Semantically Rich Smart Contract System

Jang Vijay Singh

Kellogg College
University of Oxford

A dissertation submitted in partial fulfilment of the requirements for the degree of Master of Science in Software Engineering

Abstract

Contracts are all around us in today's world. These can range from simple agreements between parties to complex legal documents that impact lives. The general public could often not be aware of the full implications of contractual terms and conditions that govern them. Expert legal advice can have prohibitive costs and be out of reach for many, but this can have serious implications. Dispute resolution often requires reliable evidence of obligations fulfilled that aligns with key stages of the contract lifecycle. The purpose of this dissertation is to explore if a technical solution involving a fusion of semantic technologies, blockchain, and BPM can alleviate some of these issues. From background research, this synthesis has not been tried before but these technologies individually offer specific value in this area. Many contractual scenarios in day to day life can be highly complex but also highly repetitive, requiring less subjective interpretation. But software contract management is still text or document based. This thesis designs and examines the feasibility of a semantically rich "smart" contract lifecycle management system. It will evaluate how a combination of these technologies can support partly automated contract verification, auditing compliance, and scaling of expert skills.

Acknowledgements

The Oxford experience has been truly remarkable, and, with its ups and downs, it instilled in me new ways of thinking, opened up new possibilities for the future, and broadened my mind in ways that I never knew possible. Not many people can claim to have studied in the same department that Sir Tim Berners-Lee teaches in or to have addressed a packed audience at the Oxford Union with world renowned dignitaries in attendance. The learning from this program is only a start and will only grow as I apply it in the real world. It would be impossible for me to name every individual who helped me in this program, but I will make a sincere attempt.

To Dr Peter Bloodsworth, my current supervisor, I owe a lot of gratitude for being so accessible, for his prompt feedback and the time he took to support and guide my work. I can count him as one of the best tutors throughout my academic life and if I excel in this program, a lot of credit goes to his guidance right from project week onwards. Prof Jeremy Gibbons is my college advisor and has remained supportive and approachable throughout. The motivation I felt from our first conversation over a pint of Oxford Scholar early in the program stayed with me throughout. I feel privileged to have met and interacted with accomplished academics like him and leading industry professionals who conducted some of the modules. Special thanks are due to Dr Monika Solanki and Dr Alejandra Gonzalez-Beltran who conducted the Semantic Technologies module and introduced me to this particular area. Prof Cas Cremers' Security Principles course was my first in the program and it was his introductory talk at Oxford on linked data that got me interested in the Software Engineering Program. Prof Ivan Flechais deserves special thanks for his early advisory sessions and ensuring I stayed on track, duly forewarned about the level of hard work required to excel in the program.

In many ways, this dissertation is the summation of my study of the last three and a half years. Every module that I attended not only added to my knowledge but also equipped me to think about Software Engineering problems from multiple perspectives. I would like to thank all my colleagues and other participants of the Software Engineering program - Sanjay Shambhu in particular for being a friend first and for proofreading one of my drafts.

Finally, for planting the seed in my mind, for making sure I aspired, aimed higher than I settled for, and persevered, I thank my father Prof Jang Bahadur Singh and my mother Dr Niranjan Kaur. Their encouragement and support meant that my sister and I matriculated in Oxford on the same day and will hopefully use our learning to make a positive difference in the world.

Contents

List of Figures

List of Tables

1 Introduction

Contracts are agreements between parties - individuals or abstract entities like corporations. More formally, [McKendrick, 2017] quotes Professor Atiyah's definition of a contract as: *"a discrete, two-party, commercial, executory exchange..."*. Landlord-tenant contracts as an exception in that they are continuing as opposed to "discrete". Simple contracts can be verbal agreements or clauses put on paper, dated, and signed. In any contract, large or small, what are the means we have of answering these questions?

- Do clauses of the contract contradict each other?
- Do any of the clauses contradict "the law", and therefore, are such clauses invalid?
- What are the legal steps needed in the performance of the contract, and, does there exist a high-integrity, trustworthy, irrevocable, and immutable audit trail of performance?

As an empirical observation, most contracts in day to day life are negotiated, executed, and concluded smoothly. Parties can also manually review or negotiate simple contracts in natural language and determine answers to the above questions using *common sense* criteria. More complex contracts could require inputs from legal experts, unless either party wishes to take on some risk of being tied in to unexpected clauses. The truth of modern life is that we are bound by a number of contracts of which we are not fully aware - purchasing a good or service puts consumers under a number of obligations set by just one party - the seller (the only protection available to customers is often general consumer protection laws such as distance selling regulations). Terms and conditions of software are even more elaborate and often consumers are never fully conversant with these. The actual enforceability of many such terms is often doubtful and is only determined once a particular contract or clause reaches litigation - legal experts often rely on precedent set by past cases when giving their 'opinion' on contractual matters. To the general public, such specialised knowledge is not very accessible, and they might well assume all parts of a contract imposed on them necessarily apply. paragraph Contracts can be considered as a set of rights, obligations, and prohibitions. [Chalkidis et al., 2018] use these terms from the legal field in the context of parsing legal text to extract obligation and prohibition clauses for use by software systems, while [Kabilan and Johannesson, 2003] offer a contract ontology using the same terminology. For parties to most modern day contracts, beyond very simple or "common sense" terms, understanding *all* their obligations, prohibitions, and rights realistically needs expert help which is not very readily available. It is, for instance, impractical for a prospective employee to hire a lawyer to fully understand their contract of employment, which involves, not only the written prose in their paper and ink contract, but also *inherited* clauses from employment related legislation, past cases, union agreements, *"generally accepted"* principles and possibly more. It is not easy for typical employees lacking professional legal support to understand if their particular instance of an employment contract puts them under some obligations that are not *well known* or *generally accepted* in a contract of employment, or, if it's taking away some rights from them. Most of the current approach around

contract technology is focused on either annotating text with machine interpretable metadata OR processing of legal texts to extract machine interpretable snippets (entities, statements). This thesis proposes complete lifecycle management of contracts by using semantically rich, machine interpretable contracts from the start. In practical terms, this means defining contracts as ontologies linked to legislation, also defined as ontologies that capture as much of the domain knowledge as possible. This could leverage the ability of machines to *infer* additional information from a given subset of data and a set of rules to help determine the "next steps" in a workflow, or identify previously unknown clauses, or identify contradictions.

1.1 What are the Problems?

From many real-world examples involving contracts, we can identify some common problems (by no means an exhaustive list, but representative):

- Access to complex legal steps is limited for general public. Expert legal support can be available from experienced individuals who are few in number, but that information can often be repetitive.
- Legal information or legislation that applies to a particular instance of a contract can come from different sources and can be frequently changing. It is hard for non-professionals to keep track of or to understand the implications of subtle changes in legislation - even if legislation might ultimately impact such people.

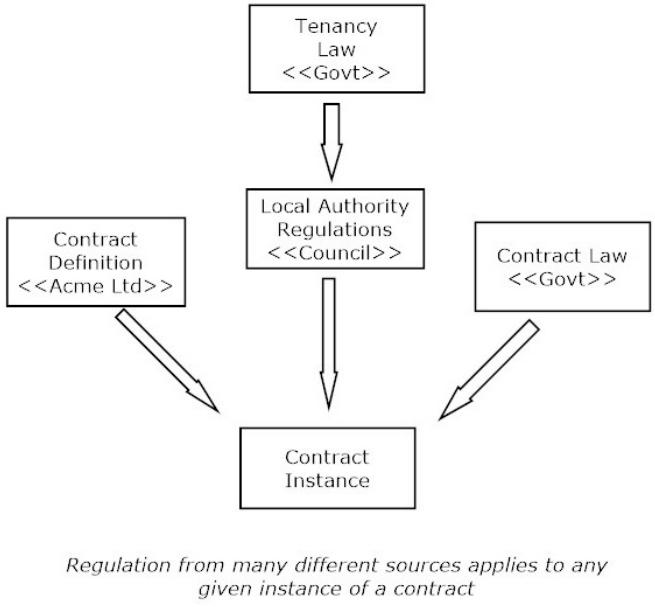


Figure 1.1: Possible inheritance of legal information for tenancies

- Legal steps involved in some contractual workflows (such as an eviction) might not be very clear or "spelt out" unambiguously to non-professionals like private landlords.
- There is an imbalance in legal information available to different types of parties. In such unequal contracts, the larger party can get away with contractually taking away legal rights of individuals who might not be aware of their rights.

- A peculiar situation can arise during contract negotiation or execution when parties often exchange documentation (purchase orders, invoices, quotes, payment advices, delivery notes) with terms and conditions written on them. The fine print often includes clauses such as "receipt of this document/contents means acceptance of certain clauses". In the event of a dispute, it can be hard to detect which final clauses applied in a given situation. A software-executed and semantically rich contract could avoid such ambiguous situations.

- For common situations that are reasonably well defined, there is a dearth of experts to help parties lacking legal knowledge, like private landlords and tenants, with repetitive processes. The cost of accessing their expertise can be prohibitive.

- In tenancies, fake tenancies appear to be surprisingly common, where unauthorised parties are able to let properties to unsuspecting tenants, who in turn assume certain legal rights by virtue of their habitation, but to the detriment of the legitimate landlord.

- 3-5% of online transactions result in disputes, according to "Digital Justice: Technology and the Internet of Disputes", book published in 2017 with an authors' talk hosted at Oxford by the FLJS[1]. It also proposes the idea of *"court as a service rather than a place"*, although critics have called such initiatives a form of *"second class justice for those with limited means"*, that could *"alienate the digitally deprived"*, and motivated by an *"unhealthy preoccupation with cost reduction"*. This concept is noteworthy, as are the criticisms, because, the areas explored in this thesis could be used to support such initiatives but in a manner that adequately addresses all stakeholder perspectives.

1.2 History and Related Applications

Legal information systems in different forms have been proposed and attempted alongside advancement in computing, often with different **goals** and different expectations from what an 'intelligent' system ought to do (see [McCarty, 1982]'s discussion on *"document"* retrieval versus *"sophisticated form of information"* retrieval). [Bashir, 2017] describes *Ricardian Contracts* developed in 1990s that consisted of documents containing legal prose (text) annotated with machine readable tags. These were digitally signed by the issuer's private key and hashed to uniquely identify the original. If they got processed by different parties such as departments in an enterprise, each party signed the hash using its own private key. In its focus on legal prose and lack of semantic richness, this is no different from the digital *text* document based contracts in common use today, as their main content remains text.
"Smart contracts", as they are understood today, focus on execution rather than semantic richness. A contract written fully as *software code* could include standard computer programs in a standard procedural language. This is somewhat the current level of maturity of blockchain *"smart contracts"* which are written in procedural languages or derivatives of popular procedural languages[2]. That being said, logic based, declarative programming has clear advantages over imperative programming in this particular area. In software executable contracts, the distinction and advantages have been illustrated by [Idelberger et al., 2016] using a sample contract as example. And this stage, we have a reasonably mature set of technologies that can be combined to deliver working solutions with this paradigm. Blockchain systems can additionally offer the trust framework for compliance monitoring, auditing, and, eventually, full

[1] Foundation for Law, Society, and Justice, based at Wolfson College, Oxford
[2] Ethereum based smart contracts are written in a 'turing complete' language called Solidity citesoliditydocs

execution of complex logic. Ontology knowledge bases can be used to bridge the semantic gap that would allow logic based programs to work (off chain or on-chain, depending on feasibility and pragmatic tradeoffs). Ontology driven, rules based semantic web expert systems already find use in the area of compliance management [IEEE, 2007]. I, therefore, commence the design and development of this project with some ideas around what the ideal solution ought to be, and what mix of technologies could help achieve it.

1.3 Rationale for Selecting this Problem Domain

Many parts of the world have serious limitations in affordable access to justice. Increasing court backlogs are a challenge[3]. A lot of this burden could be shifted to automated or semi-automated systems, facilitating compliance, and by first improving accessibility of legal information. Facilitating free and easier access to legal information is in itself a goal which technology can facilitate - [Casanovas et al., 2016] describes existing publication initiatives as a:

"...contribution to the development of the rule of law from a global point of view."

Additionally, technology can facilitate easier compliance with legal obligations, and provide guided steps to prevent mistakes in places such as the UK where mature (but complex) legislation does exist. The academic objective to utilise and apply knowledge of various relevant subject areas from the Software Engineering Program and attempt to learn some new principles and tools. I hope to apply a broad range of knowledge from relevant subject areas, tools and techniques into the design of a viable software system. Functionally, the accompanying proof of concept will address a narrow area from within a larger functional domain, so that this can be evaluated as a representative exercise in the application of Software Engineering tools and techniques. Due to the opportunity this offered to apply a novel combination and **wide range** of techniques, this was chosen over the alternative, which would have been to focus deeply on one software process, method, or tool with or without the need to address a specific functional area.

The area of *assured shorthold tenancy* (AST) was identified a relatively constrained field from the legal domain, for which, legislation and rules targeted at members of public can be easily acquired via publicly accessible material. The goal of this dissertation is to assess, with the support of a working prototype, whether the proposed system is feasible given the current state of the art, to identify bottlenecks overall design, and to conclude or reflect on technical feasibility of the system supported by suitable proof(s) of concept(s). It is not the intention of this project to be evaluated on accuracy of legal information or the linked data representations thereof, but it is to prove that (or identify specific challenges in) the solution offered is possible and offers specific value over methods commonly used now. In the process, we apply a broad combination of processes, tools, and techniques in an integrated way to support a completely unfamiliar domain. We feel this could serve as a reference framework that could be applied in similar technical scenarios to support other similar functional areas (like supporting employment contracts).

1.4 Application Area and Aims

The scope of the proposed system has been chosen to be ambitious to make it relatable to a real-world application pushing new boundaries. It also offers an opportunity to apply software

[3]From notes made at FLJS event

engineering principles at different levels: architecture of a full fledged system, design of the main components, design and proof of concept of the integration points, and proof of concept implementation of a small subset of the system. The goal is to see how far we can go with technologies currently available, and to prove the working of a yet untried collaboration of technologies. While the system design defines a higher scope to showcase the overall possibility (and to serve as a supporting rationale for this research), the implementation accompanying this research aims to test these claims (or identify future work required) with a limited proof of concept. To list the aims from this research:

a) To examine if the proposed system can be built using existing technologies. Working together, rules, semantic web, Business Process Orchestration and Management (BPM), and distributed ledger technologies can be powerful in a legal setting - something not explored yet based on published material that I studied. Whereas each of the technologies offer tangible value on its own, it is important to see how these can together benefit a software system in a synthesis not tried before. Specifically, what functional or non-functional properties will they support and how? The project will aim to identify technical challenges, bottlenecks, and a viable solution around these. There are known issues around scalability of leading blockchain platforms, but, this should not impact the solution because of the non real time nature of the usecase. but it is to find out if we can deliver a solution around these with an adequately flexible, extensible, and future-proof architecture.

b) To show, backed by previous research and examples, how legal information can be represented and used in an ontology (triples) or a combination of ontologies and rules, at least for limited, well defined and constrained areas of the law like assured shorthold tenancy.

c) To show how contractual information represented as triples, with the help of reasoners, can be used to determine validity of a contract. Combined with BPM, blockchain based smart contracts, and runtime information aggregated in a distributed ledger, this can be extended to determine the next steps in a guided process, trigger warnings, notifications, and even automatically perform obligations like rent payments.

1.5 Evaluation Strategy

The nature of the project is not exclusively product design, as there is significant effort involved in background research, technology research, prototype implementation, and evaluation. The success criteria would be to first assess if a solution can be developed with a synthesis of currently available and new technologies. This will be supported with a working prototype. Integration points and flow of data will be tested through this prototype system. Because nothing of a similar nature can be found currently, it would not be possible to make an objective comparison. Due to the time and financial constraints, and scope of an MSc dissertation, it is not possible to engage actual legal experts or users to evaluate the solution. This is addressed by creating a representative set of requirements from different stakeholder perspectives. The evaluation of this project consists of how feasible the design is backed by supporting research, to what extent it is able to address or work around known issues, and, proving the particular novel aspects of the design with a working prototype.

The technical goal of the project would be to experiment with new technologies to try to link them together in an architecture that is quite general. It would also be to evaluate how practical

or feasible such a solution is to implement, to learn lessons, and to draw conclusions. If this project delivers the required framework and working solution even for a simple scenario, it lays the groundwork for use of reasoning tools with legal contracts developed as ontologies to help ascertain validity of clauses in a contract (such as contradictory clauses or something that contradicts "the law"), determine obligations that might arise from it (like a schedule of rent payments, total liability) or draw helpful inferences on the current "state" of a contract (a delayed rent payment means the contract as the "state machine" is in a warning state). Distributed ledgers can be used as a reliable, transparent and verifiable means of tracking the steps involved in the execution of contracts - the "fulfilment of obligations" and an inbuilt monitoring of it. Based on this rationale and supporting literature, this project will, therefore, use knowledge bases defined as ontologies in combination with rules as a method of encoding contractual domains, data, rules, and contract instances. Additionally, Business Process Management (BPM) can help define transparent, flexible workflows, and facilitate the collaboration of highly specialised professionals in supporting non-technical users **at scale**. Unless this is implemented for actual users at a significant cost, testing this at scale is not possible. The evaluation will therefore, consist of justification of the architecture, proof of concept with an accompanying prototype, and testing the data flow through integration points. A sample case that guides us through the prototype implementation and evaluation is the process of creating the tenancy agreement. How this fits into a larger context of current and future requirements, possibilities, and features of the proposed system should be clearer in chapter 3 on Requirements.

1.6 Dissertation Structure

This chapter was used to set a scene and identify what this project is meant to achieve. We note the main limitations of software systems that currently provide contract lifecycle management functionality. The chapter identified how such systems could offer better, more *intelligent* functionality but that would require a fundamental shift in their implementation from the ground up. Notably, we arrive at a hypothesis for which emerging areas of technology would make such innovations possible - notably, semantic technologies including semantic reasoning, blockchain enabled distributed ledgers, and blockchain based smart contracts. Subsequent chapters will now focus on each of the identified areas one by one, and report the work that systematically leads towards a viable design.

Chapter 2 includes a detailed theoretical study of the areas of technology proposed for use in the solution. The history, context, and description of the working of these technologies would help back my original proposal. A critical examination and study of notable past failures might also identify gaps and shortcomings. A review of literature and tools made during this research would allow us to define a framework to create a viable solution.

Chapter 3 focuses on identifying main usecases, and functional and non-functional requirements that would need to be implemented for their realisation. The main purpose of this chapter is to allow a subsequent evaluation of how well the proposed technical solution might fit some real-world usecases.

Chapter 4 This chapter starts with a technology agnostic high-level architecture that could be applied to multiple domains, a more specific architecture for the chosen domain (tenancy contracts), with subsequent sections elaborating the design and implementation aspects of all components identified in the specific architecture.

Chapter 5 Although a lot of detailed design is covered in chapter 4, this chapter reports on the prototype implemented with specific technologies. It comments on the challenges, followed by an evaluation of the prototype. **Chapter 6** presents a concluding discussion on the experiences

and outcomes of the project along with future work.
In addition to the main chapters and bibliography, supporting details about the setup of test-bed environments, tools used, project tracking, and additional code snippets are presented inline.

2 Background and Literature Review

The project aims identified in chapter 1 relies on the much publicised claims of some new or emerging technologies in a combination that might be unexplored so far. This chapter starts with a brief discussion of the functional domain, and then progresses with a suitably detailed study of the theoretical basis of the proposed technologies, their history, evolution, and working. We identify literature (preference given to peer reviewed, published research or writings by notable contributors in their areas) that would either back the claims and detailed design, or identify key challenges and shortcomings. One of the primary goals of the literature review would also be to identify systems that address the legal domain, and to identify literature that points to past similar (successful or failed) uses of technologies identified. No formal methodology of literature review has been adopted as such, although principles were drawn from [Kitchenham, 2004]'s review (and reasons listed fit well with the project objectives). It is by no means an *"evaluation and interpretation of all available research"*[ibid] of the subject areas concerned, but a pragmatic summary of studied material that supports the proposed system's goals or identifies challenges that might have been encountered before.

2.1 Contracts and Contract Law

The area of contract law is vast and has matured over centuries, if not millennia. Compared to the vast repository of past contracts, legislation, cases, and judgements, any suggestion of algorithmic contract management is nascent at best if not completely unlikely. The key distinction is the deterministic nature of software systems whereas, the law consists of a heavy balance of deterministic and what might appear to be not just subjective but almost instinctive application of human faculties. Code introduces *inflexibility by design* whereas historically, contracts are an almost sociological construct to *"manage human relations"* [Levy, 2017]. This is not to say that the future is not in ambitions that at present appear to be lofty ideals or might have spectacularly failed (concepts like the DAO, algorithmic governance and so forth). The key is to identify the right mix of concepts, principles, and, yes, technologies that allow us to model real world situations in artificial intelligence enough to draw tangible value from them - self-driven cars are a notable example where specific, deterministic, and inflexible software collaborates with more probabilistic *"inferences"* (drawn on large data sets and continuous "learning"). We the previous chapter hopefully served to adequately justify this work, in this chapter, we draw some foundation from the formal subject matter in our areas of exploration. In all this, our guiding principles remain the effort to automate what is human drudgery, to help create a more level playing field where one might not exist between unequal parties. In the concern about unfair terms being imposed on weak parties, the role of institutions like the parliament and judiciary has been to step in with responses like:

- Legislation introduced to regulate employment and consumer credit contracts [McKendrick, 2017].

- Introduction of the "concept of fairness" in law when it comes to specific scenarios (large parties with a large concentration of capital and clout dealing with certain classes of ordinary citizens like employees and consumers in this example).

In other words, these are examples of governments advocating on behalf of and securing the rights of weaker contracting parties that might otherwise be compelled to accept terms vastly weighted against them (This, in fact, deviates somewhat from *"the will theory"* [McKendrick, 2017](p3)). Technology cannot fully replace these initiatives but it has a role to play in making these ideas more accessible, self-executing, and expert-guided. Further, it is to be explored if works derived from this thesis can help apply concepts such as *"consideration, illegality, frustration, duress"*[ibid], or actual statutes such as "Unfair contract terms act 1977" in a more proactive, accessible, and usable manner.
Based on an empirical observation of how contracts are currently managed, it appears current software tools lack the ability to reason on an instance of a contract and determine if a clause is illegal or if the contract instance is contradictory with another one, and this is by no means an easily tractable problem. For example, the idea of a software agent being able to assess the intention of parties in the event of arbitration or to apply the *"discretion and creative power"* [Slapper and Kelly, 2012](p38) that is the prerogative of the judiciary in its role of applying the law that parliament has created [Slapper and Kelly, 2012](page 38) or in arbitration or enforcement of private contracts. Initiatives in software based solutions dealing with the law also need to take into account the fact that any *"ambiguous words* and phrases" can often only be resolved by judicial interpretation, which is a *"creative process"* [Slapper and Kelly, 2012](page 38) with often different rules and techniques of interpretation that could possibly lead to different conclusions. Technology's role here can be to encode such terms and rules, link these with clauses, link these with past cases and keep building a repository of judicial pronouncements - in other words, given an expert system that could identify with a degree of certainty (or a high degree of probability), if a contract instance is legally sound and what the consequences of a particular action might be. This is a very ambitious goal no doubt, but in this thesis, we start with a relatively constrained and well defined area that might offer the chance to test some of these ideas, and offer a framework for continuous enhancement. Our view of the "processes" that would constitute the contract lifecycle are shown in figure2.1. (These are worth comparing with commercial offerings that cater to this area and also the lifecycle of a contract defined in [Kabilan and Johannesson, 2003]). In a mature, multi-level BPM[1] initiative, these could be considered as the top level processes to be developed further into lower levels, and finally, executable processes.

2.2 Use of Semantic Technologies in the Legal Domain

Contracts for common situations usually follow a common structure and common "blanket" clauses. Many industries follow standard form contracts, in areas like employment and assured shorthold tenancies. Most common attempts at *'digitising'* contracts have been restricted to text documents (prose) as opposed to creating more intelligent software based contracts with *semantically rich* content. Digital signatures and other usability features ensure integrity of the written prose, but correct interpretation and execution is still mainly the responsibility of contracting parties. *"Semantic web expert systems"* [Verhodubs and Grundspenkis, 2011] (SWES) have been proposed before, but so far, their role has been positioned as *"query answering"* services. This dissertation attempts to design a *semantically rich smart contract system* that

[1] Well known acronym for Business Process Management

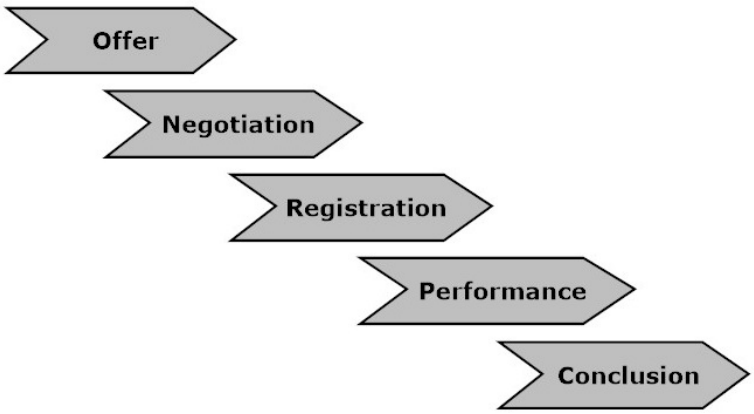


Figure 2.1: Top level processes involved in a contract lifecycle

would guide users through their obligations, offer proactive monitoring, and give them a more *accessible* and *relatable* view of their contract. The most promising way to achieve this appears to be with the use of ontologies, so a more detailed examination of this type of solution proposal follows in chapter 2.

Application of semantic technologies in law is a field with ongoing research [Boer, 2009], [Casanovas et al., 2016]. The most prominent attempts at use of ontologies for official legislation are a project called **Nomothesia** for Greek legislation [Angelidis et al., 2018] supported by the legal linked data standard called **metalex**, a legal linked data standard called LKIF [Hoekstra et al., 2007], and possibly UK legislation published as RDF [legislation.gov.uk,]. The ontologies that these initiatives define deal more with legal metadata (document, act, text) as opposed to real world entities. Other proposals also include modelling the set of permissions, obligations, prohibitions in a contract in OWL DL so that these can be reasoned upon, in addition to a library of *"interpretations a judge makes of the law"* (indicating a rich model of the law also represented in OWL-DL). [Kabilan and Johannesson, 2003] developed a *"Multi-Tier Contract Ontology"* (MTCO) as *"a framework for modelling and representing contractual knowledge"*. It highlights *"bridging the gap between business process management and contract management"* as a key contribution. The approach involves "reverse engineering" of OWL axioms from prose, rather than helping parties define semantic web contracts from start. No doubt, this approach can be matured with better natural language processing (NLP) techniques for AI based entity extraction, but, a *semantic gap* would still remain where it might not need to. This thesis, in contrast with the initiatives described previously, proposes a forward engineering approach where **the law** and domain information are made available as linked data. Linked to these, **contract instances** are then also developed as ontologies via user friendly interfaces, picklists, and restricted NLP techniques to create axioms and rules, and annotated with human-readable narrative.

2.3 Role of Distributed Ledger Technologies in Contract Management

Contract law, private contracts, and legal support for their enforcement have facilitated economic growth in the last few centuries by building a "layer of trust" in private transactions. In computing, public key cryptography has been available since the 1970s and enabled the boom in online commerce by early 2000s, but, every application has worked with a centralised infrastructure. Even small transactions between individuals have involved the presence and

mediation of large centralised financial institutions with their associated costs and overheads. The vision expressed in the original bitcoin paper was

> *"... electronic payment system based on cryptographic proof instead of trust, allowing any two willing parties to transact directly with each other without the need for a trusted third party..."*

Blockchain technology has, in recent times, enabled a "layer of trust" for practical decentralised distributed ledgers, and has the potential to allow closer mirroring of private contracts in software. Trust in parties entering into contract in the offline world is facilitated by a number of factors - past conduct, credit records, but, in many cases, concurrence of multiple neutral parties in the trustworthiness of parties entering the contract. In the blockchain world, this is mirrored by consensus algorithms matured by the bitcoin protocol where multiple participating nodes confirm the validity and sequence of a series of transactions. As this dissertation will explore further in section B, the series of activities involved in the lifecycle of a contract can be likened to a series of transactions executed in a specific sequence on a state machine - in a manner that is transparent, irrevocable, and practically non-repudiable. Bitcoin was the first to come up with a reliable way of determining which transaction came first in such an environment. The Ethereum whitepaper [Buterin et al., 2014] defines *"smart contracts* as automated systems that can move digital assets under specified conditions and after applying some logic. Decentralised Autonomous Organisations (DAOs) are by extention smart contracts that "encode the bye laws" of an entire organisation.
There are interesting analogies between a contract lifecycle and the blockchain ledger. The Ethereum whitepaper likens a blockchain to a state transition system where a contract can be modelled a "state machine" that can react to available facts by applying rules, drawing inferences, and executing transactions in reaction to these inferences. As we see later in chapter 2, the reasoner and the business process are analogous to **state transition functions** that use the current state of the contract and incoming events or new inferred facts as input to produce an output in the form of a *recommended action* or a *compliance status*. All the activity performed in the execution of the contract throughout its lifecycle is available in the ledger in the form of signed, cryptographically verifiable and searchable transactions. It is important to identify the main differences between the idea of contracts as applied in blockchain based *"smart contract"*, and our common understanding of the word "smart" in the context of computer based systems. [Bashir, 2017] quotes Szabo's definition of a smart contract:

> *A smart contract is a computerized transaction protocol that executes the terms of a contract. The general objectives are to satisfy common contractual conditions (such as payment terms, liens, confidentiality, and even enforcement), minimize exceptions both malicious and accidental, and minimize the need for trusted intermediaries. Related economic goals include lowering fraud loss, arbitrations and enforcement costs, and other transaction costs.*

The reader is invited to read a more in-depth explanation of the history, evolution, and working of distributed ledgers which is based on my understanding and literature review in the Appendix B.

2.4 Smart Contracts

The use of the term "smart" in different areas of computing needed to be critically examined first. Many modern applications or devices have been labelled as "smart" - notably those

enabled by the proliferation of sensor based, Internet connected devices classed as IoT and initiatives to embed these in different aspects of city life to develop so called "smart cities". Such devices would generate large volumes of data on which data science techniques could be applied to draw meaningful conclusions, extrapolations, and artificially intelligent predictions - effectively statistical techniques like classification and regression.

Another area in which the word "smart" has gained currency is the so called "smart contracts". [Levy, 2017] defines "blockchain based smart contracts" as systems:

> *...that aim to automatically and securely execute obligations without reliance on a centralized enforcement authority*

Interestingly, the concept finds mention as early as 1997 in a paper by Nick Szabo [Szabo, 1997], an individual whose Wikipedia listing describes him as a "computer scientist, legal scholar and cryptographer" and, possibly like many others, rumoured to be Satoshi Nakamoto himself[2][3]. Szabo's paper [Szabo, 1997], very clearly describes all the functional expectations that we have from smart contracts - mechanisms of enforcement built in to key assets in the form of executable code.

The work that seems to have popularised the idea has to be the Ethereum whitepaper [Buterin et al., 2014] which succeeds the development of the decentralised blockchain technology which then led to inflated expectations from "The DAO" project. The term "Smart Contract" in its current form has been used to describe any script capable of being executed in a blockchain environment that results in transfer of *"deemed"* value, or more accurately, a "change of state" in the distributed ledger (credit to an account, debit from another account, payment of a *fee* to the "book-keeper" or its blockchain equivalent, a miner). If the blockchain based distributed ledger is a "state machine" ([Buterin et al., 2014] specifically describes it as such), complex programs (running within contraints) that result in change of state are what is popularly referred to as "smart" contracts. Our expectations naturally include the natural expectation from the word "smart". These expectations led to the highly publicised "The DAO" project with its many inflated expectations. Briefly, it sought to implement an "organisation", similar to an incorporated company, that empowered all its shareholders (rather, token-holders) in its decision-making or in matters pertaining to use of its funds without a centralised authority wielding power disproportionate to its shareholding. All rules, regulations, and business logic of the DAO was meant to be coded within it in the form of smart contracts. The DAO was also a notable failure by 2016 [DuPont, 2017] due to a vulnerability in the smart contract code that led to the "race to zero" attack [Vessenes, 2016] that allowed the attacker to siphon off tokens worth millions and it was human intervention and collaboration of various human organisations (vigilance of some developers, Ethereum Foundation, various trading exchanges [DuPont, 2017]) that could stop the tokens from being exchanged for traditional funds and then build consensus on a hard fork" (an effective rollback measure that violated the so called "irrevocability" of the distributed ledger) - or in other words, reverting to the flexibility of "social capital" or devolution into "traditional models of sociality" [DuPont, 2017]. For any foreseeable future, we will have to live with the options involving pragmatic human override co-existing alongside trustworthy systems and for designers to revisit the inflexibility of viewing contracts as *"technical artifacts, rather than as social resources"* [Levy, 2017]

[2] https://medium.com/@altcoinbuzz/why-i-think-nick-szabo-is-satoshi-nakamoto-even-though-he-denies-it-8c999841fbbb (Accessed 28th Nov 2018)

[3] https://www.investopedia.com/news/who-nick-szabo-and-he-satoshi-nakamoto/ (Accessed 28th Nov 2018)

2.5 Semantic Technologies and their Applicatications in Law

As we identified the functional goals of distributed ledgers and smart contract, and traced their principles and history, it emerges that there are gaps between expectations and reality. There is, a very fundamental issue with the "semantic gap" - starting with a lack of appreciation of the "social" nature of contracts [Levy, 2017], and an inadequate representation of the legal viewpoint in the purely technological initiatives. The inherent process of legislative interpretation is described as a "creative" process with jurists using many *"rules of interpretation"* [McKendrick, 2017], which must be treated an evolutionary and flexible process requiring human participation AND a convergence of multiple areas of artificial intelligence. The important lesson from this is, technological initiatives for the area of contracts and legal domain in general need to incorporate a more semantically rich knowledge representation.

[McCarty, 1982] recognised the need for applying advanced "computational linguistics" and "artificial intelligence" and the need to go beyond "document" retrieval to "information" retrieval for the legal domain. Very specifically in support of the present thesis, McCarty argues for the construction of a *"conceptual model for the relevant legal domain"* as the *"most critical task in the development of an intelligent legal information system"* [McCarty, 1982, p. 266]. While Semantic Web is quite well established and justified, let us take a step back to revisit this basic principle of good software design. Object Oriented design principles stressed on creating a detailed enough ***conceptual*** model([Fowler, 1997]) in the form of "entities" or "key abstractions" and defining accurate relationships between them (composition, cardinality, inheritance).). If "state" of the world was captured in an "object graph", logic or behaviour could then operate on this data model to change that state - however, any commonalities with the OO paradigm in the context of this thesis is only restricted to aspects of the conceptual model and not on encapsulation of data with behaviour.

At any point in time, good software ought to ensure a consistent state in the object graph. **Relational database schemas**, or, entity-relationship (ER) diagrams are another form of conceptual model definitions. Web service contracts (such as **XML schema** definitions or, more recently, **JSon schemas**), serve a similar purpose by exposing a definition of the model of the "world" or domain under consideration. In some ways, the static structures defined in all these formats could be classed as "Ontology" as long as we concern ourselves with "what" is being represented rather than "how". Ontology notations evolved for this very purpose - to define conceptual models of the world and relationships among them, capable of automated reasoning, in a globally unique identification scheme (the importance of which is also described by [Fowler, 1997, p. 88-89]).

The idea of legal information (with contracts, contract law and tenancy law being a representative subset of legal information) captured in ontologies has been proposed before [Breuker et al., 1997] [Boer, 2009]. Representation of legal knowledge in technology - and visions of algorithmic law enforcement do not bridge one crucial gap - the subjective roles of "common sense" and "intent". An algorithm could be implemented with a certain intent but might still work differently for various reasons - like a bug in code or untested, corner-case scenarios [DuPont, 2017] [Vessenes, 2016]. We attempt to identify current systems that address the same or similar requirements to identify progress, challenges, and gaps. In terms of encoding legal knowledge in the form of ontologies, two important initiatives stand out:

- Metalex and LKIF - frameworks for legal knowledge interchange [Boer et al., 2008]
- The Greek legislation - project known as "Nomothesia", that uses Metalex for core concepts.

- UK Legislation published via RDF

The most prominent initiative, Nomothesia, does not actually model real world entities to a low level of granularity as its concerns are metadata surrounding high-level legal documents (like "*BibliographicThing*", documents, acts or decrees, and classifying/describing other writing that forms a source of legislation.). The core concepts are inherited from Metalex and a subset of classes used in the Nomothesia[4] ontology is shown in figure 2.2. Where initiatives like above

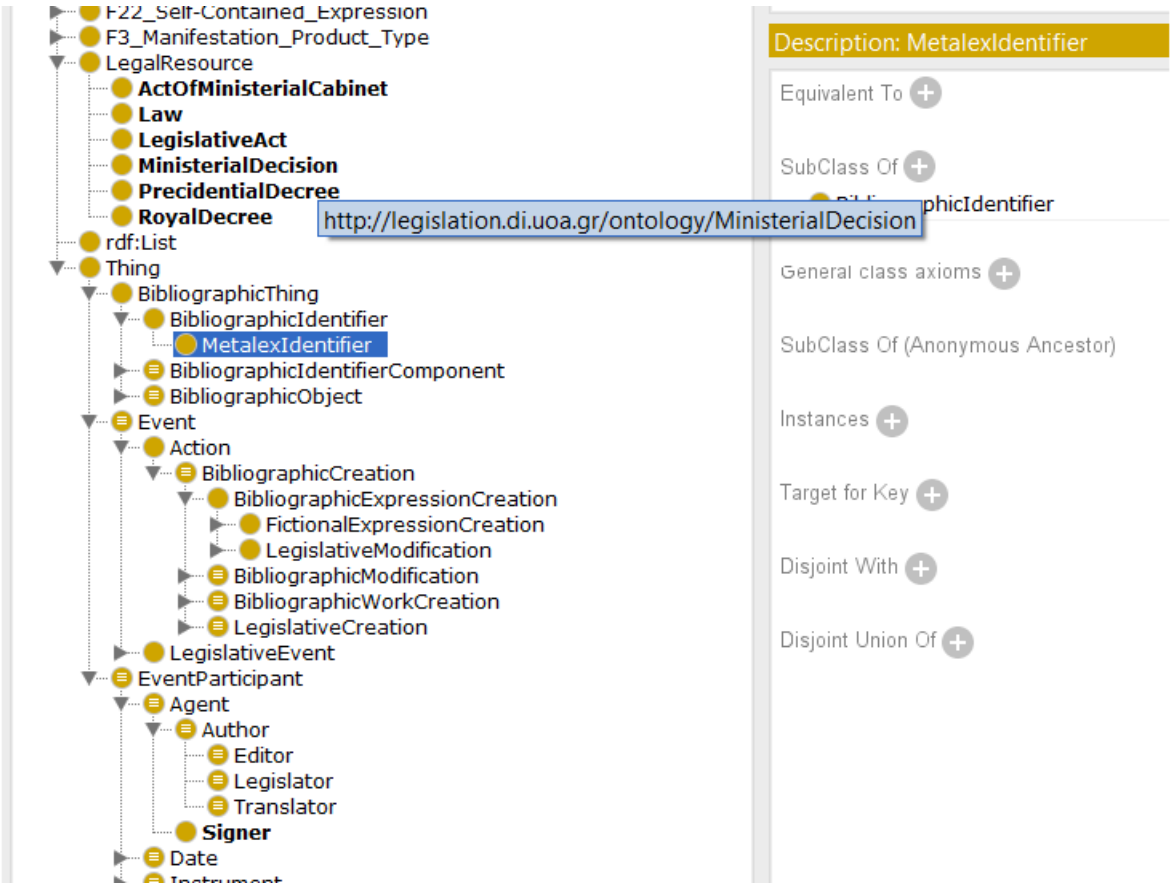


Figure 2.2: Subset of the Nomothesia domain classes seen in Protégé

fall short in meeting the aims of a "*semantically rich* smart contract" system visualised in this thesis are the granularity of models. Current commercially available software/services that address this area appear to fall in the category of "contract lifecycle management" [Sysintellects,]. These are used in conjunction with digital signatures or PKI based solutions to enforce integrity, where the actual document creation, signing and hosting is performed on a third party platform (presumably trusted), separate from parties to the contract (this observation is based on use of a commercially available, web based, tenancy management platform and multiple other digital contract management systems). Where paperless contract formation technologies fall short is that they are in essence strings of digital text. The digital signatures are also applied on basically pages of text or consisting of large or small sentences and paragraphs open to human interpretation.

An ontology backed smart contract system as visualised in this thesis would be a set of classes and machine readable (and interpretable) axioms, inheriting from authoritative sources of base

[4]http://legislation.di.uoa.gr/ontology/

legislation (possibly subsets of systems similar to Nomothesia), linked to other knowledge bases as appropriate. Current or past works closer to this approach would be Legal Knowledge Interchange Format (LKIF) [Boer et al., 2008], MTCO [Kabilan and Johannesson, 2003] mentioned earlier, and "Public Contracts Ontology" [OpenData.cz,]. LKIF is concerned with interchange of legal information whereas the last two present ontologies for contract management to drive workflows.

Role that Reasoners have to play in artificial intelligence is one of the motivators of our design and the reason for the use of semantic web technologies. The capability of 'infering' new information from a smaller subset, drawing logical conclusions, and validating consistency of provided (asserted) information has a natural application in a legal setting. A simple example of an inference drawn (instance class membership) is:

`(?B subClassOf ?C)` ^ `(?T type ?B)` → `(?T type ?C)`

One of the design decisions would be to whether use a full Description Logic (DL) reasoner (like Pellet or Jena) or a Rules engine. The tradeoffs would be guided by factors like size of the knowledge base, expected performance, and technology fit with rest of the solution. Many complex relationships cannot be expressed in plain OWL. For that, OWL can be used in combination with rules (addressed in chapter 4). Meditskos [Meditskos,] concludes that though rule based reasoners cannot reason on fully expressive ontology semantics, they scale well (from tooling review, it was seen that scalability of rules engines that use rete is not a function of the number of facts). And DL reasoners (like pellet? jena?) are sound and complete, but do not scale well. From [Meditskos,]:

> *"Rule based reasoners are the standard in large data sets e.g. in Linked Open Data cloud where there is a need for reasoning on large (interlinked) datasets".*

From an application desgin point of view, it would be important to pick a "tractable" subset of description logic and for that reason, a suitable profile of OWL2 is a natural choice due to its maturity and widespread use [Dentler et al., 2011]. Medical ontologies like SNOMED include *"over 400,000 concepts"* and, according to Horrocks [Horrocks, 2010], an extension of that with thousands of more classes takes 10 minutes to classify and check. This research supports our hypothesis about using the knowledge bases with either rules engines or preferable a full DL reasoner would offer good value to our proposed system.

2.6 Chapter Conclusion

A selection of literature, tools, and methodologies have been explored in this chapter. An effort was made to identify tools and methodologies, to identify limitations or alternatives better suited for the chosen problem are, and critically evaluate associated literature.
After having introduced a new system proposal that could benefit from a set of new technologies, this chapter helped understand the proposed technologies in detail via a study of background literature and previous work. Some new terminology often gains disproportionate prominence, but we also find that core concepts can be surprisingly enduring (*"Principles Endure"*, to quote [Simpson et al., 2015] in the context of software technologies) but it is important to recognise genuine inventive steps for the value they could add to designs of modern systems. A critical scrutiny of past failures is important to make realistic decisions and tradeoffs - overall, the literature review supported the initial choices and, in fact, helped to highlight

the novelty of certain aspects of the system proposed in this dissertation. The individual subjects studied each stands on its own, but, the aim of this project is primarily to explore how the synthesis of these can help achieve functional goals. It is with this foundation that I can confidently and meaningfully proceed to more detailed designs of the proposed system.

3 Requirements

The goals of requirements engineering in this project is to present what can be made possible with the technical areas being explored. Secondly, it is to identify a suitable subset that could be implemented in the accompanying prototype. Sources of requirements for this project were personal experience, reading, and documentaries, although no actual users or legal professionals were consulted for this dissertation project. In general, sources of requirements can be functionality required from stakeholders, rules and regulations of the business domain, and implicit non functional requirements for the system. Non functional requirements are derived from many diverse sources such as operational requirements, reference architectures for technologies chosen, industry good practices, and standards such as the ISO/IEC 25010:2011 series which defines 'quality' as:

> *"...the degree to which the system satisfies the stated and implied needs of its various stakeholders, and thus provides value."*

This project is no different as such, but the scope of requirements engineering activity is constrained by the primary goals of this project and limitations of an MSc dissertation.

3.1 Requirements Elicitation and Sources of Requirements

Drawing on principles of good requirements engineering, stakeholders of the system were identified early on and their perspectives represented using personas (an approach detailed in [Faily, 2011], [Flechais et al., 2007] in the context of combining security and usability in a comprehensive requirements engineering process). A review of approaches to requirements engineering particularly applicable to this system could be found in [Berkovich et al., 2009]. Full requirements engineering for the proposed system would involve creating representative personas for each stakeholder to participate in the requirements definition phase of the project such as workshops. Apart from use cases, it would be important to identify misuse cases [Faily, 2011] as these are an important source of both functional and non-functional requirements. This research, however, has to progress with a brief identification of some representative requirements - the goal being to address different components of the solution and prove their working together to enable such a system. My approach consisted of identifying some main use cases and actors in a specific type of contractual relationship - the assured shorthold tenancy, which is a common form of landlord-tenant contract - and modelling it in UML to the extent that would facilitate the technical discussion that follows.

For future iterations, this aspect of the requirements engineering stage will be revisited. Apart from methods, Business Process Management (BPM) as a discipline has a role to play in the requirements elicitation process. BPM tools available today offer a standard business friendly notation to model a usecase in a graphical, easy to understand notation called BPMN which is designed for non-technical users. This tool, introduced early on, can help model the sequence of events and participation of different stakeholders (or systems) in realising a usecase.

3.2 Main Usecases in a Tenancy Contract

To briefly describe the domain *in the context of this dissertation*[1]: Contracts are definitions of agreements between two parties. Parties agree on their *rights*, *obligations*, and *restrictions* under the contract, and other attributes such as start and end dates. Obligations might also have certain dates attached to them. In a procurement contract, the obligation might consist of delivering a certain good or service by a certain date and another obligation might consist of sending a payment on fulfilment of certain conditions (such as receipt of good). The choice of using a more constrained subset of contract law, *Assured Shorthold Tenancy*, is explained in section 1.3. A tenancy agreement, between parties called landlord and tenant, is a set of similar rights, obligations, and restrictions too, based primarily around the common understanding of renting a property, but also derived from various other sources (fig 1.1). A tenant's rights include having full **access to the property**, and obligations include making regular **rent payments** as per a **rent schedule** (which in turn is derived from attributes like *agreed rent* and agreed *payment frequency*). The landlord's rights include **periodic inspections** and to **serve notice** of an agreed minimum period after case they wish to terminate the agreement. The minimum possible notice period might be derived from legislation, which the landlord may not reduce but may exceed in their particular contract instance. Landlord's duties include regular *maintenance* of the property, particularly, safety inspections (gas, electric) at specific intervals. This is an oversimplified view as the idea is to highlight how a generic case of "contracts" can apply to different domains.

At a more summary level, we could identify the main usecases for the system as shown in figure 3.1 Many of the components of the use case diagram (fig 3.1) like the actors, entities, and use cases, are non-specific so can be instantiated for different domains. This is the theme kept in mind during implementation, concrete definition of data models and so forth. The "Tenant" and "Landlord", are modeled in the UML usecase diagram as actors and can be considered sub-classes of a more generic actor called Party (not pictured). "The Platform" and "Legal professional" are roles that facilitate the system in a neutral manner. Participation of legal professionals and technical semantic web experts is facilitated by human workflow activities in the BPM processes that help realise these usecases. This type of participation allows a small number of expert professionals to deal with multiple contracts, effectively *scaling up* their expertise to a large number of users.

The platform (using legal professional expertise), provides definitions of contracts (a set of legal axioms based on the law and generally accepted principles of tenancy law [gov.uk [gov.uk,]), available for the parties to use in their specific contract instance. Using this ready repository of axioms, the parties can create their own instance of a contract. Most common tenancies ought to find the standard contracts cover most of their requirements. There might be exceptional scenarios where parties negotiate and customise their instance of the contract, which again might be reviewed by the platform and the legal professional for correctness, consistency, and applicability. Once the contract instance is agreed between the parties, it is *registered* with the platform and execution starts. Parties now start performing their contractual obligations which, for a Tenant would be to *Pay Rent* according to a schedule (schedule being inferred or pre-calculated from contract properties such as start date, frequency, amount). The platform also keeps track of changing legislation and how it might impact existing and new contract instances. During the lifecycle of the contract, the platform aggregates *Compliance Events* and

[1] Paying special note again to the fact that this work does not use any legal support or domain knowledge. An introductory reading of legal texts like [McKendrick, 2017] indicates many of the terms used in this description have special meanings in the legal field.

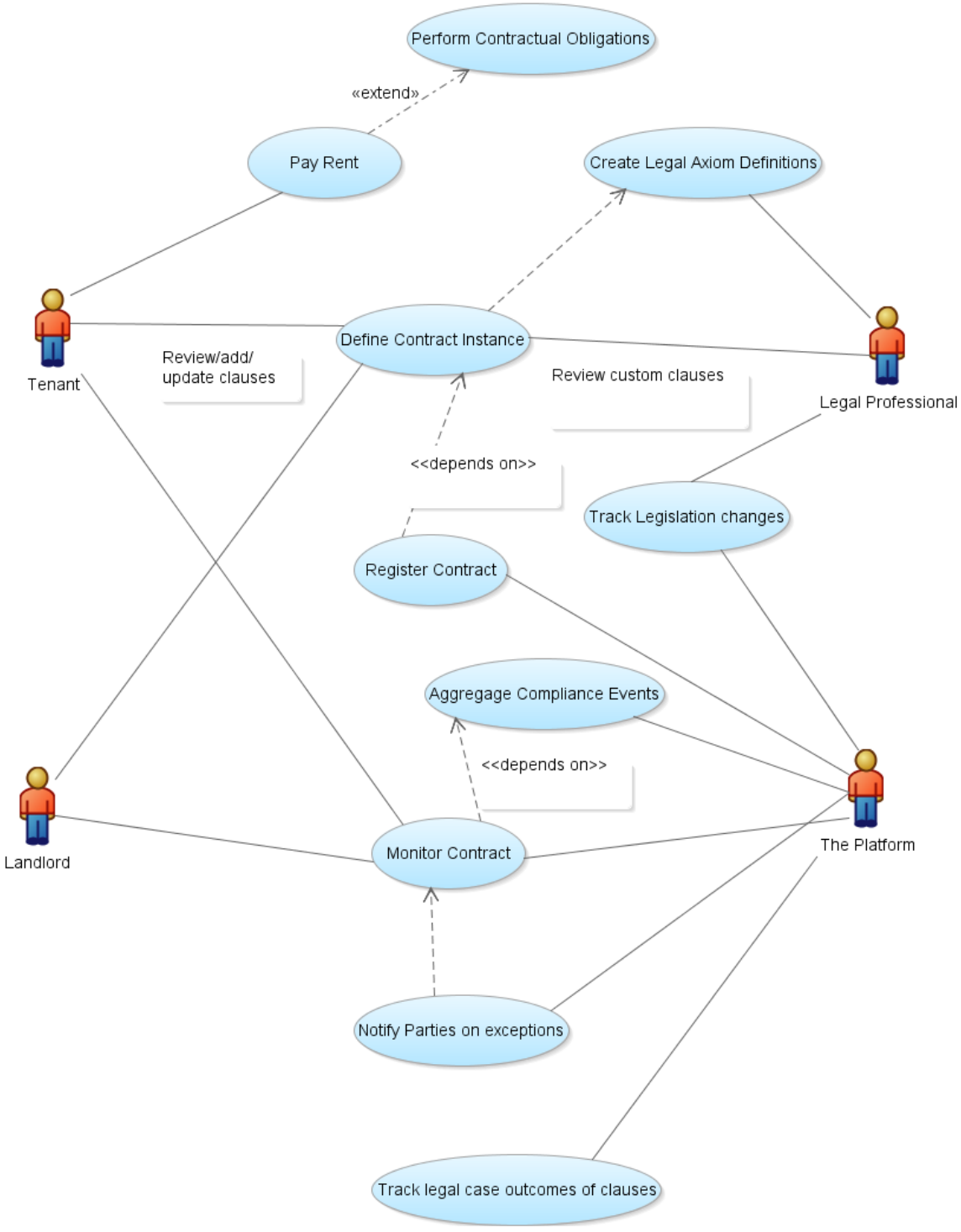


Figure 3.1: Usecases in tenancy management

associated evidence from different sources and analyses these to determine a *Compliance State* for the contract at any given moment. An example of a Compliance Event would be the payment of rent by a due date identified in the schedule. Compliance evidence might consist of verifying the target account or verifying a payment with the agreed payment processor (such as PayPal). The platform also serves as an *expert help* that proactively analyses different events,

legislation, and clauses in the contract to determine next steps a party ought to perform. A simple example of such a role would be to trigger reminder notifications close to a rent due date or after a due date in the event of missing payments. In a more complex scenario, the platform could offer a guided process to perform complex (yet repetitive) legal steps in the event of a complex workflow such as eviction. The platform could facilitate the landlord with advice around how to serve notices with acceptable evidence, ensuring that an audit trail of all steps is stored in a trustworthy manner and made accessible during a dispute resolution process. The platform could similarly support the tenant in keeping track of support requests, complaints, communications, and other events during the lifecycle of the contract.
A semi-automated service running behind the scenes can keep adding to the repository of $ClauseDefinition$ so that those forming new contracts have a readily available repository that they can refer to. These can eventually get linked to actual cases and therefore, clause definitions can have an associated *enforceability* level. Clauses that have a legal precedent can start as being enforceable. If parties draft new and untested clauses, they will start off as being low on the enforceability factor (unless legal experts who help encode the law set an explicitly high enforceability factor based on their knowledge). This *enforceability factor* could then be automatically fine-tuned over time (possibly using machine learning techniques) if contracts ever get into dispute.

3.3 Functional Requirements

We can at this stage, start to identify the the functional building blocks that would each evolve into a full-fledged system addressing a specific need, integrated in a single business process. The preceding narratives allow us to list some concrete requirements that we would implement in the prototype.

Table 3.1: Primary User Interface Prototype Requirements

Requirement ID	Description
REQ-UI-001	Landlords and Agents must be able to list properties (create Listings).
REQ-UI-002	Prospective Tenants must be able to browse listings.
REQ-UI-003	Prospective Tenants must be able to book a listing.
REQ-UI-004	Tenants and Landlords are offered a stock contract that is descriptive and easy to navigate.
REQ-UI-005	Clauses of the stock contract must be annotated with standard, descriptive wording like: *"the rent shall be ___, paid every month/fortnight/week"*.
REQ-UI-006	Stock clauses should inherit particulars like advertised rent from the Listing.
REQ-UI-007	Tenants and Landlords must be able to add clauses via drop-down menus and free text.
REQ-UI-008	Users of the system must receive notifications of important events in the tenancy lifecycle.
REQ-UI-009	Tenants, Landlords, legal professional and all interested parties must be able to view a high integrity audit trail of steps taken in the lifetime of the contract.

Table 3.2: Secondary User Interface Prototype Requirements

Requirement ID	Description
REQ-UI-010	As secondary users, systems administrators (and semantic web professionals) would get a more advanced interface to perform advanced modifications of the contract knowledge base (such as using Protégé and WebProtégé) to make updates that lead to a consistent contract.
REQ-UI-011	the immutable record of transactions performed as part of the contract lifecycle must be available for search and verification by relevant parties.
REQ-UI-012	The "process" or processes associated with the contract management must be available for inspection by authorised users and systems administrators. These should highlight the specific activities (human or automated) that are performed during contract execution - e.g. during contract negotiation, the set of activities that might be executed might include automatic reasoning, referral to a semantic web professional for technical edits, referral to a legal professional, and registration of the contract instance on the distributed ledger smart contract execution environment.

Table 3.3: Contract Repository Requirementss

Requirement ID	Description
REQ-CR-001	The repository must make the legal rules/axioms available for other parts of the system (such as for presentation to the user interface)
REQ-CR-002	Both contracts and clauses are *effective dated* - this is to ensure that new contracts are created with the appropriate legislation *effective* at the time of contract formation and also if any changes apply to existing contracts.
REQ-CR-003	The knowledge base must be created in a standard, linked data format that can *inherit* relevant legislation (possibly published by competent *"Formal sources of law"* [Boer, 2009]), and can be used to create custom views for different types of consumers (non-technical end-users of the system and trained secondary users like semantic web technologists).
REQ-CR-004	Custom clauses (essentially axioms or triples) created by users in their specific contract instances, once reviewed, must be updated into the knowledge base where appropriate. These must become available on new contract creation or when users search for *similar* clauses.

Table 3.4: Business Process Requirements

Requirement ID	Description
REQ-BP-001	Must offer a visual representation of the processes attached to contract usecases (contract registration, delayed payment, inspection, eviction must all be modelled as business processes) during runtime.
REQ-BP-002	The *runtime* view of a process instance must be close to the *design time* view and must show the actual activities executed or decisions paths followed.
REQ-BP-002	Key events in the tenancy lifecycle must leave an immutable audit trail in the contract register (the ledger) that can be inspected by all interested parties.

Table 3.5: Contract Register Requirements

Requirement ID	Description
REQ-R-001	The contract, once negotiated, finalised and reviewed, must be registered in the ledger.
REQ-R-002	Only authorised parties holding specific private keys may update the ledger.
REQ-R-003	The series of transactions associated with a contract instance must be available for inspection.

Table 3.6: Mobile Application Requirements

Requirement ID	Description
REQ-M-001	Allow tenants and landlords to receive notifications and reminders (rent due, inspection due).
REQ-M-002	The mobile app must allow secure communication between the tenant and landlord, including reliable means of serving notices and a reliable means of detecting receipt of notices.

No MoSCoW[2] style analysis has been performed, and the wording does not follow strict definitions of *must*, *should* or *would*. Although subject matter around specific technologies was discussed in the previous chapter, in this chapter we purposely keep our wording technology agnostic to keep focus on *what* the system must do rather than *how*. Realistically, the scope of this system requires a two level assessment of functional requirements. At the first stage, it allows us to determine the high level architecture (follows later in section 4.1) and justify how the components chosen help realise the identified use cases. At the second level, we would need to define functional requirements for each block - this is not a trivial task. Each solution module might require different types of skills and domain knowledge to identify their

[2] `https://en.wikipedia.org/wiki/MoSCoW_method`

functional requirements accurately - these systems involve different types of stakeholders with different motivations and their output is likely to facilitate legal steps and outcomes. Some of these components- like the user interface, mobile application - might be fairly commoditised at the technical level but require significant domain knowledge with some innovation to cater to some legal usecases. Notably, to ensure that custom contract clauses are formed correctly might need support at the user interface level. Various UI fields would need validations, mandatory input, and other safeguards to ensure that fairly reliable data passes on to the backend reasoning services (even though the reasoner components, described later, should be able to detect any inconsistencies regardless).

The second level functional requirements for the *clause repository* would require a study of all laws, rules, and regulations (such as [gov.uk,]) that apply to tenancies. For the sake of illustration, I quote some of some basic business rules identified from online resources which define a contract to be an **Assured Shorthold Tenancy** if they meet these conditions (quoted from [gov.uk,]):

- If Location is London, Yearly rent cannot be lower than £1000
- If Location is outside London, Yearly rent cannot be lower than £250
- Yearly rent cannot be higher than £100,000
- Contract is made after 15th January 1989
- Contract duration is not more than 7 years
- The landlord's residential address must be different from the property and must be available to the tenant throughout the tenancy period.

3.4 Non-Functional Requirements

Because of the nature and scope of this project, we need to first distinguish between the requirements of a complete solution and requirements that relate to the prototype associated with this project. There would be some overlap between these, which we would highlight where possible for clarity. ISO/IEC 25010:2011 served as the main reference material and I find principles of "secure and usable" software design would be very beneficial for a future design iteration.

Although a detailed security assessment of the system has not been made in this dissertation project, we realise that the most important stakeholders of this system are to be general members of the public - private landlords and tenants - for whom access to the right information at the right time (i.e. correct stage of a business process) is of importance. This means, maintaining the quality and integrity of the knowledge bases and their accurate presentation. Properties like usability and user experience can be very subjective as well but established industry standards can be applied to achieve properties like accessibility. In tems of scalability, the solution needs to cater to potentially all tenancy agreements in one country to start with (the UK). The detailed design needs to justify how this could be achieved, however, the prototype will not be evaluated on this metric. A more detailed justification of main and relevant non-functional properties will be offered in the evaluation.

3.5 Chapter Conclusion

This chapter gives a tangible set of requirements to focus and direct the subsequent work on. Based on an analysis of the problem domain, important use cases were identified and presented in UML. I then pick one of the usecases, *Define Contract Instance*, and an associated usecase

Register Contract, and present its proposed realisation in BPMN in the next chapter. The next chapter would drill down into more detailed technical design. I appreciate that the different areas touched are broad and elaborate disciplines in their own right. Secure and usable requirements engineering principles, and BPM are two non-trivial disciplines and have specific value to offer if applied with their respective methodologies. I have effectively shrunk their full application to a constrained proof of concept with the goal of showcasing the value of different software engineering tools and techniques.

4 System Design and Implementation

The goal of this chapter is to propose a viable design of the system, to identify distinct components at a higher level of abstraction first, and to gradually drill down to detailed design. The design ought to identify clear interfaces between components, and justify how this helps realise the use cases identified in the requirements section. Based on the discussions in previous chapters, especially chapter 3, I present the functional building blocks of the solution in figure 4.1.

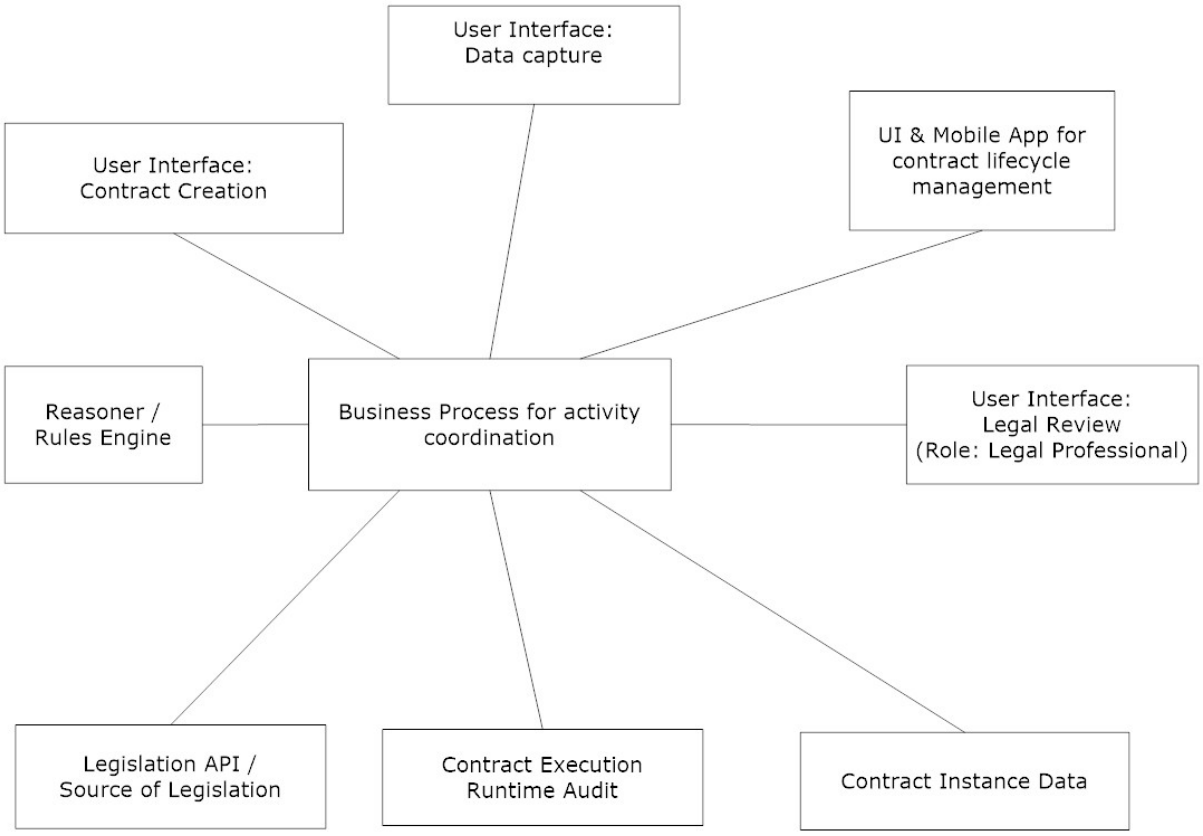


Figure 4.1: Functional building blocks of the solution

Each building block is also a full fledged application in its own right that would require its own requirements engineering. From an overall design point of view, it seems prudent to start with the most flexible, technology agnostic, and preferably business domain agnostic set of components that could be used to realise the usecases.

4.1 High-level Architecture

The functional blocks presented in figure 4.1 identify the components needed to realise the functional use-cases without tying in to specific technologies early on. This should allow the solution to be used as a reference framework at different levels of abstraction - even if other researchers or developers disagree with these choices, this flexible architecture could still be retained as a common reference point whilst being instantiated in different ways if future research supports that. When mapping to a formal architecture development methodology like TOGAF, this could serve as the reference point to develop the Business, Application, and Technology architectures [TheOpenGroup, 2018]. Even at lower levels, the design details could evolve into a common technical framework for different domains that deal with contractual relationships. The functional building blocks are described below:

Contract Lifecycle Business Process: This is envisioned as a high-level process or multilevel set of processes that chain together the whole life cycle of the contract. It coordinates the interactions between different solution components and might do so based on analysis of the current state, such as by the output of a reasoner or a new state identified in the distributed ledger. This coordination is also called *orchestration*, as a business domain subject matter expert would define how, why, and in what sequence possible activities are to be performed. At each significant milestone, this process updates the state of the contract to a distributed ledger. For this component, any of the suitable implementations of BPMN executable engines would fit well (tradeoffs and specific choices are evaluated in subsequent sections)

User Interface for Contract Creation: This particular user interface would be used by the *primary participants* of the system which are the parties that negotiate and define their contract. It presents forms for performing setup steps like creating listings and user profiles. For scenarios like tenancy creation, this is likely to be based on pre-defined templates and axioms inherited from regulation, all displayed to the users in a human readable format but internally represented as Ontologies. Beyond the minimum required clauses, parties could add more for their particular requirements, which would be facilitated by user friendly drop-down lists and restricted text matching to create valid triples. Parties can add new or custom clauses either by searching a repository of clauses or creating new ones. The new clauses thus formed, if completely unknown, could be referred to legal professional review as a human workflow step in the supporting business process.

The idea is to facilitate the contract formation process via pick-lists of entities and relationships with some support for matching natural language sentences to valid axioms (triples). Over time, such a system is likely to add to its repository of common clauses for the domain. Highly specialised clauses that don't immediately match valid triples would go for a legal review human workflow (run by the platform). Extracting RDF triples from natural language is already an area being explored [Gangemi et al., 2017], but this aspect is not the focus of this particular research. A future enhancement of this system could using NLP techniques as a fall-back for exceptional scenarios but continue using OWL triples as a baseline to form contracts.

User Interfaces for Data Capture: This is an associated user interface would involve capturing information about tenants and landlords in a form. Some of the facts captured here could have a bearing on the process further downstream (such as what clauses are available during the contract creation process).

UI and Mobile App for contract lifecycle management: This interface, supported by a mobile application with appropriate security features allows the parties to interact with the contract lifecycle as necessary:

- Secure messaging with traceability
- Search features to track the history and provenance of the contract instance. What happened till now? Is everything up to date? What are the next steps?
- Tools to create an inventory at the contract formation stage (an inventory forms part of the main contract)
- Ways to report problems with status tracking of each such request.
- Notifications and tracking of key stages of the contract lifecycle (rents, deposit protection notification, safety checks due/done etc.)
- Access to *next steps* for non-legal professionals such as private landlords. For instance, are conditions right to serve a notice to end tenancy?

Behind the scenes, all the activity could be updated to a distributed ledger in a series of irreversible records searchable by all relevant parties to monitor the lifecycle of the contract.

User Interfaces for Legal Review: The secondary users of the system are Legal experts and Semantic Web professionals for technical support. A specialised **user interface for legal reviews** would allows legal professional(s) to participate in the business process at various stages. Legal professionals might need to review highly unusual clauses created by the parties or participate in dispute resolution. In the general operations, the system (represented by *the platform* actor in the use case diagram 3.1) would be privy to a regular feed of *The Law* and changes in legislation. In practice, new versions of legislation are published with some notice after consultation and often include incremental changes. At the full maturity of the system, when legislation is fully available in the form of linked data, the platform should be able to incorporate new versions of legislation (published as OWL), and apply reasoning and test coverage to determine any deviations or unexpected inferences. Manual legal review would then be necessary to ensure the system continues to work appropriately. During early stages of maturity, when legislation is only available as text/prose, it would be the responsibility of legal professionals with help from trained technologists to populate the *clause repository* (shown in the high-level architecture in figure 4.2) with the knowledge bases derived from legislation. The clause repository serves to then allow parties to put together valid contracts via the **contract creation** UI, but the supporting business process would allow semantic web experts to intervene using tools like Protégé *when necessary* in a small subset of cases.

Contract Instance Data: This module represents the repository of definitions of individual contract instances created from the user interface. One of the main features of this solution is the creation of contract instances in the form of ontologies created from a user interface and linked to other ontologies.

Contract Execution Runtime Audit: The lifecycle of a contract will include a number of milestones, events, and communications. The events can be related to obligation fulfilment such as rent payments or serving of legal notices. This module represents the mechanism to store such data in an irrevocable, highly available, decentralised store. A background discussion around this was covered in chapter 2.

Legislation API/Source of Legislation: This module represents an external source of *legislation* from an authoritative source such as a government web service. The long term vision is that this will link to the contract definitions and contract instances created in the application so that automated reasoning can work on the full linked knowledge base. As no such service is currently available, this will only be represented in the prototype by a local ontology.

Reasoner/Rules Engine: Contract instances that are created from the user interface will be minimal local ontologies with details specific to a particular contract. This is to keep the user interaction fast and responsive. After this, the background process will combine these instances with ontologies representing legislation and these will be validated by an ontology reasoner. This is to ensure that contract instances are semantically valid and clauses do not contradict axioms inherited from the legislation.

With this, we can assign some more technical concepts to arrive at this version of the high-level architecture as shown in figure 4.2.

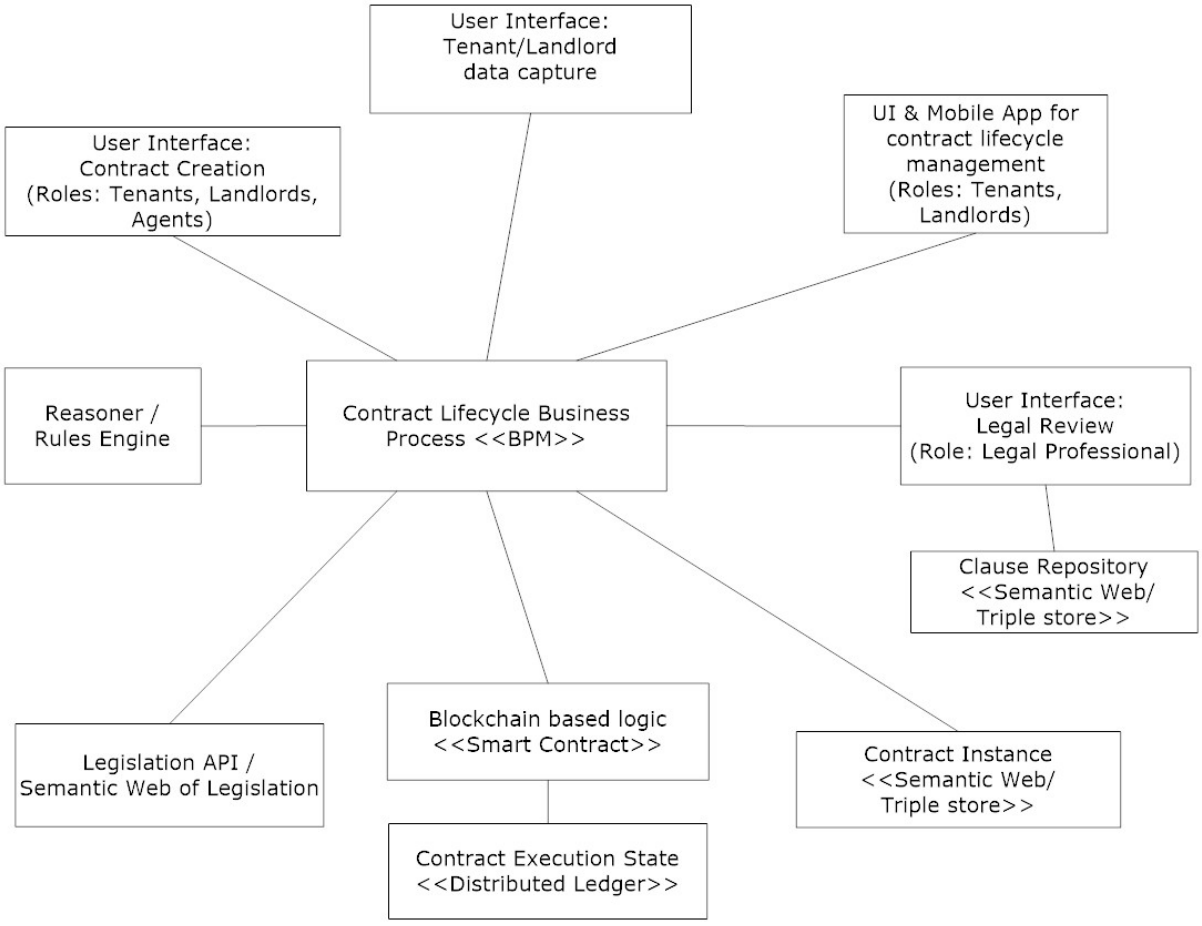


Figure 4.2: High-level technical architecture

Using the functional blocks, and the study conducted in chapter 2 as a base, high-level technology choices could be identified for the different blocks. Justification of these choices, backed by proof of concepts, critical assessment, and discussion of alternatives and trade-offs is presented throughout this dissertation. Subsequent sections in this chapter present more detailed designs of some of the key components.

4.2 The Knowledge Base

Data model in this section specifically means the ontologies that are defined to implement the **clause repository** and **contract instance** components of the solution. There are many approaches used to identify the domain model of an application. Fundamentally, it is not too different from identifying *key abstractions* deriving a suitable *conceptual model* [Fowler, 1997] from the business domain early on in an Object Oriented (OO) design process. Where objectives of this design would diverge significantly from the OO paradigm are in lack of relevance of concepts like encapsulation, data hiding, and behaviour inheritance. In an OO initiative, there would be no expectation of deducing any new information not explicitly stated. Atkinson [Atkinson, 2006] also lists significant distinctions in the *"**knowledge** capture"* and *"open world"* nature of ontologies versus more *"prescriptive"* model driven nature of traditional data models. With these distinctions clear, following an approach similar to Kabilan [Kabilan and Johannesson, 2003], I could still have used a dedicated UML tool to describe entities, their attributes, relationships, and their cardinalities - the full set of information needed in a data model equivalent of the Semantic Web TBox. Leading tools and even IDEs offer support to design entity models in a graphical or standard UML class diagram notation and automatically generate program code out of these (such as Java entity beans).

OWL, serialized in turtle, was chosen as the preferred notation. Turtle is a more concise format and easier to understand and edit by hand using text editors. The concise turtle notation was also considered suitable if contract instances or some form of ontology processing could be accomplished *onchain* now or in future. The choice of OWL (specifically, OWL3 DL) as a mature and established format for creating description logic knowledge bases is guided by rich tool support, and the fact that leading DL reasoners can process it. For creating ontologies, using Protégé was a natural choice due to its maturity and availability. The ability to define a rich schema counts in its favour with integrated tool support for reasoning and defining rules (SWRL) was particularly helpful. This is discussed further in section 4.5 on Reasoner design. Worth also highlighting is the fact that whilst a plain UML model with Java classes derived from it might work natively with many leading rules engines, it is only suitable as a tool for asserted information (*"prescriptively defined information"* as per [Atkinson, 2006]).

An *intelligent* ontology driven system would keep extending its knowledge base as it can potentially add inferred knowledge to existing asserted knowledge. The OWL vocabulary with its high expressivity, triples notation, and OWL reasoners naturally lend themselves to such format. However, such extensibility via inference is made possible due to a concept called *"Open World Assumption"*(OWA) that is not shared with a model driven approach [Atkinson, 2006] described earlier. Although this capability provides another justification for using an ontology driven approach for this system, this adds a challenge because, data constraints (or OWL data property assertions) not explicitly specified are need to be handled at the application level as data validations (perhaps even UI form validations). To give an example, even though the system ontology defined a "Contract" as having a $startDate$ property with cardinality of exactly 1, asserting (i.e. explicitly creating) an individual contract without a $startDate$ doesn't get flagged by OWL reasoners as an inconsistency. This is by design and in accordance with the OWA. For clarity, creating an individual contract in practical terms means creating an individual of class $Contract$ or one of its subclasses, associated object and data properties, and their domain classes.

One of the intentions of this design is to create individuals for key classes at a higher level of abstraction, assert as many properties as possible, and then use the reasoner to verify if the individual gets inferred to the more specialised sub-type. It should be added that a simple triples notation (turtle) was used for serialisation because it fit well with the design goal of allowing parties to negotiate and form additional contract clauses based on a pre-defined knowledge base but it eventually had to be serialised into RDF/XML due to current limitations of the UI framework.

A set of three ontologies to showcase the hierarchy, modularity, and separation of concerns were created: A generic **Contract** ontology (fig 4.3) at the top level, an **AST** (Assured Shorthold Tenancy) ontology extending it with additional knowledge about the common type of tenancy and a **Contract Instance** - which is a specific instance of a contract inheriting from the other two. Parties creating a tenancy contract essentially create an *individual* of type $AssuredShortholdTenancy$, representing a Contract Instance.

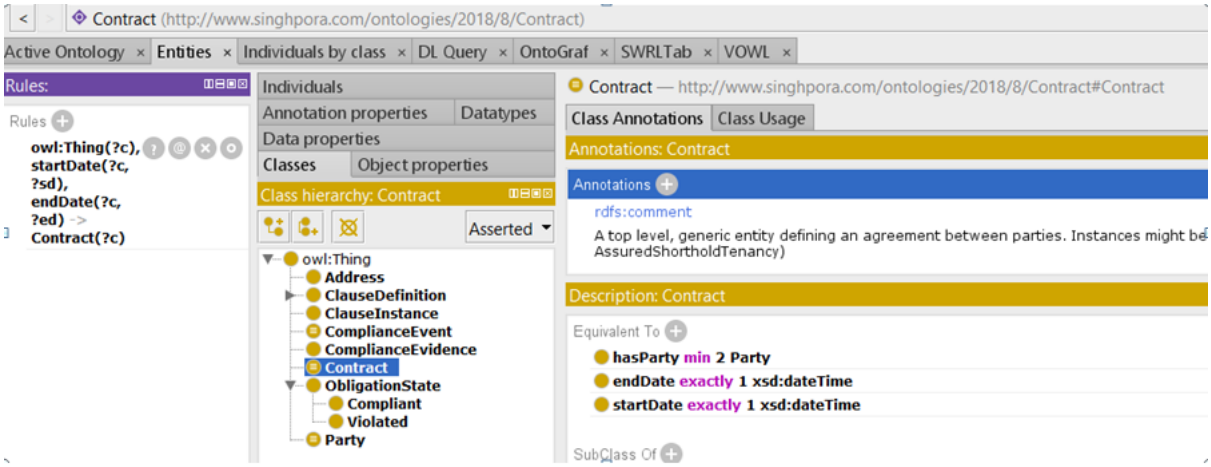


Figure 4.3: The top level contract ontology pictured in Protégé

In accordance with principles recommended by [Noy et al., 2001], my first approach was to identify existing ontology models. The work by [Kabilan and Johannesson, 2003] seemed to match my requirements closely while [OpenData.cz,] is a more current initiative but geared towards public contracts involving tenders. Finally, in line with the goals of this project, I found it prudent to create a minimalist ontology to progress with other goals of the project. The minimalist data model used in this project could still serve a viable product but for future extensibility and in the spirit of linked data, it would be important to choose and reference a formal reference ontology for common entities such as Contract, Party, and Address. With Protégé, getting good visualisation to automatically from the Ontology was a challenge (none of the main visualisation tools like OntoGraf and VOVL showed cardinality properly), therefore, for the sake of presentation, a UML class diagram describing important entities and relationships from the Contract ontology is shown in figure 4.4. In the interest of brevity, and to highlight the inheritance hierarchy in both classes and relationships, some parts of the TBox were omitted in the UML view (notably, classes like Listing, Property, and Rooms) shown in figure 4.4.

The Assured Shorthold Tenancy (AST) Ontology: The AST ontology inherits from the generic Contract ontology and creates more specialised entities for the domain. The inheritance relationships exist both for entities and for some properties. Even the AST ontology is still primarily meant to define classes and relationships (the TBox aspects) rather than individuals.

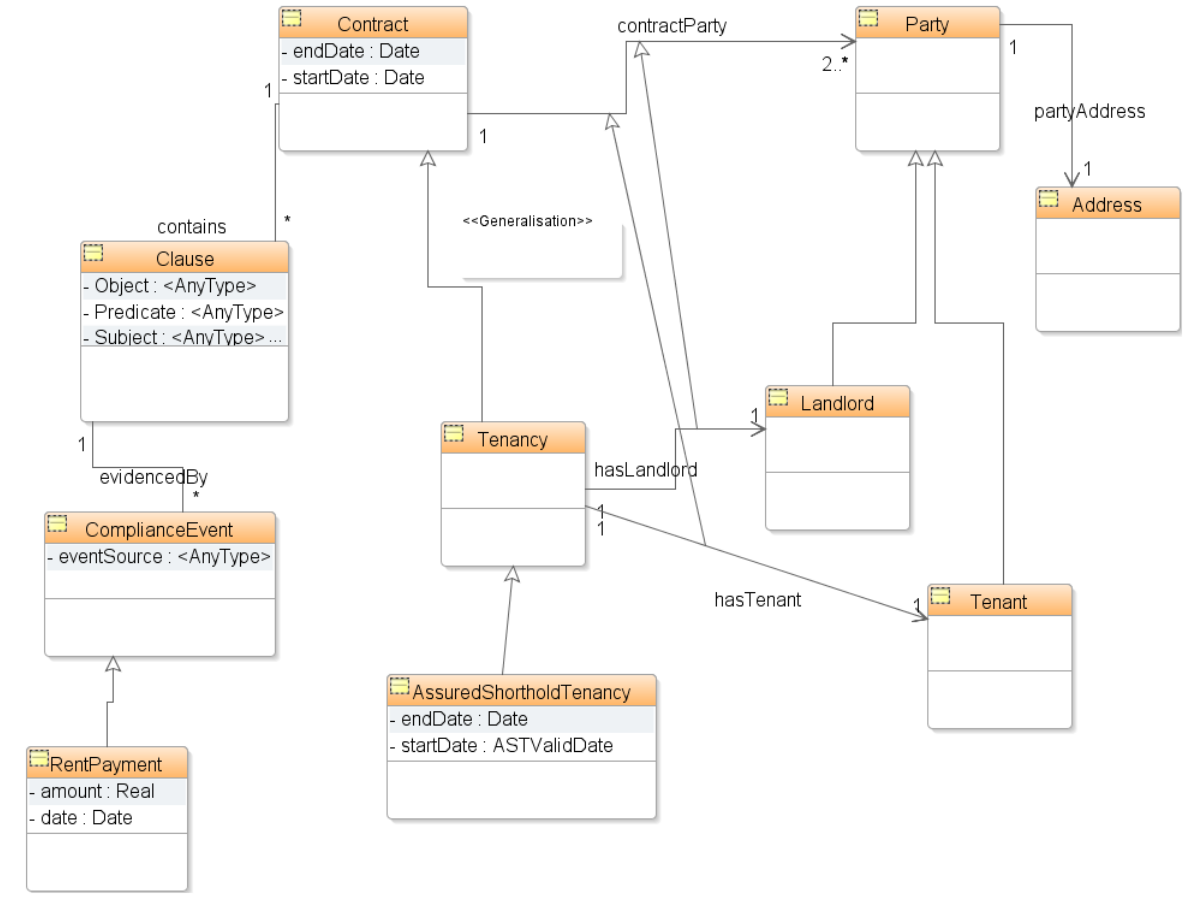


Figure 4.4: Abbreviated UML representation of the second level AST ontology

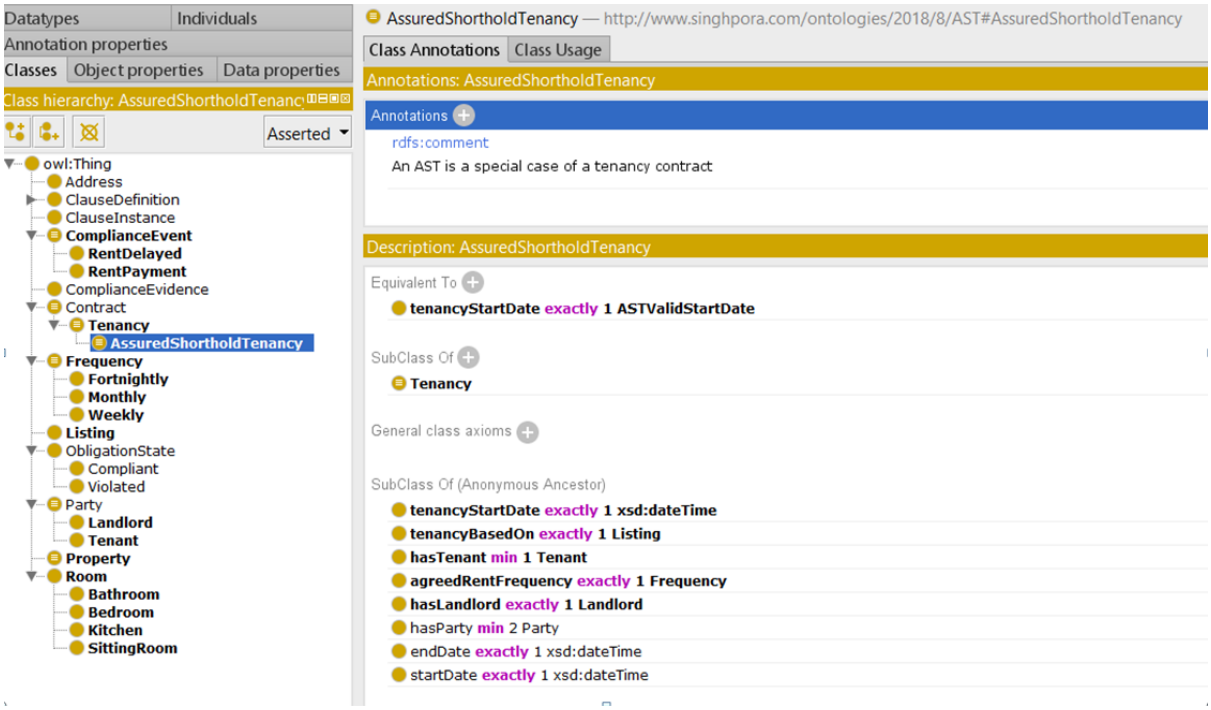


Figure 4.5: The second level AST ontology pictured in Protégé

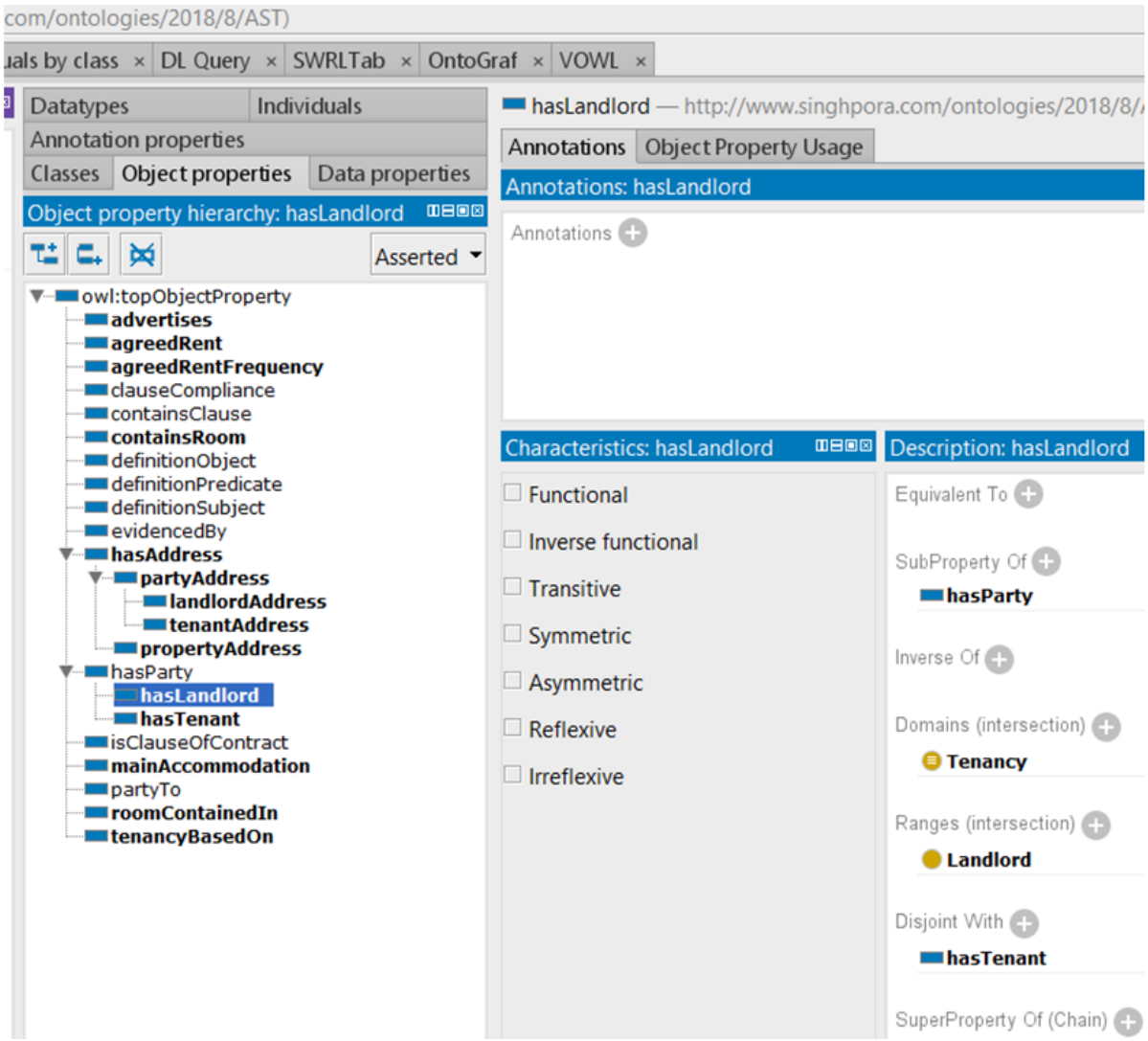


Figure 4.6: Object properties in the AST ontology pictured in Protégé

Contract Instances: The third level of data model design consists solely of individuals (instances or the ABox) representing an actual **Contract Instance**. A contract instance is created after the negotiation phase, and the outcome of that phase is a $ContractInstance$. Presented in figure 4.7 is a portion of the turtle triple serialised presentation of the OWL ontology representing a contract instance.

A point to note above is the fact that this ontology consists only of axioms (triples) defining individuals and its related asserted properties. It would have been possible for negotiating parties to override some of the base assertions if functionally permitted or if that leads to a legally sound contract - this is a check would be enforced by the application's business logic and legal support. It is also a design goal to generate the Contract Instance ontologies via a user friendly interface with most common details either pre-populated or filled in via user forms. In actual practice, users are unlikely to create many custom clauses and where they do, a human workflow should facilitate referral to legal and technical professionals so that the final contract is semantically and legally sound.

```
@prefix : <http://www.singhpora.com/ontologies/ContractInstance2#> .
@prefix owl: <http://www.w3.org/2002/07/owl#> .
@prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#> .
@prefix xml: <http://www.w3.org/XML/1998/namespace> .
@prefix xsd: <http://www.w3.org/2001/XMLSchema#> .
@prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#> .

<http://www.semanticweb.org/jvsingh/ontologies/2018/8/ContractInstance> a owl:Ontology ;
  owl:imports
  <http://www.singhpora.com/ontologies/2018/8/AST> ,
  <http://www.singhpora.com/ontologies/2018/8/Contract> .
# #    Classes (All Imported)
#
# #    Individuals
#
# http://www.singhpora.com/ontologies/ContractInstance2#tenancyagreement2
:tenancyagreement2 a owl:NamedIndividual , <http://www.singhpora.com/ontologies/2018/8/AST#Tenancy> ;
  <http://www.singhpora.com/ontologies/2018/8/AST#tenancyStartDate> "2017-01-01T00:00:00Z"^^xsd:dateTime
#
# http://www.singhpora.com/ontologies/ContractInstance2#tenant2
:tenant2 a owl:NamedIndividual , <http://www.singhpora.com/ontologies/2018/8/AST#Tenant> .

# http://www.singhpora.com/ontologies/ContractInstance2#landlord2
:landlord2 a owl:NamedIndividual , <http://www.singhpora.com/ontologies/2018/8/AST#Landlord> .

# http://www.singhpora.com/ontologies/ContractInstance2#landlordaddress2
:landlordaddress2 a owl:NamedIndividual , <http://www.singhpora.com/ontologies/2018/8/Contract#Address> .

# http://www.singhpora.com/ontologies/ContractInstance2#property2
:property2 a owl:NamedIndividual , <http://www.singhpora.com/ontologies/2018/8/AST#Property> ;
  <http://www.singhpora.com/ontologies/2018/8/AST#hasAddress> :propertyaddress1 .

# http://www.singhpora.com/ontologies/ContractInstance2#propertyaddress1
:propertyaddress1 a owl:NamedIndividual .

```

Figure 4.7: A contract instance shown as OWL triples

4.3 Distributed Ledger

Smart contract in the blockchain world refers to *immutable* executable code that runs in the constraints of a blockchain platform (virtual machines running on nodes connected to a blockchain network). When we consider a distributed ledger as a state machine, operations defined in the smart contract are used as the *state change functions* for relevant state of the distributed ledger, callable via an *address* (in a public ledger like Ethereum, both the address and the *interface or API* of the smart contract are public, which means, the callable operations must be designed carefully).

The "smartness" that exists at this level is the smartness a programmer is able to code in a programming language akin to a database stored procedure. At a very basic level, a relational database with a schema and data stored in it could very well be regarded as the "state". The data that defines a "contract" (including metadata that defines a contract such as clauses, or setup values to drive dynamic logic, details of parties) could be stored in a relational database schema. High-level programming languages such as database stored procedures written in PL/SQL or Java could be written to implement the "logic". In essence, such a system could very well function as a "smart contract". What the blockchain environment adds is that the "side effect" of the logic is cryptographically verifiable and linked to all previous transactions in the ledger in

an immutable, tamper-evident **chain** of transactions. As also discussed earlier in this chapter, a ledger can be considered as a state machine that faithfully ensures that:
`Initial or current state`
$\rightarrow$ `transaction(s) or state change function(s)` $\rightarrow$ `final state`

The first initial state is called the genesis state. The inventive step in blockchain technology pioneered by bitcoin [Buterin et al., 2014] was a reliable means to establish the *authenticity* and *order* of transactions without a single centralised trusted party. This opens up many possibilities for new applications by enabling a tamper-evident, trustworthy, public ledger with no single point of failure.
Smart contracts are (in most popular current smart contract platforms) procedural scripts that run in the blockchain and offer advanced "state change functions" to apply more complex logic than simple credits and debits. The final goal might still be to transfer value between accounts within the distributed ledger, but with more complex rules. The code that executes is itself deployed as a transaction and remains immutable. The code is, therefore, trusted, just like any other transaction. The Ethereum platform uses a procedural language called Solidity [Ethereum,] to define smart contracts that then execute on a constrained environment known as Ethereum virtual machine (VM) [Bashir, 2017].

During application design, the question that we are faced with is: *What subset of functionality is appropriate for implementation as a smart contract?*. Caterpiller [López-Pintado et al., 2017], for instance, is a whole BPMN executable process that can run completely inside the Ethereum blockchain (essentially, each activity that the business process is composed of is converted to an executable smart contract that runs on a blockchain node and can change the state of the distributed ledger according to logic defined in it).
The lifecycle of a tenancy is a series of events from the registration of the original contract to periodic activities such as maintenance events, safety checks, its conclusion or exceptional scenarios like eviction. We use a BPMN executable business process component to coordinate all these activities across systems and update the execution trail in the distributed ledger. The blockchain based contract in the prototype is currently very limited due to technology limitations, mainly that no existing reasoner was found that could execute completely *onchain*. Due to this, the BPM engine was also chosen to reside *off-chain* even though a promising solution existed for an *on-chain* BPMN engine [López-Pintado et al., 2017].
What the current prototype implementation can offer fully on-chain is the ability to track rent payments if paid between two blockchain *accounts* representing the tenant and landlord.

```
pragma solidity ^0.4.16;

/**
  @title A Tenancy contract instance (poc v0.01)
  @author jvsingh
*/
contract TenancyContract {

    ContractContent contractData;
    struct ContractContent {
       string contractId; /* Could be same as this contract instance's address on the blockchain
                             should also serve as a unique identifier */
       string contractTriples; /* contractTriples can change to a triples/struct later
                                  These only include the triples for the specific contract instance
                                  and not all the inherited/imported information.
                                  Imported information would be versioned
                                    and can always be verified using hash
                                    measurements prior to each reasoning,
                                    ensuring that system as a whole is trustworthy */
       string tenantSignature; //Tenant's signed hash
       string landlordSignature; //Landlord's signature
       string notarySignature; // Platform's signed hash

       /*parse later with a proper DateTime struct
         e.g. https://github.com/pipermerriam/ethereum-datetime/blob/master/contracts/DateTime.sol */
       uint registrationTimestamp;
       RentSchedule rentSchedule;
    }

    struct RentSchedule {
        RentPayment[] rentPayment;
    }

    struct RentPayment {
        uint expectedPaymentDate;
        uint expectedPaymentAmount;
        uint receivedPaymentTimestamp;
        string landlordSignature;
    }
```

Figure 4.8: Snippet of smart contract code showing main data structures

A rent schedule is pre-calculated from the information in the $ContractInstance$ described in section 4.2 - once the *smart contract* is deployed on the blockchain, it expects credits to the landlord account as per the schedule and publish a **state** of the contract at any given time(e.g. a warning state if the rent is a few days late); see figure 4.9. An appropriate business process can then be triggered at any time depending on this state. Other than that, key lifecycle stages of the contract (including a copy of the contract instance ontology) are also stored in the ledger to leave an immutable audit trail. This state and this trail is also publicly searchable via the 'Transactions' tab shown in figure 4.9.

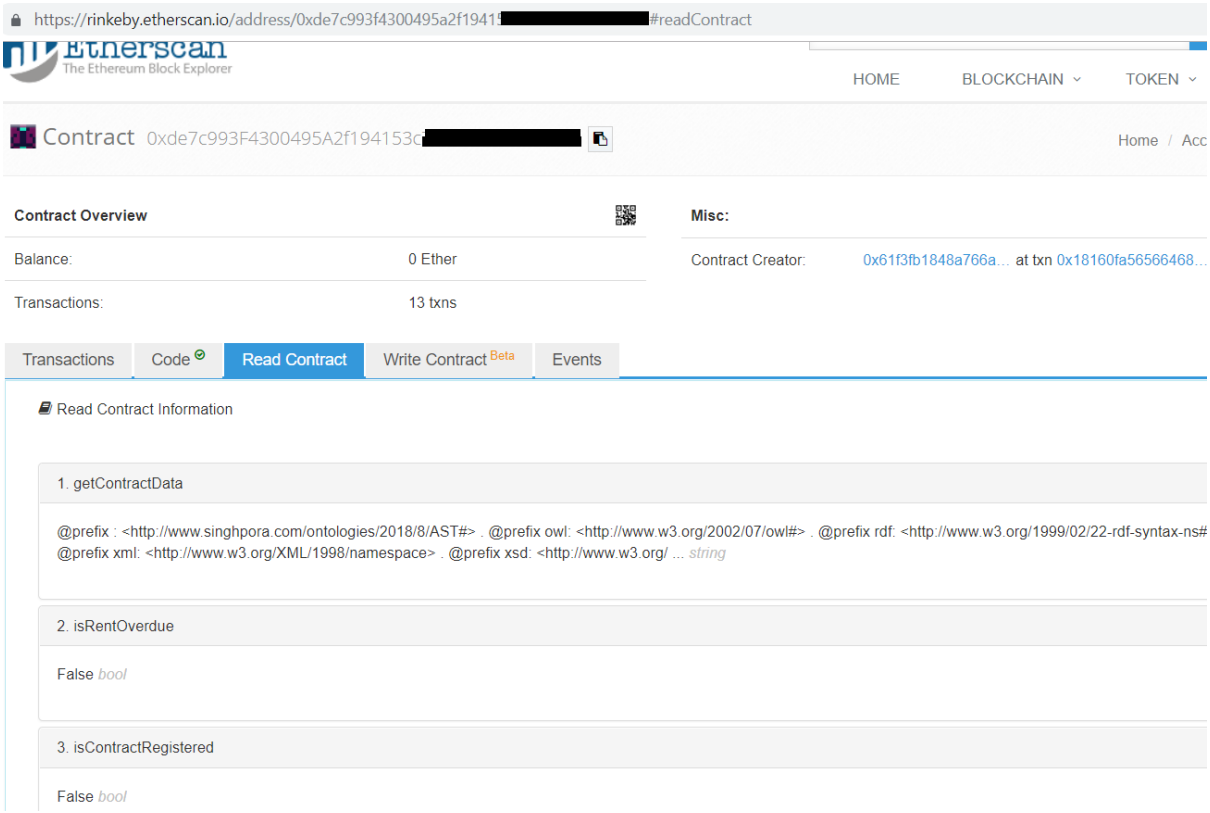


Figure 4.9: Public operations to view contract state

The execution environment initially used was a single local node started using "geth", a Go language implementation of the Ethereum virtual machine (EVM) [Hirai, 2017]. This is an execution environment similar to the Java virtual machine (JVM) and runs on each ethereum blockchain node. Before directly connecting the local geth node to a public test-bed blockchain, the prototype environment was extended to creating a private network of geth nodes on an IaaS cloud platform, effectively creating a private blockchain to test different options. The first choice of using Ethereum for the distributed ledger was due to the maturity of its high-level language program execution environment that can execute rouines written in a language called Solidity [Ethereum,]. As opposed to conventional VM based execution environments like Java, however, it emerged that the Ethereum VM is further restrictive by design and to enforce the programming paradigm whereby multiple nodes need to be able to apply the same transactions and arrive at strictly identical target state for the distributed ledger. This principle needs to be kept in mind with every chaincode function that changes state in the ledger, especially when dealing with external information. To add a technical note for completeness, the business process can make authenticated invocations on the smart contract's exposed API via a *geth* node running local to it but connected to the Ethereum *rinkeby* test network, in turn called via a custom Java wrapper that uses the web3j framework [web3j,].

Last but not the least, there are **costs** in any **transaction** that changes any state in the ledger - whether the transaction involves a simple transfer of value from one account to another or calling some non-trivial logic on a smart contract. Although the current POC solution was about exploring the extent of possibilities, a second design iteration must carefully evaluate the costs and other tradeoffs when choosing parts of the solution that would reside *on-chain*.

4.4 User Interface

The primary user interface forms an import part of the system on which success of the system will depend (measured by actual usability and user adoption). Some elements of the user interface require commodity skills to develop (stylesheets, look and feel), however, design of the user interface(s) would need deep knowledge of the domain and nature of the services. To illustrate ideas presented in this user interface, a proof of concept user interface was developed, based on the OntoPlay framework [Drozdowicz et al., 2012] discussed during research for this dissertation.

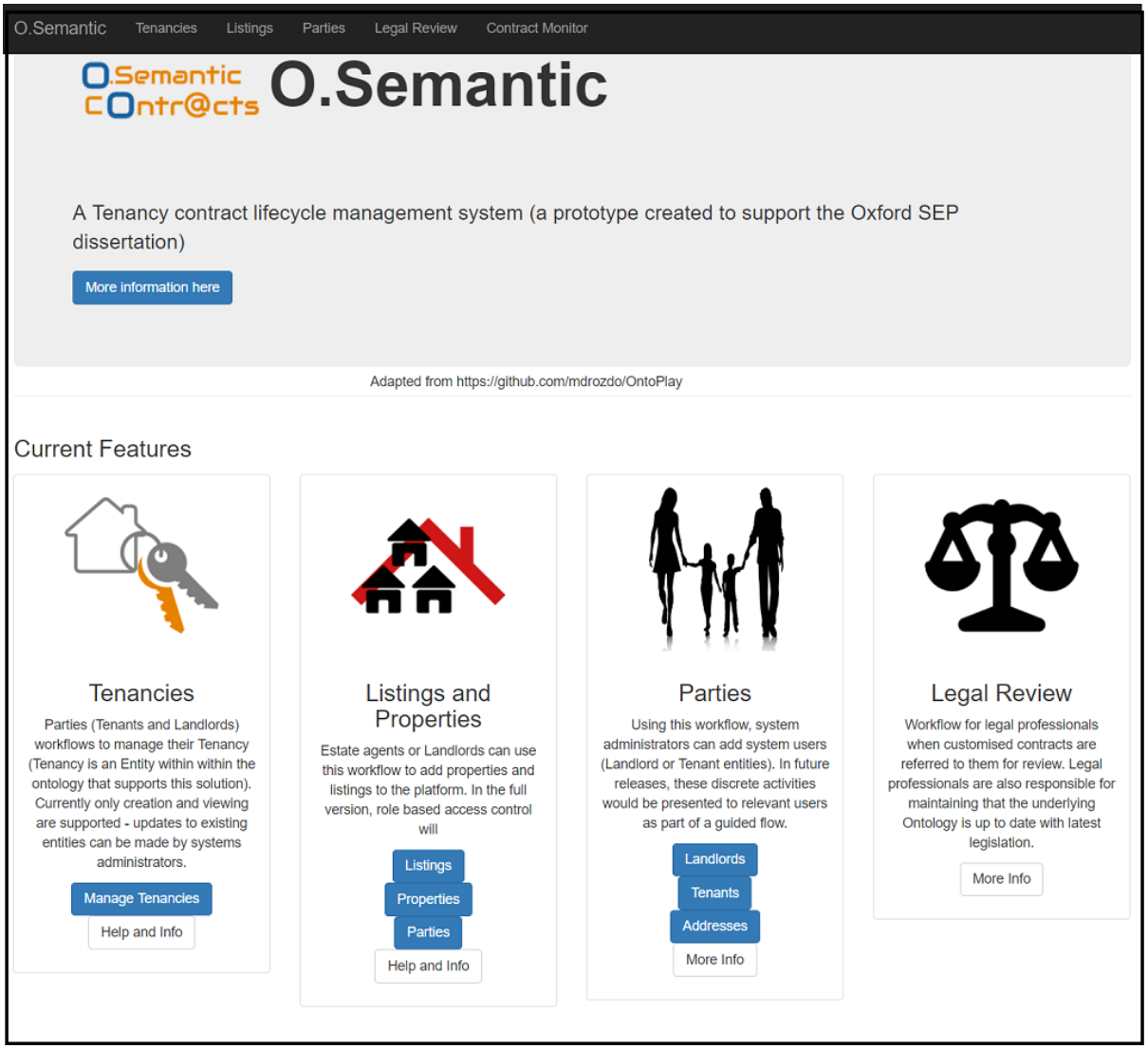


Figure 4.10: Main user interface for the prototype application

OntoPlay is a user interface framework that saved a lot of development time in creating an ontology based user interface to demonstrate the new ideas presented in this dissertation.

From a functional point of view, this interface caters to only to primary participants of the system - the non-technical primary participants like Landlords, Tenants and Legal professionals (with role based access control, not currently implemented, these users would actually see only options relevant to them). Either party (a landlord or tenant) could initiate the process of a proposed tenancy. However, an agent or landlord would have first create some setup, i.e. add

data about properties and listings. Referring back to the knowledge base, Tenancies are based on Listings and Listings advertise properties. As shown in figure 4.11, an individual can be created via a simple form auto-generated based on the OWL class definition. Properties that can be set for the new individual appear to the user in convenient dropdown menus. Targets for object properties are shown in other dropdown menus that contain other individuals from the domain of the object property.

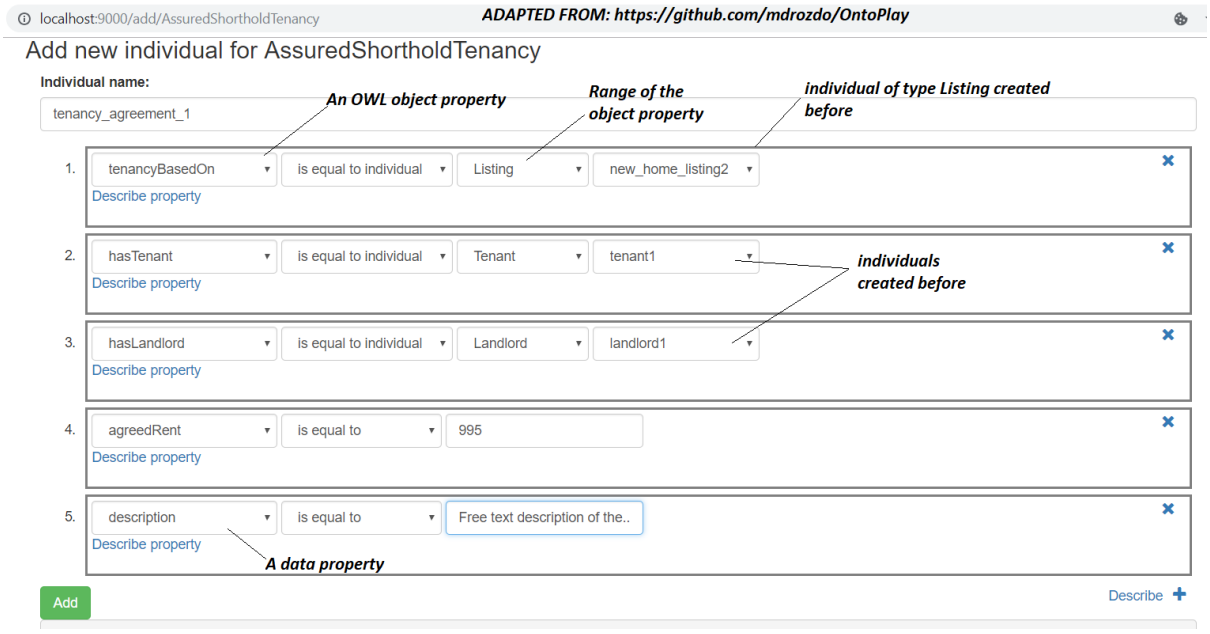


Figure 4.11: Annotated screenshot of the tenancy creation form

The tenancy agreement is created using the UI form. Internally, it is represented as an individual of the class $AssuredShortholdTenancy$ defined in the AST ontology discussed in section 4.2. Figure 4.12 shows this in Protégé.

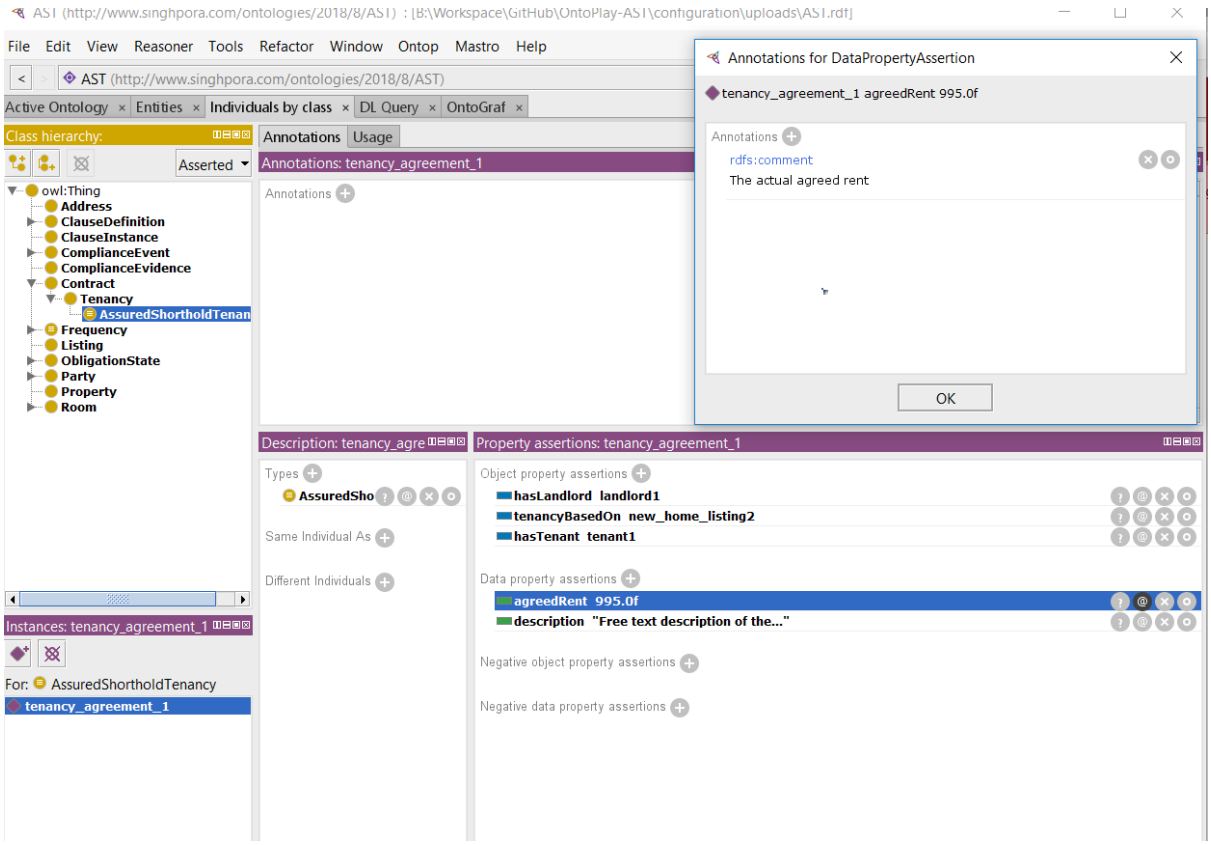


Figure 4.12: Protégé view of the tenancy instance created before

Semantic web professionals would support the system by ensuring that custom clauses added by the participants are semantically sound. Their participation in the workflow could be facilitated via a specialised interface like Protégé or WebProtégé. This kind of collaboration facilitates the scaling of scarcely available skills to a large number of users. With a sound knowledge base and well developed UI, this participation should be minimised.

There are many limitations in the prototype flow. It does not yet produce a single graphical view of a tenancy with all its properties and their properties, presented with natural language descriptions. An individual of the class $Tenancy$ or its subclass $AssuredShortholdTenancy$, its properties and transitive properties could be visualised as a tree or graph for single-view presentation. To enhance usability during contract creation, user interface *flows* - sequence of activities that constitute a single unit in UI interaction could be chained together into a guided flow by using a suitable *task flow* or *page flow* framework with further customisation of the user interface.

Although the data model is ontology based, the *presentation layer* of contract formation user interface still needs to ensure that the text displayed to users is readable in a natural language. This is achievable by ensuring that axioms or groups of axioms or triples from the A-Box have user friendly display text configured. This would in some ways be no different from common approaches for localisation or internationalisation of web applications. Briefly, these techniques involve using placeholders in code that are replaced with the real text based on the user's locale or language. Such text can also have dynamic content that displays data derived at runtime.

For instance, axioms such as:

```
:tenancyagreement1 a
http://www.singhpora.com/ontologies/2018/8/AST#Tenancy .
```

```
:tenancyagreement1
<http://www.singhpora.com/ontologies/2018/8/AST#tenancyStartDate\>
"2017-01-01T00:00:00Z"^^xsd:dateTime .
```

could have a presentation layer rendering as below: *tenancyagreement1 is an Assured Shorthold Tenancy. tenancyagreement1 starts on 1st of January, 2017.*

4.5 Reasoner

Semantic Reasoning and possibilities opened by this were the main motivation for using an ontology driven approach in this system. Briefly, *reasoning* is the ability to *infer* information that is not explicitly stated, from previously available information and rules about the domain. A general theoretical treatment of semantic technologies, reasoners, rules engines, and potential for their application in law has been done in chapter 2. The focus of this section is on describing the process that led to the choice of a particular reasoner during the prototype implementation, observations from different solutions attempted, and what might be required in the full product. From Jira task entries and comments, the first attempts started with trying HermiT and Jena via their Java APIs. As an example, with HermiT, it was possible to call the `isValid()` operation on the Reasoner interface for the well known pizza ontology (expressivity shown by Protege as SHOIN), but was not effective on the Contract ontologies described in section 4.2. When working with inferences, the **Explanation** APIs are very important both from support and usability perspectives to determine the reasoning tree followed. An example of an explanation API is the `com.clarkparsia.owlapi.explanation.ExplanationGenerator` found in the Pellet reasoner.

The second attempt identified Apache Jena as a popular reasoner and rules engine. It has (as of version 3.0.0), reasoners for both OWL and RDFS constructs. However, two challenges prevented finalising this choice for the POC: For many non-trivial ontologies, this stacktrace was a frequent occurrence and seemed to occur unexpectedly after making small changes to the ontology in Protégé:

```
Exception in thread ``main" org.apache.jena.riot.RiotException:
[line:  143, col:  24] E201 Multiple children of property element
at...
```

The second reason for not choosing Jena was its lack of support for SWRL rules - rather, the Jena documentation [Apache-Jena,] recommends:

> *"For complete OWL DL reasoning use an external DL reasoner such as Pellet, Racer or FaCT."*

The purpose of this discussion is to comment on the nature of challenges faced during the project and to present a summarised *replay* of the line of thought that led to identification of a particular reasoner.

Rules/SWRL (Semantic Web Rule Language): In an ideal situation, description of the world would be a collection of facts or axioms (entities, relationships and triples describing relationships between them) from which, all inferred knowledge could be derived by a full OWL Reasoner. This also remains the design goal of the system at least in the *Contract Instance* creation stage as we aim to allow non-technical users to put these together from a fairly pre-populated user interface. In practice, ontologies often need to be enriched by using additional

rules (specific example: from Martin Kuba's SWRL Tutorial [Kuba, 2012], inferring whether a $Person\ isA\ Driver$ is based on whether another property $hasDriverAge$ can be calculated as true. In our proposed system, this is meant to be useful for determining properties such as $nextRentDueDate$, however, the POC solution finally used a different approach of pre-calculating a rent/payment schedule for the rental agreement - also described in the Orchestration chapter). SWRL seemed a natural fit because of the choice of using OWL to represent knowledge earlier - SWRL rules can be embedded within an OWL ontology. The design goal in any case was to represent all knowledge as triples and use rules as an exception when necessary. Using rules engines such as Drools and Jena would have required a Java based object graph generated from the base Ontologies which seemed like an overhead and duplication of information even if these rules engines can be more sophisticated and more popular.

The preferred goal was to use a full OWL reasoner as opposed to rules engine. Reasoners are more *complete* [Kuba, 2012] and Pellet was the closest fit as it worked natively with OWL ontologies with embedded SWRL rules. Hermit on the other hand refused to acknowledge the rules at all and still returned the ontology as "valid". Working with Pellet was still a challenge because of often unexpected errors on making slight adjustments to the ontologies. Being closed source, it was not possible to debug easily and did not seem to be under recent development. Finally, all these challenges were overcome by using the Openllet reasoner - an open source evolution of Pellet.

Generally, when using rules in combination with ontologies, the following set of technologies collaborate: SWRL is a rule notation involving the least amount of *plumbing* code - these keep rules embedded within the ontology as noted above and OWL reasoners like Pellet can interpret these (often with the help of custom extension functions created via `BuiltIn` API extension framework.
SWRLAPI is a way to programmatically interact with these rules.
SWRLRuleEngineBridge is a framework for writing extensions so that these rules can be interpreted by external rules engines of choice and returning their inferences.
swrlapi-drools-engine is one such bridge that also seems promising as it externalises the actual reasoning to the popular drools rules engine.
SWRLTab is one of the *tabs* in Protégé that can be used to edit and run these rules (uses `swrlapi` and `swrlapi-drools-engine` under the covers) but offers the convenience of evaluating the rules there.

The challenge was some limitations in swrlapi-drools-engine which were discovered on local debugging - SWRL rules that included data ranges could not be used as the transformations were unimplemented. An example rule for determining if a Person is an Adult (from [Kuba, 2012]):

```
Person(?p), hasAge(?p, ?age), swrlb:greaterThan(?age, 18)
→ Adult(?p)
```

This narrative is recorded to serve as reference for future progress with this work and avoid the same pitfalls.

It is important to add a note on performance of reasoners - full OWL DL reasoners get slower with increase in number of concepts in the ontology [Dentler et al., 2011]. This can be a problem if we aspire for the system to reason on hundreds of millions of concepts in a linked data of the whole legal system in real time. This might never be the case - the creation of semantically valid axioms in the clause repository would be an ongoing, asynchronous process run by the platform. Moreover, techniques like [Ruster, 2016] look promising as are other techniques like federated reasoning [Idelberger et al., 2016] who proposes this in the context of a

blockchain based reasoning solution. During the course of this project, however, no blockchain based reasoning solution could be found, but, the solution has been kept flexible and modular enough to incorporate technology advances as they become available.

Finally, the choice of Pellet was finally motivated by readily available examples, its native support of SWRL rules, correctness, completeness, and support for BuiltIn functions - claims supported by [Kuba, 2012] and by observation. Eventually, I was able to migrate the POC solution to Openllet which, as mentioned before in this section, offered better stability, debugging support with an open source and recently updated code base (and is based on Pellet). These considerations ensured the proof of concept could be successfully implemented with future extensibility in mind.

4.6 Orchestration

This section covers the architectural block shown as "Contract Lifecycle Business process" in the high-level architecture (section 4.1). To ehnance modularity and loose coupling, the system is designed to ensure the high-level architectural blocks do not directly interact with each other directly, and only do so via a well defined API and middleware. Another goal is to offer a graphical tool for defining the hierarchy of business processes involved in realising the usecases of the chosen domain. This needs careful balancing with information that is represented in the underlying knowledge bases described in section 4.2. The integration middleware would contain logic governed by output from reasoning on the knowledge base and might involve human participants. To give one example of the "contract formation" process - the steps involved might be:

- Process gets initiated when the platform creates an instance of a stock contract
- Parties (landlord and tenant) fill in details
- Either of the parties add some custom clauses that they might have verbally agreed
- Once provisionally submitted, these clauses fail the validity check by the reasoner, so the process is referred to the *legal professional* in the next step.

Business Process Management is a discipline comprising areas like Business Process Modeling and activity *Orchestration*. Orchestration means coordinating multiple automated systems or human participants in a single coherent *process* that helps achieve a specific usecase or identifiable business goal. The modeling standard known as BPMN [OMG, 2014] offers a business friendly notation for non-technical users. Leading commercially available BPM systems like jBPM, Oracle, Appian and others [Gartner, 2014] offer the ability to execute BPMN process models with technical enrichment in a manner that retains the top level design-time model at runtime. BPM is as such a mature discipline in its own right and its value must be carefully assessed. In this project, the main useful features from an integration and activity orchestration would be:

- Ability to call different types of interfaces (http - Rest/SOAP, databases, Java etc.) in a more or less declarative or less verbose manner compared to doing these things in a programming language.
- Ability to model and execute human tasks as part of an overall process involving many other types of activities

- Error handling, maintenance of state (e.g. a process instance "waits" while a human participant performs their tasks
- Tracking analytics and efficiency - understanding how long activities take and identifying performance bottlenecks.
- Keeping the high-level process flow graphical and visible in a business friendly notation

A simple enterprise service bus (ESB) or even some Java code could easily achieve most of the above requirements, especially if there are no human participants involved. But, it diminishes from the user and business friendliness, and long term maintainability of the overall solution. Oracle BPM was used in the prototype solution under a permitted developer license while jBPM and Caterpiller were also assessed. Caterpiller [López-Pintado et al., 2017] could be an important technology to use in a future version of the system due to its ability to run *onchain*. Oracle BPM was found to be the easiest to set up and run so was used in the prototype implementation - one of the prototype processes at runtime can be seen in figure 4.13.

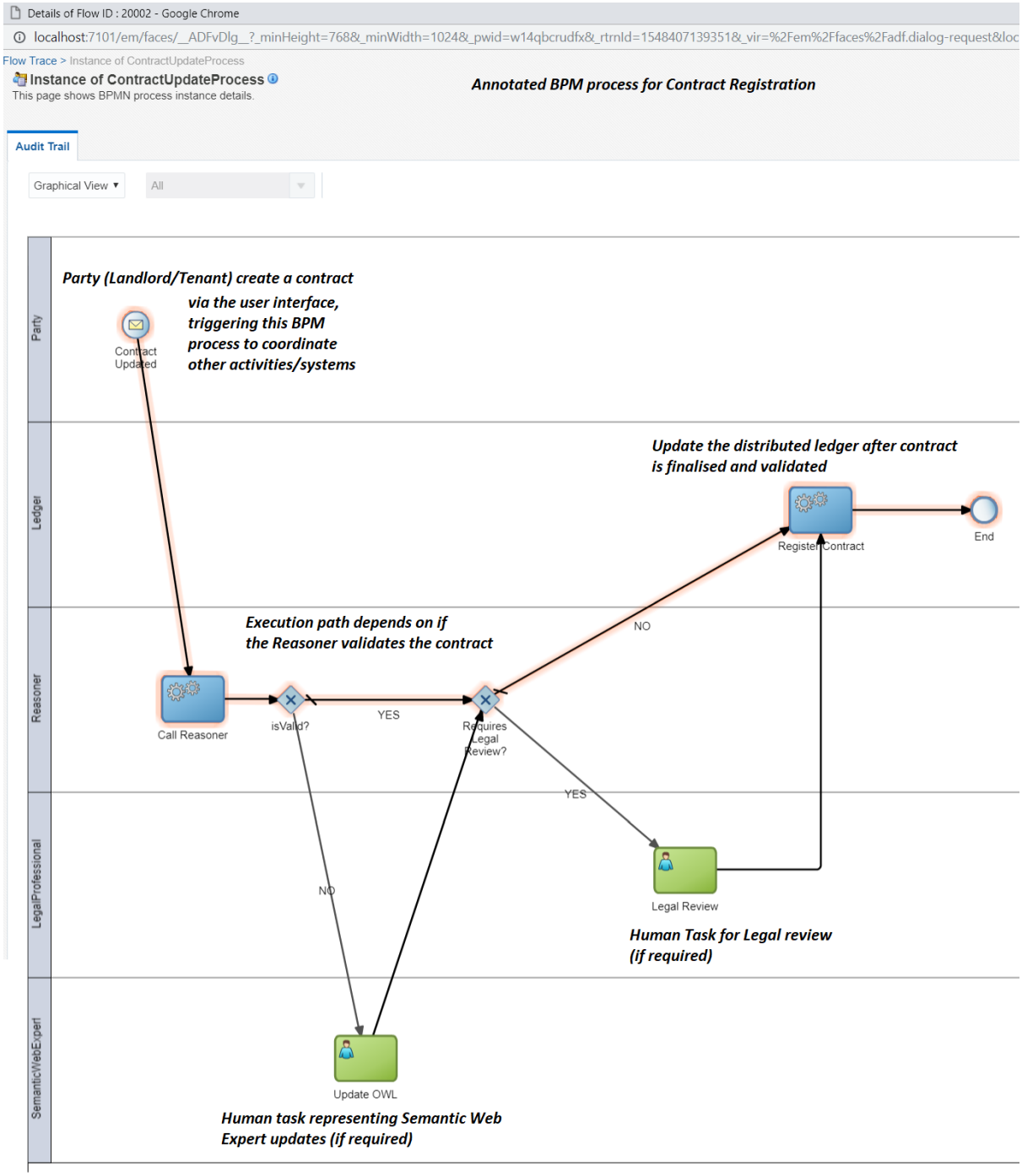


Figure 4.13: Runtime view of the contract registration BPMN executable process

4.7 Project Tracking

To reiterate, the goal of this dissertation is to demonstrate use of Software Engineering principles and tools to a real life potentially commercially viable project. It is also to enhance knowledge, test untried ideas, and contribute to advancement of knowledge using good responsible research practices[1]. A software project heading towards MVP (minimum viable product)

[1]The Singapore Statement on Research Integrity describes a number of fundamental responsibilities to be used as a checklist and which I attempted to apply during different phases of this project.

stage might need to go through somewhat different stages of development with a heavier focus on market research and commercial viability in early stages. This dissertation is primarily geared towards technical feasibility of specific novel areas of such a project. It defines a number of components that would go into a realistic product. However, rather than drilling down into one particular area as a technical curiosity, it backs multiple aspect of the product with technology assessment, proofs of concepts, and offers a framework for how various components would integrate into a single functioning product. Rather than commenting on the actual law and legislation, this project restricts itself to the technology foundation that would support such a system or what might allow domain experts to make their knowledge accessible. To organise and track this disparate set of activities over the duration of the project, it was important to use some Agile eigineering tool support, some of which are described below.

Jira: A private Jira installation was use with a simple Kanban board for tracking tasks. The project involved a number of unknowns. Kanban suited the exploratory structure of the individual tasks. The tasks initially started off as being of high granularity (create an ontology for contracts/assured shorthold tenancy) but eventually became granular (create a rule for inferring an obligation violation as a result of delayed payments). Notes made on the Jira task were helpful in extracting content for the dissertation and reflecting on experiences during the project.

Github: A private Github repository was found invaluable in tracking POC code and artefacts, and for tracking code changes related to a particular Jira item (using a Jira plugin).

Documentation: Brief notes on tools assessed were kept in a simple spreadsheet with a tab allocated to each tool. A wiki tool would have better served the purpose but it was not felt too important for a single person team. A project with a team of more than one would equally benefit from a wiki for keeping notes and sharing knowledge, but, other than that, our setup adequately facilitated an iterative and agile process that larger teams could easily participate in.

4.8 Conclusion

This chapter presented an actual possible design of the proposed system supported by a working prototype. The chapter started with a technology and domain agnostic architecture representing the *functional* building blocks derived from the functional requirements. Section 4.1 adapted this to a high-level architecture with the technologies identified and studied in chapter 3. Subsequent sections in this chapter then described the detailed design and implementation of each of these components. Each block is a full fledged, standalone, and cohesive system in its own right, integrated to achieve specific functional goals, guided by principles of modularity and reusability. Apart from novel aspects of the integration, much effort was required in understanding and applying individual and new technologies like the blockchain, semantic technologies, and reasoners. Given the ambitious scope of the project and background study required, it was not possible to compare and contrast many different architectures, however, multiple parameters and tradeoffs had to be considered in both component design and integration. Results of hands-on technology trials, alternatives, and aspirational goals are noted within the narrative. With this background, I present a concrete architecture for the working prototype in figure 4.14.

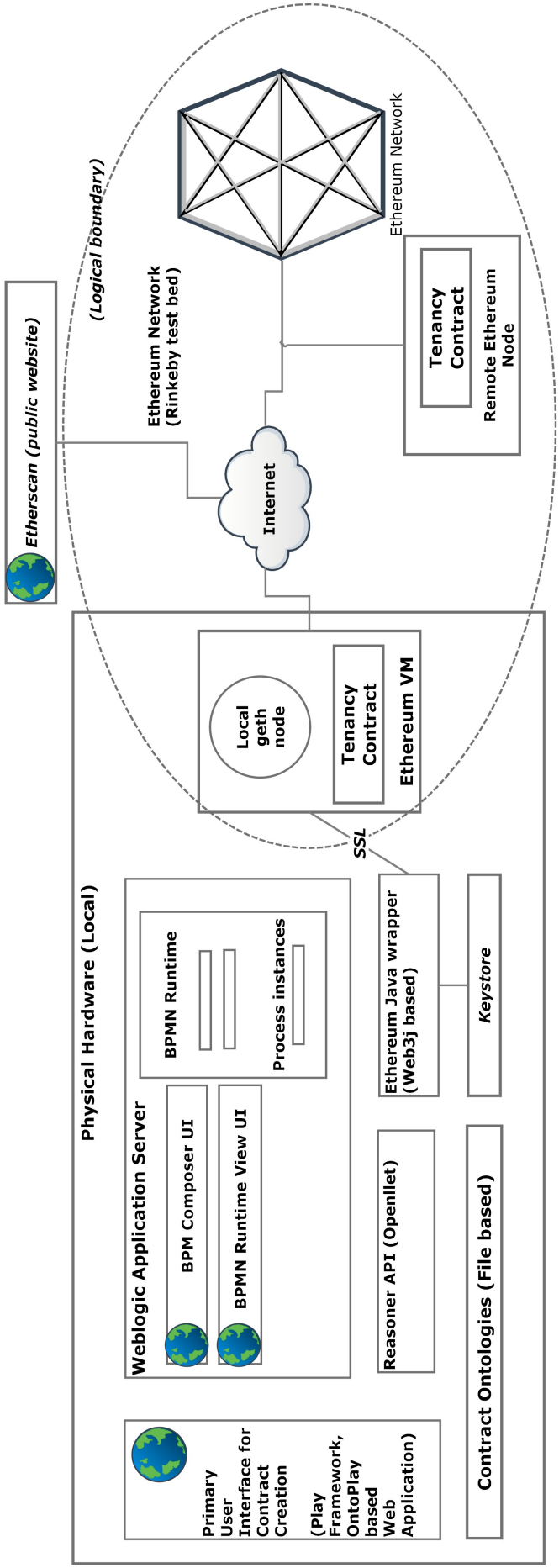


Figure 4.14: Detailed architecture for the proof of concept solution

5 Implementation and Evaluation

The narrative in chapter 4 covers many low level design details about how the individual components of the solution work. The final outcome of the design process was a detailed architecture diagram shown in figure 4.14 that supports the project aims and requirements, but that can be open to many implementation choices. This chapter lists the specific technologies used to implement that design in the working prototype and then progresses with an evaluation.

5.1 Concrete Instantiation of the Architecture

Much of prototype implementation was readily available at the conclusion of the design process reported in chapter 4. The section reports on exactly what the final prototype implementation of detailed design was, using specific technologies and versions. A concrete implementation of the detailed architecture defined in chapter 4, figure 4.14 was realised using these technologies:

Table 5.1: Tools used in the proof of concept implementation

Tool	Notes
Java	Version 1.8.0_191
Geth	Version 1.8.13-stable-225171a4
Solidity	Version 0.4.16
Solidity Windows compiler	Release 0.4.16
Ethereum	Ropsten and Rinkeby testbeds
Web3j	Release 3.5.0
OntoPlay	Based on latest master branch from `https://github.com/mdrozdo/OntoPlay` as of Dec 2018
Oracle BPM Suite	Fusion Middleware 12.2.1.3.0
Weblogic Application Server	Fusion Middleware 12.2.1.3.0
Openllet Reasoner	Local build v2.6.5-SNAPSHOT of integration branch from `https://github.com/Galigator/openllet` as of Dec 2018
Protégé	Version 5.1.0
Operating System	Windows 2010 Professional
Hardware	Intel i7 based system

Transitive dependencies are not listed unless these involved significant separate setup or not clear from release notes of the versions listed. SNAPSHOT versions or local builds of development branches use versions of code as of December 2018. From a more technical development perspective, the interaction of components is represented by sequence diagrams

such as figure 5.1. This should allow the reader to visualise a technical flow better instead of the lowest level of program code (even though some relevant snippets of code have been shown in the previous chapter).

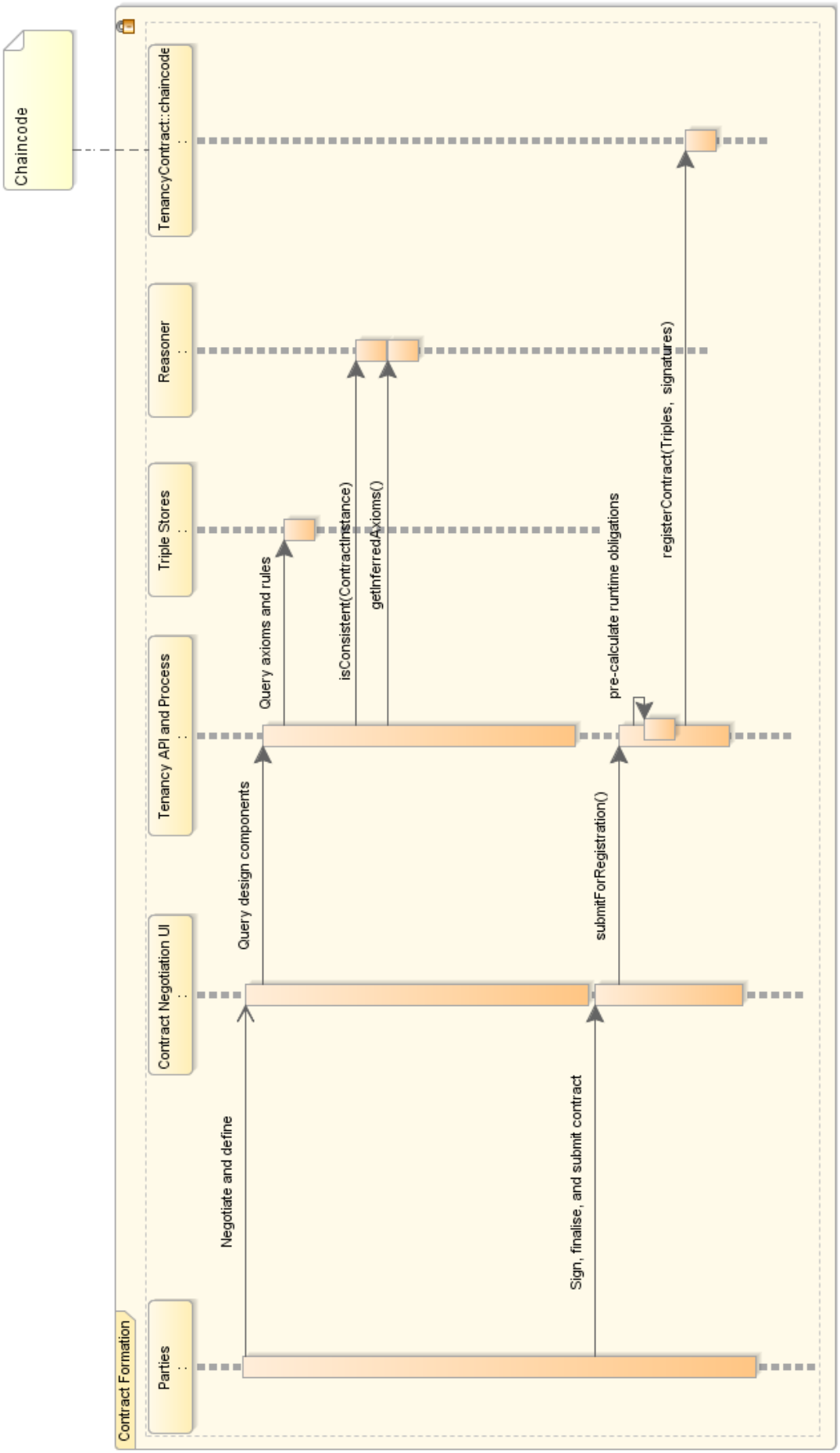


Figure 5.1: Sequence diagram for the contract registration flow

5.2 Implementation Challenges

To reiterate, each high-level architecture block that was identified at the beginning of chapter 4 is a full fledged application in its own right. The prototype implementation had to shrink the full application development lifecycle for all of these into a concise scope. Neither the design goals of this project, nor this type of synthesis of technologies was found in a previously existing system during background research. There were however, examples and documentation available for the individual technologies that were referred to, even if sometimes sketchy. The work that supported the prototype implementation involved background research, learning curve, reading technology documentation, understanding many different areas, and then creating the design in which they worked together. Each component had to be analysed for suitable implementation options that would also work well with others - for example, even the choice of serialisation format of OWL ontologies had to be revisited after the user interface framework was unable to process turtle. Many different reasoners were evaluated before converging on Openllet. Trials involved testing different tools, identifying appropriate APIs or libraries, and debugging open source libraries when faced with obstacles. Some important tools that were evaluated but not used in the implemented prototype are listed in appendix A.

Most of the technologies listed in table 5.2 involved a non-trivial setup, including a number of transitive dependencies with their own setup. Often, multiple versions of a tool had to be attempted before confirming a version that was suitable for the overall prototype environment. Before discovering the *OntoPlay* framework, my attempts had been to develop a simple user interface technology to render forms based on Ontologies. The adoption of blockchain tools involved not just the initial learning curve, but also extremely time-consuming setup. Although the final design converged on using an Ethereum testbed (both *rinkeby* and *ropsten* were used at different stages), for initial testing a single local *geth* node was installed. This allowed the initial testing of ideas and identifying an approach for integrating with rest of the system (finally achieved using the *web3j* library and custom code exposed as SOAP). To run some realistic tests with blockchain technology, a number of other installations were attempted, including first setting up a private network of geth nodes, both local and on *IaaS* cloud, and then finally using a public blockchain testbed network called rinkeby. Setting up a local node connected to a public network involved the *syncing* stage before transactions could be executed. This took more than 48 hours even though this was not a full fledged mining node. On more than one occasion, when I paused my system to document the work or during work related travel, my local node would go out of sync and syncing it back was very time-consuming. Luckily, during last stages of evaluation, I found a **public** ropsten node that anyone could connect to (Making transactions on behalf of an account still requires the account's private key, which only I had). One of the features offered by testbeds like rinkeby and ropsten is provision of test *Ether* tokens for testing via *faucets*[1]. These tokens are required because, each transaction that is created on the blockchain costs some tokens which can either be acquired or *mined*. This narrative is recorded here to give the reader an idea about the background work involved in setting up this working prototype.

5.3 Technical Evaluation

With the context set by the previous sections, I now report the technical evaluation of the prototype with help of the usecase that was identified at the conclusion of chapter 3. To perform

[1] An example faucet is `https://faucet.ropsten.be/`

an end to end execution of the prototype system, we start by defining a small tenancy agreement or `ContractInstance` containing the following data set up from the primary user interface. This triggers the contract creation business process. The goal is to follow the execution of the different components that are implemented in the solution, all coordinated by the BPM process: User Interface(s), Reasoner, Blockchain wrapper, and finally, transactions created on the distributed ledger (an Ethereum test network).

Table 5.2: Subset of test data used for evaluation

Setup Element	Test Data Value
Property:	Property1
Address:	Property1Address
addressLine 1:	**2 Regents Street**
addressLine 2:	Slough
Listing:	new_home_listing1
advertises: advertises:	Property1 (created above)

The *Setup Element* value in the table above corresponds to Classes and data or object properties in the Contract Instance ontology while *Test Data Value* corresponds to instances or data property values that are important for this test. Data or object property names begin with a small case.

For evaluation, we start by creating a small test contract using the data listed above. See figure 5.2 for part of the prototype UI flow used for contract creation. (Figure 4.11 in chapter 4 that is also involved in tenancy contract creation is not repeated here to retain focus on the test data.)

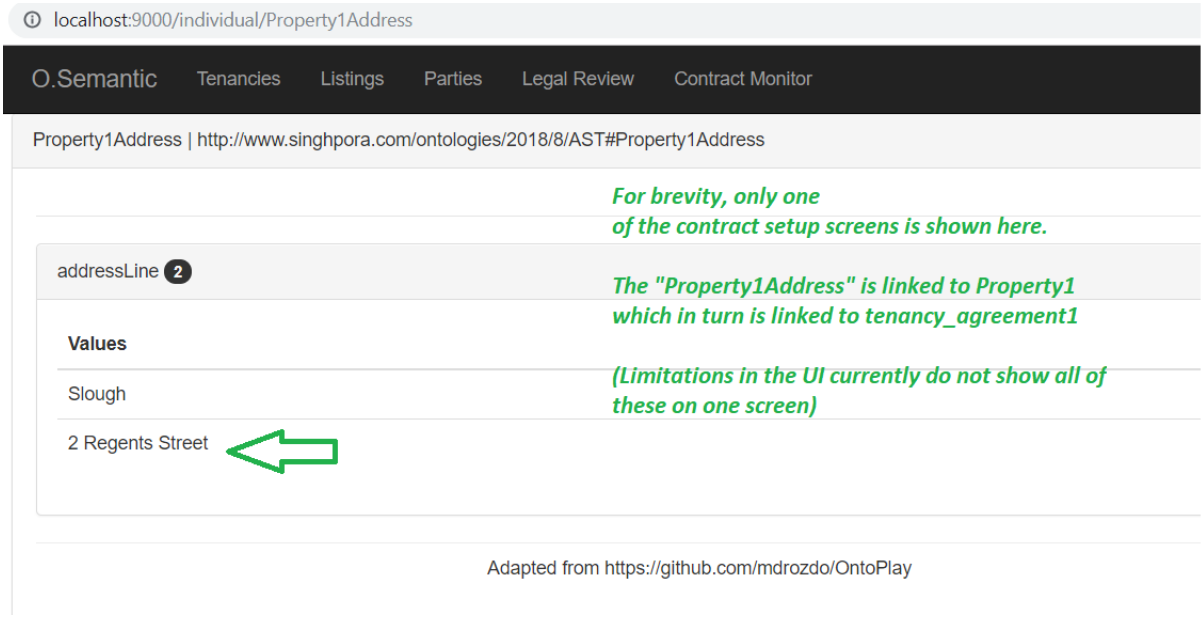


Figure 5.2: Screen showing a part of contract creation on the UI (Property address highlighted)

Once the contract is finalised on the primary user interface, the BPM process is triggered and it orchestrates the other systems involved. Figure 5.3 shows a single view with a section of

the BPM process with the actual execution path highlighted. Below that, the same image shows log outputs from the chaincode wrapper, i.e. Java code that calls the Ethereum geth local node over local TCP. The BPM process invokes the chaincode wrapper and reasoner components over a local SOAP web service interface exposed for the prototype. The **Deploy Contract** activity deploys the contract on the blockchain (an Ethereum testbed network), and returns a public address, using which, subsequent invocations can be made to the smart contract. One such subsequent invocation is the **Sign Contract** activity that represents an acceptance of the contract by the parties. In the actual implementation, this might involve a human task where the parties explicitly review and digitally sign the contract after it has undergone Reasoner validation or expert intervention if required.

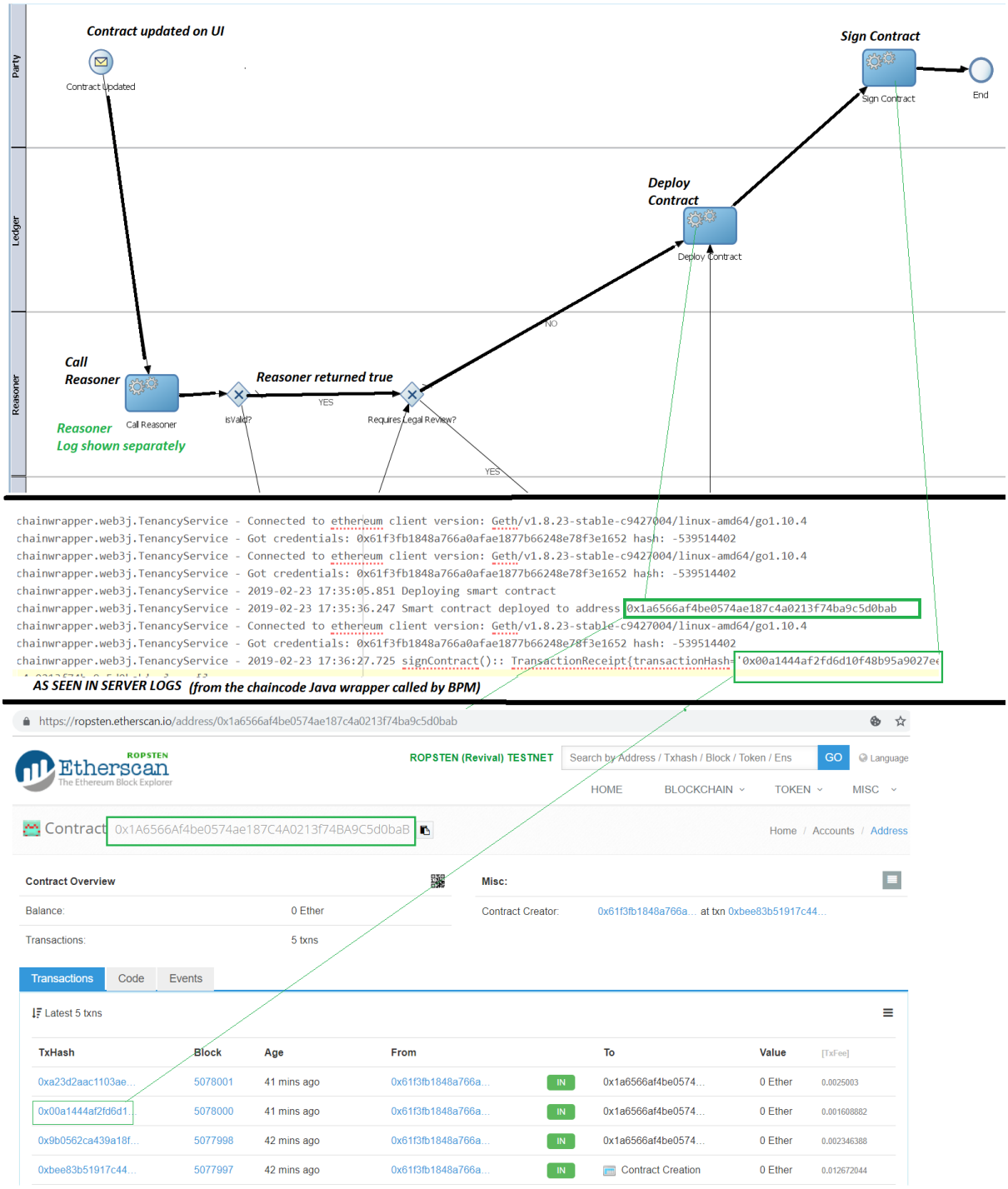


Figure 5.3: Flow of control and data during execution of the Contract Registration process

Related to the above test, the **Call Reasoner** activity results in a call to the reasoner. In the prototype, this only leads to a validation of the Contract Instance ontology, returning a true or false. The Java implementation of this operation calls the `OWLReasoner` API as below:

```
OWLReasonerFactory reasonerFactory =
OpenlletReasonerFactory.getInstance();
OWLReasoner reasoner =
reasonerFactory.createReasoner(ontology,
```

```
 new SimpleConfiguration());
boolean isConsistent = reasoner.isConsistent();
```

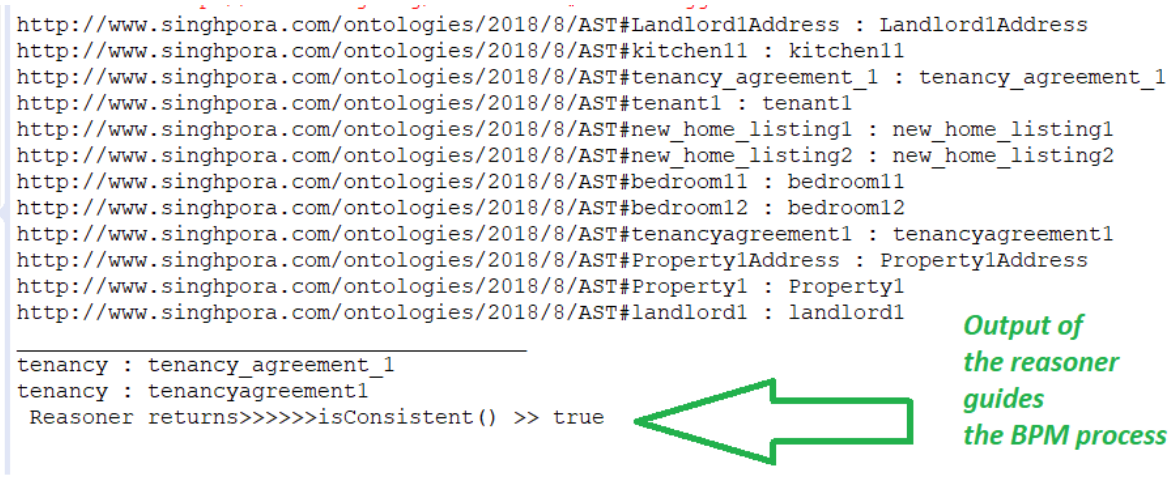


Figure 5.4: Part of the reasoner output (simple validation of the contract)

The simple example shown in figure 5.6 later in this chapter shows how this is called by the BPM process over a SOAP web service interface. As a final verification step, we can explore the contract's audit trail in multiple ways. Each individual transaction can be viewed via a blockchain explorer user interface such as Etherscan (shown in multiple preceding figures in this section). A purpose built function `getAuditTrail` is also written to view the different lifecycle steps in the contract. A purpose built user interface can be provided for this in future, but currently, this can be invoked via the public Ethereum blockchain explorer tool called Etherscan as shown in figure 5.5. The property address from the original contract is only used as a mnemonic identifier for verification purposes - as such, other data like signed values, the contract address, or individual transaction hashes are more useful for future verification. The timestamps in the figure can be tallied with server logs shown before.

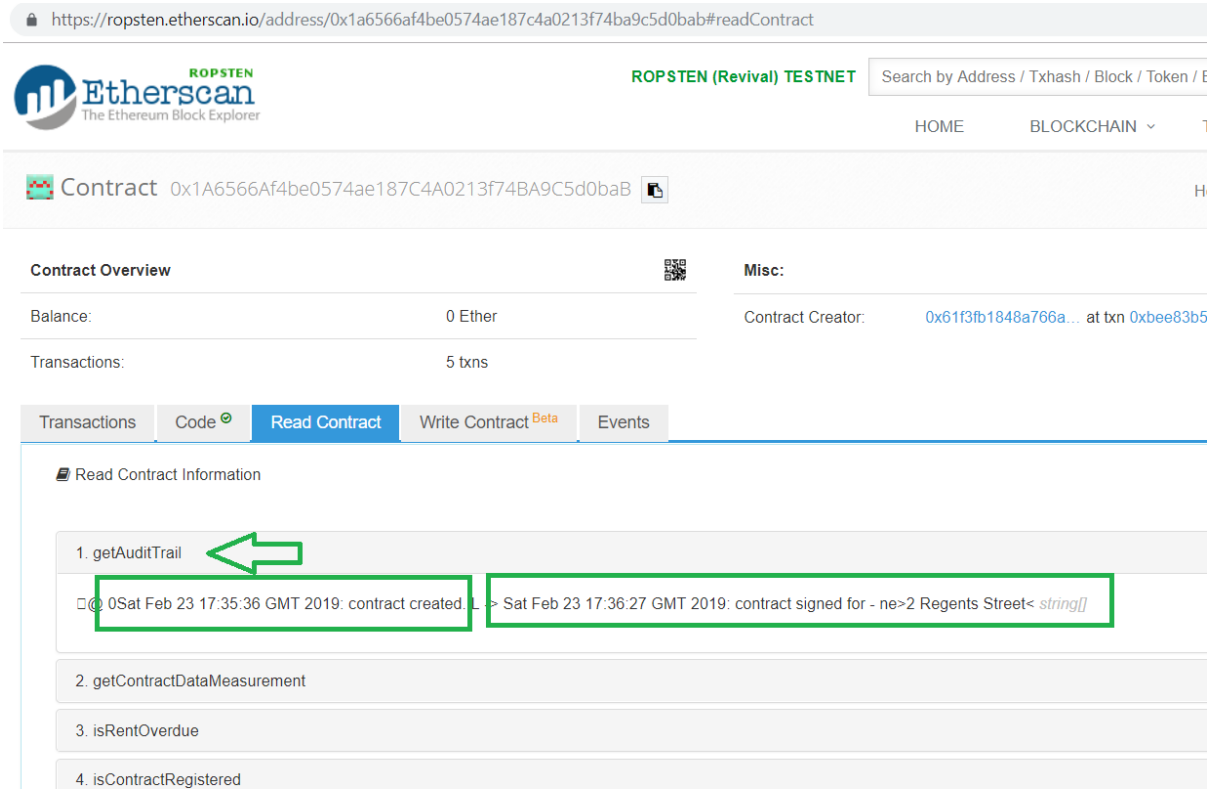


Figure 5.5: Output of the `getAuditTrail` function showing two lifecycle steps

5.4 Non Functional Properties Achieved

This section reflects on how the architecture and design achieves important non functional requirements from ISO/IEC 25010:2011.

Scalability and Performance: The user interface is fairly decoupled from processing that takes place behind the scenes so can be independently scaled up based on increase in usage. The more heavy processing like complex reasoning, potential human activities, and creating blockchain transactions (a potentially time-consuming step) execute asynchronously behind the scenes, coordinated by the BPM process instances. Leading BPM platforms, including Oracle BPM chosen for the prototype implementation, also provide capabilities like KPI monitoring, so processes can be improved over time after observation.

Reusability: Components created for this solution actually serve a general needs. The reasoning service that was created fits into the contract negotiation business process, but can also be used in other contexts. The wrapper service used to interact with the distributed ledger is another such example that can be 'plugged' in to our business process for contract management, but is, reusable enough to be applied in completely different contexts. The only components not specifically reusable in other types of contract scenarios is the Orchestration (BPM) component and the User Interface. There could be scope for identifying some common process templates in a future design iteration. The user interface, even though it uses a framework (OntoPlay) that can render any Ontology, still had to be customised for the current requirements and would need more customisation for usability in future iterations.

Modularity: The high-level architecture identified components early on, before proceeding with a detailed solution. Any of the solution's building blocks could be replaced or implemented in completely different ways and in different related domains of contract management. The detailed design maintains clean interface between components for loose coupling of sys-

tems. The Orchestration module implemented in BPM allows coordination between systems. Had there been no need for stateful, long running processes, even other middleware technologies like an Enterprise Service Bus would have served the purpose.

Cost and Licensing: For the prototype, the main user interface component is created using open source technologies under permitted license terms. The proprietary Oracle BPM solution was chosen over jBPM for the Orchestration module after evaluation because of ease of access, well documented support, ease of setting up, integration with a variety of interfaces (SOAP, Rest, JMS, Files etc.), and a licensing model that allows free use for development, testing, and prototyping[2]. The tradeoff in this space is between costs, development effort, and vendor lock-in risk, but the modular design ought to allow replacement without significant impact on other parts of the system like the user interfaces or the register. All other components were developed using tools and technologies with permissive, open source licenses.

Extensibility: The architecture and design proposed at different levels can be extended to other areas of law such as the employment law - the high level architecture, including choice of components caters for that by simply instantiating it with a different knowledge base. Even the building blocks identified in figure 4.1 leave room for different detailed designs for one or more blocks. The development of knowledge bases in a standard format (like OWL) also enhances extensibility. The architecture supports evolution to a 'future state' as technology matures (for instance, if reasoning could be supported *onchain* in future).

General Performance: Typical interactions that users have with the user interface would involve data entry, reviewing pre-built contract clauses, using pick-lists to add new clauses, and possibly searching for entering new custom clauses. Although these features enhance user experience, the last two operations that could involve performance challenges during the contract drafting stage, if we expect a reasoner to be invoked there in real-time. The overall workflow was thus optimised to ensure that heavy reasoning on a large knowledge base, such as contract instance linked to a large legal repository, only take place asynchronously. With this approach, user interface responsiveness was decoupled from activities that could be offloaded to back-end processing. For instance, once a party has reviewed and finalised a contract, the registration can proceed behind the scenes and parties would be notified once confirmed. This approach also decoupled real-time user interactions from known blockchain scalability problems. Further means to enhance performance of individual components is applying standard principles of high availability for components in our control.

5.5 General Test Strategy

Due to the number of unknowns and potential for identification of new requirements based on actual user behaviour, a comprehensive test coverage of all technology components would be of utmost importance. This is the only way the system will remain agile and responsive to different types of change, like:

- User experience related changes based on actual user interaction and feedback
- Changes in business logic, and business flows
- Changes in the Knowledge base as a result of changes in legislation or as recommended by expert feedback

[2] `https://www.oracle.com/technetwork/licenses/standard-license-152015.html`

The benefits of a test driven approach to software development (TDD) are well known and do not need to be reiterated here. The reader could refer to *"Physics of TDD"* and the benefits of a test driven approach over *"debug later programming"* by [Grenning, 2011]. This section rather focuses on how a good test strategy can support both evaluation and long term evolution of the system.

The main goal from a technical test strategy point of view would be to ensure that for a given version of the legal knowledge base, for a given version of a contract instance, and for a given system configuration including technical component versions, the system behaves in a deterministic and expected manner. A set of repeatable test cases must consistently produce expected outputs. New tests are always added when some new scenario is identified from actual usage. Any changes in legislation, a functional feature change in the system, or even a technical change would be incorporated in a controlled manner only when we know precisely what changes in existing behaviour it would cause. This is possible with an extensive suite of repeatable test cases.

To unit test the different components of the prototype, I ensured that the *interfaces* and *integration points* are both well documented - this means, inputs, outputs, expected fault scenarios, unexpected fault handling. It helps that in the prototype solution, the Reasoner and BlockchainWrapper (Ethereum Java Wrapper shown in 4.14) had SOAP interfaces for easy integration with the other components via the BPM process engine. The BPM process to *Register Contract* is itself triggered when the OWL file representing a contract instance is ready for background processing. For ease of unit testing, a SOAP interface has also been exposed for this.

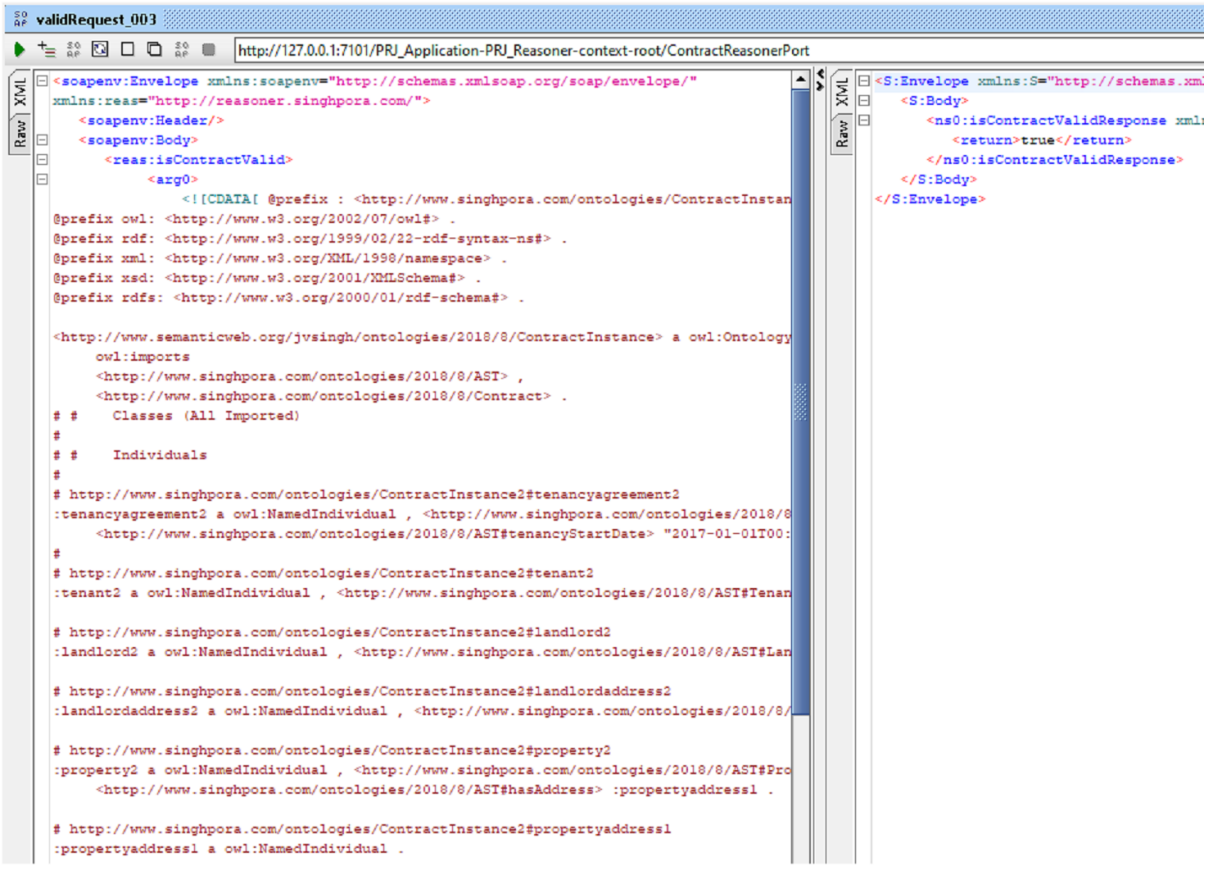


Figure 5.6: Example manual unit test for the reasoner

The simple example shown in figure 5.6 checks if the ontology representing the contract instance is valid. The Java implementation of this operation calls the OWLReasoner API as below:

```
OWLReasonerFactory reasonerFactory =
OpenlletReasonerFactory.getInstance();
OWLReasoner reasoner =
reasonerFactory.createReasoner(ontology,
 new SimpleConfiguration());
boolean isConsistent = reasoner.isConsistent();
```

5.6 Contributions and Aims Achieved

Now that the design and prototype development stage is complete, this is a good opportunity to reflect on how much of the original aims of the project have been achieved. I re-visit section 1.4 now in light of the results reported in previous sections. While these aims offered an initial direction and guided the subsequent work, the success of this project cannot be the literal achievement of these aims but the research, learning, design, and possibilities identified. The previous chapter concluded with what I think is the most important contribution of this project - an actual detailed design that was the outcome of research accompanied by hands on technology trials. The most important technical goal of creating a viable design in which these technologies work together was achieved. This design and its prototype implementation supported the original aims as discussed below:

It is definitely proved that the proposed system - a semantically contract lifecycle management system - is possible. It can be created with semantic technologies that allow linking to different knowledge bases (which in future could be official legislation). This already is a big step forward from plain text based digital contract documents currently in use. The work also proved that BPM technologies can be integrated effectively to perform long-running coordination of activities that might involve human participants and that such a process technology offers value to contract lifecycle management.

The reported work identified multiple challenges. Whereas the initial motivation was to build smartness literally into 'blockchain based' programs also known as *smart* contracts, pragmatic considerations and limitations (high cost of developing missing features, currently high cost of running complex transactions on a popular blockchain platform) meant only a subset of logic is created as *chaincode* while the rest is kept *offchain*. This does not diminish any functional features of the system. As technology matures, the design is kept flexible enough to incorporate new features. Logically, this means, the dotted line shown in figure 4.14 representing the blockchain components could shift to incorporate the Reasoner and BPM engine if this is found suitable in future.

It was shown by some simple examples that the outline of legal contracts, including details of parties, rights, and obligations, could be represented in Ontologies. The more specialised case of tenancy agreements was used as an example for this dissertation. Research and literature review described in chapter 2 further supports this. It would have been more satisfying to have created a more detailed knowledge base representing the chosen area of tenancies. For example, Inventories representing contents of rented properties are an important part of any tenancy agreement and can have a bearing on final monetary obligations at the conclusion of a

tenancy. But this and many other aspects were not addressed despite the know-how to ensure all system components received adequate coverage.

Technical know-how was researched and then applied in the prototype where a contract instance was defined as an ontology, validated supported by OWL Reasoners, and then a rent schedule (representing obligations) pre-calculated and stored in a blockchain smart contract that expects rent payments to be made to a blockchain account representing the landlord. This is unfortunately still very disjointed as the smart contract code does not naturally derive from the ontology but is custom-written specifically for one obligation, which is the rent schedule derived from individuals and properties in the ContractInstance ontology. So even if for this one obligation can be met *onchain*, the bigger value added by the use of blockchain technology currently is the faithful and non-repudiable record of the contract lifecycle. This could change in future if reasoning capability can be achieved *onchain*. Also see section 6.3.

Finally, the experience has been mixed in terms of ease of implementation. There was a steep learning curve for assessing different possible technologies that would suit the different building blocks identified. This was preceded and accompanied by theoretical study of some unfamiliar functional areas and also technology paradigms as reported in chapter 2. Setup of each component was complex even aside from the learning curve. Due to the project time spent on these activities, it was not possible to develop a more complex working solution with more functional features, even though the design and prototype reported in chapter 4 adequately supports my claims.

Strategic Evaluation of the System At the top level, a meaningful evaluation of the system would be actual adoption by users, which, in turn, would be governed by the immediate usability features. At the very least, the contract formation experience should become vastly simplified and easier than text based contracts, mainly due to the usability features of the user interface facilitated by the underlying ontology knowledge base. The reasoning capability, linked data that incorporates relevant areas of legislation or property data, facilitating participation of legal experts, and the audit trail in a distributed ledger all enhance the overall processes. Theoretically we can look towards a future where disputes are minimised, easier to resolve, delays avoided, and fraud prevented. However, measuring these achievements in operation would involve more focused research. Key performance indicators, such as percentage of tenancy disputes or duration of dispute resolution would need to be collected for suitable sized scientific samples now and compared with similar samples of customers using the new platform. Another main strategic evaluation of the system would be full participation of legal professionals in encoding the law for various different domains. After all, the present system is merely a framework to facilitate that outcome.

6 Discussion and Conclusion

6.1 Summary of Reflection

It was my attempt that reflection on different choices, decisions made, and pros and cons of different alternatives should permeate throughout this dissertation. This section is intended to draw attention to some important ideas with the further benefit of hindsight. Due to the nature of the project, it was not possible to obtain metrics on performance and scalability from the prototype implementation, even though existing research and design choices detailed within the main content suggest no major scalability bottleneck. Many unfamiliar components were studied and multiple options or ideas were tested iteratively - design elements were often noted in retrospect. Had the nature of fields been more familiar, design and development might have followed in a more conventional and sequential fashion. I feel this did not diminish the depth or breadth of the final outcome and the Agile approach sufficiently supported this work.

As the investigative nature of this work was intense, there was little scope for user trials. In a more commercially driven initiative, a purely user interface driven solution with all the promised usability features could potentially be released to actual test users to gain first hand insights into user expectations. For this dissertation, such an approach would have compromised the design of a flexible enough solution that would cater for aspirational goals and incorporate new developments.

Besides the sources of requirements listed in chapter 3, there was no real stakeholder participation or consultation during this project. Using a real problem statement that is likely to evolve into a product, focus in the dissertation was maintained on restricting this to a primarily academic project to demonstrate a wide range of principles of tools from Software Engineering. Any future development or derivatives of this work would need to involve closer collaboration with the legal profession, in addition other stakeholders. With the technical know-how and possibilities learnt from this project, any potentially commercial enhancements must first start with a market survey or consultation to identify the actual demand for such a system and identify additional, corollary requirements that could be fulfilled.

As a blockchain based distributed ledger, Ethereum was the only technology applied in this project as it had the most readily accessible tooling and support including a test-bed network. This area was new and unfamiliar, exploration and practical understanding was time consuming. However, I feel the core principles around decentralised computing and distributed ledger technology were adequately addressed and their value demonstrated. I hope the discussion of this subject from its basics contributes to overall knowledge of the field and serves as a tool for clear understanding.

The limitations in the prototype system are primarily the result of complexity and lack of

similar precedent that I could find. In fact, finding a way of achieving this combination of technologies to deliver the kind of functionality discussed early in this dissertation was the point of this project. This, in my view, was achieved and demonstrated conclusively. The agile approach was very helpful, and, I believe, was applied in a very meaningful way as a Software Engineering practice. Overall, the project offered the opportunity to apply Software Engineering processes, tools, and techniques to a very unfamiliar and challenging problem.

6.2 Conclusion

The biggest contribution of the project is the architecture defined by the end of chapter 4. Apart from functional requirements, how this architecture achieves many desirable non-functional properties, including working around known scalability issues of blockchain technology is also detailed in section 5.4. The architecture was proven by a working prototype developed using a set of specific technologies. As the approach was to first start from a flexible high-level architecture, it leaves room for alternate implementations. The detailed design can also be instantiated for a different contract domain with the very same set of technologies.

Without access to any similar previous applications, the work offered an opportunity for challenging research and pushing boundaries of what is possible. This meant starting many times over with different tools and adjustments. The integration points were proved using currently available technologies but in an original combination. Each of the very diverse technology areas that were involved in the project are non-trivial and were studied both for their specific value addition and fit into the overall solution. Due to the unfamiliarity of both the domain and technologies involved, much of the project time was spent in proving functional aspects of the solution could be achieved with the use of current technology.

During the course of my research, I found a lot of existing research backing my ideas (notably [Boer, 2009], [Idelberger et al., 2016]) and line of reasoning. Therefore, although I cannot claim the idea of using Semantic Technologies in contract law as original as I thought at the beginning of the project, the specific application and combination of tools to develop the proof of concept is unique as far as I could conclude. The application of software engineering principles and techniques (often learnt in seemingly unrelated SEP modules) to an unfamiliar domain was quite satisfying and demonstrated value derived from the program.

Another unique contribution of this work is that it could serve as a reference at many different levels - from a flexible high-level architecture, a more detailed technology specific architecture, to lower level implementation details. Both the high level architecture and the technical framework detailed in chapter 4 can serve as a reference for similar solutions in other domains. One of the lessons was the need for initial tradeoffs between the amount of chaincode that is technically feasible to deliver the solution on a public blockchain. More chaincode is good for transparency and trustworthiness, but very costly to run. But, if higher initial investment for development could be sustained, it should be possible to deliver the BPM and reasoner components on a fully private or hybrid blockchain. Finally, it was very satisfying to study past works, knowledge that other scholars and practitioners have contributed to the different subjects that this research drew from.

6.3 Future Work

The aspirational future goal of the project is to *access the law* via a linked data publication API similar to [legislation.gov.uk,] or [Angelidis et al., 2018]. Where these initiatives are top-down, the design proposed in this project works form the **bottom-up**. At some point, both sides would mature to enable a natural **convergence** - linking of individual contracts to actual law. Until that happens, there is still value to be derived from developing this design with enough of the *law* for a relevant domain encoded in the appropriate knowledge bases (section 4.2) supporting this system. Another natural enhancement that could be somewhat easily realised is to link with pubic linked data like land registry records. With adequate user adoption and buy-in from the government, this system could serve to prevent some kinds of fraud like fake or unauthorised tenancies.

The features and properties desired from distributed ledgers are achievable now, as they offer a usable enough trustworthy store of the contract state and for tracking contratual "obligations" like periodic rent payment. But, the full scope of activities like semantic reasoning is not yet practical in blockchain *chaincode* or what are popularly called *smart contracts*. A pragmatic decision was taken to offload such execution "offchain" and keep the chaincode logic minimalistic and focused on maintaining state, serving as a non-repudiable, trustworthy, queryable, and highly available log of the contract's runtime audit trail. Besides the prohibitive transaction costs on major public blockchains today, the limitations of current blockchain virtual machines make it impractical and many of these restrictions are by design. After re-assessing these tradeoffs carefully, I still think technically it is possible to create a custom solution by extending the VMs to include reasoning libraries. As one area of future research, in the Ethereum environment [Merriam, 2017] [Hirai, 2017], the python based py-EVM variant could for instance be extended to include python based reasoners [Lamy, 2017] and to make these available to chaincode at the platform level. This would initially involve creating a private blockchain for the proposed system, unless or until the public blockchain incorporates this enhancement.

Even the full state of the executable BPMN process is impractical to maintain in a public blockchain due to the cost involved. BPM executable engines are typically stateful and tend to save the state of process instances in traditional databases. But, technically, the capability exists [López-Pintado et al., 2017] and can be incorporated into a working solution once an on-chain reasoner is achieved. Practically, this would mean the dotted boundary of the blockchain network seen in the detailed architecture shown in figure 4.14 would move leftwards to include more components. It was not possible to adequately assess other smart contract technologies (Ripple) due to time constraints but this did not inhibit a conceptual study and application. From a requirements engineering point of view, a second level and more detailed requirements analysis is required for most components, with the involvement of specialised personnel like legal domain experts and semantic web practitioners.

Bibliography

[Angelidis et al., 2018] Angelidis, I., Chalkidis, I., Nikolaou, C., Soursos, P., and Koubarakis, M. (2018). Nomothesia: A linked data platform for greek legislation.

[Apache-Jena,] Apache-Jena. *Reasoners and rule engines: Jena inference support*. [`https://jena.apache.org/documentation/inference/`; Online; accessed 29-Dec-2018].

[Atkinson, 2006] Atkinson, C. (2006). Models versus ontologies-what's the difference and where does it matter. *VORTE2006, Regal Kowloon Hotel, Hong Kong*.

[Bashir, 2017] Bashir, I. (2017). *Mastering Blockchain*. Packt Publishing Ltd.

[Berkovich et al., 2009] Berkovich, M., Esch, S., Leimeister, J. M., and Krcmar, H. (2009). Requirements engineering for hybrid products as bundles of hardware, software and service elements-a literature review. In *Wirtschaftsinformatik (1)*, pages 727–736.

[Boer, 2009] Boer, A. (2009). Legal theory, sources of law and the semantic web. In *Proceedings of the 2009 conference on Legal Theory, Sources of Law and the Semantic Web*, pages 1–316. IOS Press.

[Boer et al., 2008] Boer, A., Winkels, R., and Vitali, F. (2008). Metalex xml and the legal knowledge interchange format. In *Computable models of the law*, pages 21–41. Springer.

[Breuker et al., 1997] Breuker, J., Valente, A., Winkels, R., et al. (1997). Legal ontologies: a functional view. In *Procs. of 1st LegOnt Workshop on Legal Ontologies*, pages 23–36.

[Buterin et al., 2014] Buterin, V. et al. (2014). A next-generation smart contract and decentralized application platform. *white paper*.

[Casanovas et al., 2016] Casanovas, P., Palmirani, M., Peroni, S., van Engers, T., and Vitali, F. (2016). Semantic web for the legal domain: the next step. *Semantic Web*, 7(3):213–227.

[Chalkidis et al., 2018] Chalkidis, I., Androutsopoulos, I., and Michos, A. (2018). Obligation and prohibition extraction using hierarchical rnns. *arXiv preprint arXiv:1805.03871*.

[Dentler et al., 2011] Dentler, K., Cornet, R., Ten Teije, A., and De Keizer, N. (2011). Comparison of reasoners for large ontologies in the owl 2 el profile. *Semantic Web*, 2(2):71–87.

[Drozdowicz et al., 2012] Drozdowicz, M., Ganzha, M., Paprzycki, M., Szmeja, P., and Wasielewska, K. (2012). OntoPlay-A Flexible User-Interface for Ontology-based Systems. In *AT*, pages 86–100.

[DuPont, 2017] DuPont, Q. (2017). Experiments in algorithmic governance: A history and ethnography of the dao, a failed decentralized autonomous organization. In *Bitcoin and Beyond*, pages 157–177. Routledge.

[Ethereum,] Ethereum. *Solidity*. [https://solidity.readthedocs.io/en/latest/; Online; accessed 28-Oct-2018].

[Faily, 2011] Faily, S. (2011). *A framework for usable and secure system design*. PhD thesis, University of Oxford.

[Flechais et al., 2007] Flechais, I., Mascolo, C., and Sasse, M. A. (2007). Integrating security and usability into the requirements and design process. *International Journal of Electronic Security and Digital Forensics*, 1(1):12–26.

[Fowler, 1997] Fowler, M. (1997). *Analysis patterns: reusable object models*. Addison-Wesley Professional.

[Gangemi et al., 2017] Gangemi, A., Presutti, V., Reforgiato Recupero, D., Nuzzolese, A. G., Draicchio, F., and Mongiovì, M. (2017). Semantic web machine reading with fred. *Semantic Web*, 8(6):873–893.

[Gartner, 2014] Gartner (2014). Reviews for business process management platforms. https://www.gartner.com/reviews/market/business-process-management-platforms/vendors. [Online; accessed 30-Jan-2019].

[gov.uk,] gov.uk. Tenancy agreements: a guide for landlords (england and wales). https://www.gov.uk/tenancy-agreements-a-guide-for-landlords/tenancy-types. [Online; accessed 28-Dec-2018].

[Grenning, 2011] Grenning, J. W. (2011). *Test Driven Development for Embedded C*. Pragmatic bookshelf.

[Hirai, 2017] Hirai, Y. (2017). Ethereum virtual machine (evm) awesome list. https://github.com/ethereum/wiki/wiki/Ethereum-Virtual-Machine-(EVM)-Awesome-List. [Online; accessed 29-Dec-2018].

[Hoekstra et al., 2007] Hoekstra, R., Breuker, J., Di Bello, M., Boer, A., et al. (2007). The LKIF Core Ontology of Basic Legal Concepts. *LOAIT*, 321:43–63.

[Horrocks, 2010] Horrocks, I. (2010). Scalable ontology-based information systems. In *EDBT*, page 2.

[Idelberger et al., 2016] Idelberger, F., Governatori, G., Riveret, R., and Sartor, G. (2016). Evaluation of logic-based smart contracts for blockchain systems. In *International Symposium on Rules and Rule Markup Languages for the Semantic Web*, pages 167–183. Springer.

[IEEE, 2007] IEEE (2007). Rules and Ontology in Compliance Managemen.

[Kabilan and Johannesson, 2003] Kabilan, V. and Johannesson, P. (2003). Semantic representation of contract knowledge using multi tier ontology. In *Proceedings of the First International Conference on Semantic Web and Databases*, pages 378–397. CEUR-WS. org.

[Kitchenham, 2004] Kitchenham, B. (2004). Procedures for performing systematic reviews. *Keele, UK, Keele University*, 33(2004):1–26.

[Kuba, 2012] Kuba, M. (2012). Owl 2 and swrl tutorial. *Institute of Computer Science, Masaryk University*. [Online; accessed 22-Sep-2018].

[Lamy, 2017] Lamy, J.-B. (2017). Owlready: Ontology-oriented programming in python with automatic classification and high level constructs for biomedical ontologies. *Artificial intelligence in medicine*, 80:11–28.

[legislation.gov.uk,] legislation.gov.uk. legislation.gov.uk - rdf/xml format. [https://www.legislation.gov.uk/developer/formats/rdf; Online; accessed 30-Jan-2019].

[Levy, 2017] Levy, K. E. (2017). Book-smart, not street-smart: blockchain-based smart contracts and the social workings of law. *Engaging Science, Technology, and Society*, 3:1–15.

[Lim and Huebel, 1979] Lim, J. and Huebel, J. (1979). Tempering of accounts and records to disguise snm theft. Technical report, California Univ., Livermore (USA). Lawrence Livermore Lab.

[López-Pintado et al., 2017] López-Pintado, O., García-Bañuelos, L., Dumas, M., and Weber, I. (2017). Caterpillar: A blockchain-based business process management system. In *Proceedings of the BPM Demo Track and BPM Dissertation Award co-located with 15th International Conference on Business Process Modeling (BPM 2017), Barcelona, Spain*.

[McCarty, 1982] McCarty, L. T. (1982). Intelligent legal information systems: Problems and prospects. *Rutgers Computer & Tech. LJ*, 9:265.

[McKendrick, 2017] McKendrick, E. (2017). *Contract law*. Palgrave law masters, twelfth edition.

[Meditskos,] Meditskos, G. Rule-based applications on top of ontologies - architectures and challenges. https://www.iti.gr/iti/files/document/seminars/MeditskosSeminar.pdf. [Online; accessed 29-Dec-2018].

[Merriam, 2017] Merriam, P. (2017). The python ethereum ecosystem. https://medium.com/@pipermerriam/the-python-ethereum-ecosystem-101bd9ba4de7. [Online; accessed 29-Dec-2018].

[Nofer et al., 2017] Nofer, M., Gomber, P., Hinz, O., and Schiereck, D. (2017). Blockchain. *Business & Information Systems Engineering*, 59(3):183–187.

[Noy et al., 2001] Noy, N. F., McGuinness, D. L., et al. (2001). Ontology development 101: A guide to creating your first ontology.

[OMG, 2014] OMG (2014). About the business process model and notation specification version 2.0.2. https://www.omg.org/spec/BPMN. [Online; accessed 30-Jan-2019].

[OpenData.cz,] OpenData.cz. Public contracts ontology. https://github.com/opendatacz/public-contracts-ontology. [Online; accessed 30-Oct-2018].

[Ruster, 2016] Ruster, M. (2016). Large-scale reasoning with owl. *arXiv preprint arXiv:1602.04473*.

[Simpson et al., 2015] Simpson, A., Martin, A., Cremers, C., Flechais, I., Martinovic, I., and Rasmussen, K. (2015). Experiences in developing and delivering a programme of part-time education in software and systems security.

[Slapper and Kelly, 2012] Slapper, G. and Kelly, D. (2012). *Law: The Basics*. Routledge.

[Stonebraker, 2010] Stonebraker, M. (2010). Errors in database systems, eventual consistency, and the cap theorem. *Communications of the ACM, BLOG@ ACM*.

[Sysintellects,] Sysintellects. 9 Stages of Contract Lifecycle Management. `https://www.contractexperience.com/resources/resources-main.html`. [Online; accessed 29-Dec-2018].

[Szabo, 1997] Szabo, N. (1997). Formalizing and securing relationships on public networks. *First Monday*, 2(9).

[TheOpenGroup, 2018] TheOpenGroup (2018). *TOGAF: Core Concepts*. [`https://pubs.opengroup.org/architecture/togaf9-doc/arch/chap02.html`; Online; accessed 01-Jan-2019].

[Verhodubs and Grundspenkis, 2011] Verhodubs, O. and Grundspenkis, J. (2011). Towards the semantic web expert system. *Scientific Journal of Riga Technical University. Computer Sciences*, 44(1):116–123.

[Vessenes, 2016] Vessenes, P. (2016). More ethereum attacks: Race-to-empty is the real deal. *Peter Vessenes*. [`https://vessenes.com/more-ethereum-attacks-race-to-empty-is-the-real-deal/`; Online; accessed 20-Nov-2018].

[web3j,] web3j. *web3j: Web3 Java Ethereum Dapp API*. [`https://github.com/web3j/web3j`; Online; accessed 29-Oct-2018].

A Technical Notes

A.1 Additional Tools Assessed

Tools, frameworks, and software used in the implementation are described in section 5.1. Various other significant tools considered or evaluated are noted below.

Table A.1: Other tools considered but not part of implementation

Tool	Notes
swrlapi-drools-engine	Version 2.0.6-SNAPSHOT
swrlapi	Version 2.0.5
pellet-core	Version 2.2.2 (Has associated dependencies like pellet-el, pellet-rules, pellet-owlapiv3, pellet-explanation)
com.hermit-reasoner:org.semanticweb.hermit	Version 1.3.8.4
drools	Version 5.5.0.Final
Jena	Version 3.0.0
jBPM	Version 7.7.0.Final
Caterpiller BPM	v1.0 from https://github.com/orlenyslp/Caterpillar

A.2 Task and Bug Tracking

Jira was used to manage progress during the project execution. Because of the exploratory nature of the project and number of unknowns, an Agile engineering approach supported by some Agile tools was a good fit. This helped with progress as ideas were quickly added to a backlog to be picked up at later suitable times, tasks could be grouped into stories, and progress notes helped derive content for the work reported in this dissertation.

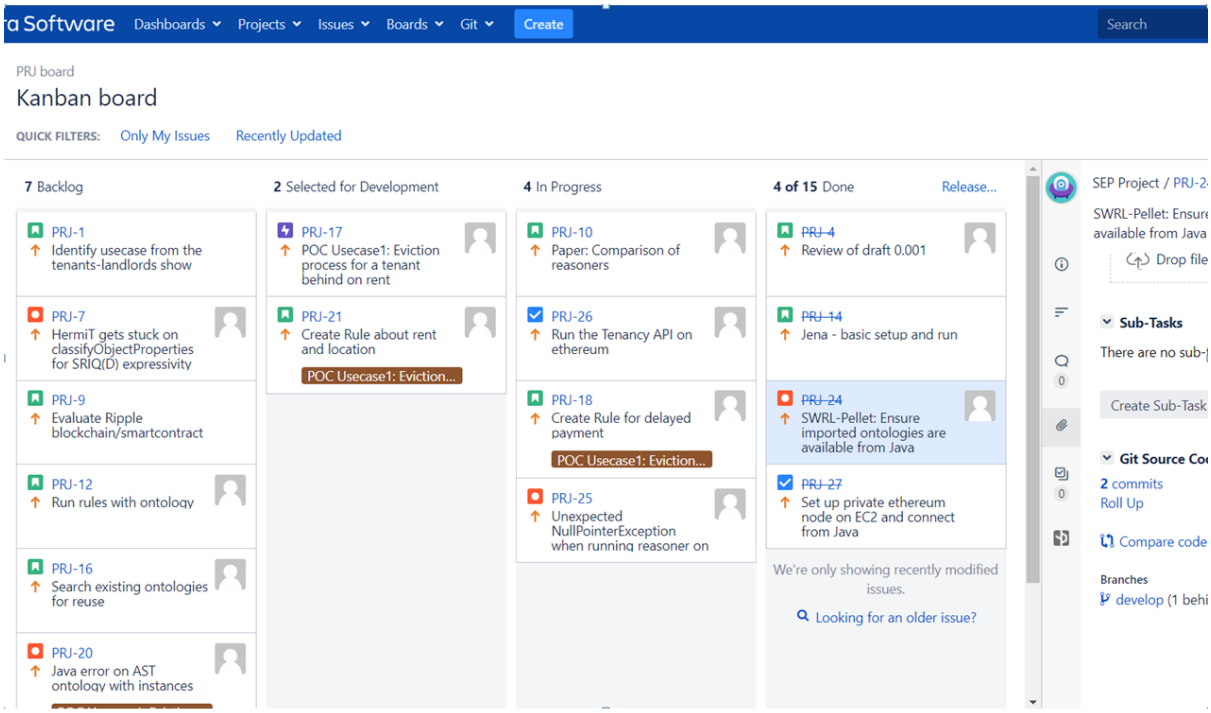


Figure A.1: A view of the Jira board during an early stage of the project

SEP Project / PRJ-7

HermiT gets stuck on classifyObjectProperties for SRIQ(D) expres:

Edit | Comment | Assign | More | Backlog | Selected for Development | Workflow | Admin

Details

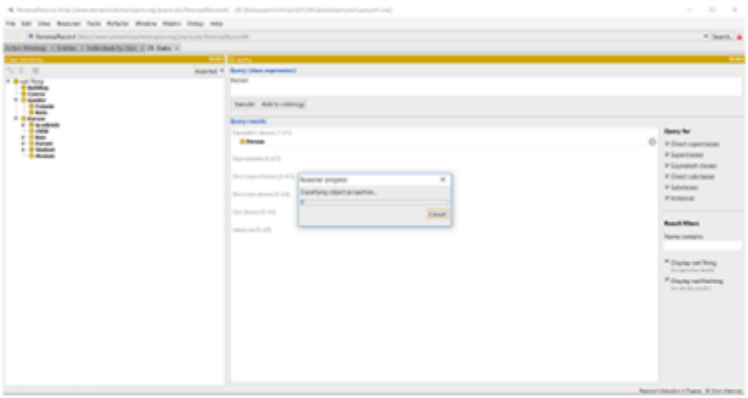
Type:	Bug	Status:	BACKLOG (View Workflow)
Priority:	Medium	Resolution:	Unresolved
Labels:	None		

Description

On the sample1 (family) ontology, HermiT gets stuck on "classifying object properties".

DL Expressivity is: SRIQ(D)

HermiT does work on the pizza ontology whose expressivity is SHOIN



Attachments

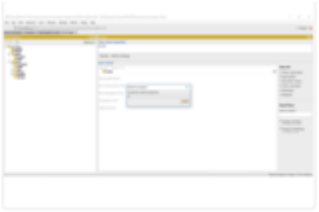


image-2018-09-01-05-03-59-
01/Sep/18 5:04 AM 31 kB

Activity

All | Comments | Work Log | History | Activity | Git Roll Up | Git Commits

JV Singh added a comment - 01/Sep/18 5:43 AM - edited

Observed when running:

```
org.semanticweb.HermiT.Reasoner::dumpHierarchies(new java.io.PrintWriter(System.out), true,
true, true);
```

Figure A.2: A open bug related to the use of Hermit

B Working of Distributed Ledgers and Smart Contracts

At a high level, we note that different lifecycle stages of a contract must be supported by a trustworthy, ordered, and immutable audit trail. Conventional databases with enhanced redundancy and availability would seem to be a natural choice today and are widely used. However, as the ensuing discussion will attempt to explain, it would be pragmatic for modern systems to leverage emerging technologies that are demonstrably able to offer these properties.

Principles and history of ledgers and bookkeeping: Before exploring this area using sophisticated concepts in vogue today, we need to understand the idea of ledgers and then "distributed ledgers" from its basics. A ledger (like an accounting ledger or what is often known as 'book'[1] in a 'book-keeping' practice) is a book of records, typically monetary records, that starts with a certain "value" (such as initial money in a bank account or even zero). Over a given period (such as a financial year or even a day), various transactions either credit to or debit from the base amount. Books are said to "balance", when certain conditions meet - such as, the final value in a bank account matches the sum of all credits and debits on the books. Any spurious credit or debit entry would cause a mismatch and be easily detectable, but, a *combination* of such transactions might obfuscate tampering. We could represent a simplified book of accounts for a business as a set of dated entries that either credit to or debit from the "current balance" (fig. B.1). A real book of accounts would be far more complex, and a balance sheet for even a small business would account for values of physical items or assets, debit their depreciation, taxes and much more.

Balance Brought forward			100
		CR	DR
Transaction number	**Date**		
Tx1	Month1	10	
Tx2	Month1		20
...	...		
TxN	MonthX	100	
Current balance:			190

Figure B.1: A simple book of accounts

Historically, such accounts (and similar records like land registers) were maintained on

[1] In accounting and project management software, it is common to find terms like book of accounts. In popular media, accounting scandals are often described as "clever accountants" fudging or "cooking" the "books"

thick bound paper based books (perhaps clay tablets or sheep skin prior to paper). Each transaction is entered consecutively and dated. Verifying totals requires tedious manual effort and focus. This is a very oversimplified view to help us understand what features and functionality present systems aim to achieve. The bound ledger books with consecutive entries would involve painstaking effort in "balancing" books or auditing, especially if this involved historical records or detecting errors. There could be various motivations for a malicious attempt to tamper with such accounting record. Not recording an entry altogether might have been possible in the past and harder to detect. Making fraudulent entries could be another one with the motive of siphoning off funds. In modern times, the Enron accounting scandal was essentially perpetuated by fraudulent accounting practices and possible tampering of accounts. Such frauds were not undetectable by audits but the detection could be made harder by sophistication of the tampering and collusion. Accounting and audit practices have definitely matured and "double entry bookkeeping" has been used since middle ages to account for money and materials, with sophisticated accounting methods to detect theft or prevent "disguise" of theft of sensitive material like nuclear material [Lim and Huebel, 1979]. The basic principle of record keeping has remained the same in all these use-cases:

> *...to ensure accuracy and integrity of records, the accounts are balanced... so that the fundamental account equations of* assets = liabilities *and* credits = debits *are satisfied. Any imbalance causes a* ***book balance discrepancy****... [Lim and Huebel, 1979]*

Now, for an adversary, one of the tactics might be to tamper with records in a way that can avoid detection by ensuring the malicious tampering does not cause a book balance discrepancy. The

Balance Brought forward				100
		CR	DR	
Transaction number	**Date**			
Tx1	Month1	10		
Tx2.1	*Month1*		*100*	
Tx2.2	*Month1*	*80*		
...	...			
TxN	MonthX	100		
Current balance:				190

Figure B.2: A tampered accounting book without book balance discrepency

simplistic example in figure B.2 may or may not cause actual harm but provides an example of a tampered book of accounts which can be a serious matter. If we were to treat "current balance" as "state" of the ledger, for any given period, we ought to be able to replay the set of credits and debits on an initial "balance brought forward" figure, and arrive at the same figure every time for every date including intermediate dates. In a tampered ledger, this would not be possible for all intermediate dates.

If, however, all individual entries were to be stored alongside a unique cryptographic measurement (a hash), and, each subsequent entry in the ledger was a hash of the previous hash, current entry, and some unique temporal values like timestamp and nonce, then, the the ledger

Balance Brought forward				100	Cryptographic Measurement
		CR	DR		
Transaction number	**Date**				
Tx1	Month1	10			ff1c.....a000
Tx2.1	*Month1*		*100*		*95xd.....314b*
Tx2.2	*Month1*	*80*			....
...	...				....
TxN	MonthX	100			*390c.....fab4*
Current balance:				190	*7f39.....1ac0*

Figure B.3: A simple tamper-evident ledger

would be more tamper evident. Although what figure B.3 depicts can only be implemented realistically as a digital ledger, attempts to secure the integrity and even confidentiality of ledgers can be found in history. A patent (US954791A) from year 1909, for instance, describes a device to physically secure the contents of a ledger to avoid inspection and tampering.

Distributed Ledgers: In physical ledgers, various methods could be employed in a malicious attempt to tamper with old entries - erasing previous writing, or replacing a whole sheet of paper with one that includes tampered entries. Modern digital records can in some ways make tampering easier as, in practice, simple digital records without adequate safeguards don't have the limitations or natural tamper-evidence of physical ink and paper.
As a means to increase ledger integrity, if, instead of one, multiple identical copies of sensitive ledgers were maintained, this could make tampering harder, though not impossible if the book-keepers happen to collude with one another and with the adversary. Yet another level of complexity could be introduced by geographically separating the ledgers and somehow making it impossible for them to communicate with one another except with a limited set of information: a) contents of the new transaction (CR/DB x from current state) to be applied to the ledger and b) the new value of the ledger. A transaction could be said to get *confirmed* if all these hypothetical (and somewhat improbable) bookkeepers could agree on the new value via a previously agreed protocol (consensus?) for deciding what the current value total of the ledger is? Even if some of the bookkeepers become corrupt or make a mistake, the protocol could be defined to tolerate a certain percentage of tolerance for corrupt book-keepers.
What if the ledger and its contents were all public - surely, any tampering could be detectable if the precise state of the ledger was part of public memory and therefore, any tampering would immediately be detected. Such a system would work on various assumptions: a) that all members of the public or a significant subset of them would be *interested* in doing this or "someone" might incentivise them to do so b) the general public or those who participate in this system are *capable* of doing this accurately and precisely - as after all, we could be dealing with financial accounts, and even other complex transactions or transaction chains like in land records/transfers, corporate share issuance/ownership/transfers (potentially help prevent cases like sale excess shares of Bear Stearns [Nofer et al., 2017]) to maintaining inventory of nuclear material [Lim and Huebel, 1979].

If not 100% capable, at least there should exist some way of determining what most (or most credible) record keepers have to say about the state of the current balance after the application of a transaction c) The records can be made public or at least, the public version of the records can be provably tamper-evident without disclosing content that is intended to be private (one would assume the ledger for nuclear material inventory is highly confidential). We note that there isn't a 100% fool-proof and practical means of ensuring ledger integrity.

This situation is not made easier with the advent of digital record-keeping - perhaps the complexity and magnitude of the problem is compounded. Digital records could on the one hand introduce additional checks and balances to improve *integrity* and *confidentiality* - audit entries, timestamps, authenticated access and so forth. Redundancy and disaster recovery techniques combined with data replication are standard and mature techniques used to improve the availability of sensitive data such as important ledgers.

But digital records do offer certain advantages: Many auditing mechanisms can be made easier or fully automated. Computer programs are exceptionally good and fast at it, and once developed, they can run over and over again - facilitating both record-keeping and compliance. These are sophisticated functionally oriented business systems. On the technical side, measurements (hashes) of system backups have been a tested means of ensuring system integrity used by systems administrators or database administrators.

All these means still do not offer sufficient (mathematically provable?) fool-proofing against malicious tampering. Two or more redundant copies of a key database are still vulnerable if their administrators (book-keepers) collude to change details that favour an adversary. Timestamps, audit records, everything could be altered in a sophisticated attack. Numerous techniques are employed to enhance properties like availability, consistency, integrity, transaction speed/throughput/response time of data persistence systems (databases), many of which rely on distribution of the system across multiple network nodes. These mechanisms involve tradeoffs between consistency, availability, and partition tolerance, (the CAP theorem [Stonebraker, 2010]), leading some Big Data systems to describe themselves as achieving *"eventual consistency"*.

Back to our simple account book, we note that a slight alteration in the sequence or number of entries led to an undetectable change in the ledger (in this simplistic example, the total figure remained unchanged but the spurious transactions could have caused damage - they could represent an illegal loan or asset transaction for instance for which profits were siphoned away). This isn't an impossible problem to detect via an honest audit (or even prevent when all the bookkeepers are honest) - but we work with the assumptions that the system needs to continue working with reliable results when some of these actors (nodes), and it could be any of them, turns corrupt. This problem is formally described as the Byzantine General's problem. The pioneering Blockchain based system, the Bitcoin distributed ledger in 2009 was the first to apply a *"Practical Byzantine Fault Tolerance"* algorithm called *"Proof of work"* [Bashir, 2017] as a mechanism to achieve consensus. Transaction throughput and scalability has always been difficult to achieve with different variants of distributed ledger consensus algorithms achieving different results in terms of throughput but retaining the essential design goal of being a decentralised linked list of blocks of transactions. Caveat: as the DAO[2] hard fork in 2016 shows, the decentralisation works until it needs to be overridden, but, technically, the hard fork could be seen as a case of *all* nodes colluding.

[2]Decentralised Autonomous Organisation