decentralized Autonomous Organization and Governance Issues." *Social Science Research Network*. 4 Dec. Accessed Sep 11, 2019. <https://ssrn.com/abstract=3082055>.
- Choudhury, Olivia, Nolan Rudolph, Issa Sylla, Noor Fairiza, Amar Das. 2018. "Auto-Generation of Smart Contracts from Domain-Specific Ontologies and Semantic Rules." *2018 IEEE International Conference on Internet of Things (iThings) and IEEE Green Computing and Communications (GreenCom) and IEEE Cyber, Physical and Social Computing (CPSCom) and IEEE Smart Data (SmartData)* (IEEE) 963-970.
- City of Melbourne. 2018. "Chief Executive Officer's Employment Contract." *City of Melbourne*. 30 Nov. Accessed Sep 11, 2019. <https://www.melbourne.vic.gov.au/about-council/governance-transparency/council-information/registers-inspection/Pages/default.aspx>.
- Cognology. 2019. *360 Degree Feedback System*. 11 Sep. Accessed Sep 11, 2019. https://www.cognology.com.au/products/360_degree_feedback/?gclid=EAlaIqobChMI4qiYv8mg5QIVQiUrCh2rXwSOEAAyAAEglx3_D_BwE.
- Confideal. 2019. *Trusting people is easy again*. 11 Sep. Accessed Sep 11, 2019. <https://confideal.io/>.
- CSIRO. 2018. "Smart legal contracts for Australian businesses." *CSIRO News*. 29 Aug. Accessed Aug 29, 2019. <https://www.csiro.au/en/News/News-releases/2018/New-blockchain-based-smart-legal-contracts>.
- Curtotti, Michael, Eric McCreath. 2010. "Corpus Based Classification of Text in Australian Contracts." *Proceedings of Australasian Language Technology Association Workshop*. Melbourne, Australia: Australasian Language Technology Association Workshop. 18-26.
- Data61, CSIRO. 2013. *SPINdle*. 30 May. Accessed June 10, 2019. <http://spindle.data61.csiro.au/spindle/index.html>.
- Davis, Amanda. 2015. "A History of Hacking." *The Institute. IEEE. March 2015*. 6 Mar. Accessed Sep 11, 2019. <http://theinstitute.ieee.org/technology-topics/cybersecurity/a-history-of-hacking>.
- Discord Inc. 2019. *Discord App*. 11 Sep. Accessed Sep 11, 2019. <https://discordapp.com/>.
- DSLFIN. 2019. "Financial Domain-Specific Language Listing." *DSLFIN Workshop on Domain-Specific Languages for Financial Systems*. 11 Sep. Accessed Sep 11, 2019. <https://dslfin.org/resources.html>.

- EconoTimes. 2017. "Smart Contracts Have Massive Potential to Take Trigger Massive Adoption of Blockchain." *EconoTimes*. 3 Nov. Accessed Sep 11, 2019. <https://www.econotimes.com/Smart-Contracts-Have-Massive-Potential-to-Take-Trigger-Massive-Adoption-of-Blockchain-987848>.
- Ellsworth, Phoebe. 2005. "Legal Reasoning." In *The Cambridge Handbook of Thinking and Reasoning*, by Holyoak and Morrison, 685-704. New York: Cambridge Univ. Press.
- Engelbart, Douglas. 1968. *oN-Line System*. Menlo Park, 9 Dec.
- Epic Games. 2019. "Blueprints Visual Scripting." *Unreal Engine*. 11 Sep. Accessed Sep 11, 2019. <https://docs.unrealengine.com/en-US/Engine/Blueprints/index.html>.
- Ethereum. 2019. *Intro to Ethereum Wallets*. 11 Sep. Accessed Sep 11, 2019. <https://docs.ethhub.io/using-ethereum/wallets/intro-to-ethereum-wallets/>.
- Ethereum SCW. 2019. *Smart Contract Wallets*. 11 Sep. Accessed Sep 11, 2019. <https://docs.ethhub.io/using-ethereum/wallets/smart-contract-wallets/>.
- Event-B.org. 2018. *Event-B and Rodin Documentation Wiki*. 18 Mar. Accessed Sep 11, 2019. <http://www.event-b.org/>.
- Falkon, Samuel. 2017. "The Story of the DAO — Its History and Consequences." *medium.com*. 24 December. Accessed June 10, 2019. <https://medium.com/swlh/the-story-of-the-dao-its-history-and-consequences-71e6a8a551ee>.
- FindLegalForms. 2019. "Real Estate Contract for Sale - Cash or Seller Financing (Australia)." *FindLegalForms.com*. 11 Sep. Accessed Sep 11, 2019. <https://au.findlegalforms.com/product/real-estate-contract-for-sale-cash-or-seller-financing-australia/>.
- Fitzpatrick, Scott. 2019. "Banking on Stone Money: Ancient Antecedents to Bitcoin." *Economic Anthropology* 1-13.
- Forbes. 2018. "How Smart Contracts Could Change The Way You Do Business." *Forbes*. 16 May. Accessed Sep 11, 2019. <https://www.forbes.com/sites/forbescoachescouncil/2018/05/16/how-smart-contracts-could-change-the-way-you-do-business/#136e291a1288>.
- Fortune. 2018. "Blockchain Firm R3 Is Running Out of Money, Sources Say." *Fortune*. 7 June. Accessed Sep 11, 2019. <https://fortune.com/2018/06/07/blockchain-firm-r3-is-running-out-of-money-sources-say/>.
- Fruehwald, Edwin. 2011. "Tip of the Week: Five Methods of Legal Reasoning." *The Law Professor Blogs Network*. 3 Aug. Accessed Aug 29, 2019. https://lawprofessors.typepad.com/legal_skills/2011/08/tip-of-the-week-five-methods-of-legal-reasoning.html.
- Glantz, Kissell. 2013. *Multi-Asset Risk Modeling*. Cambridge, Massachusetts: Academic Press.
- Goldfarb, Charles. 1992. "Appendix A, A brief history of the development of SGML ." In *SGML Handbook*, by Charles Goldfarb, 663. Oxford, UK: Clarendon Press.
- Governatori, Guido, Antonino Rotolo, and Erica Calardo. 2012. "Possible World Semantics for Defeasible Deontic Logic." *Deontic Logic in Computer Science*. Springer. 46-60.
- Governatori, Guido, Florian Idelberger, Zoran Milosevic, Regis Riveret, Giovanni Sartor, Xiwei Xu. 2018. "On legal contracts, imperative and declarative smart contracts, and blockchain system." *Artificial Intelligence and Law* 377-409.
- Grigg, Ian. 2004. "The Ricardian Contract." *Proceedings of the First IEEE International Workshop on Electronic Contracting*. Washington, DC, USA: IEEE Computer Society Washington, DC, USA. 25-31.
- Haber, Stuart, and W. Scott Stornetta. 1991. "How to time-stamp a digital document." *Journal of Cryptology* 99-111.
- Harrison, Amelia. 2015. "Formal Methods for Answer Set Programming." *Proceedings of the Technical Communications of the 31st International Conference on Logic Programming (ICLP 2015)*. Cork, Ireland: dblp computer science bibliography. 1-8.

Reference List and Appendices

- Hernandez, Alejandro. 2018. *Are You Trading Stocks Securely? Exposing Security Flaws in Trading Technologies*. White Paper, Seattle: IOActive.
- HSC CoWorks. 2019. "HSC ENGLISH LANGUAGE TECHNIQUES." *HSC CoWorks*. 11 Sep. Accessed Sep 11, 2019. <https://hsccoworks.com.au/hscenglishlanguagetechniques/>.
- Hyperledger Composer. 2019. "Welcome to Hyperledger Composer." *github Hyperledger Composer*. 11 Mar. Accessed Mar 11, 2019. <https://hyperledger.github.io/composer/v0.19/introduction/introduction.html>.
- Hyperledger. 2019. *Hyperledger Fabric*. 11 Sep. Accessed Sep 11, 2019. <https://www.hyperledger.org/projects/fabric>.
- IBM. 2020. *Working with web3js API and JSON to Build Ethereum Blockchain Applications*. 2 Jan. Accessed Jan 2020, 2020. <https://developer.ibm.com/recipes/tutorials/working-with-web3js-api-and-json-to-build-ethereum-blockchain-applications/>.
- Idelberger, Florian, Guido Governatori, Regis Riveret, and Giovanni Sartor. 2016. "Evaluation of Logic-Based Smart Contracts for Blockchain Systems." doi:10.1007/978-3-319-42019-6_11.
- Idris. 2019. *Idris, A Language with Dependent Types*. 11 Sep. Accessed Sep 11, 2019. <https://www.idris-lang.org/>.
- Inria. 2019. *The Coq Proof Assistant*. 11 Sep. Accessed Sep 11, 2019. <https://coq.inria.fr/>.
- Investopedia. 2019. *Business-to-Consumer (B2C)*. 20 May. Accessed Sep 11, 2019. <https://www.investopedia.com/terms/b/btoc.asp>.
- Jäger, Oliver. 2013. "Technologien für das Privacy Wallet." paper, Trier.
- JCT. 2019. *About JCT*. 11 Sep. Accessed Sep 11, 2019. <https://corporate.jctltd.co.uk/about-us/>.
- Knuth. 1967. "Semantics of Context-Free Languages." *Mathematical Systems Theory* 127-145.
- Koons, Robert. 2017. "Defeasible Reasoning." Accessed Sep 11, 2019. <https://plato.stanford.edu/entries/reasoning-defeasible/>.
- Kowalski, Robert. 2019. "Logic Production Systems." *Imperial College London - Department of Computing*. 11 Sep. Accessed Sep 11, 2019. <http://lps.doc.ic.ac.uk/>.
- Kramer, Frank. 2015. "Privacy Enhancing Technology through a Privacy Wallet." *International / European-Japanese Conference on Information Modelling and Knowledge Bases (EJC)*. Maribor, Slovenia: EJC. 137-156.
- Kuhn, Tobias. 2014. "A Survey and Classification of Controlled Natural Languages." *Computational Linguistics, Vol. 40, No. 1* 121-170.
- LawDepot. 2019. "Free Last Will and Testament." *LawDepot*. 15 Feb. Accessed Feb 15, 2019. <https://www.lawdepot.com/contracts/last-will-and-testament-au>.
- Lierler, Yuliya. 2017. "What is answer set programming to propositional satisfiability." *Constraints* 307-337.
- Lifschitz, Vladimir. 2008. "What is answer set programming?" *AAAI'08 Proceedings of the 23rd national conference on Artificial intelligence - Volume 3*. Chicago: AAAI Press. pp.1594-1597.
- Lloyd, John. 1994. "Practical Advantages of Declarative Programming." *Conference Proceedings: Joint Conference on Declarative Programming*. Peñíscola, Spain. 3 - 17.
- Marks, Eric. 2018. "The Case for Graphical Smart Contract Editors." *Medium*. 30 Apr. Accessed Sep 11, 2019. <https://medium.com/pennblockchain/the-case-for-graphical-smart-contract-editors-8e721cdcde93>.
- Mazières, David, and Dennis Shasha. 2002. "Building Secure File Systems out of Byzantine Storage." *Proceedings of the Twenty-first Annual Symposium on Principles of Distributed Computing*. Monterey, California: ACM. 108-117.
- McCarthy, John. 1988. "Mathematical logic in artificial intelligence." *Daedalus (Common Sense)* 297-311.

Reference List and Appendices

- McKinsey. 2018. "Blockchain beyond the hype: What is the strategic business value?" *McKinsey Digital*. June. Accessed Sep 11, 2019. <https://www.mckinsey.com/business-functions/mckinsey-digital/our-insights/blockchain-beyond-the-hype-what-is-the-strategic-business-value>.
- McNamara, Paul. 2010. "Challenges in Defining Deontic Logic." *Stanford Encyclopedia of Philosophy*. Accessed Sep 11, 2019. <https://stanford.library.sydney.edu.au/entries/logic-deontic/challenges.html>.
- Nakamoto, Satoshi. 2008. "Bitcoin: A Peer-to-Peer Electronic Cash System." *bitcoin*. 31 October. Accessed Sep 2019, 11. <https://bitcoin.org/bitcoin.pdf>.
- Nicholas, Barry. 1962. *An Introduction to Roman Law*. Oxford: Clarendon Press.
- NIST. 2019. *National Institute of Standards and Technology - Information Technology Laboratory - Computer Security Resource Center*. 11 Sep. Accessed Sep 11, 2019. <https://csrc.nist.gov/glossary/term/personally-identifiable-information>.
- Northchain. 2019. *OLE Blockchain technology*. 11 Sep. Accessed Sep 11, 2019. <https://www.northchain.tech/en/how-ole-works/>.
- NSW LRS. 2019. *Torrens Title Register*. 31 May. Accessed Sep 08, 2019. <https://www.nswlrs.com.au/Public-Register/Torrens-Title-Register>.
- NSW RMS. 2019. *Registration*. 11 Sep. Accessed Sep 11, 2019. <https://www.rms.nsw.gov.au/roads/registration/index.html>.
- OpenLaw. 2019. "About." *OpenLaw Docs*. 11 Sep. Accessed Sep 11, 2019. <https://app.openlaw.io/about>.
- OpenZeppelin. 2019. "Build Secure Smart Contracts in Solidity." *OpenZeppelin - Contracts*. 11 Sep. Accessed Sep 11, 2019. <https://openzeppelin.com/contracts/>.
- Panetta, Kasey. 2019. "The CIO's Guide to Blockchain." *Smarter With Gartner*. 23 Sep. Accessed Sep 25, 2019. <https://www.gartner.com/smarterwithgartner/the-cios-guide-to-blockchain/>.
- PEXA. 2019. *About PEXA*. 11 Sep. Accessed Sep 11, 2019. <https://www.pexa.com.au/company>.
- Potassco. 2019. *Getting Started*. 11 Sep. Accessed Sep 11, 2019. <https://potassco.org/>.
- Prestwich, James. 2018. *Declarative Smart Contracts*. 2 Sep. Accessed Sep 11, 2019. <https://prestwi.ch/declarative-smart-contracts-2/>.
- Prowell, Stacy. 2005. "CORRECT-BY-DESIGN SOFTWARE IS FUNDAMENTAL TO HIGH-CONFIDENCE DEVICES." *High Confidence Medical Device Software and Systems (HCMDSS) Workshop*. Philadelphia, USA: Computer and Information Science, University of Pennsylvania.
- Qt. 2017. "Supported HTML Subset." *Qt v5.9 Documentation*. 31 May. Accessed Mar 9, 2019. <https://doc.qt.io/qt-5.9/richtext-html-subset.html>.
- Randall, Vernellia. 2009. "Race and Wealth Disparity: The Role of Law and the Legal System." *Law - Fall 2009 Racism, Health Disparities, and the Law*. Accessed Sep 11, 2019. <academic.udayton.edu/health/syllabi/disparities/07WealthInequality/wealth02.htm>.
- Ray, Baishakhi, Daryl Posnett, Vladimir Filkov, and Premkumar Devanbu. 2014. "A large scale study of programming languages and code quality in github." *SIGSOFT FSE*. Hong Kong: ACM Digital Library. 155-165.
- Rice, Michael. 2018. "solidity-legal-contracts/contracts/Will.sol." *github*. Accessed Mar 11, 2019. <https://github.com/mrice/solidity-legal-contracts/blob/master/contracts/Will.sol>.
- Ryan, Philippa. 2018. "Herbert Smith Freehills, King & Wood Mallesons back Australian National Blockchain." *Australian Financial Review*. 6 Sep. Accessed Aug 29, 2019. <https://www.afr.com/companies/professional-services/herbert-smith-freehills-king--wood-mallesons-back-australian-national-blockchain-20180905-h14yg3>.

Reference List and Appendices

- Ryan., Philippa. 2018. "A proposed new Taxonomy for Autonomous Smart Contracts." *LinkedIn*. 12 Aug. Accessed Sep 11, 2019. <https://www.linkedin.com/pulse/proposed-new-taxonomy-autonomous-smart-contracts-dr-philippa-ryan/>.
- Schwitter. 2018. "Specifying and Verbalising Answer Set Programs in Controlled Natural Language." (EasyChair).
- scilla-doc. 2019. *Scilla*. 11 Sep. Accessed Sep 11, 2019. <https://scilla.readthedocs.io/en/latest/>.
- Sirer, Emin Gun. 2016. *Thoughts on the DAO Hack*. 17 Jun. Accessed Sep 11, 2019. <http://hackingdistributed.com/2016/06/17/thoughts-on-the-dao-hack/>.
- Smith, William. 1875. *A Dictionary of Greek and Roman Antiquities*. London: John Murray.
- Snowden, Dave. 2011. "Typology or Taxonomy?" *Cognitive Edge*. 22 Oct. Accessed Sep 11, 2019. <http://cognitive-edge.com/blog/typology-or-taxonomy/>.
- Solidity. 2019. *Read The Docs v0.5.12*. 1 Oct. Accessed Oct 3, 2019. <https://solidity.readthedocs.io/en/v0.5.12/>.
- Solidity. 2017. *Read The Docs*. 6 July. Accessed Sep 11, 2019. <https://solidity.readthedocs.io/en/v0.4.13/>.
- Stanford Law School. 2019. *Developing a Legal Specification Protocol: Technological Considerations and Requirements*. 14 Feb. Accessed Sep 11, 2019. <https://law.stanford.edu/publications/developing-a-legal-specification-protocol-technological-considerations-and-requirements/>.
- Sutherland, Ivan. 1963. *Sketchpad*. PhD Thesis, Boston: Massachusetts Institute of Technology.
- Swamy, Aalok. 2018. "Pros and Cons of Solidity." *sixpl*. 18 May. Accessed Sep 11, 2019. <https://sixpl.com/pros-and-cons-of-solidity/>.
- Szabo, Nick. 1994. "Smart Contracts." *Nick Szabo's Essays, Papers, and Concise Tutorials*. Accessed June 10, 2019. <http://www.fon.hum.uva.nl/rob/Courses/InformationInSpeech/CDROM/Literature/LOTwinterschool2006/szabo.best.vwh.net/smart.contracts.html>.
- Tableau. 2019. *Tableau Mission: Principles of visual understanding*. 11 Sep. Accessed Sep 11, 2019. <https://www.tableau.com/about/mission#understanding>.
- Thomson Reuters. 2020. *Document automation - Contract Express*. 02 Jan. Accessed Jan 02, 2020. <https://legal.thomsonreuters.com/en/products/contract-express>.
- Tönnissen, Stefan, Teuteberg, Frank. 2018. *TOWARDS A TAXONOMY FOR SMART CONTRACTS*. Research Paper, Osnabrück, Germany: Osnabrück University.
- Treasury. 2019. "WORKING WITH CONTRACTS Practical assistance for small business managers." *Australian Government, The Treasury*. 11 Sep. Accessed Sep 11, 2019. <https://treasury.gov.au/sites/default/files/2019-03/WorkingWithContractsGuide.pdf>.
- Voshmgir, Shermin. 2019. *Blockchainhub - Blockchain Oracles*. 1 Jul. Accessed Jan 2, 2020. <https://blockchainhub.net/blockchain-oracles/>.
- Vyper. 2019. *Vyper*. 11 Sep. Accessed Sep 11, 2019. <https://vyper.readthedocs.io/en/v0.1.0-beta.13/>.
- W3C. 2019. *Semantic Web*. 11 Sep. Accessed Sep 11, 2019. <https://www.w3.org/standards/semanticweb/>.
- W3C. 2010. *XML Linking Language (XLink) Version 1.1*. 06 May. Accessed Sep 11, 2019. <https://www.w3.org/TR/xlink11/>.
- Walport, Mark. 2016. *Distributed Ledger Technology: beyond block chain*. Government Report, London: UK Government, Office for Science.
- Wood, Gavin. 2017. "Ethereum: A secure decentralised generalised transaction ledger EIP-150 REVISION." Yellow Paper.
- Xerox PARC. 1973. *Xerox Alto*. Palo Alto, 1 Mar.

Reference List and Appendices

Yang, Zheng, and Hang Lei. 2018. "Formal Process Virtual Machine for Smart Contracts Verification." *International Journal of Performability Engineering* 1726-1734.

Zatko, Peiter. 2011. "How a Hacker Has Helped Influence Government and Vice Versa." *Black Hat 2011*. Las Vegas.

Appendix A – Simple ‘Will’ – Solidity vs ASP

Annotated smart contract in Solidity for a ‘Will and Testament’ (Rice 2018) since depreciated.

Solidity Code	Explanation
<code>pragma solidity ^0.5.1;</code>	Compiler directive with version number. Latest stable version Feb 2019 is ^0.5.4.
<code>contract Will {</code>	Declares a ‘Contract’ (like a class in java).
<code> //declarations</code> <code> address owner;</code> <code> address trustee;</code> <code> uint fortune;</code> <code> bool isDeceased;</code> <code> address payable[9] wallets;</code> <code> uint nextWallet = 0;</code> <code> mapping(address => uint) inheritance;</code> <code> //constructor</code> <code> constructor(address _trustee) public payable {</code> <code> owner = msg.sender;</code> <code> fortune = msg.value;</code> <code> trustee = _trustee;</code> <code> isDeceased = false;</code> <code> }</code> <code> //function modifiers</code> <code> modifier onlyOwner {require(msg.sender == owner); _;}</code> <code> modifier onlyTrustee{require(msg.sender == trustee);_;}</code> <code> modifier deceased {require(isDeceased == true); _;}</code> <code> //setter</code> <code> function setInheritance(address payable _wallet,</code> <code> uint _inheritance) public onlyOwner {</code> <code> wallets[nextWallet] = _wallet;</code> <code> inheritance[_wallet] = _inheritance; }</code> <code> //functions</code> <code> function payout() private deceased {</code> <code> for (uint i=0; i<wallets.length; i++) {</code> <code> wallets[i].transfer(inheritance[wallets[i]]);}</code> <code> function recordDeceased() public onlyTrustee {</code> <code> isDeceased = true;</code> <code> payout(); }</code> <code>}</code>	 special type (Ethereum wallet address)... for ‘Owner’. for ‘Trustee’. unsigned integer (i.e.whole dollars)... the ‘Fortune’. boolean. array of addresses modified to allow payments. unsigned integer (i.e. whole dollars). map – ‘inheritance’ holds a uint for every address. Constructor creates object at ‘Contract’ deployment. public & payable modifiers mean ‘can pay Ether’. Owner’s ‘Ethereum wallet address’ Amount the owner is distributing owner specifies a trustee when calling this code (msg.sender and msg.value are global built-ins) Modifiers are used like access modifiers. An ‘if’ is pasted at the start of function code. _ ; means the code of the modified function. setInheritance can only be called by the ‘Owner’. Loads ‘Ethereum wallet address’ of each ‘Inheritor’. Loads the inheritance \$ against the ‘wallet address’. Locked unless ‘isDeceased’ is set to true. Traverses the ‘wallets’ array and moves the allocated inheritance to ‘Ethereum wallet address’ of that child recordDeceased can only be called by the ‘Trustee’. It executes the ‘Will’ by distributing the ‘Fortune’.

Figure 21 - A simple smart contract written in Solidity and annotated

ASP **logic program** equivalent for the above ‘Will and Testament’ (uses percentages) is:

```
transfer(TestatorWallet, InheritAmount, Wallet) :-
    deceased,
    InheritAmount = Amount*Percent/100,
    inherits(Child, Percent),
    estate(Amount, TestatorWallet),
    id(Child, Wallet).

#show transfer/3.
```

Facts needed to instantiate this code (not shown for Solidity) are:

```
estate(1000000, "89gbmbmscss").
id(mary,"bkbkbbjk3"). id(fred,"i78tgqebgk"). id(bec,"b979beqjs")
inherits(mary, 40). inherits(fred, 20). inherits(bec, 40).
```

The **event** needed to trigger payout is:

```
deceased :- #true.
```

Appendix B – ‘Will and Testament’ Template

LAST WILL AND TESTAMENT OF _____

I, _____, presently of _____, hereby revoke all former testamentary dispositions made by me and declare this to be my last Will.

PRELIMINARY DECLARATIONS

Prior Wills and Codicils

1. I revoke all prior Wills and Codicils.

Marital Status

2. I am married to _____.

Children

3. My living children are _____.

EXECUTOR

Executor

4. The expression ‘my Executor’ used throughout this Will includes either the singular or plural number, or the masculine or feminine gender as appropriate wherever the fact or context so requires. The term ‘executor’ in this Will is synonymous with and includes the term ‘executrix’.

Appointment

5. I appoint _____ of _____, New South Wales as the sole Executor of this Will.

Powers Of My Executor

6. I give and appoint to my Executor the following duties and powers with respect to my estate:
 - a. To pay my legally enforceable debts, funeral expenses and all expenses in connection with the administration of my estate and the trusts created by my Will as soon as convenient after my death, except for any debt secured by real and/or personal property which is to be assumed by the recipient of such property.
 - b. To take all legal actions to have the probate of my Will completed as quickly and simply as possible, and as free as possible from any court supervision.
 - c. To retain, exchange or dispose of any personal property without liability for loss or depreciation.
 - d. To purchase, maintain, convert and liquidate investments or securities, and to exercise voting rights in connection with any shareholding, or exercise any option concerning investments or securities.
 - e. To open or close bank accounts.

- f. To maintain, continue, dissolve, change or sell any business which is part of my estate, or to purchase any business if deemed necessary or beneficial to my estate by my Executor.
- g. To sell, mortgage, exchange, lease or otherwise dispose or deal with any real property in my estate and to pay, alter, improve, add to or remove any buildings thereon and generally to manage such real property.
- h. To maintain, settle, abandon, make a claim against or defend, or otherwise deal with any claims or actions against my estate.
- i. To employ any solicitor, accountant or other professional.
- j. Except as otherwise provided, to act as my Trustee by holding in trust the share of any minor beneficiary, and to keep such share invested, pay the income or capital or as much of either or both as my Executor considers advisable for the maintenance, education, advancement or benefit of such minor beneficiary and to pay or transfer the capital of such share or the amount remaining to such beneficiary when he or she reaches the age of majority or, during the minority of such beneficiary, to pay or transfer such share to any parent or guardian of such beneficiary subject to like conditions and the receipt of any such parent or guardian discharges my Executor.

The above authority and powers granted to my Executor are in addition to any powers and elective rights conferred by statute or common law or by other provision of this Will and may be exercised as often as required, and without application to or approval by any court.

DISPOSITION OF ESTATE

Distribution of Residue

- 7. To receive any gift or property under this Will a beneficiary must survive me for thirty (30) days. Beneficiaries of my estate residue will receive and share all of my property and assets not specifically bequeathed or otherwise required for the payment of any debts owed, including but not limited to, expenses associated with the probate of my Will, the payment of taxes, funeral expenses or any other expense resulting from the administration of my Will. The entire estate residue is to be divided between my designated beneficiaries with the beneficiaries receiving a share of the entire estate residue. All property given under this Will is subject to any encumbrances or liens attached to the property.
- 8. I direct my Executor to distribute the residue of my estate as follows ("Share Allocations"):
 - a. All of the residue of my estate to _____ of _____, _____, for their own use absolutely.

Wipeout Provision

- 9. I HEREBY DIRECT that the residue of my estate or the amount remaining be divided into one hundred (100) equal shares and to pay and transfer such shares as follows:
 - a. 100 shares to be divided equally between my parents and siblings, or survivors thereof, for their own use absolutely, if all or any of them are then alive.

Individuals Omitted from Bequests

- 10. If I have omitted to leave property in this Will to one or more of my heirs as named above or have provided them with zero shares of a bequest, the failure to do so is intentional.

GENERAL PROVISIONS

No Contest Provision

11. If any beneficiary under this Will contests in any court any of the provisions of this Will, then each and all such persons shall not be entitled to any devises, legacies, bequests, or benefits under this Will or any codicil hereto, and such interest or share in my estate shall be disposed of as if that contesting beneficiary had not survived me.

Severability

12. If any provisions of this Will are deemed unenforceable, the remaining provisions will remain in full force and effect.

Signature

13. I, _____, the within named Testator, have to this my last Will contained on this and the preceding pages, set my hand at the City of _____, in the Commonwealth of Australia, this 1st day of April, 2019 I declare that this instrument is my last Will, that I am of the legal age in this jurisdiction to make a Will, that I am under no constraint or undue influence, and that I sign this Will freely and voluntarily.

WITNESSES

This instrument was signed on the above written date by _____, and in our presence the Testator declared this instrument to be their last Will. At the Testator's request and in the presence of the Testator, we subscribe our names as witnesses hereto.

Each of us observed the signing of this Will by _____ and by each other subscribing we witness and affirm that each signature is the true signature of the person whose name was signed. Each of us is now the age of majority, a competent witness and resides at the address set forth after their name.

To the best of our knowledge, the Testator is the age of majority or otherwise legally empowered to make a Will, is mentally competent and under no constraint or undue influence.

We declare under penalty of perjury under the laws of the Commonwealth of Australia that the foregoing is true and correct this 1st day of April 2019, at _____, New South Wales.

Signed by _____ in our presence and then by us in their presence.

Signature _____

Name _____

Address _____

City/Town _____

Postcode _____

Signature _____

Name _____

Address _____

City/Town _____

Postcode _____

Appendix C – Low Complexity – ‘Will and Testament’

Analysis

Who is testator.

I, **John Wallace**, presently of **Lot 49, Cowpasture Road, Mulgoa, 2745, NSW**, hereby revoke all former testamentary dispositions made by me and declare this to be my last Will.

PRELIMINARY DECLARATIONS

Prior Wills and Codicils

1. I revoke all prior Wills and Codicils.

Marital Status

2. I am not married.

Children

3. I have five living children.

EXECUTOR

Executor

4. The expression ‘my Executor’ used throughout this Will includes either the singular or plural number, or the masculine or feminine gender as appropriate wherever the fact or context so requires. The term ‘executor’ in this Will is synonymous with and includes the term ‘executrix’.

Appointment

Who is executor.

5. I appoint **James Stewart of McPhee & Associates, Springwood, 2777, NSW**, New South Wales as the sole Executor of this Will.

Powers of My Executor

6. I give and appoint to my Executor the following duties and powers with respect to my estate:

- a. To pay my legally enforceable debts, funeral expenses and all expenses in connection with the administration of my estate and the trusts created by my Will as soon as convenient after my death, except for any debt secured by real and/or personal property which is to be assumed by the recipient of such property.
- b. To take all legal actions to have the probate of my Will completed as quickly and simply as possible, and as free as possible from any court supervision.
- c. To retain, exchange or dispose of any personal property without liability for loss or depreciation.
- d. To purchase, maintain, convert and liquidate investments or securities, and to exercise voting rights in connection with any shareholding, or exercise any option concerning investments or securities.
- e. To open or close bank accounts.
- f. To maintain, continue, dissolve, change or sell any business which is part of my estate, or to purchase any business if deemed necessary or beneficial to my estate by my Executor.
- g. To sell, mortgage, exchange, lease or otherwise dispose or deal with any real property in my estate and to pay, alter, improve, add to or remove any buildings thereon and generally to manage such real property.
- h. To maintain, settle, abandon, make a claim against or defend, or otherwise deal with any claims or actions against my estate.
- i. To employ any solicitor, accountant or other professional.
- j. Except as otherwise provided, to act as my Trustee by holding in trust the share of any minor beneficiary, and to keep such share invested, pay the income or capital or as much of either or both as my Executor considers advisable for the maintenance, education, advancement or benefit of such minor beneficiary and to pay or transfer the capital of such share or the amount remaining to such beneficiary when he or she reaches the age of majority or, during the minority of such beneficiary, to pay or transfer such share to any parent or guardian of such beneficiary subject to like conditions and the receipt of any such parent or guardian discharges my Executor.

The above authority and powers granted to my Executor are in addition to any powers and elective rights conferred by statute or common law or by other provision of this Will and may be exercised as often as required, and without application to or approval by any court.

To benefit,
death must
occur no less
than 30 days
after testator.

% to each child.
If a death then
divide portion.
Use same ratios.

If all
beneficiaries die
then equal split
between
parents and
siblings.

Any contestor is
cut out.

two witnesses

DISPOSITION OF ESTATE

Distribution of Residue

7. To receive any gift or property under this Will a beneficiary must survive me for thirty (30) days. Beneficiaries of my estate residue will receive and share all of my property and assets not specifically bequeathed or otherwise required for the payment of any debts owed, including but not limited to, expenses associated with the probate of my Will, the payment of taxes, funeral expenses or any other expense resulting from the administration of my Will. The entire estate residue is to be divided between my designated beneficiaries with the beneficiaries receiving a share of the entire estate residue. All property given under this Will is subject to any encumbrances or liens attached to the property.

8. I direct my Executor to distribute the residue of my estate as follows ("Share Allocations"):

a. All the residue of my estate to:

10% to **Fred Wallace** of 1/16 Ronald Avenue, Freshwater, 2096, NSW

20% to **Bec Smith** of 4 Arnold Street, Cremorne, 2090, NSW

10% to **Sam Wallace** of 3 View Street, Coffs Harbour, 2450, NSW

20% to **Jim Wallace** of 123 Soldiers Road, Anna Bay, 2316, NSW

40% to **May Fredricks** of 7 Payne Road, Castle Hill, 2154, NSW

for their own use absolutely.

Wipeout Provision

9. I HEREBY DIRECT that the residue of my estate or the amount remaining be divided into one hundred (100) equal shares and to pay and transfer such shares as follows:

a. 100 shares to be divided equally between my parents:

Tom Wallace of 54 Bland Rd, Springwood NSW, 2777, NSW

Aida Wallace of 54 Bland Rd, Springwood NSW, 2777, NSW

and siblings:

Anne Patrick of 16 Park Avenue, Cammeray, 2450, NSW

Ines Brown of 13 Raymond Road, Neutral Bay, 2089, NSW

Steve Wallace of 40 Euroka Street, North Sydney, 2154, NSW

or survivors thereof, for their own use absolutely, if all or any of them are then alive.

Individuals Omitted From Bequests

10. If I have omitted to leave property in this Will to one or more of my heirs as named above or have provided them with zero shares of a bequest, the failure to do so is intentional.

GENERAL PROVISIONS

No Contest Provision

11. If any beneficiary under this Will contests in any court any of the provisions of this Will, then each and all such persons shall not be entitled to any devises, legacies, bequests, or benefits under this Will or any codicil hereto, and such interest or share in my estate shall be disposed of as if that contesting beneficiary had not survived me.

Severability

12. If any provisions of this Will are deemed unenforceable, the remaining provisions will remain in full force and effect.

Signature

13. I, **John Wallace**, the within named testator, have to this my last Will contained on this and the preceding pages, set my hand at the City of **Sydney**, in the Commonwealth of Australia, this **Thu, 04 Jul 2019** I declare that this instrument is my last Will, that I am of the legal age in this jurisdiction to make a Will, that I am under no constraint or undue influence, and that I sign this Will freely and voluntarily.

ygiyiback

WITNESSES

This instrument was signed on the above written date by **John Wallace**, and in our presence the testator declared this instrument to be their last Will. At the testator's request and in the presence of the testator, we subscribe our names as witnesses hereto.

Each of us observed the signing of this Will by **John Wallace** and by each other subscribing we witness and affirm that each signature is the true signature of the person whose name was signed. Each of us is now the age of majority, a competent witness and resides at the address set forth after their name.

To the best of our knowledge, the testator is the age of majority or otherwise legally empowered to make a Will, is mentally competent and under no constraint or undue influence.

We declare under penalty of perjury under the laws of the Commonwealth of Australia that the foregoing is true and correct

Signed by **John Wallace** in our presence and then by us in their presence.

Signature **walletaddress**

Name **Brian Bellhaus**

Address **McPhee & Associates, Bayview, 2104, NSW**

Signature **walletaddress**

Name **Margaret Talbot**

Address **14 Prince Street, Underdale, 5032, SA**

Address **McPhee & Associates, Springwood, 2777, NSW** Address **14 Prince Street, Underdale, 5032, SA**

ASP Code – Logic Program

ASP 'logic program' for contract type 'Will and Testament': (load to Will_v01_contract.lp file)

```
% Define Intermediate Concepts ===== %Group 1
witnessed() :-                                     %1①
    entity(witness1, Witness1, _, _, _),
    entity(witness2, Witness2, _, _, _),
    Witness1 != Witness2.

executable(Testator, DOD, Wallet, Residue, ExecuteDate, Executor, Costs) :- %1②
    witnessed,
    death(Testator, DOD),
    execute_will(Testator, Debt, Fees, ExecuteDate, Executor),
    Residue = Estate-Costs,
    Costs = Debt+Fees,
    entity(testator, Testator, _, Wallet, Estate),
    entity(executor, Executor, _, _, _).

disqualifying_death(Entity, DaysAfter) :-          %1③
    DaysAfter = Date-DOD,
    DaysAfter < 30,
    death(Entity, Date),
    executable(Testator, DOD, _, _, _, _, _).

qualifying_beneficiary(Entity, Wallet) :-          %1④
    entity(beneficiary, Entity, _, Wallet, _),
    not disqualifying_death(Entity, _),
    not contests(Entity, _),
    inherits(Entity, Percent),
    Percent > 0.

-wipeout :-                                       %1⑤
    qualifying_beneficiary(Entity, Wallet).

wipeout :-                                       %1⑥
    not -wipeout.

% Distribution to Beneficiaries adjusted for deaths ===== %Group 2
orig_percent(Entity, Percent) :-                 %2①
    entity(beneficiary, Entity, _, _, _),
    inherits(Entity, Percent).

qual_percent(Entity, Percent) :-                 %2②
    entity(beneficiary, Entity, _, _, _),
    inherits(Entity, Percent),
    qualifying_beneficiary(Entity, _).

sum(original, Sum) :-                           %2③
    Sum = #sum{ Percent, Entity : orig_percent(Entity, Percent) }.

sum(qualifys, Sum) :-                           %2④
    Sum = #sum{ Percent, Entity : qual_percent(Entity, Percent) }.

adjust(Ratio) :-                                %2⑤
    not wipeout,
    Ratio = (OrigSum*1000/QualSum),
    sum(original, OrigSum),
    sum(qualifys, QualSum).

transfer(Testator, SourceWallet, InheritAmt, Beneficiary, Wallet) :- %2⑥
    not wipeout,
    executable(Testator, DOD, SourceWallet, Residue, ExecuteDate, Executor, Costs),
    qualifying_beneficiary(Beneficiary, Wallet),
    InheritAmt = ((Residue * Percent/100) * Ratio)/1000,
    inherits(Beneficiary, Percent),
    adjust(Ratio).

% Distribution if Wipeout (all Beneficiaries dead) ===== %Group 3
qualifies_on_wipeout(Entity, Wallet) :-          %3①
    wipeout,
    entity(parent, Entity, _, Wallet, _),
    not disqualifying_death(Entity, _),
    not contests(Entity, _).

qualifies_on_wipeout(Entity, Wallet) :-          %3②
    wipeout,
    entity(sibling, Entity, _, Wallet, _),
    not disqualifying_death(Entity, _),
    not contests(Entity, _).
```

```

wipeout_count(Count) :-                                     %3③
    Count = #count{ Entity, Wallet : qualifies_on_wipeout(Entity, Wallet) }.

transfer(Testator, SourceWallet, InheritAmt, Entity, Wallet) :- %3④
    wipeout,
    executable(Testator, DOD, SourceWallet, Residue, ExecuteDate, Executor, Costs),
    qualifies_on_wipeout(Entity, Wallet),
    InheritAmt=Residue/Count,
    wipeout_count(Count).

% Distribution of Fees and Debt ===== %Group 4
transfer(Testator, SourceWallet, Costs, Executor, Wallet) :- %4①
    executable(Testator, DOD, SourceWallet, Residue, ExecuteDate, Executor, Costs),
    entity(executor, Executor, _, Wallet, _).

% Outputs ===== %Group 5
message_to_human(Executor, Testator, "insert messages to the executor here...") :- %5①
    transfer(Testator, _, _, Executor, _),
    entity(executor, Executor, _, _, _).

#show message_to_human/3. %5②
#show transfer/5. %5③

```

ASP Code – Facts

ASP ‘fact’ templates supplied by a central authority

```

entity(iph_type, iph_entity, iph_entityDOB, iph_entitywallet, iph_entitybalance)
inherits(iph_beneficiary, iph_percentage).
creation(iph_location, iph_date).

```

These templates are instantiated to produce facts: (load to Will_v01_facts.lp file)

```

creation(sydney, 43616).
entity(testator, "john wallace", 20088, "johnwalletaddress", 1000000).
entity(executor, "james stewart", 18218, "jameswalletaddress", 350).
entity(beneficiary, "fred wallace", 29041, "fredwalletaddress", 5).
entity(beneficiary, "bec smith", 29432, "becwalletaddress", 8).
entity(beneficiary, "sam wallace", 29969, "samwalletaddress", 15).
entity(beneficiary, "jim wallace", 30376, "jimwalletaddress", 140).
entity(beneficiary, "may fredricks", 31310, "maywalletaddress", 24).
inherits("fred wallace", 10).
inherits("bec smith", 20).
inherits("sam wallace", 10).
inherits("jim wallace", 20).
inherits("may fredricks", 40).
entity(parent, "tom wallace", 12965, "tomwalletaddress", 200).
entity(parent, "aida wallace", 13378, "aidawalletaddress", 200).
entity(sibling, "anne patrick", 19378, "annewalletaddress", 9000).
entity(sibling, "ines brown", 20255, "ineswalletaddress", 5000).
entity(sibling, "steve wallace", 21459, "stevewalletaddress", 6000).
entity(witness1, "brian bellhaus", 22692, "brianwalletaddress", 800).
entity(witness2, "margaret talbot", 24042, "margaretwalletaddress", 16).

```

ASP Code – Events

ASP ‘event’ templates supplied by a central authority

```

death(iph_entity, iph_date).
contests(iph_entity, iph_date).
executewill(iph_testator, iph_debt, iph_fees, iph_date, iph_executor).

```

These templates are instantiated produce events: (load to Will_v01_events_43680.lp file)

```

death("john wallace", 43646).           The Testator, ‘John Wallace’ died 8Jun2019
contests("sam wallace", 43660).         John’s son Sam has contested the Will
execute_will("john wallace", 1249, 1000, 43680, "james stewart").
                                         The Executor pays out John’s Credit Card debt of $1,249.
                                         The fees for winding up the estate are $1,000.
                                         The Executor, ‘James Stewart’ decides the Will can be executed.

```

This example is test case 3.

ASP Execution Command

```

clingo Will_v01_facts.lp Will_v01_contract.lp Will_v01_events_43680.lp

```


Sample Test Cases showing ASP Event code and test Results as Answer Sets

Test	Explanation	Events File (Will_v01_events_43680.lp)	Expected Answer Set
1	Death of Testator recorded on 2Jul2019 (43646 days fm 1/1/1900). No activity because Executor has not authorised execution of Will.	death("john wallace",43646). execute_will("john wallace",1249,1000,43680,"james stewart").	transfer("john wallace","johnwalletaddress",99775,"fred wallace","fredwalletaddress") transfer("john wallace","johnwalletaddress",199550,"bec smith","becwalletaddress") transfer("john wallace","johnwalletaddress",99775,"sam wallace","samwalletaddress") transfer("john wallace","johnwalletaddress",199550,"jim wallace","jimwalletaddress") transfer("john wallace","johnwalletaddress",399100,"may fredricks","maywalletaddress") transfer("john wallace","johnwalletaddress",2249,"james stewart","jameswalletaddress") message_to_human("james stewart","john wallace","insert messages to the executor here...")
2	All 5 children are alive so distribution is as per allocated %. That is: Fred 10%, Bec 20%, Sam 10%, Jim 20%, May 40%. Note that only \$997,751 of the original \$1,000,000 (the Residue) is distributed because of debts (\$1,249) and fees (\$1,000). The 'message_to_human()' sends instructions to the Executor about the Testator's Will.	death("john wallace",43646). execute_will("john wallace",1249,1000,43680,"james stewart"). contests("sam wallace").	transfer("john wallace","johnwalletaddress",110850,"fred wallace","fredwalletaddress") transfer("john wallace","johnwalletaddress",221700,"bec smith","becwalletaddress") transfer("john wallace","johnwalletaddress",221700,"jim wallace","jimwalletaddress") transfer("john wallace","johnwalletaddress",443400,"may fredricks","maywalletaddress") transfer("john wallace","johnwalletaddress",2249,"james stewart","jameswalletaddress") message_to_human("james stewart","john wallace","insert messages to the executor here...")
3	Sam has contested the Will, and this has been recorded by the Executor. Consequently, Sam is cut out of the Will (Clause 9). Note that Sam's share has been reallocated to the other children.	death("john wallace",43646). execute_will("john wallace",1249,1000,43680,"james stewart"). contests("sam wallace").	transfer("john wallace","johnwalletaddress",249437,"tom wallace","tomwalletaddress") transfer("john wallace","johnwalletaddress",249437,"arne patrick","arnewalletaddress") transfer("john wallace","johnwalletaddress",249437,"ines brown","ineswalletaddress") transfer("john wallace","johnwalletaddress",249437,"steve wallace","stevewalletaddress") transfer("john wallace","johnwalletaddress",2249,"james stewart","jameswalletaddress") message_to_human("james stewart","john wallace","insert messages to the executor here...")
4	Children and Mother wiped out in plane crash on way to funeral before the 29 days after the Testator, so no longer qualify as beneficiaries. The Residue will be shared between Parents (Tom) and Siblings (Anne, Ines, Steve).	death("john wallace",43646). execute_will("john wallace",1249,1000,43680,"james stewart"). death("fred wallace",43675). death("bec smith",43675). death("sam wallace",43675). death("jim wallace",43675). death("may fredricks",43675). death("aida wallace",43675).	transfer("john wallace","johnwalletaddress",249437,"tom wallace","tomwalletaddress") transfer("john wallace","johnwalletaddress",249437,"arne patrick","arnewalletaddress") transfer("john wallace","johnwalletaddress",249437,"ines brown","ineswalletaddress") transfer("john wallace","johnwalletaddress",249437,"steve wallace","stevewalletaddress") transfer("john wallace","johnwalletaddress",2249,"james stewart","jameswalletaddress") message_to_human("james stewart","john wallace","insert messages to the executor here...")
5	Child Fred Wallace was not on the plane, but dies in a car accident 2 days later (31 days), so still qualifies as a beneficiary. The Will rules say Fred's Estate gets the entire Residue. Notes: 43646 = 02Jul2019 death of Testator 43675 = 31Jul2019 29 days wipeout of Mother and 4 of 5 children 43677 = 02Aug2019 31 days death of last child 43680 = 05Aug2019 34 days Will is executed	death("john wallace",43646). execute_will("john wallace",1249,1000,43680,"james stewart"). death("bec smith",43675). death("sam wallace",43675). death("jim wallace",43675). death("may fredricks",43675). death("aida wallace",43675). death("fred wallace",43677).	transfer("john wallace","johnwalletaddress",997750,"fred wallace","fredwalletaddress") transfer("john wallace","johnwalletaddress",2249,"james stewart","jameswalletaddress") message_to_human("james stewart","john wallace","insert messages to the executor here...")

Group	ID	Test Case	Expected Result	Success	Notes
0		Missing data			
0	1	no facts or events	no answer	✓	
0	2	no testator	no answer	✓	
0	3	no executor	no answer	✓	
0	4	no beneficiaries	wipeout distribution	✓	
0	5	one witness missing	no answer	✓	
0	6	both witnesses missing	no answer	✓	
0	7	no beneficiaries, no witness	no answer	✓	
0	8	no distribution	wipeout distribution	✓	Unlikely as SCE controls input

1	Testator death only			
1	1	no activity	no answer	✓ Case 0
1	2	testator dies	no answer	✓ Case 1
1	3	testator dies, executor authorises	full distribution	✓ Case 2
2	No Testator death + other events			
2	1	executor dies	no answer	✓ Executor is a business, this is a role
2	2	beneficiary dies	no answer	✓
2	3	parent dies	no answer	✓
2	4	sibling dies	no answer	✓
2	5	witness dies	no answer	✓
2	6	executor authorises	no answer	✓
3	Testator death + other events			
3	1	testator dies, executor dies, executor authorises	full distribution	✓ Executor is a business, this is a role
3	2	testator dies, beneficiary dies, executor	modified distribution	✓ remaining beneficiaries get more
3	3	testator dies, parent dies, executor authorises	full distribution	✓ impact only on wipeout
3	4	testator dies, sibling dies, executor authorises	full distribution	✓ impact only on wipeout
3	5	testator dies, witness dies, executor authorises	full distribution	✓ no impact
3	6	testator dies, executor authorises	full distribution	✓ full distribution
4	A Beneficiary contests will			
4	1	testator dies, executor authorises	modified distribution	✓ Case 3
4	2	testator dies, beneficiary dies, executor authorises	modified distribution	✓ 2 beneficiaries don't qualify
4	3	testator dies, all beneficiaries contest, executor authorises	wipeout distribution	✓
5	If Wipeout			
5	1	testator dies, all beneficiaries die within 30 days	wipeout distribution	✓
5	2	testator dies, all beneficiaries die after 30 days	full distribution	✓
5	3	case 51 + one parent dies	modified wipeout distribution	✓ Case 4
5	4	case 51 + only 1 sibling alive	modified wipeout distribution	✓ 1 sibling gets the entire amount
5	5	case 51 + no parents or siblings left alive	Message to executor	✓ message to executor
6	Timing impacts			
6	1	testator dies, wipeout, executor authorises	wipeout distribution	✓
6	2	testator dies, wipeout, sole beneficiary dies 31 days later, executor authorises	dead beneficiary gets everything	✓ Case 5
6	3	case 62 but 30 days later	dead beneficiary gets everything	✓ corner case

- | | | |
|------------------------------|------------------------------|------------------------------|
| Will_v01_other.lp | Will_v01_other_testcase24.lp | Will_v01_other_testcase54.lp |
| Will_v01_other_testcase01.lp | Will_v01_other_testcase25.lp | Will_v01_other_testcase55.lp |
| Will_v01_other_testcase02.lp | Will_v01_other_testcase26.lp | Will_v01_other_testcase61.lp |
| Will_v01_other_testcase03.lp | Will_v01_other_testcase31.lp | Will_v01_other_testcase62.lp |
| Will_v01_other_testcase04.lp | Will_v01_other_testcase32.lp | Will_v01_other_testcase63.lp |
| Will_v01_other_testcase05.lp | Will_v01_other_testcase33.lp | Will_v01_other_tst.lp |
| Will_v01_other_testcase06.lp | Will_v01_other_testcase34.lp | |
| Will_v01_other_testcase07.lp | Will_v01_other_testcase35.lp | |
| Will_v01_other_testcase08.lp | Will_v01_other_testcase36.lp | |
| Will_v01_other_testcase11.lp | Will_v01_other_testcase41.lp | |
| Will_v01_other_testcase12.lp | Will_v01_other_testcase42.lp | |
| Will_v01_other_testcase13.lp | Will_v01_other_testcase43.lp | |
| Will_v01_other_testcase21.lp | Will_v01_other_testcase51.lp | |
| Will_v01_other_testcase22.lp | Will_v01_other_testcase52.lp | |
| Will_v01_other_testcase23.lp | Will_v01_other_testcase53.lp | |

```
c:\TST> cllingo Will_v01_contract_tst.lp Will_v01_other_testcase63.lp  
c\lingo version 4.5.4  
Reading from Will_v01_contract_tst.lp ...  
Solving...  
Answer: 1  
transfer("john wallace","johnwalletaddress",997750,"fred wallace","fredwalletaddress")  
transfer("john wallace","johnwalletaddress",2249,"james stewart","jameswalletaddress")  
message to human("james stewart","john wallace","insert_message_to_Executor_here...")  
SATISFIABLE
```

Appendix D – Mid Complexity – ‘Real Estate Sale’

Analysis – Instantiation Variables required

Complex types

```
entity      ( __seller__, __buyer__, __earnestmoneyholder__,  
              __nominatedmediator__, __nominatedarbitrator__ )  
asset  
list       ( __property__ )  
            ( __propertylegaldescription__, __included__, __excluded__,  
              __exceptions__, __closingconstraints__, __buyersclosingobligations__,  
              __costspayerlist__, __additionalagreements__ )
```

Simple types

```
__earnestmoneydeposit__  
__sellerloan__  
__sellerinterestrate__  
__sellerloanrepayfrequency__  
__sellerloanrepaystart__  
__cashatclosing__  
__price__  
__creditcheckdate__  
__notifydate__  
__responsesdays__  
__demandreturndays__  
__loancommitmentbydays__  
__maxloaninterestrate__  
__minloanyears__  
__minloanamount__  
__reportnames__  
__deadlinedate__  
__repairpercentage__  
__repaircredit__  
__utilitiesstatus__  
__utilitiespayer__  
__casualtymaxpercent__  
__warrantypayer__  
__warrantymaxdeductible__  
__warrantymaxcost__  
__titleinsurancepayer__  
__maxmortgagetermschange__  
__surveyrequired__  
__taxprorationrequired__  
__closingdate__  
__closingdeliverable__  
__disputeprocedure__  
__expirationdate__  
__expirationtime__  
__buyerassign__  
__earnestmoneysignature__
```