TECHNISCHE UNIVERSITÄT DRESDEN

Faculty of Computer Science

Institute of Software and Multimedia Technology
Chair of Software Technology

Master Thesis

Smart Contract Development with JASON BDI Agents

Mostakim Mullick

Born on: 20th February 1998 in Hooghly, India
Matriculation number: 4993151

5th January 2023

First referee
Prof. Dr. rer. nat. Uwe Aßmann

Second referee
Dr.-Ing. Sebastian Götz

Supervisor
M. Sc. Orçun Oruç

TECHNISCHE UNIVERSITÄT DRESDEN

Faculty of Computer Science Chair of Software Technology

Task for the preparation of a Master Thesis

Course:	Distributed System Engineering
Name:	Mostakim Mullick
Matriculation number:	4993151
Matriculation year:	2020
Title:	Smart Contract Development with JASON BDI Agents

Objectives of work

Agent-oriented programming has been developed over the few decades in order to comprehend the relationship between dynamic environments and software applications. Belief-Desire-Intention agents can implement a plan execution library that consists of objectives and goals. Beliefs are the environmental status and these agents can update the status of the environment. For instance, weather degree is a dynamic environment variable that can be updated by agents to use with rule-based conditions such as desires and intentions. In order to realize agent-based abstraction, one can use the Jason Belief-Desire- Intention (BDI) agent with AgentSpeak. Jason Agent Framework is an interpreter of the logic- based BDI language called AgentSpeak so that agents can be easily integrated into a general-purpose language.

A smart contract has preconditions and postconditions to support a handshake mechanism between participants in a blockchain network. When a transaction has been commenced in a smart contract, a transaction can update the state of the ledger. However, transactions or meta-transactions (only data) can be initiated by an externally owned account or client application. An agent can manage autonomous entities according to its internal planning library. Each plan can have multiple subgoals and subgoals can deploy existing smart contracts according to preconditions and postconditions in a case study.

A case study can be realized as follows: An adaptive supply chain can adapt to the environmental changes between retailer, distributor, and producer agents. Each agent has a main goal and sub-goals that are relevant to the particular main goal. Storage of retailers, distributors, producer agents can be registered into a programmable ledger. Storage capacity of retailer-distributor agents and production capacity of producer agents should be tracked by the internal planning library of agents. An agent can decide raw materials and goods can be sent according to parameters such as production ready (boolean), capacity level of storage, and production level with preconditions and postconditions. Each smart contract should be managed by agents with regards to this simulation.

TECHNISCHE UNIVERSITÄT DRESDEN

Faculty of Computer Science Chair of Software Technology

This work should investigate and identify the feasibility of decentralized smart contract execution with agents. To this end, a literature review should be provided by the thesis author regarding agent-oriented programming, contract-oriented programming, and smart contract programming. After the literature analysis, a proof-of-concept application that is relevant to a particular case study should be realized. Finally, evaluations need to be defined with corresponding criteria and the student should discuss results to come up with conclusions.

Focus of work

- Realizing the literature review of BDI-based agent-oriented programming, decentralized smart contract execution with autonomous agents.
- Carrying out a demonstration of the feasibility of a case study.
- Comparing with other BDI agent frameworks with regards to drawbacks and advantages.

Task Description

The following items are possible tasks:

1. Analyzing the feasibility of the agent-oriented planning library with decentralized smart contracts.
2. Researching the feasibility of the language abstraction in on-chain programming with language features of an agent such as belief-desire-intention.
3. Define types of agents, which are required in a particular case study and implement the case study through the Jason-Agentspeak framework with Solidity language.
4. In order to ensure software functionality, the student should implement behavioral and unit test cases.

A demo and presentation will be organized.

First referee:	Prof. Dr. rer. nat. Uwe Aßmann
Second referee:	Dr.-Ing. Sebastian Götz
Supervisor:	M. Sc. Orçun Oruç
Issued on:	17th October 2022
Due date for submission:	20th March 2023

Prof. Dr. rer. nat. Uwe Aßmann
Supervising professor

Statement of authorship

I hereby certify that I have authored this document entitled *Smart Contract Development with JASON BDI Agents* independently and without undue assistance from third parties. No other than the resources and references indicated in this document have been used. I have marked both literal and accordingly adopted quotations as such. There were no additional persons involved in the intellectual preparation of the present document. I am aware that violations of this declaration may lead to subsequent withdrawal of the academic degree.

Dresden, 5th January 2023

Mostakim Mullick

Abstract

Over the last few decades, agent-oriented programming has been constituted in an attempt to better understand how software applications interact with dynamic circumstances. Agents with the Belief-Desire-Intention (BDI) framework can create a plan execution library with goals and objectives. Beliefs represent the state of the environment, and these agents can update that state. One may utilize the JASON AgentSpeak(L), ASTRA or any preferred AOP language to implement agent-based abstraction.

Utilising of BCT has skyrocketed in recent years too, and various blockchains customized to specific use cases have emerged. One of the most essential blockchain uses is smart contracts. Most present smart contract systems presume that when contracts are executed over a network of decentralized nodes, the majority's decision can be trusted. However, we have seen that people associated with a smart contract may intentionally take steps to manipulate contract execution in order to improve their own gains. To solve this issue, we suggest an agent model as the underlying mechanism for contract execution over a network of decentralized nodes and a public ledger, and we analyze the prospect of prohibiting users from influencing smart contract execution.

Numerous research and application sectors are being impacted by the agent programming as well as decentralized blockchain technology and idea, and as a result, many see this as an opportunity to find new solutions to old issues or reap innovative advantages. Several writers in the agent community are proposing their own mix of agent-oriented technology with blockchain to address both old and new difficulties. This thesis paper aims to define the prospects, factors to consider, and analyzing information about the content for combining agents with decentralized smart contracts while adapting it in a supply chain.

Contents

Symbols and Acronyms

BDI	Belief-Desire-Intention
AOP	Agent Oriented Programming
OOP	Object Oriented Programming
MAS	Multi-Agent System
BCT	Block Chain Technology
NFT	Non-Fungible Tokens
ASTRA	AgentSpeak(TR) Agents
DeFi	Decentralized Finance
PoA	proof of authority
PoW	proof of work
PoS	proof of stake
BOID	Beliefs Obligations Intentions Desires
PRS	Procedural Reasoning System
dMARS	Distributed Multi-Agent Reasoning System
SSI	Self-sovereign identification
DAO	Decentralized Autonomous Organizations
AI	artificial intelligence
API	Application Programming Interface
Dapp	Decentralized Application
EVM	Ethereum Virtual Machine
NPM	Node Package Manager
ERC	Ethereum request for comment
UI	User Interface
CLI	Command Line Interface
JSON	JavaScript Object Notation
RPC	Remote Procedure Calls
UPC	Universal Product Code
SKU	Stock Keeping Unit
npm	Node Package Manager
JADE	Java Agent DEvelopment Framework
IDE	Integrated Development Environment
pip	preferred installer program
JVM	Java Virtual Machine
NatSpec	Ethereum Natural Language Specification Format
ACL	Agent Communication Language
FIPA	Foundation for Intelligent Physical Agents

1 Introduction

The term *"agent"* is commonly used these days. This applies to both artificial intelligence (AI) and fields outside of it, such as databases and automated manufacturing. Despite growing in popularity, the phrase has been employed in so many different contexts that it has lost all meaning when not connected to a certain understanding of agenthood. The terminology "agent" is most frequently used in the context of AI to describe an object that operates constantly and autonomously in a context that also contains other processes and agents. Perhaps this is the single characteristic that all practitioners of the word in AI agree upon. Although the emphasis is on the word "autonomy," the definition of autonomy is vague, but it is generally understood to suggest that the behaviors of the agents do not need ongoing human direction or interference, and presumptions are frequently made about the environment of the agents, such as that it is mostly unpredictable.

Agents are frequently assumed to be robotic agents, in which case additional problems including sensory input, motor control, and time constraints are stated. Finally, agents are frequently perceived as "high-level." Although this meaning is somewhat nebulous, many people use some variation of it to separate agents from other pieces of hardware or software. The "high level" is exhibited by symbolic representation and/or cognitive-like skills. Agents may be "informable," have symbolic planning in addition to stimulus-response rules, and even be able to speak. This meaning is not universally assumed in AI, so clearly, the concept of agenthood in AI is far from clear. As a result, We should be explicit about what we mean when we use the term "agent," which is as follows: An entity whose state is seen to be composed of mental elements including beliefs, capabilities, decisions, and commitments is referred to as an agent. These elements are precisely specified and roughly correspond to their equivalents from common sense. Therefore, according to this perspective, an agent only exists in the programmer's head. Any hardware or software component only qualifies as an agent if its analysis and management have been done in this way.

In latest years, there has been a lot of interest in the research of computational agents that can behave rationally. The architecture of systems required to conduct high-level management and control activities in complex, dynamic situations is becoming increasingly important in the commercial world. Air traffic control, telecommunications networks,

commercial operations, spacecraft, and medical services are examples of such systems. Experience with using traditional software methodologies to design such systems has revealed that they are extremely complicated and costly to build, validate, and sustain. Agent-oriented systems, which are based on a fundamentally different perspective of computational entities, provide opportunities for a qualitative shift in this stance.

One such architecture views the system as a rational agent with particular mental attitudes of BDI, reflecting, respectively, the informational, motivating, and deliberative states of the agent. When deliberation is subject to resource constraints, these mental attitudes govern the system's behavior and are crucial for obtaining adequate or optimal performance.

BDI agents can be considered for both a formulation of a theory and a practical design standpoint. *AgentSpeak(L)* is a generalization of one of the implemented BDI systems that allows agent programs to be developed and processed in the same way as a horn-clause rationale programs. It is a language of programming based on a limited first-order language with events and actions. The programs created in AgentSpeak(L) govern the agent's behavior (i.e., its interaction with the environment). The agent's beliefs, desires, and intentions are not explicitly stated as modal formulae. Instead, as designers, one may attribute these ideas to AgentSpeak(L) agents. The agent's present belief state may be considered a model of itself, its environment, and other agents; the states that the agent seeks to bring about based on external or internal stimuli can be viewed as desires, and the adoption of programs to meet such stimuli can be viewed as intents.

It is therefore easy to design an agent capable of controlling its surroundings on its own, but this becomes troublesome when the environment is shared by more than one agent. AgentSpeak(L) is extended by languages such as JASON and ASTRA. They could be used to implement that language's operational semantics and provide a foundation for the development of Multi-Agent System (MAS) with different user-customizable features.

The implementation of these agents in specialized fields would be helpful when discussing the agents' ability to communicate using AgentSpeak(L). One of the most significant areas will be supply chain management, where we may assign roles to various agents such as manufacturers, wholesalers, and retailers. An adaptable supply chain allows retailer, distributor, and manufacturer agents to adjust to environmental changes. Each agent has a core objective as well as related secondary objectives. It seems logical that whenever agents interact in the supply chain, there will be a transaction, and that transaction should be recorded. The transaction can be stored in any database by agents, but will it be optimal? A programmable ledger can be used to record the movement and ownership of items between each agent. So rather than keeping these interactions in the form of transactions in databases or spreadsheets, it will be perfect for storing them in smart contracts, which can be inspected for tracking or future references.

The concept of *smart contracts*, or computer protocols intended to automatically facilitate, authenticate, and implement the negotiation and application of digital contracts without the

need for centralized authorities, has been revived in recent years has resulted of the rapid development of cryptocurrencies and the BCT that underpins them. Smart contracts have been incorporated into popular blockchain-based development platforms like Ethereum and Hyperledger. They have a wide range of potential application areas in the globalised era and intelligent sectors of the economy, including financial sector, management, healthcare, and Internet of Things, among many others.

The combination of the agent and blockchain domains appears to be effective. In order to study the operation, compatibility, and behavior pattern of smart contracts with agents, some work has been done in the area of integrating agents with smart contracts.

In this thesis, a ***Smart contract development with JASON BDI Agents*** is presented. An environment that will show off several agents, allowing agents throughout the supply chain—namely, manufacturers, retailers, and wholesalers—to communicate with one another utilizing a primary agent. This ecosystem will also make it possible to maintain relationships between agents, store transactions using smart contracts and offer a hash for each transaction that may be used to examine further information.

1.1 Motivation

The blockchain is having an increasing influence on a variety of research and application fields, from distributed computing and storage to supply chain management and healthcare accountability. In fact, there are many distinct use cases where the promise of supplying a secure and fault-tolerant ledger that mutually untrusted parties may use to monitor computations and interactions completely dispersed and decentralized without the need for a central authority is intriguing.

The agent community is no different; it began expressing interest in the opportunities presented by the blockchain to either solve long-standing problems (such as trust management in open systems, accountability of actions for liability, among others), or to take advantage of expected benefits to endow a system with new properties (such as novel infrastructures for MAS, trustworthy coordination).

Smart contracts play a critical role in enabling the blockchain to function as a general-purpose distributed computing engine while retaining its core properties of security, trust, and fault-tolerance, and they actually extend their reach to cover computations other than data storage and management.

The convergence of the agent and blockchain worlds appears to be promising: Agents are distributed autonomous entities whose interactions should be managed and mediated in trustworthy ways; blockchains and smart contracts, on the other hand, are trust-sensitive mechanisms for mediating and managing interactions.

1.2 Our contributions

The contribution of the paper are as follows:

- We demonstrate and analyze the aspects of MAS-BCT created by integrating agents with smart contracts, as well as the implications for agent-oriented practice and blockchain. The primary goal was to utilize AgentSpeak(L) to program agents and store their interactions using smart contracts.

- The fundamental concept was to construct MAS using JASON, an extension of AgentSpeak(L), and smart contracts using Solidity and integrating them together.

- Our research enabled the usage of JASON with both the Python interpreter and Java. As an alternative to JASON, we also tested using ASTRA. Because Java was the prior implementation language, ASTRA's type system is based on it. By using a comparable type system, translating between ASTRA and Java is made easier and more clear. Vyper was also taken under consideration for Smart contracts while switching from Java to Python, but was later abandoned owing to some of its drawbacks.

In order to integrate them, we conducted extensive study while doing that. When choosing a programming language to integrate agents written in agentSpeak(L) with smart Contracts written in Solidity, we run into a myriad of challenges.

1.3 State of the Art

Most methodologies to bridging MAS and BCT are of placing a MAS and a blockchain side by side, allowing agents to utilize blockchain services opportunistically as needed. We contend that these techniques do not fully integrate agents and blockchain on a technical level, because the MAS just exploits the blockchain like any other software library. In fact, the agents are viewed essentially like any other off-chain entity, such as blockchain clients, whose contact with the blockchain is restricted to making transactions and deploying/invoking smart contracts—just like any other service.

Efforts striving to put elements of agent-oriented models and technologies directly into the blockchain. For instance, adopting agent programming languages for implementing smart contracts would be an integration effort belonging to this category, as well as empowering smart contracts' computational model (usually, object-oriented) with features defining agent-orientation. Another approach would comprise integration efforts attempting to introduce concepts and techniques from the BCT domain into agents. An agent-programming platform, for example, can be offered, in which agents are written in a smart contract language and their interactions are enabled and governed by a BCT's transactions mechanism.

The existing research focuses on both computational and interactional aspects. Many strategies focus on the computational aspect, aiming to use the blockchain to store data

in a safe and traceable manner in order to hold agents who manipulate it accountable for their actions; other strategies concentrate on the interaction aspect, making use of the blockchain transaction mechanism, frequently in conjunction with smart contracts, to validate and control interactions among agents. However, we stress that not all aspects of the interaction dimension have been taken into account. Existing approaches always assume that interactions will take place between off-chain entities, whether they contain agents or not. However, this leaves open the possibility of using agent-oriented abstractions to mediate on-chain entity interactions, such as those involving smart contract reciprocal invocations or node communication protocols.

In this thesis, we will primarily concentrate on agent-oriented models and technologies that are directly integrated into the blockchain.

1.4 Evaluation Results

Our evaluation was conducted using three separate criteria. (1) Smart Contract Functionality (2) Choosing AOP Language (3) Outcomes of combining smart contracts with Agents. Experimental findings from these many angles are systematically examined.

Smart Contract Functionality

We examined how well smart contracts run on several networks, including the local network, Infura utilizing Rinkeby, and Alfajores, as well as how long it takes to construct each contract. Local networks consistently had the lowest delay time across all the networks after testing the prototype for smart contracts, thus we utilized this network to test our subsequent scenarios involving agents. Since CELO is simpler to obtain than RinkenyETH, testing on Alfajores test network was likewise simpler than on Rinkeby test network. In none of our test scenarios did we use the mainnet.

Along with that, we also stated reason and provided information on why Solidity is preferred over Vyper. Vyper, on the other hand, is built on Python. Solidity is similar to JavaScript, however Vyper lacks several functions that Solidity has, helping to make Solidity more efficient.

Choosing AOP Language

In order to understand how they operate and determine whether they are compatible with other languages to be used to include them into blockbhain, we developed the `.asl` files using JASON and ASTRA. We utilized Java and Python based interpreters to create MAS utilizing JASON AgentSpeak(L), and we also tested ASTRA to learn more about the differences between the languages and see if it works well with Java and makes use of all Java packages like `org.web3j`.

Our evaluation revealed that, despite the fact that the code in the `.asl` files for JASON's Java and Python absed interpreters is identical, the two nevertheless use distinct instructions to communicate. The key difference between the two is that the Java based interpreter requires a `mas2j` file to execute MAS, but the Python based interpreter just requires a python script that establishes an environment for multi agents to communicate with each other.

Although ASTRA-written agents are somewhat unique since they are more likely to resemble Java-style syntax. There is no need to create separate files for the interaction between agents because all the agents may be written in a single file with the `.astra` extension.

Outcomes of smart contracts with Agents

The principal goal of this thesis was to operate the agents in a MAS while adapting a supply chain and collaborating BCT to it in the form of smart contracts. We experimented with combining several AgentSpeak languages with web3 libraries for the assessment. Only `web3.py` and a Python-based interpreter for JASON were found to be effective for the implementation. With other web3 libraries, such as the `web3j` and `web3.js` libraries, we had problems with jar files and missing packages while using AOP, since ASTRA and java-based interpreter of JASON can't work with the core libraries required to run smart contracts.

In order to make it easier to grasp, the evaluation of the thesis is outlined in more detail later in the chapter.

1.5 Outline of the thesis

- The research and associated work that has been done in the area of our study are covered in Chapter 2.
- The technical information needed to comprehend the ideas offered in this thesis is provided in Chapter 3.
- The system's high-level design, architecture, technical details and several sample use cases created for this work are covered are covered in Chapter 4 of this thesis.
- Chapter 5 summarizes the system's evaluation findings.
- The design enhancements for upcoming and additional study subjects that result from this effort are covered in Chapter 6.

2 Related Work

The pertinent research in the field of agent programming using smart contracts will be covered in this part. The associated papers' subjects were arranged to address the following inquiries:

- How to better comprehend agents as well as AOP and distinguish it from Object Oriented Programming (OOP). How to construct MAS using JASON and other domain-specific AOP languages, and how to use AOP languages to create agents and have them interact in supply chain management.

- After understanding agents, how to include BCT into them, and how to use AOP and BCT to record agent interactions in blockchain utilizing smart contracts and stashing transactions on a blockchain.

2.1 Assimilating Agents and AOP

To address our first question about learning about agents, AOP, and other concepts to explore with our thesis, the articles listed below assisted in gaining a comprehensive view on all of the topics and understanding them better.

Exploring AOP from an OOP perspective

Researchers in agent-oriented programming have created a number of agent programming languages that successfully connect theory and practice. Unfortunately, despite these languages' popularity inside their own communities, the larger community of software engineers has not found them to be as intriguing. The need to bridge the cognitive gap that exists between the notions behind standard languages and those underpinning AOP is one of the key issues facing AOP language developers.

In this paper [CRL15], a conceptual mapping between OOP and the AgentSpeak(L) family of AOP languages was made in an effort to create such a linkage. This mapping examines

how AgentSpeak(L) notions connect to OOP principles and the concurrent programming concept of threads. After that, they used the study of this mapping to inform the creation of the ASTRA programming language.

BDI Agent Programming in AgentSpeak Using JASON

This study [BH06] is premised on the instruction that was offered as a part of CLIMA-instructional VI's program. The tutorial's goal was to provide a general overview of the functionality offered by JASON, a MAS development platform built around an interpreter for an enhanced version of AgentSpeak. The most well-known and extensively researched architecture for cognitive agents is the BDI architecture, and AgentSpeak is a sophisticated, logic-based programming language that was inspired by the BDI design.

The paper also discusses how agent-based technology has become increasingly popular for a number of reasons, including how well-suited it has proven to be for the invention of a wide range of applications, such as for air traffic control, autonomous spacecraft control, healthcare, and industrial systems control, to mention a few. These are undoubtedly application areas where reliable systems are needed. The fact that formal verification techniques tailored specifically for MAS are also an area that is luring much research attention and is likely to have a major impact on the uptake of agent technology is one of the benefits of the approach to programming MAS that results from the research reviewed in this paper.

Domain specific agent-oriented programming language based on the Xtext framework

One of the most reliable methods for creating distributed systems is the agent technology. A runtime environment that allows the execution of software agents is presented by the multi-agent middleware XJAF. The researchers suggest a domain-specific agent language called ALAS, whose major function is to facilitate the construction and execution of agents across heterogeneous platforms, in order to address the issue of incompatibility. According to the demands and needs of the agents, a metamodel and grammar of the ALAS language have been developed to describe the language's structure. The development of the compiler and the creation of Java executable code that can be run in XJAF are both covered in this paper [Sre+15].

Agent-Oriented Supply-Chain Management

In order to build an agent-oriented software architecture for controlling the supply chain at the tactical and operational levels, this paper [FBT01] examines problems and offers solutions. The method is based on the employment of an agent building shell, which offers assured, reusable, and generic components. It sees the supply chain as being made up of a group of intelligent software agents, each of which is in charge of one or more supply chain activities.

These agents collaborate with one another to plan and carry out their tasks and provide services for communicative-act-based communication, conversational coordination, role-based organization modeling, and other things. They attempted to demonstrate two nontrivial agent-based supply-chain designs using these elements that might allow intricate cooperative work and the control of disruption brought on by stochastic occurrences in the supply chain.

The objective of developing models and methods that allow MAS to do coordinated work in practical applications has been advanced by the research in a number of different ways. These strategies are carried out by the agents, which causes several organized dialogues to occur amongst the agents. These concepts have been supported by the development of a useful, application-neutral coordination language that offers tools for describing coordination-enhanced plans as well as the interpreter supporting their execution. The researcher of the study has tested the coordination language and the shell on a number of issues, including supply chain coordination initiatives carried out in collaboration with industry. Despite the fact that the number of solutions they built and the number of users of our system are both limited, the evidence they have so far shows that their methodology is promising in terms of the naturalness of the coordination model, the effectiveness of the representation and power, and the usability of the provided programming tools.

2.2 Linking Blockchain with Agents

The principal objective was to include BCT into MAS, which is a very current and innovative subject that may be applied in a futuristic supply chain. Some researchers have already worked on it, which has given us some ideas on how to construct smart contracts with JASON BDI agents. The papers listed below are aligned to our work and can be read to have a stronger insight.

From Agents to Blockchain: Stairway to Integration

The researchers who conducted this study attempted to throw some insight on the most recent integration efforts, mostly from a "agent-vs.-blockchain" perspective. They stressed the possibility of an alternate strategy, which they referred to as "agent-to-blockchain." Finally, they described the "agent-to-blockchain" approach along a pathway upgrading smart contracts towards complete agency, in both dimensions, after acknowledging the presence of two integration dimensions, a computational and an interactional one.

In this article [Cia+20], the researchers provide a roadmap and highlight the challenges that still need to be resolved in order to understand which are the opportunities, the dimensions to take into account, and the different ways available for combining agents and blockchain. They then discussed the case of Tenderfone [Maf20], a custom blockchain that offers proactive smart contracts as the initial step along the road-map, equipping smart contracts with control flow encapsulation, re-activeness to time, and asynchronous communication means, as both validation of their road-map and grounds for future development.

Decentralized Execution of Smart Contracts: Agent Model Perspective and Its Implications

After close scrutiny, the authors of the paper [Che+17] concluded that users who are connected with a smart contract may deliberately try to influence the contract's execution in order to improve their own advantages. In order to address this issue and discuss the possibility of preventing users from manipulating smart contract execution by using game theory and agent based analysis, the authors of this paper propose an agent model as the underlying mechanism for contract execution over a network of decentralized nodes and public ledger.

To simulate the execution of smart contracts over a decentralized network of nodes and participants utilizing blockchain and public ledger, the authors of this study developed an agent-based framework. In contrast to the widely held belief that the results of smart contract execution can be trusted, the agent-based model of smart contract execution makes the assumption that nodes may have an incentive to influence or lie about the results of execution of the contract in exchange for personal benefits or financial gains, even if they are not directly involved in a contract. It had been noted that users who are directly or indirectly involved in a smart contract may behave intelligently to influence the execution outcome of the smart contract. According to the agent-based approach, it might be possible to stop users from cheating when it comes to contract execution or lying about the outcome.

It has also been demonstrated that it is realistic to prohibit users from lying about outcomes or manipulating contract execution results if penalties are applied during contract execution and the assumption is made that users are not totally confident in the rationality of other participants. Furthermore, it had been thought that studying irrationality will help us understand how users behave in a decentralized crypto-currency or smart contract system. An important outstanding challenge is the systematic study of irrationality in relation to the execution and consensus of smart contracts. If it is feasible to employ other mechanisms, such than a monetary penalty, to discourage users from lying about contract outcomes when it benefits them, that would be an intriguing open challenge raised in the paper.

MAS and Blockchain: Results from a Systematic Literature Review

The creation of intelligent distributed systems that manage sensitive data makes extensive use of the MAS technology. The usage of BCT for MAS is encouraged by current trends in order to promote accountability and trustworthy connections. The researchers explained that as most of these techniques have just recently begun to investigate the subject, it is important to build a research road map and identify any relevant scientific or technological hurdles.

This paper[Cal+18] includes a thorough literature evaluation of trials using MAS and BCT as conciliatory remedies as the first required step toward achieving this aim. The authors examined the reasons, presumptions, prerequisites, characteristics, and limits offered in the current state of the art in an effort to give a thorough review of their application fields.

They also lay out their vision for how MAS and BCT may be coupled in various application situations while noting upcoming hurdles.

3 Background

This section discusses the background information needed to comprehend the subject and the technologies that are essential to this research. Starting with AgentSpeak(L), which is one of the key use cases for this thesis, we will go through a variety of subjects. Then we switch to Smart contracts built on the BCT protocol. The core of this thesis is the creation of supply chain agents and the initiation of their interaction in BCT smart contracts, which are thoroughly detailed in this chapter. We go into great depth on the various implementations that enable us to operate Agents with smart contracts. We would next discuss the planning and execution of the key points of this thesis.

3.1 Agent

What is an agent?

An *agent* is a reactive system with some autonomy in the sense that if a task is assigned to it, the system determines the optimal way to complete the task. These systems are referred to as "agents" because they are regarded as active, purposeful producers of actions: they are sent out into their environment to achieve objectives, and actively pursue these objectives, figuring out how tasks are to be done for themselves rather than being told in low-level detail how to do so. If they are robotic agents, they may be assigned tasks like as organizing a trip for us, buying tickets to booking hotels, bidding on our behalf in an online auction, and many other tasks that we can conceive of delegating.

Characteristics of Agents

We define agents as systems that exist in a certain context. This means that agents can sense their surroundings (through sensors) and have a repertoire of possible actions to do (via effectors or actuators) in order to affect their surroundings. The essential question for the agent is how to transition from sensor input to action output: how to decide what to do depending on sensor data. An agent's environment can be physical (in the case of robots

inhabiting the physical world) or software-based (in the case of a software agent inhabiting a computer operating system or network).

Decisions regarding what action to do are turned into real actions by some method external to the agent; often, this is accomplished through some type of Application Programming Interface (API). In practically all realistic applications, agents have very limited influence over their surroundings. As a result, while individuals may take acts that alter their surroundings, they cannot entirely control it. This is frequently due to the presence of other agents in the environment who have influence over their portion of the environment. Aside from being located in an environment, the following characteristics are expected of a rational agent:

- Autonomy

  At its most basic, autonomy implies being able to work freely in order to attain the goals assigned to an agent. Thus, an autonomous agent, at the very least, makes independent judgments about how to attain its designated goals; its decisions (and thus actions) are under its own control and are not influenced by others.

- Proactiveness

  Being proactive entails being able to engage in goal-directed behavior. If an agent has been assigned a specific goal, we expect the agent to make every effort to fulfill that goal. Proactivity eliminates completely passive agents who never strive to accomplish anything. As a result, we don't normally conceive of an object as an agent in the Java sense: such an object is effectively inert until somebody runs a method on it, i.e. instructs it what to do.

- Reactivity

  It is not difficult to design a system that merely responds to external stimuli in a reflexive manner; such a system can be constructed as a lookup table, which just translates environment states directly to actions. Similarly, creating a fully goal-driven system is not difficult. (After all, typical computer programs are ultimately just chunks of code designed to fulfill certain goals.) However, putting in place a system that achieves an appropriate mix of goal-directed and reactive behavior is difficult. This is one of the primary design goals of AgentSpeak.

- Social ability

  In this context, social ability refers to an agent's ability to collaborate and coordinate actions with other agents in order to achieve goals. To realize this type of social ability, it is necessary to create agents that can interact not just in terms of exchanging bytes or calling procedures on one another, but also at the knowledge level. That is, agents should be able to communicate with one another about their opinions, aims, and plans.

3.1.1 MAS

'Single agent systems' are uncommon in practice. The more usual scenario is for agents to coexist in an environment with other agents, resulting in a multi-agent system. Each agent has the unique capacity to control a portion of its surroundings, but more generically, and more problematically, the domains of influence in the environment may overlap, implying that the environment is jointly controlled by more than on agent. This complicates life for our agents because, in order to accomplish the desired outcome in the environment, our agent must consider how the other agents with some influence are likely to respond.

Agents may have different organizational ties to one another; one agent may be a peer to another or have line authority over another. Eventually, these agents will have some awareness of one another, however one agent may not have comprehensive knowledge of the other agents in the system.

A language that supports goal-level delegation, support for goal-directed problem solving, lends itself to the creation of systems that are responsive to their environment, should cleanly integrate goal-directed and responsive behavior, and should support knowledge-level communication and cooperation are all necessary for making all the agents in MAS interact with one another.

3.2 BDI Model

3.2.1 Introduction

The BDI software model is a programming approach for intelligent agents. Although it appears to be defined by the implementation of an agent's beliefs, desires, and intentions, it really employs these notions to address a specific challenge in agent programming [Sho97].

In essence, it offers a technique for distinguishing between the action of picking a plan and the execution of already active plans. As a result, BDI agents may balance the time spent discussing about plans and implementing those plans. The system designer and programmer are in charge of the third activity, which is planning—making the plans in the first place. This activity is outside the model's purview.

3.2.2 Architecture

This section outlines the BDI system's modeled architecture, also shown in Figure 3.1.

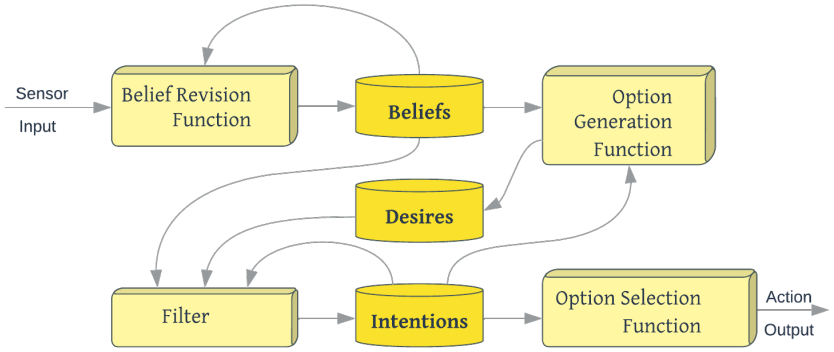

Figure 3.1: BDI Architecture

Beliefs

Beliefs reflect the agent's informational state, or its perspective (including itself and other agents). Inference rules can also be included in beliefs, allowing forward chaining to lead to new beliefs. Using the word belief rather than knowledge acknowledges that what an agent believes may or may not be accurate (and in fact may change in the future). One or more belief states and a belief set constitute a belief model.

Belief set: A collection of object diagrams that outline the subject matter of an agent class's beliefs serve as the specification for the belief set. A set of instance diagrams that specify a specific instance of the belief set is referred to as a belief state. Although this is an operational decision, beliefs are maintained in a database (also known as a belief foundation or a belief set).

Desires

Desires indicate the agent's motivational state. They reflect goals or scenarios that the agent wishes to achieve or create.

Goals: A desire that has been embraced for active pursuit by the agent is referred to as a goal. The additional requirement that the collection of active wants must be consistent is added when the term "goals" is used.

Intentions

Intentions depict the agent's deliberate state, or what the agent has decided to do. Intentions are aspirations that the agent has made some commitment to. This indicates that the agent has started carrying out a strategy in implemented systems.

Plans: Plans are collection of instructions that an agent can follow to carry out one or more of its objectives. They can be thought of as recipes or knowledge bases. Plans could incorporate different plans. Three different categories of nodes—start states, end states, and internal states—as well as one kind of directed nape of the neck the plan graph's constituent parts.

- Plan Execution: An *activation event* and *activation condition*, which specify when and in what context the plan should be activated, are labeled on the plan graph's first transition. The activation event could be a goal event that happens as a result of the accomplishment of a sub goal activity in another plan, leading to goal-driven activation, or a belief event that happens when an agent's beliefs change or when certain external changes are noticed. If more than one plan is relevant to an event in a particular context, they are either engaged concurrently if the activation is event-driven or progressively until they are successfully terminated if the activation is goal-driven. When a plan is activated, an action may be conducted with the help of an optional activation action.

  Events pass and fail may be used to indicate the success or failure of the activity connected with the active state during transitions between them. Activities may be halted by transitions from active states that are denoted with the event any whenever their condition is satisfied. The transition of a plan that is aborted is a specific instance of this. The plan fails if the abort condition is true at any point during the execution of its body after it has been triggered. The final transitions of the plan graph may be labeled with the steps to be taken in the event that the plan is successful, unsuccessful, or abandoned.

- Failure: The semantics of plan graphs includes the idea of failure. When an action on a transition fails, when there is an explicit transition to a fail state, or when the activity of an active state ends in failure with no outbound transition enabled, failure inside a graph can happen.

Events

In a nutshell, events function as triggers for the agent to react. An event may alter aims, activate plans, or update beliefs. Externally produced events may be picked up by sensors or other connected systems. Events may also be produced internally to start decoupling updates or activity plans.

In order to include obligations, norms, and commitments of agents acting within a social context, the BDI was further enhanced with an obligations component, giving rise to the Beliefs Obligations Intentions Desires (BOID) [Bro+01] agent architecture.

Agent Programming

In the AOP paradigm, the idea of software agents serves as the foundation for the development of the software. AOP has externally stated agents (with interfaces and messaging capabilities) at its core as opposed to OOP, which has objects (offering methods with variable parameters) at its foundation. They might be viewed as abstractions of actual items. Messages are interpreted in a class-specific manner by receiving "agents."

AOP versus OOP

AOP specialising the framework by fixing the state (now called mental state) of the modules (now called agents) to consist of components such as beliefs (along with beliefs about the world, about themselves, and about one another), capabilities, and decisions, each of which enjoys a precisely defined set of properties. OOP proposes viewing a computational system as being composed of modules that are able to communicate with one another and that have individual ways of handling incoming messages. Table 3.1 summarizes the relation between AOP and OOP.

A variety of restrictions are imposed on an agent's mental state, which generally match to restrictions placed on their common sense equivalents. These agents notify, request, offer, accept, reject, compete, and help one another to do a computation. This concept is directly adapted from the literature on speech acts [Pra77]. Speech is categorized according to speech acts, which include informing, requesting, offering, and more. Each of these communication acts has a unique set of assumptions and outcomes. AI, natural language processing, and plan recognition all use speech act theory. Table 3.1 summarizes the relation between AOP and OOP

Table 3.1: AOP versus OOP

	AOP	OOP
Fundamental Unit	Agent	Object
Parameters defining state of basic unit	beliefs, commitments, capabilities, choices	Unconstrained
Computation process	message passing and response methods	message passing and response methods
Types of message	inform, request, offer, promise, decline	Unconstrained
Constraints on methods	honesty, consistency	None

AgentSpeak(L): BDI Agents Interaction

AgentSpeak(L) is a simplified textual version of Procedural Reasoning System (PRS) [GI89] or Distributed Multi-Agent Reasoning System (dMARS) [dIn+04]. In most ways, the language and its operational semantics are comparable to the implemented system. The implemented system includes extra language constructs to facilitate agent programming.

AgentSpeak(L) is a programming language that uses a restricted first-order language with events and actions as its core. The programs created in AgentSpeak(L) govern the agent's behavior (i.e., its interaction with the environment). The agent's beliefs, desires, and intentions are not explicitly stated as modal formulae.

The agent's existing belief state may be considered as a model of itself, its environment, and other agents; states that the agent intends to bring about based on external or internal stimuli can be considered as desires; and the adoption of programs to meet such stimuli can be characterized as intentions. This shift in perspective of using a basic specification language as an agent's execution model and then ascribing mental attitudes such as beliefs, desires, and intentions from an external standpoint is more likely to integrate practice and theory.

JASON

The agent program and an agent architecture can be differentiate as, the software framework in which an agent program operates is referred to as the agent architecture. The PRS is an example of an agent architecture, and the plans are the software that exists within it. We design the program that will drive the agent's behavior, but the architecture itself determines most of what the agent does without the programmer having to worry about it. JASON's language is an extension of AgentSpeak, which is based on the BDI architecture. As a result, a belief basis is one of the components of the agent architecture. Another key component is the agent's goals, which are realized through plan execution.

It has become customary to include a simple example program to aid comprehension. Hence, an example of JASON AgentSpeak code:

```
started.

+started <- .print("Hello,_I'm_an_Agent").
```

Let us attempt to grasp some of what is going on here. The first thing to realize is that this is the definition of a single agent. This specification is generally kept in a single file with the `'.asl'` extension. Agent's basic beliefs is described in the first line. The full stop, `'.'`, serves as a syntactic separator in the same way as a semicolon does in Java or C. The next line establishes a plan for the agent, which is the sole plan this agent possesses. This action will display `'Hello, I'm an Agent'` on the user's terminal. Later, more goals and sub-goals also can be added, according to the plans and requirements.

ASTRA

ASTRA is also based on AgentSpeak(L) in that it provides all of the same fundamental capabilities as AgentSpeak(L), but it also adds a number of extra features that makes it more practical Agent Programming Language. The fundamental distinction between beliefs, aims,

and events is that words and variables are typed. ASTRA's type system is based on the Java type system, mainly because Java is the underlying implementation language, and using a comparable type system makes translating between ASTRA and Java easier and more visible.

As a result, ASTRA programs are more organized than AgentSpeak(L) applications. The basic structure is seen below:

```
package path.to.folder;
import java.lang.Object;

agent Hello {
    module Console console;

    initial !init();

    rule +!init() {
        console.println("Hello,I'm_Agent");
    }
}
```

As can be observed, ASTRA has embraced the Java `package` and `import` nomenclature. The `agent` keyword denotes the beginning of the agent program, and curly braces represent the body of that program (known here as an `agent class`). This specification is generally kept in a single file with the `'.astra'` extension. This action will also display `'Hello, I'm an Agent'` on the user's console.

3.3 Web3

3.3.1 Overview

The terms "Web 1.0" and "Web 2.0" refer to phases in the development of the World Wide Web using different technology and formats. Web 1.0 refers to a time when the bulk of websites just had static pages and the great majority of people consumed information rather than created it. Web 2.0 is centered on user-generated content that is published to forums, social media and networking sites, blogs, and wikis, among other services. It is founded on the concept of "the web as platform."

An notion for a new version of the World Wide Web called Web3 (sometimes referred to as Web 3.0) integrates ideas including decentralization, blockchain technology, and token-based economy. Some journalists and engineers have compared it to Web 2.0, where they claim that content and data are concentrated in a select few businesses frequently referred to as "Big Tech."

Web3's specific goals vary, and the phrase has been characterized as "ambiguous," but they all revolve on the concept of decentralization and frequently include BCT, including multiple

crypto-currencies and Non-Fungible Tokens (NFT). The concept of Web3 would include financial resources, in the form of tokens. It can alternatively be described as the ostensible next generation of the web's technological, legal, and monetary infrastructure, which includes crypto-currencies, blockchain, and smart contracts. Web3's three core architectural enablers were recognized as a mix of decentralized or federated platforms, safe interoperability, and verifiable computing via distributed ledger technologies.

The idea of Decentralized Autonomous Organizations (DAO) serves as the foundation for several concepts. On addition to owning your data in Web3, you can also use tokens that function like stock to collectively own the platform. DAO enables decentralized platform ownership coordination and future platform decision-making. Another important idea is Decentralized Finance (DeFi), which allows for money exchange without the intervention of banks or the government. Self-sovereign identification (SSI) enables users to identify themselves without depending on an authentication system like OAuth [Lei12], where identity verification requires contacting a trusted party. Web3 would most likely coexist with Web 2.0 websites, with Web 2.0 websites most likely adopting Web3 technology to keep their services updated.

3.3.2 Concept

Web 3.0 may be recognized by a collection of traits that are altering how people connect outside of the physical world and have an impact on how businesses are made [New22]:

- Individualized

  Information may be divided into context-relevant segments based on network contacts and personal preferences. In fact, one of the features of Web 3.0 is the capacity to deal with unstructured content on the Web more intelligently by giving the published data about individual or group factors contextual meaning. The social networks are a well-known method of connection with a very broad audience of adopters. People may choose their own exposure and contacts depending on information preferences and "proximity".

- Ubiquity

  People can connect at any time and from any location. Staying connected is made easier by a variety of communication alternatives, including mobile networks, Wi-Fi networks, cable networks, or cellular networks, as well as many device kinds, including laptops, desktop computers, tablets, and an endless array of mobile devices. Regardless of the architectures or systems being used

- Efficient

  Both people and computing devices are capable of filtering information based on context, meaning, and relevance. This trait has an implied connection to the individualized trait. Regarding individual variances, knowledge causes various responses in various persons. The capacity to filter content based on user interests appears to be a benefit of Web 3.0. Efficiency requires that the information must be organized such

that algorithms can read it and comprehend it as clearly and completely as humans can.

3.4 Blockchain

3.4.1 Introduction

Blockchain is a decentralized ledger system that may be used to store data across numerous cluster nodes. A typical blockchain network's cluster of nodes cannot be controlled by a single organization, hence the system is decentralized. The blockchain data is computationally authenticated by the majority of unconnected nodes in a cluster of arbitrary 'N' nodes. This fundamental architecture of blockchain assures that it is not controlled by a centralized authority and that the system as a whole is 'trustless.' We imply that there is no need to trust the source of data or the provider since mathematical computations and decentralized validation make the system computationally trustworthy.

Due to the lack of a single administrator who has access to change and remove the recorded data, this feature of blockchain technology assures that the data has not been altered. It displays characteristics like immutability, irrevocability, and non-repudiation as a result of its construction. Due to these characteristics, it is the perfect answer for several significant real-world use cases, including financial applications.

The modern blockchains allow users to create their own custom code that is then performed as functions inside the ecosystem and the outputs of these functions are added to the blockchain's updated state. These functions are often run at numerous network nodes and a consensus on the status of outcomes is reached. These tasks or workloads are known as smart contracts in Ethereum and several other public blockchains, and chaincodes in Hyperledger Fabric.

3.4.2 Architecture

According to its technical definition, a blockchain is a series of interconnected blocks that, as seen in Figure 3.2, include the data on a group of transactions up to the most recent block. The block headers retain the details of their own hash, that of their preceding block, a list of transactions, and the time stamp. In the blockchain network, these characteristics are utilized to track a specific transaction. All blockchain nodes save a block as the most recent state once it has been approved and confirmed by the majority of network participants.

Different procedures, including as proof of work (PoW) [Ger+16], proof of stake (PoS) [Ngu+19], and proof of authority (PoA) [De +18], etc., are available to reach agreement on the choice of a block.

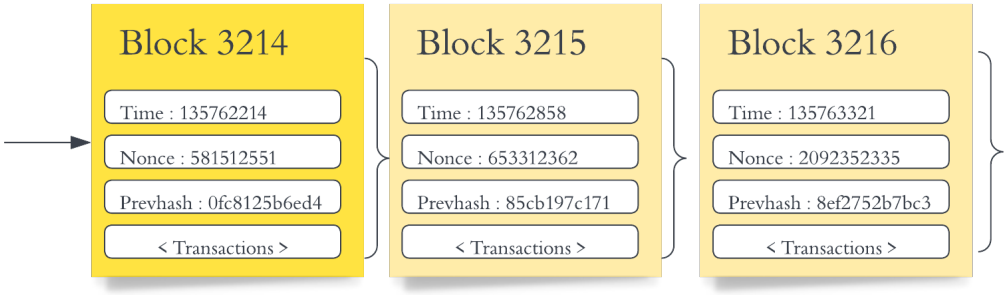

Figure 3.2: Blocks chained together in a blockchain

On a higher level, blockchains may be split into two categories: permissioned and permissionless. Consortial and private blockchains are also included in the permissioned blockchains, whereas public blockchains are included in the permissionless blockchains. Below, we explore these categories' characteristics.

Public Blockchains

Public blockchains are permission-less blockchains in which anybody with a personal computer may run an open-source protocol and connect to the network as an equal node. The public blockchains run completely decentralized peer-to-peer systems. They are the perfect approach for implementing digital currency. The most well-known public blockchains are Ethereum and Bitcoin.

Private Blockchains

Private blockchains are distributed node clusters that operate within a secure private network and store and process data according to blockchain rules. Private blockchains are governed by a centralized organization rather than being decentralized in nature. They primarily serve as a distributed ledger that is scalable. Private blockchains often employ the lightweight PoA protocol to add new blocks.

Consortial Blockchains

Consortial blockchains are permissioned blockchains, just as private blockchains, and the general public cannot join them. They belong to a group of priveleged individuals who could be associated with this network. The cluster of nodes is dispersed throughout the intranets of the participating organizations. They are especially helpful when two mutually trusted parties wish to use the distributed ledger's privacy-preserving features. Hyperledger Fabric, Hyperledger Indy, and other consortium blockchains are the most well-known.

3.4.3 Smart Contracts

To begin, we will define smart contracts. Szabo's [Sza97] definition is as follows:

Definition 1 *A smart contract is a set of promises, specified in a digital form, including protocols within which the parties perform on these promises.*

In order to automatically execute, control, or document legally significant events and activities in accordance with the provisions of a contract or an agreement, a smart contract is a computer program or transaction protocol. The goals of smart contracts are to decrease the need for trustworthy intermediaries, arbitration fees, fraud losses, and malicious and unintentional exceptions. The smart contracts provided by Ethereum are widely regarded as a crucial building block for DeFi and NFT applications. Smart contracts are frequently linked to cryptocurrencies.

A smart legal contract, as opposed to a smart contract, is a conventional, legally-binding pact with specific provisions defined and put into machine-readable code. Smart legal contracts should not be confused with smart contracts.

Solidity: Build Smart Contracts

The Ethereum blockchain's popularity and disruptive power are directly related to its capacity to execute smart contracts. BCT may be effectively used to create Decentralized Application (Dapp) using Ethereum platform. A Dapp is a tool used to connect individuals and groups on various sides of an interaction without the usage of a centralized intermediary.

Solidity, a Javascript-like language designed expressly for creating smart contracts, is the primary language used in Ethereum. Among its other characteristics, Solidity is statically typed and enables complicated user-defined types, libraries, and inheritance. The solidity compiler converts source code into Ethereum Virtual Machine (EVM) bytecode so that it may be deployed into the Ethereum network. The owner of the contract is responsible for paying the additional transaction fees associated with contract deployments and smart contract interactions in the form of *gas*.

EVM

The second-largest blockchain system in the world is believed to be Ethereum. With smart contracts, Ethereum enhances the blockchain paradigm. The applications running on the Ethereum blockchain are known as smart contracts. Ethereum provides the EVM to parse the source code of the contracts into an opcode sequence that is predefined by Ethereum in order to execute the smart contracts. For the Ethereum blockchain to correctly execute contracts and handle transactions, each node requires an EVM.

The EVM may be conceptualized practically as a vast, decentralized system with large numbers of objects, called *accounts*, that can maintain an internal database, run code, and communicate with one another.

3.4.4 Truffle Suite

Truffle

As a programming environment, testing framework, and asset pipeline for blockchains running on the EVM, truffle basically aims to simplify the workload of a developer. Truffle offers built-in binary management, deployment, linking, compilation, and testing for smart contracts in addition to automated contract testing for quick development. It offers configurable build pipeline with support for tight integration, as well as pNPM ackage management using Node Package Manager (NPM) and the ERC190 standard, i.e. Ethereum request for comment (ERC).

Truffle supports transactions and deployments with MetaMask to safeguard your mnemonic, which is a pattern of letters, and it offers enhanced debugging with breakpoints, variable analysis, and step functionality. With an interactive terminal for direct contract communication, it is an external script runner that runs scripts inside the Truffle environment. Provides a scriptable, extendable framework for deployment and migrations, as well as network management for deploying to any number of public and private networks.

Ganache

Ganache is a private blockchain enabling speedy production of Corda and Ethereum distributed applications. Ganache may be used throughout the whole development cycle, allowing you to create, distribute, and test Dapp in a secure and predictable setting.

Both a User Interface (UI) and a Command Line Interface (CLI) are available with ganache. A desktop program called Ganache UI supports both Corda and Ethereum. Ethereum programming is possible using the powerful command-line tool ganache. It supports snapshot/revert state, Ethereum JSON-RPC compatibility, Zero-config Mainnet and testnet forking, console-log in Solidity, and the ability to impersonate any account without the need for private keys.

3.4.5 Vision

As blockchain technology continues to develop, a number of new blockchains have recently come to light, each with a unique architecture and implementation that has been carefully chosen for the intended use case. Additionally, there are significant flaws in the conventional blockchain architecture that remain unresolved, and other implementations aim to address these concerns. (1) Lack of anonymity in transaction execution (2) High transactional delay

owing to complex consensus procedures that cause scalability concerns are the two main outstanding challenges with blockchains. (3) Communication between several blockchains; (4) Integration with the outside world.

Each of the more recent blockchain implementations, including Zcash, Polkadot, Ethereum 2.0, Chainlink, etc., is working to find a solution to these problems. With the use of trustworthy execution environments, we also hope to address the aforementioned issues with conventional blockchains in this thesis. Because the burden from the blockchain is transferred to secure execution environments, the blockchains are lightweight and offer transaction execution privacy. In addition, the Hyperledger Avalon has developed blockchain connectors for a number of distinct blockchains that may be used to promote interoperability between them. The Avalon infrastructure may also be used to build "attested oracles," which connect blockchains with data from the outside world.

4 Methods and Implementation

This chapter describes the layout of the thesis' proposed solution. It also discusses implementation specifics and the tools used to build smart contracts and JASON BDI agents. The BDI agents work in a MAS via JASON AgentSpeak and communicate via solidity and python-based smart contracts. We would talk about the application's design objectives and general overview. We will also go through the application's design and functionality for both creating smart contracts independently and after doing so, as well as for creating agents. We discuss the MAS and BCT implementation specifics, package and library versions utilized, and system configuration. Additionally, the procedures necessary to combine both in order to contribute to this thesis are also presented. This chapter's prerequisites include understanding the design and architecture of blockchain, AgentSpeak, and AOP as described in the previous chapter. The prior chapter's contents should be reviewed in order to fully comprehend the chapters that follow.

4.1 Roadmaps

The initial roadmap will involve generating multiple agents in a MAS and running them simply to see how they fulfill their objectives. Goals are the driving force that directs an agent's proactivity, as agents are expected to actively pursue the goals assigned to them without needing to be constantly stimulated, as opposed to object-orientation with service/method/function invocation, where services/objects are purely reactive entities that provide functionalities. Examine how agents summon each other by sending messages, as well as how they behave and interact with one another.

The next step will be to create smart contracts for each process on a blockchain, complete with state and events. Additionally, while creating the contract, select the appropriate language and version. After ensuring that everything is in order, combine MAS and BCT and try to run the overall program to ensure compatibility.

4.2 Design Goals

The purpose of this project is to develop an application that integrates MAS and BCT. To construct such an application, we took design considerations that influenced our architecture. We have certain particular goals in mind while contemplating integrating both technologies. The following design objectives are listed:

- Autonomous

  To fulfill the objectives assigned to an agent, autonomy simply implies being able to work autonomously. An autonomous agent, thus, at the very least, makes independent judgments about how to accomplish its given goals; these decisions (and subsequently, its actions) are within its own control and are not influenced by other forces.

- Efficiency

  Effectiveness strategies improve the functioning of a smart contract or lower the expenses connected with its use. These patterns can help operators and consumers save time and money.

- Access Control Patterns

  Particular smart contract functionalities are only accessible by certain people, according to access control rules. For a specific task, permissions and authorizations can be managed, such as limiting access to the admin. On a public blockchain ledger, where everyone can observe the contract but you want to limit who may do what within the contract, the ability to restrict access is especially helpful. Certain access control patterns, such multi-authorization, ownership, and role-based access control, have names that make it obvious what they are meant to do.

- Goal-directed behaviour

  If an agent has been assigned a specific objective, it is assumed that the agent would attempt to attain the goal. Proactivity eliminates completely passive actors who never strive to accomplish anything.

- Reactiveness

  Implementing an application that achieves an appropriate mix of goal-directed and reactive behavior becomes difficult. This is one of the primary design goals of AgentSpeak.

- Security

  Protection patterns are created to optimize a smart contract's level of security against any threat. Reentrancy attacks, overflow assaults, or the problematic behavior of the actual smart contracts are all prevented by using them. There are various different types of frequently utilized security patterns, which is not surprising given the quantity of assets connected to smart contracts. Numerous of these patterns, such as circuit breakers and evacuation plans, are intended to safeguard contracts against failure in the event that the worst occurs.

4.3 System Configuration And Model Description

The application is designed to be as basic as possible in order to learn how a supply chain works in the real world, i.e. each entity interacts with the other entity on a different level in order to complete the chain.

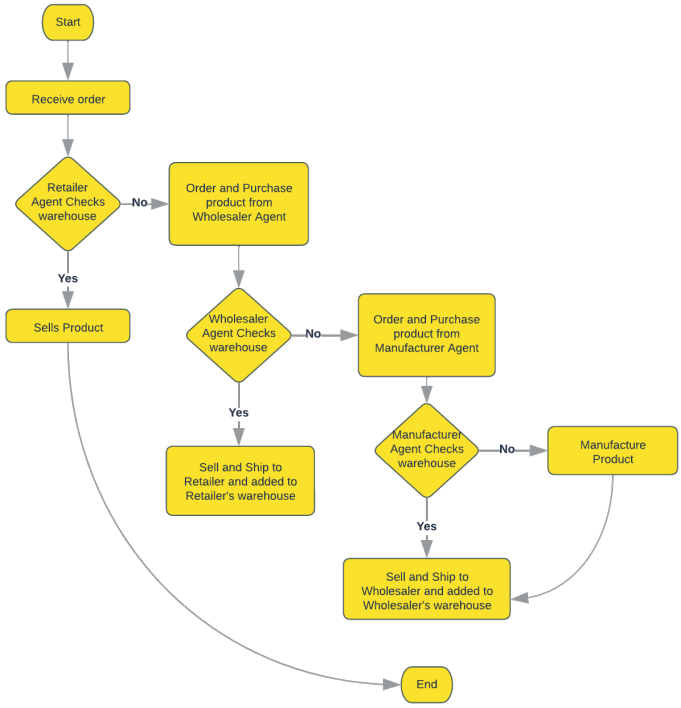

Figure 4.1: Supply Chain Flow Chart

The application flow for the *Smart contract development with JASON BDI agents* is depicted in Figure 4.1. These three agents, a manufacturer, a wholesaler, and a retailer, each perform a specific function in the supply chain. All three of the other agents are being invoked by another primary agent in the MAS. Figure 4.2 can be used for brief demonstration on which basis the contracts are to be created.

The application was created and tested on a Linux Ubuntu 22.04.1 LTS system. Visual Studio and InetlliJ are the Integrated Development Environment (IDE)s used to create the application. jEdit is also used somehow to run agents using JASON while creating `.asl` and running `.mas2j` file. To test the compatibility of web3j with gradle for agents, many versions of Java Standard Edition Development Kit were utilized, however the most commonly used was Java SE Development Kit 18.0.2.1.

Other standards for the development of Smart contracts and agents are detailed in the sections below.

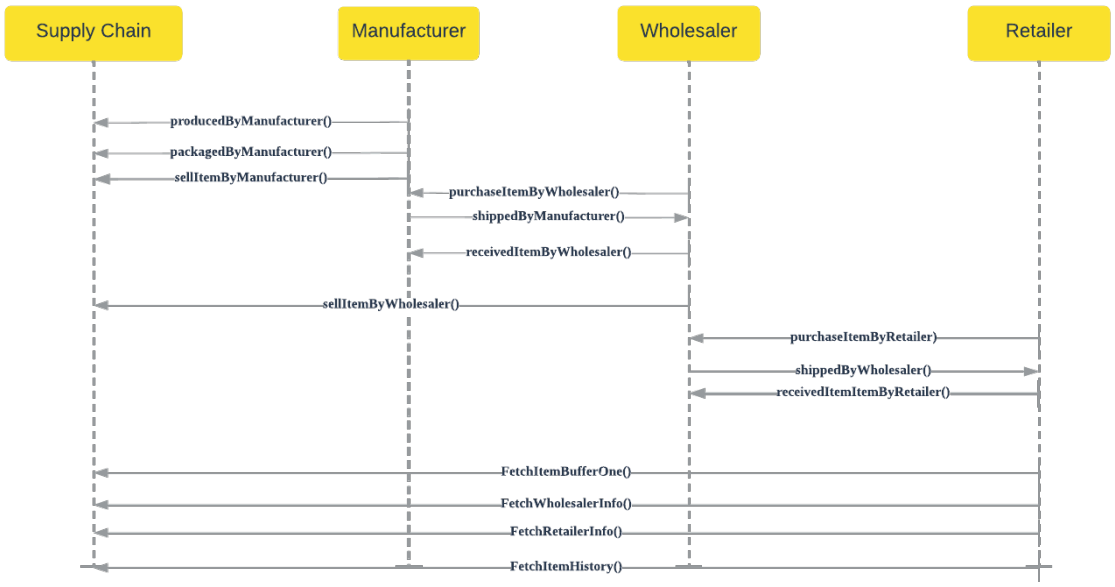

Figure 4.2: Supply Chain Sequence Diagram

4.4 Smart Contracts Development

While taking into consideration the scenario of a supply chain where all information on suppliers, recipients, products, and business circumstances is dispersed over supply chain databases. It is possible to execute the sale of products or services as a transaction that is cryptographically signed by the seller and the buyer and attached to a smart contract for sales transactions. When the sale occurs and all other conditions, such as documentation and quality checks, are met, the execution of the transfer of the corresponding funds and rights can be enforced; in other words, smart contracts can ensure that collaborative and entrepreneurial processes are carried out correctly.

As aforementioned, the terminology blockchain refers to two things: a distributed database and a data structure (i.e. a linked list of blocks containing transactions, where each block is cryptographically chained to the preceding one by incorporating their hash value and a cryptographic signature, in such a manner that changing an earlier block requires re-creating the whole chain since that block). The idea of smart contracts, which are scripts that run every time a specific kind of transaction takes place and may read and write to the blockchain, is related to the blockchain technology. Smart contracts enable parties to enforce the requirement that while one transaction occurs, other transactions also occur.

The programming for smart contracts related supply chain is done using Solidity Language while keeping the version in mind in order to integrate it with the agent programming language so that they can work together. Smart contracts perform the following functions: (1) The product is produced by the manufacturer, (2) the manufacturer completes the packaging, (3) the product is placed on the market, (4) the wholesaler purchases the product, (5) the

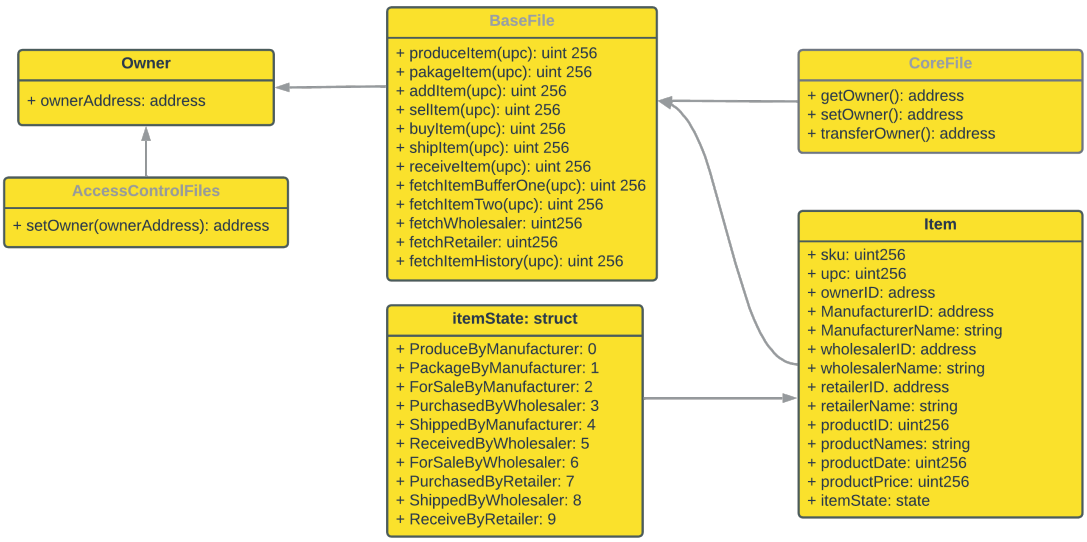

Figure 4.3: Smart Contract's Data Model Diagram

manufacturer ships the product, (6) the wholesaler receives the product, (7) the wholesaler places the product on the market, (8) the retailer purchases the product, (9) the wholesaler ships the product, and (10) the retailer receives the product as the final result.

Each function is regarded as an event, and every event is given a state, as shown in Figure 4.3. It has always been a rule that each event must occur after the one before it has concluded. For example, a manufacturer cannot sell a product before producing it, while a wholesaler cannot receive a product before buying it. It is carried out with the aid of a state check. A product can also be tracked using the Universal Product Code (UPC) or by utilizing the Stock Keeping Unit (SKU) to trace the entire batch.

Solidity language is used to create smart contracts, while JavaScript is used for testing. The `.sol` files were compiled using Solidity v0.8.13, which also produced `.abi` and `.bin` files. The truffle tool, especially truffle v5.6.5, has been used for deployment and testing. We have used ganache v7.5.0 and ganache-UI v2.5.4 to examine state and manage chain behavior. All of the packages listed below have been obtained using node v14.0.0 (npm v6.14.4) in the table 4.1.

Table 4.1: Package Version

Node Package	Version
web3	1.7.5
truffle	5.6.5
@truffle/contract	4.5.22
@truffle/hdwallet-provider	2.0.13
dotenv	16.0.1
geth	0.4.0
openzeppelin	4.7.3

Figure 4.4 was used as a reference for developing smart contracts connected to supply chain. The following events were included in the supply chain: To complete the supply chain, (i) the manufacturer manufactures the product, (ii) the manufacturer packages the product, (iii) the manufacturer sells the product to the wholesaler, (iv) the wholesaler purchases the product, (v) the wholesaler receives the product, (vi) the wholesaler sells the product to the retailer, (vii) the retailer purchases the product, (viii) the retailer receives the product, and restocks his/her inventory.

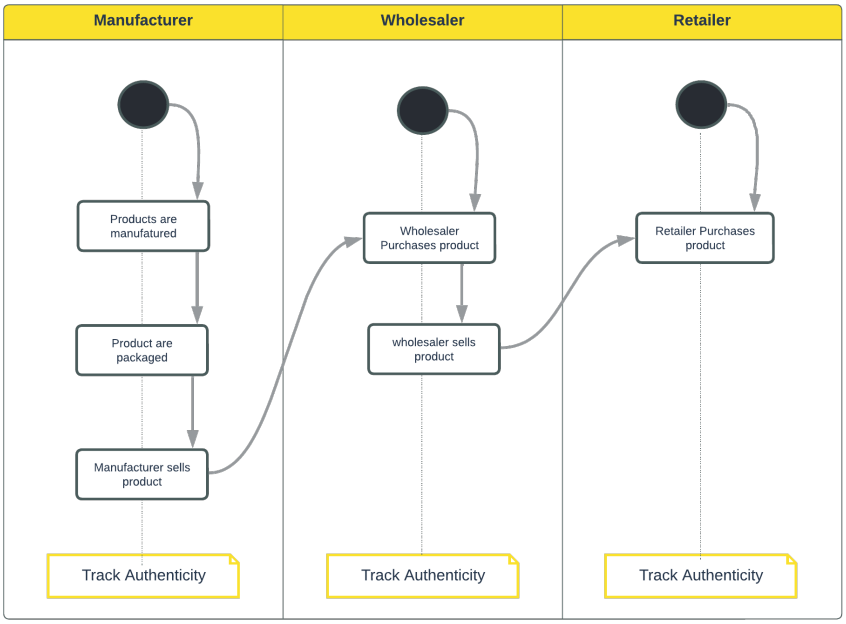

Figure 4.4: Supply Chain Activity Diagram

The way in which files organised an classes imported using inheritance can be understand from Figure 4.3, but for the in depth understanding Figure 4.5 should be taken into consideration. Solidity contracts can make use of a particular type of remark to give detailed documentation for functions, return variables, and other features. The name of this unique format is the Ethereum Natural Language Specification Format (NatSpec). The formatting for comments used by the author of a smart contract and recognized by the Solidity compiler is included in NatSpec. Additionally, third-party tool annotations may be included in NatSpec. The `@custom:<name>` tag is most likely how they are completed, and analysis and verification tools are an excellent example of this in usage.

Vyper was also chosen to construct smart contracts, which is now one of DeFi's most popular languages. Vyper is a high-level programming language identical to Python. However, due to its limitations over Solidity, the proposal was subsequently abandoned.

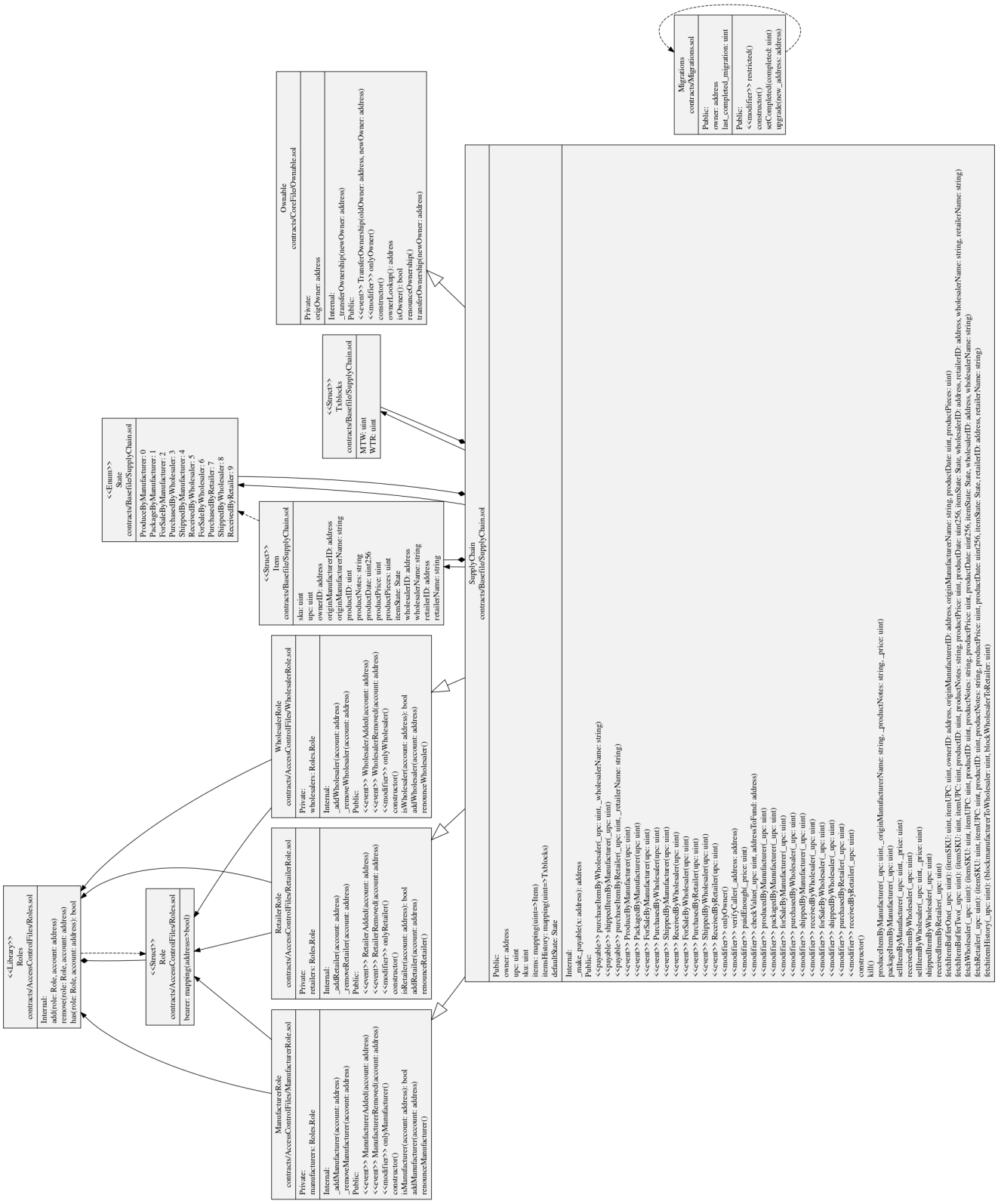

Figure 4.5: Smart Contract Class Diagram

4.5 Agent Model Implementation

An agent continually perceives its environment, makes decisions about how to act to achieve its objectives, and then takes action to alter the surroundings. The speech-act theory is

commonly used to describe agent communication in MAS. The speech-act hypothesis is based on the premise that language is action.

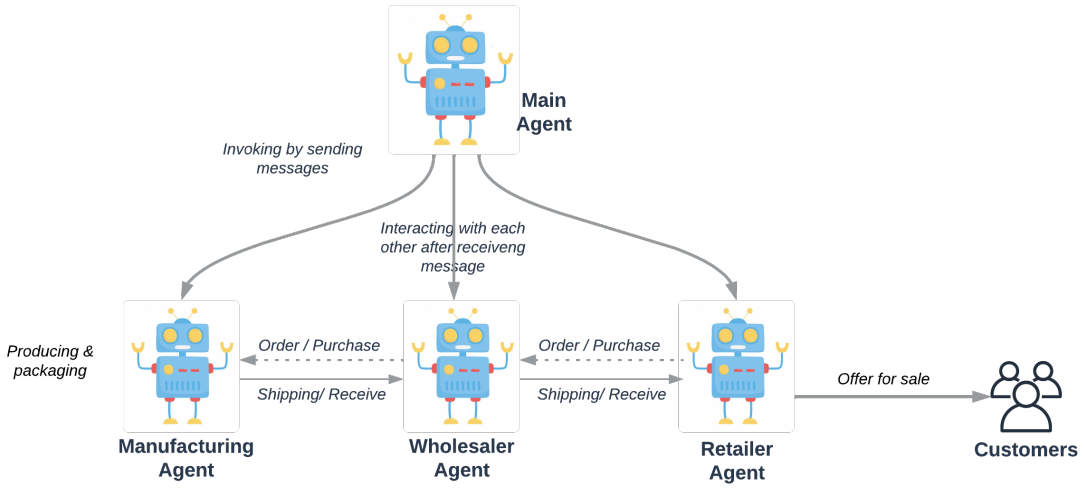

Figure 4.6: Agent Interaction in Supply chain

An agent program is run by the JASON interpreter. An agent uses a reasoning cycle to carry out its operations, which may be broken down into the following steps: first, it perceives the environment; second, it updates the belief base; third, it receives communication from other agents; fourth, it selects "socially acceptable" messages; fifth, it chooses an event; sixth, it retrieves all relevant plans; seventh, it determines the applicable plans; eighth, it chooses one applicable plan; and last, it executes one step of an intention.

In our MAS, each participant's function within the supply chain will be handled by an agent , see Figure 4.6. The retailer agent will work to ensure that the product is available in the warehouse before ordering it from the wholesaler agent, who will then ask the wholesaler to check its inventory, get in touch with the manufacturer to produce the product, and ship it to the wholesaler, who will then send it to the retailer agent. There will be a *main agent*, who will launch the retailer agent's efforts to sell items to consumers and also other agents by sending messages.

The BDI architecture is the most common technique to implementing "intelligent" or "rational" agents. The specification of a set of base beliefs and a set of plans results in the creation of an AgentSpeak(L) agent. AgentSpeak(L) differentiates between two kinds of goals: achievement goals and test goals. However, we use achievement targets for our agents. We employed a variety of techniques while writing our agents and ensuring that they could be utilized with smart contracts. We tested several different interpreters and wrote our agents using those. We pursue the following strategies:

- JASON (Java-based interpreter)

- ASTRA
- JASON (Python-based interpreter)

The solutions listed above have been defined later in this chapter.

The structure of the agent program is governed by the fact that there will be four agents in the MAS, as shown in Figure 4.7 in MAS, and they will interact with each other as shown in Figure 4.6. Our implementation strategy is as follows:

- The supply chain will be initiated by the main agent, who will then engage the retailer agent;
- Retailer agent will inspect its inventory and sell products to customers; if a product is not in stock, retailer agent will ask the main agent to contact the wholesaler agent;
- The wholesaler agent will check its warehouse and ship the product to the retailer agent; if the product is not available, the wholesaler agent will request that the manufacturer agent be contacted by the main agent;
- Upon checking its inventory, the manufacturer agent will ship the product to the wholesaler agent. If the product is not in stock, the manufacturer agent will manufacture the product, does the package and deliver it to the wholesaler.

Every implementation uses the same approach to how agents cooperate and communicate, and each implementation is detailed in more detail in the section below.

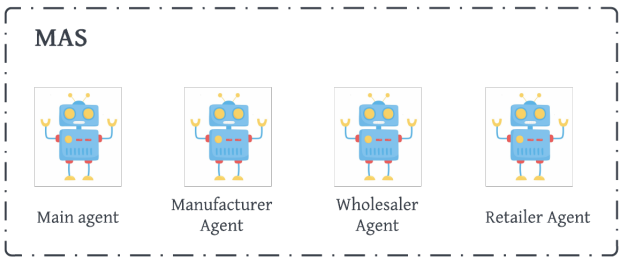

Figure 4.7: Agents in MAS

JASON (Java-based interpreter)

We first used JASON with a Java-based interpreter to develop the agents in MAS. We set up the home variables, downloaded the necessary scripts and libraries, then used gradle as well as maven for simple configuration. AgentSpeak's expanded dialect has an interpreter named JASON. It implements the operational semantics of that language and offers a platform for creating MAS with a variety of characteristics that may be altered by the user. Version 3.1 of

JASON, which is the most recent version, was installed. The agents are constructed using figure 4.6 as a guide.

In addition to being able to interpret the original AgentSpeak, JASON possesses a few other crucial abilities. Inter-agent communication based on speech acts and strong negation are included, allowing for the use of both closed- and open-world assumptions. Additionally, it supports creating environments (which are not normally to be programmed in AgentSpeak; in this case they are programmed in Java). It offers the ability to deploy distributed MAS via a network (using Java Agent DEvelopment Framework (JADE)); the user may also add other distribution infrastructures. Furthermore, it offers an IDE in the form of an Eclipse or jEdit plugin; the IDE has a "mind inspector" that aids with debugging.

ASTRA Implementation

ASTRA programs are divided into agent classes, which may be expanded using a multiple inheritance paradigm. Each agent class is written in a separate file with the same name as the agent class and the `.astra` extension. ASTRA is distinct from AgentSpeak (L). Because ASTRA applications can refer to Java classes, support for delivering fully qualified class names is required. ASTRA programs incorporate partial plans (called plan bodies) in addition to plan rules to promote code/class resuability. ASTRA strives to be familiar to developers who are familiar with mainstream programming languages, particularly Java.

Agent Programming Languages are intended to aid in the creation of MAS. Such systems are intended to have more than one agent and more than one agent type by default. Support for deploying numerous agents is given in many Agent Programming Languages via deployment files, which let the developer to define the initial community of agents to be deployed. ASTRA does not support this; instead, you construct an agent that generates new agents. The System API provides the essential functionality to allow this approach. In ASTRA, coding one agent to produce another agent is extremely straightforward. You just invoke the System API's `createAgent(...)` operation.

JASON (Python-based interpreter)

An interpreter for JASON, an agent-oriented programming language, built on Python. It can be installed in the system using `preferred installer program (pip)`. For our implementation, we utilized agentspeak 0.1.0. Python-agentspeak is similar to JASON, except you don't need to create a `.mas2j` file to construct a multi-agent system; instead, all the agents may be called together by calling them in a `.py` file.

4.6 Agent-Contract Collaboration

We are largely focused on agent-oriented models and technology, and we have smart contracts directly incorporated into the blockchain. With reference to the illustration 4.6,

the model will be the same for the agents, but their communication will take place via smart contracts, be recorded as transactions on the blockchain, and be subsequently verified using the transaction hashes.

The thesis' core premise is the integration of the two technologies, MAS and BCT. We have tried to integrate both the technologies using several tries by using several web3 libraries, i.e., *web3.js* for JavaScript, *web3.py* for Python and *web3j* for Java in order to interact with Ethereum.

JASON and Web3j

We attempted to test each smart contract using web3js after generating them with Solidity. We considered migrating to web3j since it would be simpler as opposed to continuing to utilize JASON AgentSpeak, which is based on Java. Through the Command Line Tools tool, Web3j facilitates the development of Java smart contract function wrappers from Solidity ABI files or straight from Truffle's Contract Schema. To reveal the contract's per-network deployment address, a wrapper is "improved" and produced. When the wrapper is created, these addresses are from the truffle deployment.

The plan to utilize Web3j with JASON was eventually scrapped since, in order for JASON to run MAS, the `.mas2j` file had to be executed, and when it did, it couldn't find the `org.web3j` package. We attempted to download the jar files locally, change the version of web3j and JASON, and switched from gradle to maven in order to make the program run, but the problem persisted.

ASTRA and Web3j

After several attempts with web3j and JASON, we considered switching to ASTRA, a programming language that is quite similar to Java and aims to be familiar to developers with language. A successful agent construction was followed by the same problem as with JASON when importing the web3j package.

JASON and Web3.py

Web3.py is a Python package that allows you to connect with Ethereum. It is often used in Dapps to support a number of use cases, including sending transactions, communicating with smart contracts, accessing block data, and more. The Web3.js JavaScript API served as the foundation for the original API, which has since expanded to meet the demands and conveniences of Python developers.

JASON's Python interpreter and Web3.py worked well together. In order to communicate or convey messages from one agent to another, `.asl` files for agents were produced in Python

using AgentSpeak, which is a bit different from JASON constructed using Java. Additionally, running a `.mas2j` file is not necessary for MAS in JASON-style AgentSpeak for Python.

JASON and Jython

Jython is a Python programming language implementation meant to operate on the Java platform. JPython was the previous name for the implementation. Any Java class may be imported and used by Jython apps. Jython applications, with the exception of a few common modules, employ Java classes rather than Python modules.

Jython provides practically all of the modules found in the standard Python programming language package, with the exception of a few modules written in C. Jython either dynamically or statically converts Python source code to Java bytecode (an intermediate language). The following tasks are especially well suited for Jython:

- In order to interact with Java packages or run Java programs, Jython provides an interactive interpreter. This makes it possible for developers to experiment with and debug any Java system using Jython.

- Users can write simple or intricate scripts using embedded scripting to increase an application's capabilities. The Jython libraries are available to Java programmers for their systems.

- Python scripts are frequently 2–10 times shorter than their Java equivalents, enabling quick development of applications. This has a direct impact on how efficiently programmers work. Python and Java get along well with one another, allowing programmers to mix the two languages at will during both product development and release.

JASON AgentSpeak's smart contract development for Python and Web3.py was a success that we explained in the next chapter. So we decided to give it another shot and try to develop it using Jython. The Jython project provides Python implementations in Java, giving Python the benefits of operating on the JVM and access to Java classes.

The next chapter explains the results from our suggested approaches for taking the steps along smart contracts deployed in a bespoke blockchain with their own flow of control among agents constructed utilizing agent oriented language.

5 Results

This chapter focuses on reviewing all of the work completed while utilizing all of the technologies to construct the application. Prior to integrating the technologies, each technique is evaluated independently.

5.1 Smart Contract Performance

Solidity-based smart contracts were created with the intention of integrating them into supply chains to maintain records of goods ownership and movement from one entity to another. It was vital to determine if the smart contracts were functioning properly or not shortly after developing them while keeping in mind the supply chain's sequence as shown in Figure 4.2.

To verify this, we created several test cases and ran them using the Truffle tool. We kept in mind to verify the states, which are essentially the order of the supply chain, as well as store the ownership and movement of the product from one entity to another entity in each contract when doing the testing. The test cases are illustrated below utilizing a local network named "development," in which we are testing each contract by confirming the state of the product displayed in the Figure 4.3, as well as movement and ownership of the product using SKU, UPC, and owner ID and storing it into a buffer and then the cross checking them.

```
Using network 'development'.
Compiling your contracts...
===========================
> Everything is up to date, there is nothing to compile.
<----------------ACCOUNTS---------------->
Contract Owner: accounts[0]  0xad0BC114B5CF3F0797346fF1Fb1Daf1Cf5123395
Manufacturer: accounts[1]  0x4A9fe326Edc88F1f22940DC9F70BD391fB4218f8
Wholesaler: accounts[2]  0x5fB0Cd136C7A19E8E12F062548002B4460B0dC0d
Retailer: accounts[3]  0xc7D1C50D87B82E85b959DBC2cD9959bfc0480A5E
<-------TESTING CONTRACT FUNCTIONS------->
  Contract: SupplyChain
    Testing smart contract function produceItemByManufacturer() (14051ms)
    Testing smart contract function packageItemByManufacturer() (2723ms)
    Testing smart contract function sellItemByManufacturer() (2526ms)
    Testing smart contract function purchaseItemByWholesaler() (2037ms)
    Testing smart contract function shippedItemByManufacturer() (1566ms)
    Testing smart contract function receivedItemByWholesaler() (1542ms)
    Testing smart contract function sellItemByWholesaler() (1486ms)
    Testing smart contract function purchaseItemByRetailer() (2440ms)
```

```
    Testing smart contract function shippedItemByWholesaler() (1438ms)
    Testing smart contract function receivedItemByRetailer() (1380ms)
  10 passing (1m)
```

According to the test scenarios, smart contracts are functioning well. Each contract will produce a transaction hash, which is simple to get and can be used to further verify information like as the date and time, amount of gas consumed, account used to deploy the contract, etcetera.

Performance Over Test Networks

Due to a built-in feature of the truffle tool that indicates how long it takes to build each contract as well as how long it takes to generate all of the contracts, we had to consider other options for our smart contracts. We tried to check the time taken by each network in order to get more knowledge, specifically taking into account the local network, Alfajores, and Rinkeby. We also considered testing out a different programming language to write our contracts in order to determine why Solidity is more effective than all of them.

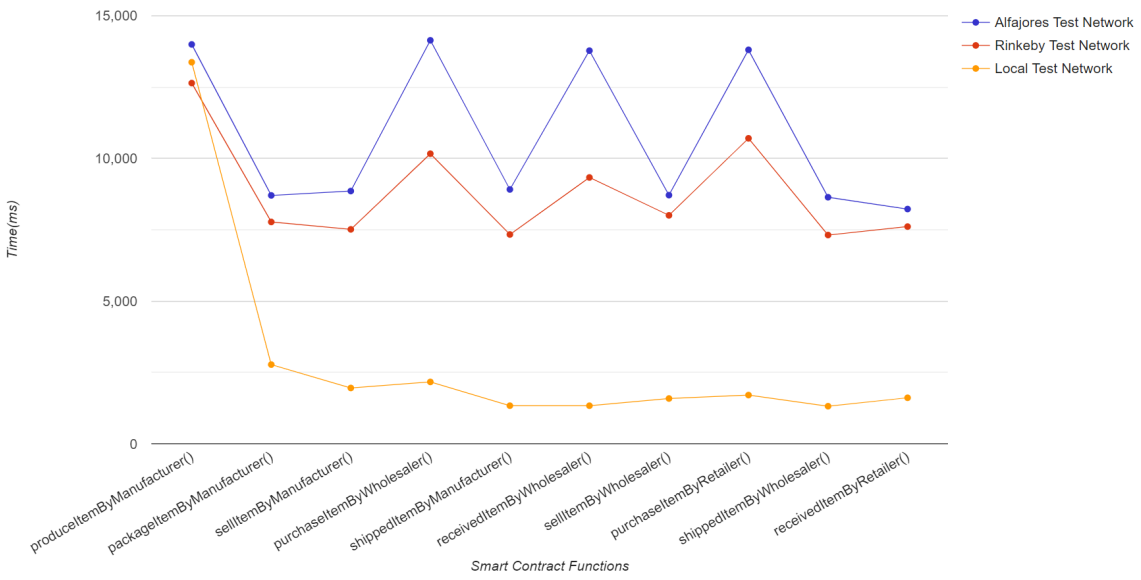

Figure 5.1: Performance across various networks

It is obvious that the local network used far less time than the other two networks from Figure 5.1. However, it was simpler to obtain CELO for Alfajores, the blockchain currency we used as gas for deployment and payment, than it was to acquire RinkebyETH for the Rinkeby test network. On the other hand, we can see that contract deployment was a little faster using the Rinkeby test network compared to the Alfajores test network.

Contract Language Analogy

We were decided from the outset to utilize Solidity to construct the smart contracts since it is a curly-bracket language meant to target the EVM. C++, Python, and JavaScript have all

had an impact on it. Solidity is statically typed and, among other things, enables inheritance, libraries, and sophisticated user-defined types. Furthermore, it receives frequent upgrades and breaking modifications, and new features are released on a regular basis.

JASON's and some issue with web3 libraries gave us the opportunity to explore more and to switch from Solidity to Vyper and learn more about the language. Because Vyper lacks *Modifiers*, *Class Inheritance*, *Inline Assembly*, *Function Overloading*, *Operator Overloading*, and *Binary Fixed Point*, a thorough examination of Vyper prompted us to chose Solidity once more. The usage of the following constructs might result in confusing or challenging to comprehend code, hence they are not included and to create final smart contracts, we continued to use Solidity as our primary language.

5.2 MAS Development

We originally utilized JASON with a java-based interpreter to execute MAS. To run MAS, we downloaded the necessary scripts and libraries. Gradle and maven were then used for straightforward configuration and setting up the home variables. Then, in order to enable agent interaction, we created `mas2j` files after designing `asl` files for each agent. We introduced four agents: `supplyChainAgent`, `manufacturerAgent`, `wholesalerAgent`, and `retailerAgent`. The `supplyChainAgent` is the primary agent who will engage other agents to achieve their objectives. As their names imply, the other agents will perform their respective tasks in an adaptive supply chain. Running the agents in the MAS environment produced the following results.

```
<----------------INTERACTION BETWEEN AGENTS----------------->
supplyChainAgent : Starting SupplyChain with SmartContracts
supplyChainAgent : Hi, I am the owner of Contract
supplyChainAgent : Creating RetailerAgent
retailerAgent    : Hi, I am here
retailerAgent    : Checking Warehouse, and order
retailerAgent    : Ordering to wholesalerAgent
supplyChainAgent : Creating WholesalerAgent
wholesalerAgent  : Hi, I am here
wholesalerAgent  : Checking Warehouse, and order
wholesalerAgent  : Ordering to manufacturerAgent
supplyChainAgent : Creating ManufacturerAgent
manufacturerAgent: Hi, I am here
manufacturerAgent: Checking Warehouse, and Manufacturing
manufacturerAgent: Manufacturing Product
manufacturerAgent: Packaging Product
manufacturerAgent: Selling product to wholesalerAgent
wholesalerAgent  : Purchasing product from manufacturerAgent
manufacturerAgent: Shipping product to wholesalerAgent
wholesalerAgent  : Received product from manufacturerAgent
wholesalerAgent  : Selling product to retailerAgent
retailerAgent    : Purchasing product from wholesalerAgent
wholesalerAgent  : Shipping product to retailerAgent
retailerAgent    : Received product from wholesalerAgent
retailerAgent    : NOW SELL TO CUSTOMER!!
supplyChainAgent : SUPPLYCHAIN COMPLETE
```

Due to the limitations of Java-based interpreter with `web3` package, we immediately switched to ASTRA and JASON with a Python-based interpreter. However, all of them produced the same outcome as described above, despite the fact that the scripting of agents and the MAS environment differed.

ASTRA agents, unlike JASON, are not written in `asl` files, and it also does not require a `mas2j` file to make all of the agents interact with one another. In ASTRA, all of the agents' primary and secondary goals may be expressed in a single `astra` file. Agents are written in Java-style syntax, which makes it easier for coders to comprehend and write in the format. The primary means of interaction between agents in ASTRA is through the usage of an Agent Communication Language (ACL). ASTRA allows for direct contact through Foundation for Intelligent Physical Agents (FIPA) ACL-based message forwarding.

Although ASTRA is simpler to grasp, it suffers from the same restriction as JASON with its Java-based interpreter in that it cannot leverage the `web3` package to infuse BCT into the MAS. As a result, we decided to use JASON with a Python-based interpreter. It is being used after installing the `agentspeak` package using pip. It utilizes the same `asl` file, but instead of a `mas2j` file, it initiates the agents interaction with a python script. It is as simple to run as any other Python script and works flawlessly when smart contract functionalities are added to it as `actions` of agents.

5.3 Infuse BCT in MAS

Following the creation of contracts, we began looking for an appropriate AOP language that is compatible with the `web3` library and can be used by agents in a MAS to generate smart contracts. In the last chapter, we discussed every solution we considered.

Finale Outcome

Our main objective was to find a way to incorporate BCT into MAS and create smart contracts using JASON BDI agents. One of our ideas was successful, and the infuse was effective. We tested the code several times to make sure it was running correctly by looking at the `contract address` and `transaction hashes` from ganache as we were deploying using local network, and each try was successful, as can be seen in the output from one of the evaluations below.

```
<---------------------SMART CONTRACTS AND AGENTS----------------------->
Deployed Contract Address: 0xc9f78D73aCAf603Fe2319682316268A39Cc5CBB7
Owner Address: accounts[0] 0xad0BC114B5CF3F0797346fF1Fb1Daf1Cf5123395
Manufacturer Address: accounts[1] 0x4A9fe326Edc88F1f22940DC9F70BD391fB4218f8
Wholesaler Address: accounts[2] 0x5fB0Cd136C7A19E8E12F062548002B4460B0dC0d
Retailer Address: accounts[3] 0xc7D1C50D87B82E85b959DBC2cD9959bfc0480A5E
<---------------------INTERACTION BETWEEN AGENTS---------------------->
supplyChainAgent : Starting SupplyChain with SmartContracts
supplyChainAgent : Hi, I am the owner of Contract, with account: 0xad0BC114B5CF3F0797346fF1Fb1Daf1Cf5123395
supplyChainAgent : Creating RetailerAgent
retailerAgent    : Hi, I am here, with account: 0xc7D1C50D87B82E85b959DBC2cD9959bfc0480A5E
retailerAgent    : Checking Warehouse, and order
retailerAgent    : Ordering to wholesalerAgent
supplyChainAgent : Creating WholesalerAgent
wholesalerAgent  : Hi, I am here, with account: 0x5fB0Cd136C7A19E8E12F062548002B4460B0dC0d
wholesalerAgent  : Checking Warehouse, and order
wholesalerAgent  : Ordering to manufacturerAgent
supplyChainAgent : Creating ManufacturerAgent
manufacturerAgent: Hi, I am here, with account: 0x4A9fe326Edc88F1f22940DC9F70BD391fB4218f8
manufacturerAgent: Checking Warehouse, and Manufacturing
manufacturerAgent: Manufacturing Product
manufacturerAgent: Tx produceItemByManufacturer successful with hash: 0x653ff850d6fdee68554ded12d07c3c570c1a8a7d9aafe64b6b3fe54abe1b05f8
manufacturerAgent: Packaging Product
manufacturerAgent: Tx packageItemByManufacturer successful with hash: 0xaaefbe4aaf9c4533f7bc0c83323140151e33cc896695deeb4d9596123683f4f2
manufacturerAgent: Selling product to wholesalerAgent
```

```
manufacturerAgent: Tx sellItemByManufacturer successful with hash: 0xadd6ee6dfe24c84b04562f66509842c74d7d5c9c20fbe15883773e30632ce582
wholesalerAgent  : Purchasing product from manufacturerAgent
wholesalerAgent  : Tx purchaseItemByWholesaler successful with hash: 0x9dc089278a6f8491d891c77215136ece5d69c94a37081f35428d9e6184b6da70
manufacturerAgent: Shipping product to wholesalerAgent
manufacturerAgent: Tx shippedItemByManufacturer successful with hash: 0xa020ef5eaed13aa05cbda1f92cb331fa6f4c493164a22ff11a43073b1ab289e0
wholesalerAgent  : Received product from manufacturerAgent
wholesalerAgent  : Tx receivedItemByWholesaler successful with hash: 0x19b414d448c142dbba17468c42f05d2012089e067442913f4c45a9c5011e5d9f
wholesalerAgent  : Selling product to retailerAgent
wholesalerAgent  : Tx sellItemByWholesaler successful with hash: 0xd562b49a09473dc00304abbb2d3be55f451a27406197a011d750329c62ee3c19
retailerAgent    : Purchasing product from wholesalerAgent
retailerAgent    : Tx purchaseItemByRetailer successful with hash: 0xfd490f1fa472c42d33612392f42756115d340df8685b95bb0a85203917ea0ff6
wholesalerAgent  : Shipping product to retailerAgent
wholesalerAgent  : Tx shippedItemByWholesaler successful with hash: 0x4104a12dbb37f565fa07191135eb35c249591e6c5832c2ba703f9c8746ac1bf8
retailerAgent    : Received product from wholesalerAgent
retailerAgent    : Tx receivedItemByRetailer successful with hash: 0xf19d8bc9b0ddd26e010b7fb19a319ddb8d0d390cf673756bff3299d8d36d2873
retailerAgent    : NOW SELL TO CUSTOMER!!
supplyChainAgent :  TxHashes stored in BLOCKCHAIN
supplyChainAgent : SUPPLYCHAIN COMPLETE
```

After a successful implementation, we remained adamant on running it in Java and used the `web3j` package to integrate BCT into MAS. We attempted to use JASON with Java by converting the Python code into a Java application. The creation of a Java application containing Python code was successful, but when the `.mas2j` file was executed, it was unable to import the package `org.python` , which is necessary to start the program.

6 Conclusion and Further Work

In this thesis, we discussed about the MAS and BCT, which are well-known technologies that will likely be employed in the future in a significant supply chain sector. We provide an implementation and a sample regarding the creation of smart contracts with JASON BDI agents, whereby the smart contracts are constructed using Solidity, the agents are developed with the JASON Python-based interpreter, and both are integrated with the web3 package.

As part of our ongoing efforts, we may expand the functionality of our program and create a fantastic user interface, which will allow us to modify the code and make it usable. In order to utilize the web3j package, we also anticipate having a successful implementation with our other choices, maybe following a significant update in all the other languages we tried to employ that include a Java-based interpreter.

A Appendix I

B Appendix II

List of Figures

List of Tables

List of Listings

Bibliography

[Pra77] Mary Louise Pratt. *Toward a speech act theory of literary discourse*. Indiana University Press, 1977.

[GI89] Michael P Georgeff and Felix Ingrand. "Decision-making in an embedded reasoning system". In: *International joint conference on artificial intelligence*. 1989.

[Sho97] Yoav Shoham. "An overview of agent-oriented programming". In: *Software agents* 4 (1997), pp. 271–290.

[Sza97] Nick Szabo. "Formalizing and securing relationships on public networks". In: *First monday* (1997).

[Bro+01] Jan Broersen et al. "The BOID architecture: conflicts between beliefs, obligations, intentions and desires". In: *Proceedings of the fifth international conference on Autonomous agents*. 2001, pp. 9–16.

[FBT01] Mark S. Fox, Mihai Barbuceanu, and Rune Teigen. "Agent-Oriented Supply-Chain Management". In: *Information-Based Manufacturing: Technology, Strategy and Industrial Applications*. Ed. by Michael J. Shaw. Boston, MA: Springer US, 2001, pp. 81–104. ISBN: 978-1-4615-1599-9. DOI: 10.1007/978-1-4615-1599-9_5. URL: https://doi.org/10.1007/978-1-4615-1599-9_5.

[dIn+04] Mark d'Inverno et al. "The dMARS architecture: A specification of the distributed multi-agent reasoning system". In: *Autonomous Agents and Multi-Agent Systems* 9.1 (2004), pp. 5–53.

[BH06] Rafael H. Bordini and Jomi F. Hübner. "BDI Agent Programming in AgentSpeak Using Jason". In: *Computational Logic in Multi-Agent Systems*. Ed. by Francesca Toni and Paolo Torroni. Berlin, Heidelberg: Springer Berlin Heidelberg, 2006, pp. 143–164. ISBN: 978-3-540-33997-7.

[Lei12] Barry Leiba. "Oauth web authorization protocol". In: *IEEE Internet Computing* 16.1 (2012), pp. 74–77.

[CRL15] Rem W Collier, Seán Russell, and David Lillis. "Exploring AOP from an OOP perspective". In: *Proceedings of the 5th International Workshop on Programming Based on Actors, Agents, and Decentralized Control*. 2015, pp. 25–36.

[Sre+15] Dejan Sredojević et al. “Domain specific agent-oriented programming language based on the Xtext framework”. In: *5th International Conference on Information Society and Technology, Society for Information Systems and Computer Networks, Mar*. 2015, pp. 8–11.

[Ger+16] Arthur Gervais et al. “On the security and performance of proof of work blockchains”. In: *Proceedings of the 2016 ACM SIGSAC conference on computer and communications security*. 2016, pp. 3–16.

[Che+17] Lin Chen et al. “Decentralized execution of smart contracts: Agent model perspective and its implications”. In: *International conference on financial cryptography and data security*. Springer. 2017, pp. 468–477.

[Bra+18] Juliao Braga et al. “Blockchain to improve security, knowledge and collaboration inter-agent communication over restrict domains of the internet infrastructure”. In: *arXiv preprint arXiv:1805.05250* (2018).

[Cal+18] Davide Calvaresi et al. “Multi-agent systems and blockchain: Results from a systematic literature review”. In: *International conference on practical applications of agents and multi-agent systems*. Springer. 2018, pp. 110–126.

[De +18] Stefano De Angelis et al. “PBFT vs proof-of-authority: Applying the CAP theorem to permissioned blockchain”. In: (2018).

[Ngu+19] Cong T Nguyen et al. “Proof-of-stake consensus mechanisms for future blockchain networks: fundamentals, applications and opportunities”. In: *IEEE Access* 7 (2019), pp. 85727–85745.

[Cia+20] Giovanni Ciatto et al. “From Agents to Blockchain: Stairway to Integration”. In: *Applied Sciences* 10.21 (2020). ISSN: 2076-3417. DOI: 10.3390/app10217460. URL: https://www.mdpi.com/2076-3417/10/21/7460.

[Maf20] Alfredo Maffi. *Tenderfone Smart Contracts*. Version v1. Last accessed by Alfredo Maffi authored 2 years ago. gitlab. 2020. URL: https://gitlab.com/pika-lab/blockchain/tenderfone/tenderfone-sc.

[New22] Cointelgraph Newsletter. *What is Web 3.0: A beginner’s guide to the decentralized internet of the future*. 2022. URL: https://cointelegraph.com/blockchain-for-beginners/what-is-web-3-0-a-beginners-guide-to-the-decentralized-internet-of-the-future.