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

24th February 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, 24th February 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
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
IoT	Internet of Things

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 agent-hood. 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. 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 assumed to be robotic agents, in which case additional aspects including sensory input, motor control, and time constraints are stated. Finally, agents are frequently perceived as "high-level." Although this definition is rather ambiguous, many people use it to distinguish agents from other bits of hardware or software. Symbolic representation and/or cognitive-like capabilities are examples of "high-level" abilities. Agents can be "informable," have symbolic planning as well as stimulus-response rules, and even talk. Because this notion is not commonly understood in AI, the concept of agenthood in AI is hazy. As a result, when we use the term "agent," we need to be clear about what we mean, which is as follows: An agent is an entity whose state is regarded to be formed of mental aspects such as beliefs, capabilities, decisions, and commitments. These elements are precisely specified and roughly correspond to their equivalents in 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.

Recent years have seen a surge in interest in the study of computational agents with the ability to behave rationally. In the commercial world, the architecture of the systems needed to carry out high-level management and control operations in complex, dynamic circumstances is becoming more and more crucial. Examples of these systems include those used in air traffic control, telecommunications networks, business operations, spacecraft, and healthcare. Such systems are exceedingly difficult and expensive to develop, validate, and maintain

when utilizing typical software approaches. Agent-oriented systems, which are based on a fundamentally different perspective on computational entities, provide opportunities for a qualitative shift in this stance.

One such architecture envisions the system as a sensible agent with distinct mental attitudes of BDI, reflecting the informative, motivational, and deliberative states of the agent, respectively. 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 allow agent programs to be developed and processed in the same way as a horn-clause rationale program. 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 retailers, distributors, and manufacturer agents to adjust to environmental changes. Each agent has a core objective as well as related secondary objectives. BDI agents can be used to improve decision-making in supply chain management by allowing for the simulation and optimization of different scenarios and strategies. For example, a BDI agent could be used to forecast demand for a product and adjust inventory levels accordingly.

Additionally, BDI agents can also be used for negotiation and coordination with other agents to obtain better prices or delivery schedules. Despite what might be expected, there will always be a transaction anytime agents engage in the supply chain, which must be documented. Any centralized database could be utilized by agents to store the transaction, but is that the best option? 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 as a result 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 globalized era and intelligent sectors of the economy, including the financial sector, management, healthcare, and Internet of Things (IoT), among many others. Smart contracts offer a tamper-proof and auditable record of all the transactions between various parties, which may be used to improve traceability and transparency in the supply chain. In lengthy and complicated supply chains, retailers and distributors follow the items' flow, which might be very helpful.

While conducting a further in-depth study on programming agents and smart contracts, we would want to investigate the viability of an interaction mechanism between agent-oriented programming and smart contracts in this thesis. We shall attempt to address the research questions (RQ) listed below.

> RQ1. What character do we need in smart contracts to coordinate with the beliefs, desires, and intentions of agents?
>
> RQ2. How to represent supply chain roles (manufacturer, wholesaler, retailer) in Solidity language such that BDI model agent library can coordinate and synchronize AgentSpeak plans and goals with on-chain smart contract transaction payload?

In this thesis, ***Smart contract development with Jason BDI Agents***, we will attempt to develop an application that will display various agents, allowing agents across the supply chain to interact with one another while comparing different BDI agent Framework to answer the above questions. This ecosystem will allow agents to maintain relationships, store transactions using smart contracts, and determine whether both technologies can be used to create a fair and efficient dispute resolution mechanism in the supply chain by providing a transparent and tamper-proof record of the agreements and interactions between parties.

1.1 Motivation

The blockchain is increasing influence on various research and application fields, from distributed computing and storage to supply chain management and healthcare accountability. 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, and 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). BDI agents can be used to build more resilient and sustainable supply chains by simulating and optimizing various sustainability scenarios, such as examining the environmental effect of different transportation routes or manufacturing processes. BDI agents can also be programmed to automatically initiate smart contracts with suppliers or customers to enforce sustainability-related commitments.

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 and trust, and they actually extend their reach to cover computations other than data storage and management. Smart contracts can be utilized to improve supply chain traceability and transparency by providing a tamper-proof and auditable record of all transactions between multiple parties. This is especially beneficial in complicated and extensive supply chains where it is difficult to maintain track of products' movements.

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. Combining MAS with smart contracts can have several potential benefits for supply chain management and other industries:

- Improved coordination
  The supply chain can be made more efficient and successful by coordinating the operations of numerous actors using smart contracts. Smart contracts could be used to set the rules and restrictions that govern interactions between supply chain agents, which can assist to coordinate the operations of multiple agents and limit the risk of mistakes or inconsistencies.

- Decentralized decision-making
  By programming agents to make decisions based on their beliefs, desires, and intentions, multiple agent systems can provide a decentralized and autonomous approach to decision-making. This allows agents to make decisions based on the most current information and adapt to changing conditions in the supply chain.

- Transparency
  Smart contracts and multiple agent systems can provide transparency in the supply chain. Smart contracts can be used to record all activities and interactions among agents, allowing all parties to see the flow of goods and the state of the supply chain.

- Improved security
  By using smart contracts to define the rules and constraints that govern the interactions between agents, the system can be more resistant to malicious attacks and fraud.

Additionally, multiple agents can work together to detect and defend against any malicious attempt.

- Scalability
  Smart contracts can be used to handle a large number of transactions and interactions between agents, which can help scale the system as the number of agents and transactions increases.

- Interoperability
  By using smart contracts to facilitate communication and coordination among multiple agent systems, the system can be designed to be interoperable with other blockchain and non-blockchain platforms.

- Flexibility
  By using multiple agents with BDI approach, the system can adapt to changing conditions and goals, making the system more flexible.

It is crucial to remember that while merging MAS with smart contracts might have numerous advantages, doing so requires knowledge of both disciplines, and the system's design must take into consideration the supply chain's complexity and dynamic nature.

1.2 Our contributions

The contribution of the paper are as follows:

- We are addressing the literature study on BDI-based agent-oriented programming and autonomous agent-based decentralized smart contract execution.
- 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 is to construct MAS using Jason, an extension of AgentSpeak(L) as well as comparing with other BDI agent frameworks and finally creating smart contracts with Solidity and integrating them altogether.

- 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

Some research reveals methods for connecting the MAS and BCT by putting agents and blockchains side by side, enabling agents to utilize blockchain services as needed. Current model research on BDI agents and smart contracts is on leveraging BDI concepts to create and build optimization techniques that can communicate with blockchain-based smart contract systems. This involves the integration of BDI agents with blockchain-based decision-making processes, such as consensus protocols and smart contract execution, as well as the application of BDI-based reasoning to manage uncertainty and adapt to changing situations in decentralized contexts.

Use of BDI agents for decentralized autonomous organizations (DAOs), which are decentralized, self-governing organizations that run on smart contracts built on the blockchain, is one illustration of this. A DAO's members can be represented by BDI agents, who can then employ BDI-based reasoning to decide on their behalf. Using BDI agents for smart home automation is another field of research; these agents can operate as a bridge between users and gadgets and can make choices based on the users' intentions, beliefs, and desires.

Overall, the present state of the art for BDI agents and smart contracts is still in its early stages; however, with growing interest in blockchain technology, research, and development in this subject is expected to expand in the future years. This thesis will primarily focus on agent-oriented models based on the BDI framework and connected to the blockchain for the goal of running a supply chain utilizing agents. In Chapter 2, we dug further into the subject by asking some Literature review questions (LRQ) to gain a better understanding.

1.4 Evaluation Results

In order to answer our research questions, our evaluation was conducted using three separate criterias: (1) Choosing among various AOP Language with BDI framework, (2) Smart Contract Functionality, and (3) Outcomes of combining smart contracts with Agents. Experimental findings from these many angles are systematically examined.

Choosing AOP Language with BDI Framework

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 analysis indicated that, while the code in the `.asl` files for Jason's Java and Python-based interpreters is similar, the two employ different commands to communicate. The main distinction between the two is that the Java-based interpreter requires a `mas2j` file to execute MAS, but the Python-based interpreter only requires a python script that creates an environment for several 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.

Smart Contract Functionality

We will investigate if it is possible to describe supply chain members (such as a retailer, wholesaler, and manufacturer) in Solidity, the programming language used for writing smart contracts on the Ethereum blockchain. We examined h,ow well smart contracts run on severbasedtworks, includareg the local network, Infura utilizing Rinkeby, and Alfajores, as well as how long it takes to construct each contract and Java-basedption amount. Local networks consistently had the lowest delay time acrosPython-basedtworks 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.

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 with literature research questions(LRQs),
- 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,
- Chapter 5 details the application's outcomes and evaluation.
- Chapter 6 comprises a self-discussion of the primary concepts identified when researching study subjects,
- Conclusions of the system's evaluation are summarized in Chapter 7,
- Design enhancements for upcoming and additional study subjects that result from this effort are covered in Chapter 8.

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 literature research questions (LRQ):

> **LRQ1.** How to build intelligent agents within MAS using AOP language while comparing it with Object Oriented Programming (OOP)?
>
> **LRQ2.** What are the frameworks available for creating agents using the BDI model?
>
> **LRQ3.** How to include smart contracts into MAS in order to store transactions on a blockchain?
>
> **LRQ4.** What is the purpose of conducting further study when considerable research has already been done on the decentralized execution of smart contracts using the agent model?

2.1 Assimilating Agents and Agent-Oriented Programming

To address **LRQ1** about learning about AOP, MAS, and other concepts to explore with our thesis, the articles listed below assisted in gaining a comprehensive view of all of the topics and understanding them better.

> Paper Title: *Exploring AOP from an OOP perspective*
>
> Researchers in agent-oriented programming have created several agent programming languages that successfully connect theory and practice. Unfortunately, despite these languages' popularity in their 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; to create 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.

Paper Title: *Jason* for BDI Agent Programming in AgentSpeak

This research [BH06] is based on the teaching provided as part of the CLIMA instructional VI program. The purpose of the lesson was to present a broad overview of the capability provided by Jason, a MAS development platform based on an interpreter for an upgraded version of AgentSpeak. The BDI architecture is the most well-known and thoroughly investigated architecture for cognitive agents, and AgentSpeak is a sophisticated, logic-based programming language inspired by the BDI design.

The study also highlights how agent-based technology has grown in popularity for a variety of reasons, including its suitability for the development of a wide range of applications, such as air traffic control, autonomous spacecraft management, healthcare, and industrial system control, to name a few. These are unquestionably application areas where dependable systems are required. The fact that formal verification approaches developed particularly for MAS are also attracting a lot of academic attention and are expected to significantly influence agent technology adoption.

Paper Title: *A Multi-agent System for Optimizing Urban Traffic*

This article [FG03] presents the creation of a hierarchical multi-agent framework for controlling an urban traffic system, which is made up of numerous locally running agents, each representing a traffic system intersection. Two traffic scenarios are studied to evaluate the operation of the proposed urban traffic multi-agent system. The multi-agent system efficiently controlled the network's progressive congestion in both circumstances (traffic accidents and morning rush hour).

Paper Title: *Decision Making in Multiagent Systems: A Survey*

In this study [RAT18], the researchers look at the most recent work on cooperative MAS decision-making models, such as Markov decision processes, game theory, swarm intelligence, and graph theoretic models. Reinforcement learning, dynamic programming, evolutionary computing, and neural networks are examples of algorithms that produce optimum and suboptimal policies.

The researchers also talk about how these models may be used in robotics, wireless sensor networks, cognitive radio networks, intelligent transportation systems, and smart electric grids. Furthermore, the researchers define key terms in the field and discuss remaining challenges such as incorporating big data advances into decision making, developing autonomous, scalable, and computationally efficient algorithms, tackling more complex tasks, and developing standardized evaluation metrics.

Paper Title: *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. All the 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.

There is Jason framework that we are about to use in order to create agents which facilitates reasoning about beliefs, desires, and intentions, as well as agent communication. However there are other frameworks available that support the BDI paradigm and MAS. LRQ2 led us to these research articles presented below to understand better.

Paper Title: *Languages for Programming BDI-style Agents: an Overview*

In this paper [MDA05], nine languages and systems are reviewed and compared namely PRS, dMARS, JACK, JAM, Jadex, AgentSpeak(L), 3APL, Dribble, and Coo-BDI. There are additional links to other systems and languages based on the BDI concept, as well as surveys dealing with relevant themes.

Paper Title: *A Review of Agent Platforms*

The goal of the paper [LPG15] is to give information on various frameworks and conduct an in-depth analysis. It offers details on active projects, projects whose status is unknown in relation to certain platform types, such as general-purpose, special-purpose, modeling-simulation platforms, and platforms that are no longer being developed. It also shows MAS development techniques.

Table 6.1 in Chapter 6 has been likewise structured with the assistance of this research.

Paper Title: *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].

Paper Title: *A Survey of Agent Platforms*

The article [KB15] provides an up-to-date comparative analysis of the most promising current agent platforms that can be deployed. It is built on universal comparison and assessment standards, offering classes to assist readers understand which agent platforms have roughly comparable qualities and which decisions should be made in which scenarios.

In this article, the researchers offered a practical guideline recommendation for comparing and evaluating agent platforms, which is directly related to the scope of platform operation and the quality of each platform's given feature. This guidelines proposal is divided into five criterion categories: platform qualities, usability, operational abilities, pragmatics, and security management. Each of these categories has a number of subcriteria suggesting an in-depth assessment of each agent platform.

Paper Title: *Multi-agent oriented programming with JaCaMo*

The researchers presented a complete strategy for multi-agent oriented programming that combines the agent, environment, and organizational levels, giving a programming model that tries to connect relevant abstraction elements in an effective and synergistic way. To put the concept into reality, the researchers developed the JaCaMo platform, which integrates and expands current techniques and supporting technologies, particularly Jason (agent dimension), CArtAgO (environment dimension), and Moise (communication dimension) (organisation dimension). In this paper [Boi+13], a case

study and various applications were used to demonstrate the overall strategy.

Paper Title: *Brahms: A multiagent modelling environment for simulating work processes and practices*

The researchers suggest in this paper [SCV07] that the Brahms language is appropriate for researching many social and work practice issues of relevance to the organizational process modeling community. Their modeling work practice experience and findings imply that wider social problems may also be simulated. The Brahms modeling language provides significant advantages for researchers because, when compared to other tools like as Swarm, it allows for a more "natural" description of human behavior at the level of activities, reasoning, communication, contact with objects, and mobility in the world.

Brahms, like other BDI languages, is a declarative language, but it is distinct from the other BDI languages in numerous ways: Brahms is an activity-based language, whereas the majority of other BDI languages are task-based. Brahms has a subsumption design, whereas most other BDI languages employ a goal-based architecture; it supports environment modeling (geography), agent mobility in the environment, and so on; Finally, Brahms provides a distinct fact-state for modeling the world state outside of the agent's belief-set, whereas typical BDI-languages only describe agents with an independent belief-state.

Paper Title: *Jadex: A BDI-Agent System Combining Middleware and Reasoning*

This article [BPL05] describes a method for integrating an agent middleware with a reasoning engine in order to maximize the benefits of both strands. The reason for agent-oriented middleware was discussed, as well as an overview of the BDI model, and the design and implementation of the Jadex BDI engine as an extension to the widely used JADE agent platform. The Jadex system enables the creation of rational beings with goal-directed (rather than task-oriented) behavior. The Jadex agents are built using well-established software engineering approaches such as XML, Java, and OQL, allowing software professionals to swiftly capitalize on the possibilities of the mentalistic approach.

Paper Title: *LightJason, a Highly Scalable and Concurrent Agent Framework: Overview and Application*

The paper [Asc+18] provides an overview and use of LightJason, a highly scalable Java-based AOP and simulation platform. The researchers describe the architectural elements of LightJason and demonstrate its applicability via the use of an example of a browser-based web application that implements a traffic serious game designed to instruct an interdisciplinary student team in MAS and AOP.

The use-case model illustrates a highway traffic scenario that is used to teach MAS and AOP to students. Vehicles, road segments with speed restrictions, and traffic rule enforcement are all modeled as BDI agents. The researchers focused on how

LightJason supports integrating agent technology into cutting-edge browser-based web applications, which is difficult with existing agent frameworks, which are mostly stand-alone with "hard-wired" integrated development environments, runtime, graphical user interface, and source editor. These components are modular in LightJason, making the structure lightweight, versatile, and simple to incorporate.

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 run smart contracts by the agents, created using Jason framework. The papers listed below are aligned to our work and can be read to have a stronger insight to answer the LRQ3 we posed in this chapter.

Paper Title: *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 paper [Cia+20], the researchers give a roadmap and emphasize the issues that still need to be addressed in order to comprehend the prospects, dimensions to consider, and several methods for merging agents with blockchain. They then discussed the case of Tenderfone [Maf20], a custom blockchain that offers proactive smart contracts as the first step along the roadmap, equipping smart contracts with control flow encapsulation, reactivity to time, and asynchronous communication means, as both validation of their roadmap and a foundation for future development.

Paper Title: *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.

The authors of this paper created an agent-based framework to simulate the execution of smart contracts over a decentralized network of nodes and participants using blockchain and public ledger. Unlike the wias held belief that smart contract execution

results can be trusted, the agent-based model of smart contract execution assumes that nodes may have the incentive to influence or lie about contract execution results 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.

Paper Title: *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.

Paper Title: *Self-Aware Smart Contracts with Legal Relevance*

This paper [Nor18] introduces a revolutionary, blockchain-agnostic paradigm for cross-organizational peer-to-peer cooperation. When new blockchain technology enabled smart contracts are paired with intelligent multi-agent systems, they provide so-called self-aware contracts, which offer a high degree of automation for such peer-to-peer partnerships.

It was revealed that combining BDI agents with the declarative SAC-Language results in self-aware contracts, where the former ensure that trustworthy information is routed into contract-based collaborations as a merged set. The researchers accomplished human manageability of the self-aware contract framework by offering a declarative

smart-contract language that defines cross-organizational contract-collaborations, in addition to deploying agents that give a degree of artificial intelligence in a collaboration.

Understanding how applications employing MAS and smart contracts are deployed inside a supply chain is crucial to ascertain if smart contracts and agents can be used to build a fair and efficient dispute resolution procedure in a supply chain.

As mentioned in LRQ4, some work has been done combining agents and blockchain, however we are conducting additional research in this area because the agents we are about to use are BDI agents with Jason framework, which are a type of software agent that can make autonomous decisions based on its beliefs about the environment, desires or goals, and intentions to achieve those goals. We are attempting to integrate blockchain technology into a multi-agent environment created using Jason framework in which intelligent agents can communicate with one another.BDI agents can assist with supply chain management by arranging the flow of commodities, monitoring inventory levels, and negotiating with suppliers and consumers.

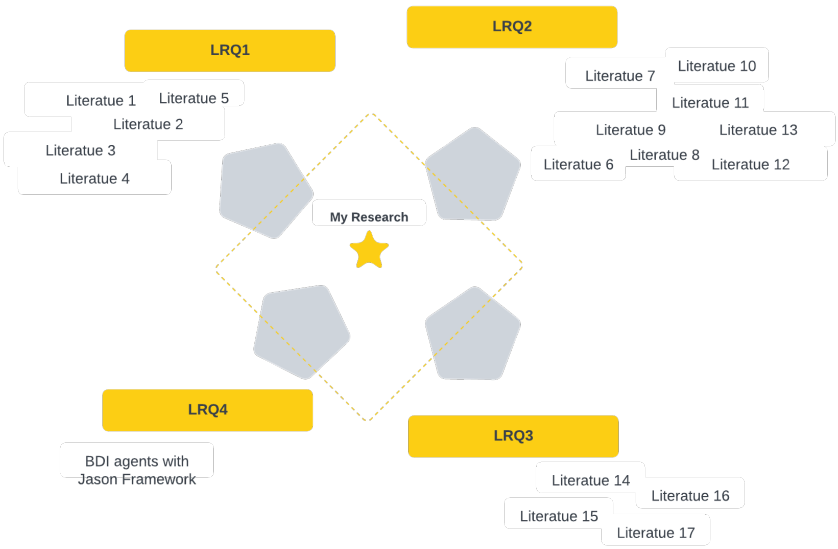


Figure 2.1: Literature Contribution

Figure 2.1 depicts a visual representation of the above-mentioned paper's contribution to our literature study and addressing the LRQs. The next chapters attempt to construct our own application utilizing solidity-written smart contracts with Jason BDI agents to investigate if BDI agents can enhance the collaboration, coordination and negotiation between supply chain members and if it can be used with smart contracts to manage unforeseen events and contingencies in supply chain.

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 blockchain technology's protocol. The core of this thesis is the creation of supply chain agents and the initiation of their interaction using smart contracts, which are thoroughly detailed in this chapter. We go into great depth on the various topics that enable us to operate agents with smart contracts.

3.1 Agent Introduction

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 [Ing+97]. 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

Agents are defined as systems that exist in a certain context. This means that agents can sense their environment and have a repertoire of possible actions to do in order to affect their environment. 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).

Aside from being located in an environment, the following characteristics are expected of a rational agent [JW95]:

- 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 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 Multiple Agent System (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 one agent.

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 Belief-Desire-Intention (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, its emphasis is on the 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 plans and implementing those plans.

3.2.2 Architecture

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

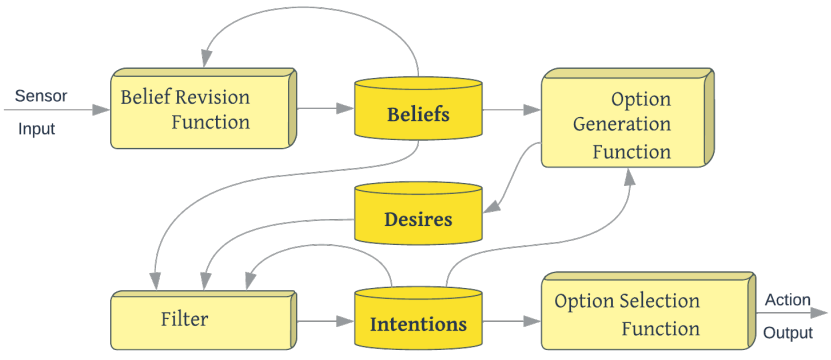


Figure 3.1: BDI Architecture [Abb+18]

- Beliefs
  Beliefs reflect the agent's informational state or 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

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 specifying a specific instance of the belief set is called 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 the 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 collections of instructions an agent can follow to carry out one or more objectives. They can be thought of as recipes or knowledge bases. Plans could incorporate different plans.

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 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 if 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 fails 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.

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-Oriented 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 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), desires, and intentions, 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.

Table 3.1: AOP versus OOP

	AOP	OOP
Focus	Autonomous agents	Objects
Structure	Multi-agent systems	Class hierarchies
Interaction	Agent-to-agent communication	Object-to-object interaction
Execution	Concurrent execution of agents	Sequential execution of objects
Autonomy	High degree of autonomy for agents	Limited autonomy for objects

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 differentiated 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 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's 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 are 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 several extra features that makes it a more practical agent programming language. The fundamental distinction between beliefs aims,

and events are 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 on forums, social media, networking sites, blogs, and wikis, among other services. It is founded on the concept of "the web as a platform."

A notion for the new version of the World Wide Web called Web3 (sometimes referred to as Web 3.0) integrates ideas including decentralization, blockchain technology, and a 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 around 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 cryptocurrencies, 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. In addition to owning your data in Web3, you can also use tokens that function like a 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. One of the features of Web 3.0 is the capacity to deal with unstructured content on the web more intelligently by giving published data about individual or group factors contextual meaning. Social networks are a well-known method of connection with a very broad audience of adopters. People may choose their 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 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.

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 chain codes 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 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 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.

On a higher level, blockchains may be split into two categories: permissioned and permissionless. Consortial and private blockchains are also included in the permissioned

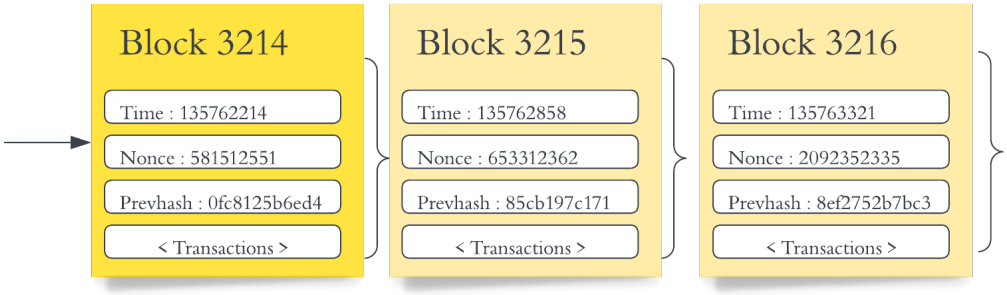


Figure 3.2: Blocks chained together in a blockchain [AC21]

blockchains, whereas public blockchains are included in the permissionless blockchains. Below, we explore these categories' characteristics [SKM19; ZL20].

- 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 like private blockchains and the general public cannot join them. They belong to a group of privileged 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*.

Ethereum Virtual Machine

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 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 a configurable build pipeline with support for tight integration, as well as NPM package 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 the 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, several 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, Hyperledger Avalon has developed blockchain connectors for several 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 are created using the Jason framework and communicate by sending messages and storing the interaction details using Solidity-based smart contracts. We will discuss 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 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 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. 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. 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 goal of this project is to create an application that combines MAS and BCT. We used design considerations that guided our architecture to build such an application. In considering integrating both technologies, we have specific aims in mind. The following design objectives are listed:

- Decentralization
  One of the main advantages of using smart contracts and BDI agents in supply chain management is that they allow for decentralized decision-making and coordination. The design of the system should aim to enable decentralization.

- Autonomous
  Autonomy simply implies being able to work autonomously. To fulfill an objective, an autonomous agent will make independent judgments about how to accomplish its given goals; these decisions and subsequently, its actions are within its control and are not influenced by other forces.

- Transparency
  Smart contracts and BDI agents are designed to provide transparency in the supply chain. The design of the system should aim to provide transparency of all activities and interactions among agents, allowing all parties to see the flow of goods within the supply chain.

- 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.

- Interoperability
  The design of the system should aim for interoperability with other blockchain and non-blockchain platforms in order to enable the integration of existing systems and the exchange of information among different platforms.

- Flexibility
  Supply chains are complex and dynamic systems. The design of the system should be flexible enough to adapt to changing conditions in the supply chain. This can be achieved by programming the agents to reason and decide based on their beliefs, desires and intentions which can change over time.

- 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
  As MAS on the blockchain will handle sensitive information, the security of the system is crucial. The design of the system should aim to protect agents and transactions from malicious actors.

- Scalability
  As the number of agents and transactions in a supply chain increases, the scalability of the system can become a challenge. The design of the system should aim for scalability solutions that enable large-scale deployment of agent-based systems on the blockchain.

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.

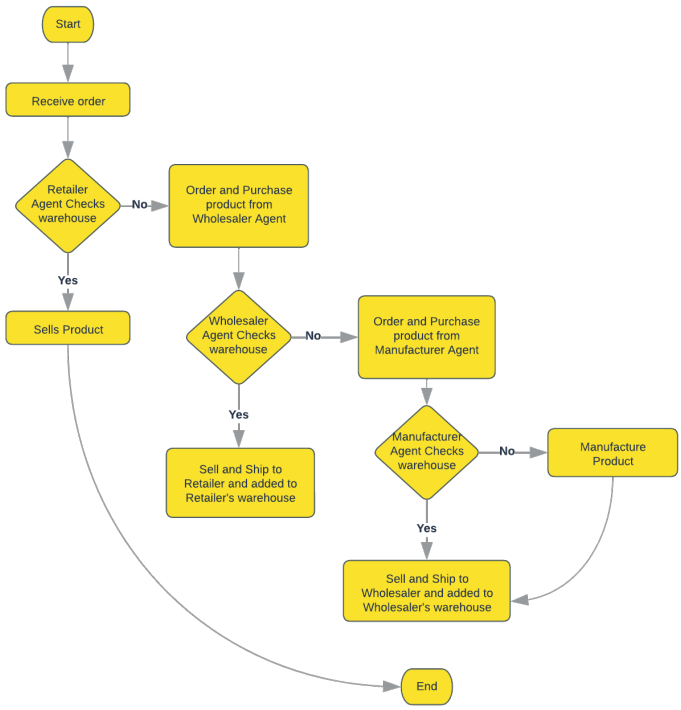


Figure 4.1: Supply Chain Flow Chart

The application flow for the *Smart contract development with Jason BDI agents* is depicted in figure 4.1. These three agents, `manufacturerAgent`, `wholesalerAgent`, and `retailerAgent`, each perform a specific function in the supply chain. All three of the other agents are being invoked by another primary agent i.e., `mainAgent` in the MAS. Figure 4.2 shows the sequence of generating the smart contracts within the supply chain. Later, figure 4.7 and 4.6 shows the agent interaction, and figure 4.8 shows the integration of BDI agents and smart contracts together.

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) used to create the application. *jEdit* is also used somehow to run agents using Jason framework 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 BDI agents are detailed in the sections below.

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 are 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 as shown in figure 3.2, 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 blockchain technology is connected to the concept of smart contracts, which are scripts that run every time a certain type of transaction occurs and can read and write to the blockchain. Smart contracts allow parties to enforce the requirement that additional transactions occur concurrently with one transaction.

The programming for smart contracts related supply chain is done 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 with reference to figure 4.2: (1) The product is produced by the manufacturer, (2) the manufacturer completes the packaging,

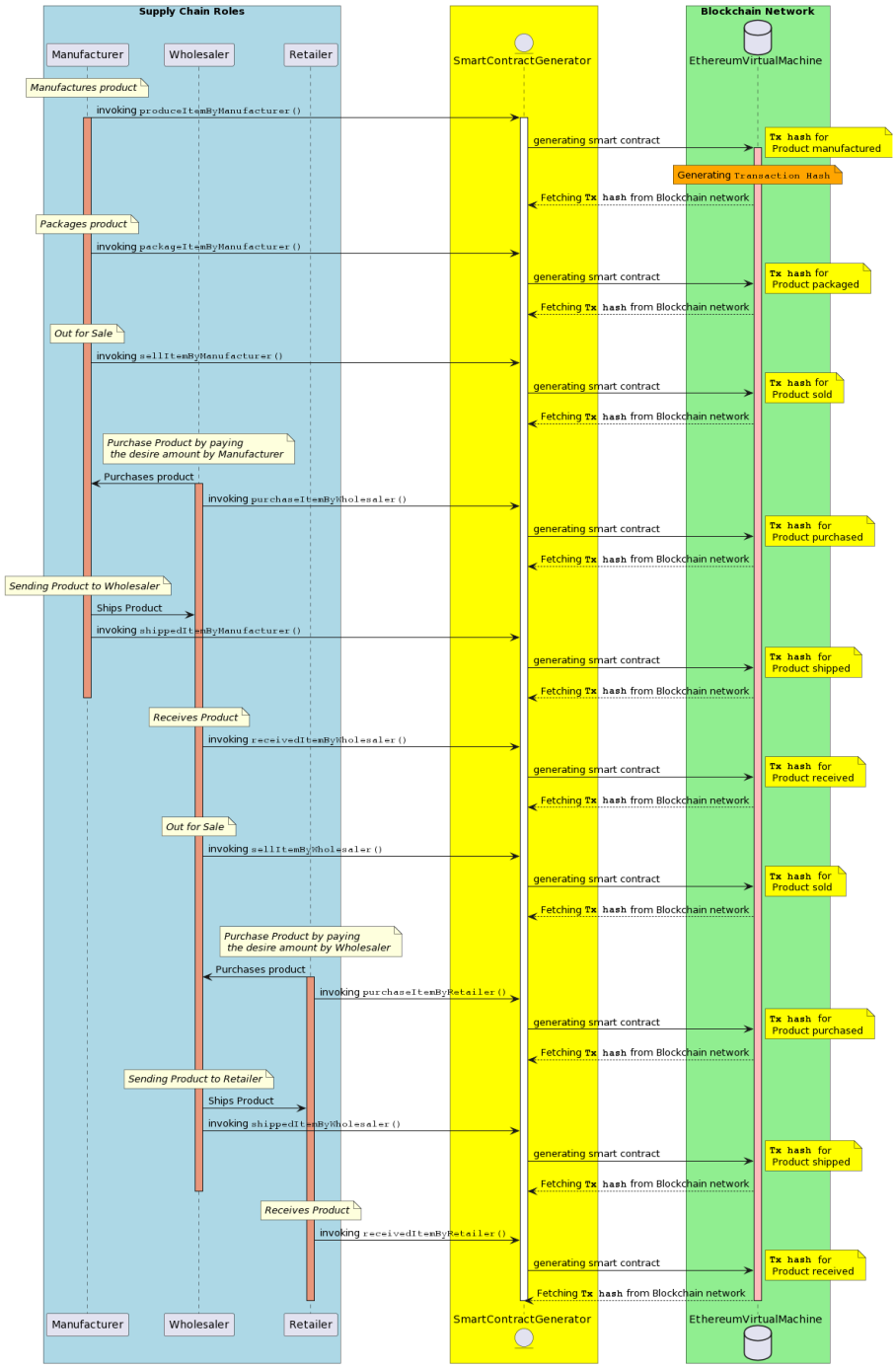


Figure 4.2: Smart Contract Sequence Diagram

(3) the product is placed on the market, (4) the wholesaler purchases the product, (5) the 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. 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. Later, while integrating smart contracts with agents, the state element was removed to make the supply chain adaptive because there can be sometimes a scenarios when a manufacturer, wholesaler or both will be not a part of the particular supply chain due to the availability of sufficient inventory. Scenarios pertaining to these possibilities are added in the subsequent chapter as a result.

Solidity language is used to create smart contracts, while JavaScript is used for testing. The `.sol` files were compiled using Solidity v0.8.13, to retrieve `.abi` and `.bin` files for further implementation and deployment. 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 4.3 was used as a reference for developing smart contracts connected to the 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. These events are also covered earlier in this chapter.

To understand the model of the application for smart contracts and the files organized and classes imported using inheritance in-depth, figure 4.4 should be taken into consideration. Solidity contracts can make use of a particular type of format 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[1]. Additionally, third-party tool annotations can be also 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.

Aside from Solidity, there are several more languages available for building smart contracts. Vyper was also chosen to construct smart contracts, which is now one of DeFi's most popular

[1] Check more about NatSpec : https://docs.soliditylang.org/en/v0.8.17/natspec-format.html

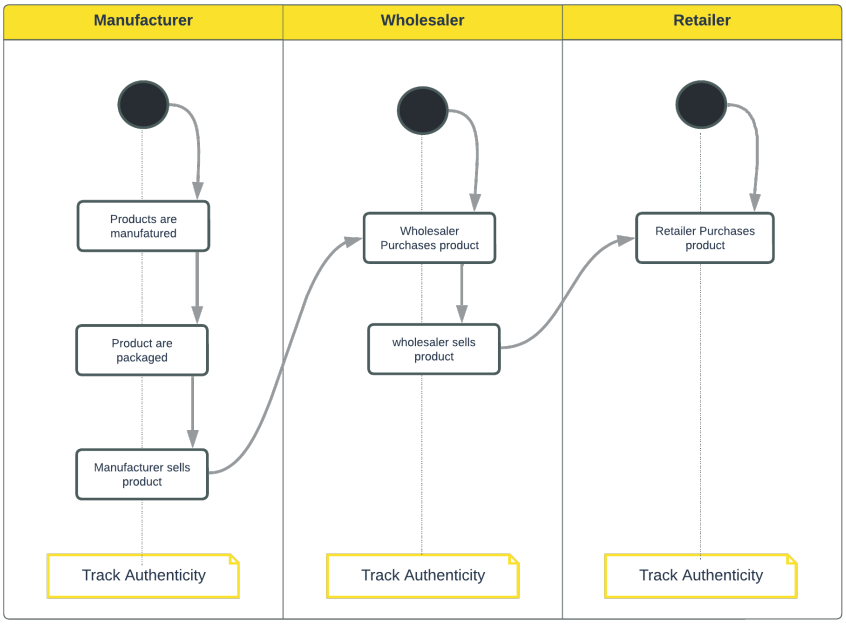


Figure 4.3: Supply Chain Activity Diagram

languages. Vyper is a high-level programming language identical to Python. However, due to its limitations over Solidity, the proposal was subsequently abandoned.

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. An agent program is run by the Jason framework. 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 agent within the supply chain will act autonomously , see figure 4.5 to get an idea of how the agents are communicating according to their beliefs, updating them, and executing plans. The `retailerAgent` will work to ensure that the product is available in the warehouse before ordering it from the `wholesalerAgent`, who will then ask the `wholesalerAgent` to check its inventory and if insufficient stock then, get in touch with the `manufacturerAgent` to produce the product, and ship it to the `wholesalerAgent`, who will then send it to the `retailerAgent`. The quantity of stock available and the quantity ordered will be set as belief for each agents as '`inventory(A)`.' and '`order(B)`.', respectively and it will be compared while checking the warehouse before selling product. There will be a

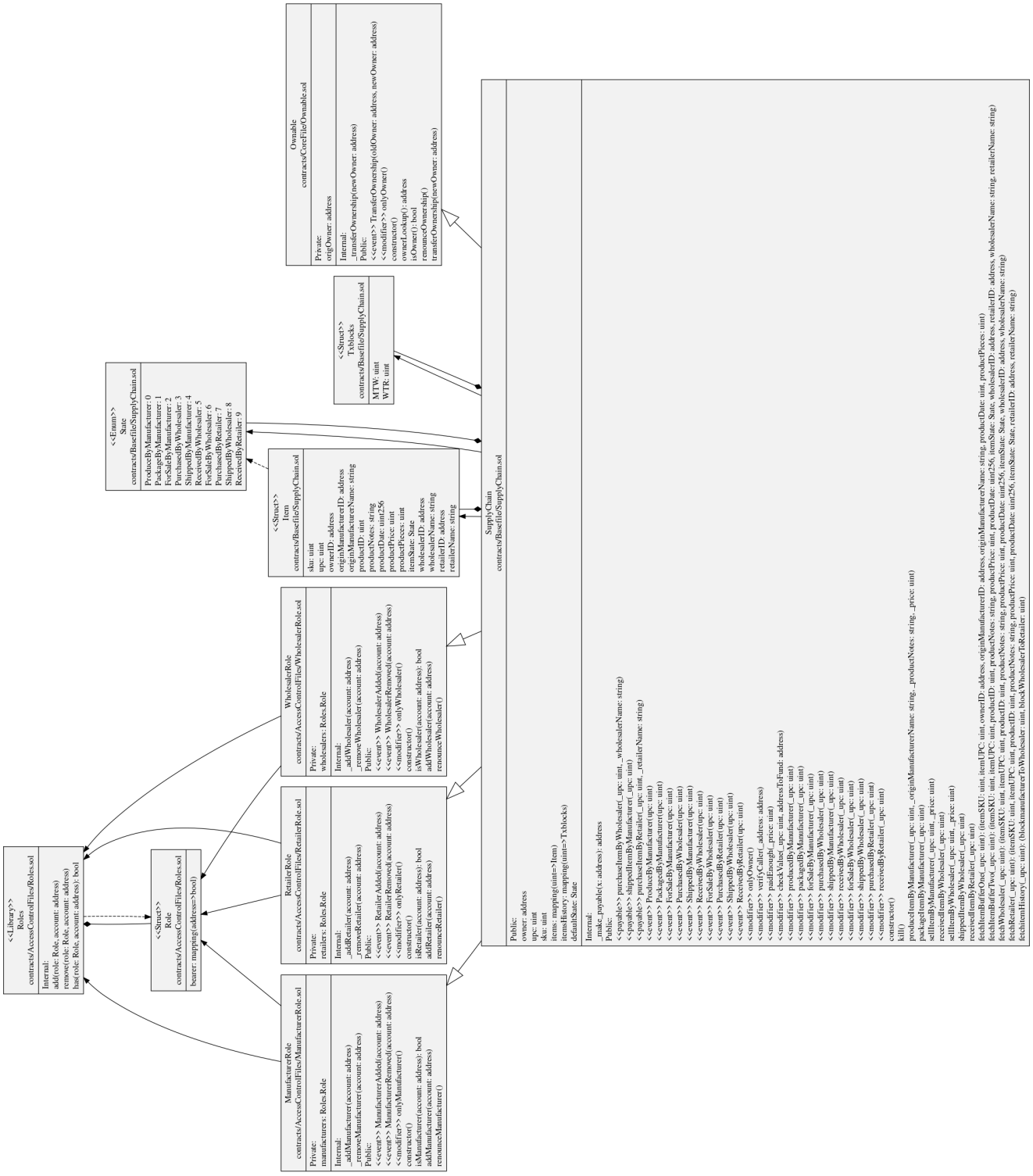


Figure 4.4: Smart Contract Class Diagram

`mainAgent`, 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"

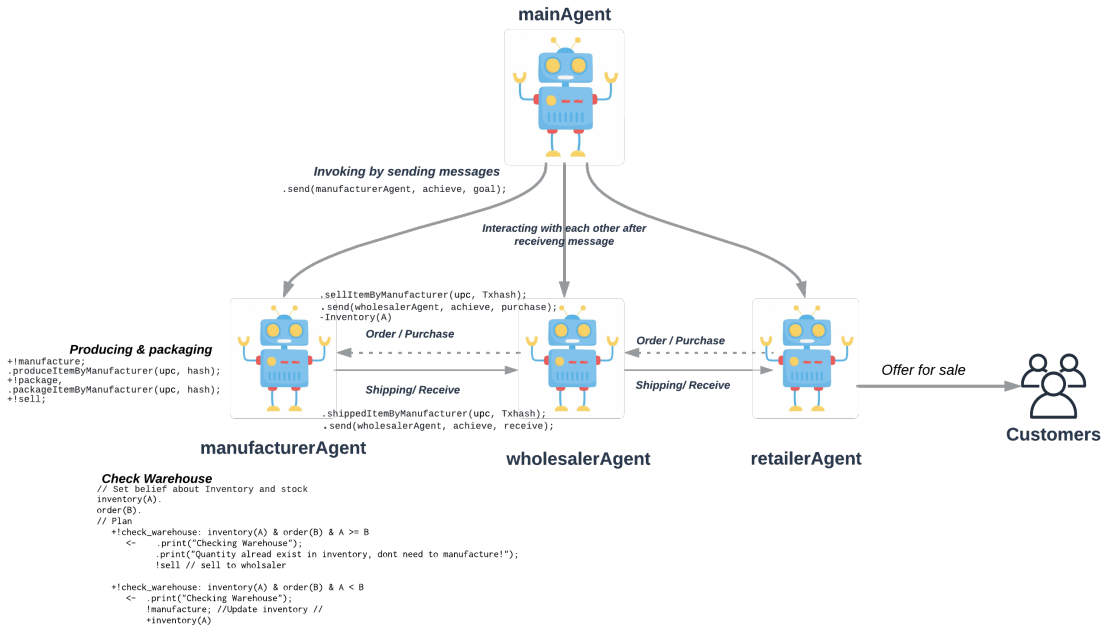


Figure 4.5: Agent Interaction in Supply chain

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, and put them appropriately as actions within the suitable plans. 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 4.5. The sequence of their proper interaction is 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;
- `retailerAgent` will inspect its inventory and sell products to customers; if a product is not in stock, `retailerAgent` will ask the `mainAgent` to contact the `wholesalerAgent`;

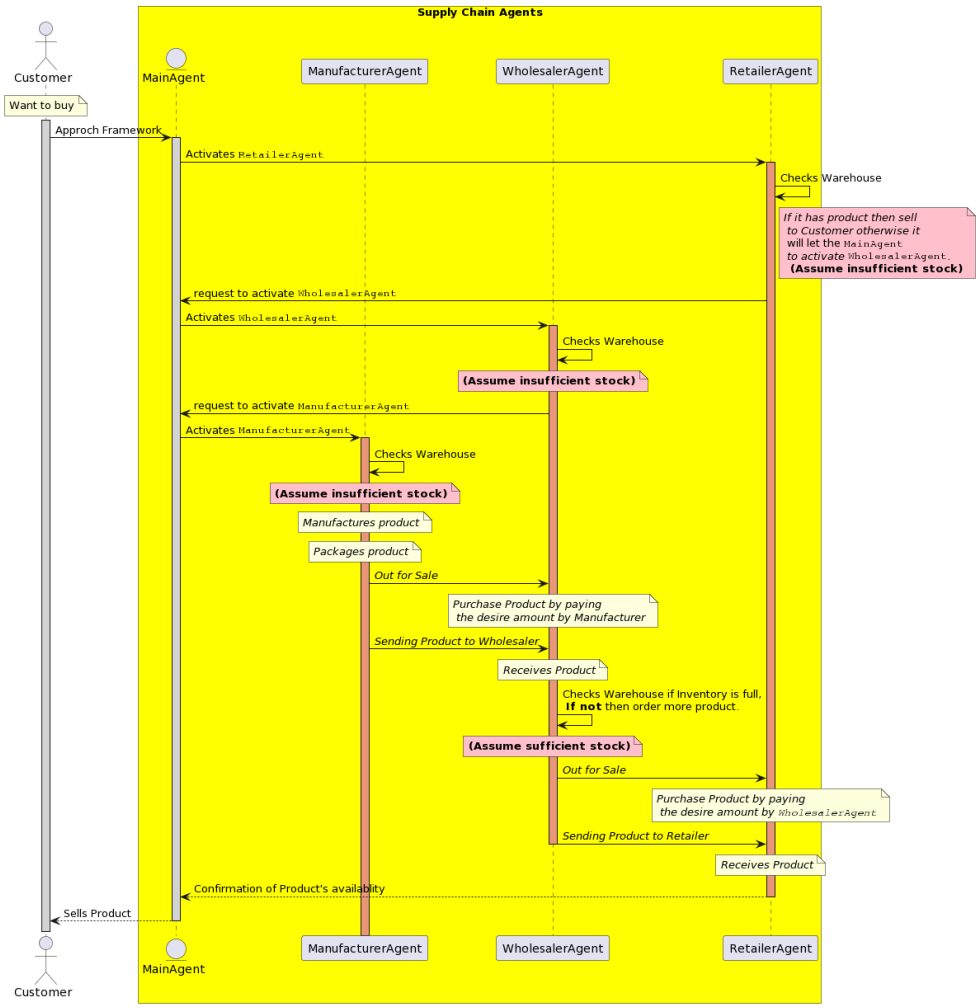


Figure 4.6: Agent Sequence Diagram

- The `wholesalerAgent` will check its warehouse and ship the product to the `retailerAgent`; if the product is not available, the `wholesalerAgent` will request that the `manufacturerAgent` be contacted by the `mainAgent`;

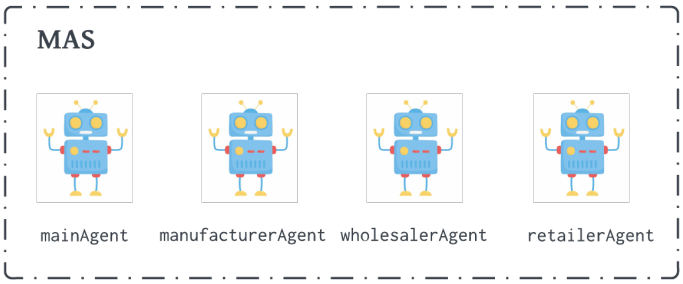
- Upon checking its inventory, the `manufacturerAgent` will ship the product to the `wholesalerAgent`. If the product is not in stock, the `manufacturerAgent` will manufacture the product, does the package and deliver it to the `wholesalerAgent`.

- `WholesalerAgent` will strive to keep its inventory full so that it does not have to order every time, therefore it will check its warehouse and order more if it is not full.

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

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




```
# Adding actions for agents
@actions.add_funtion(":function_name")
def fun_name():

# Creating a MAS using agents
env = agentspeak.runtime.Environment()

with open(os.path.join(os.path.dirname(__file__), "asl/mainAgent.asl")) as source:
    agents = env.build_agents(source, 2, actions)

with open(os.path.join(os.path.dirname(__file__), "asl/manufacturerAgent.asl")) as source:
    agents.append(env.build_agent(source, actions))

with open(os.path.join(os.path.dirname(__file__), "asl/wholesalerAgent.asl")) as source:
    agents.append(env.build_agent(source, actions))

with open(os.path.join(os.path.dirname(__file__), "asl/retailerAgent.asl")) as source:
    agents.append(env.build_agent(source, actions))
```

Figure 4.7: Agents in MAS

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 4.5 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, 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 Application Programming Interface (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 Jason framework is similar to Jason framework with Java, 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 as shown in figure 4.7.

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 4.5, 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 application flow is developed in accordance with figure 4.8, which is an extension of figure 4.2 and figure 4.6.

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. The attempt of linking each framework with each web3 library is detailed below.

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 framework, which is based on Java. Through the command line tools, 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 can be produced. When the wrapper is created, these addresses are from the truffle deployment.

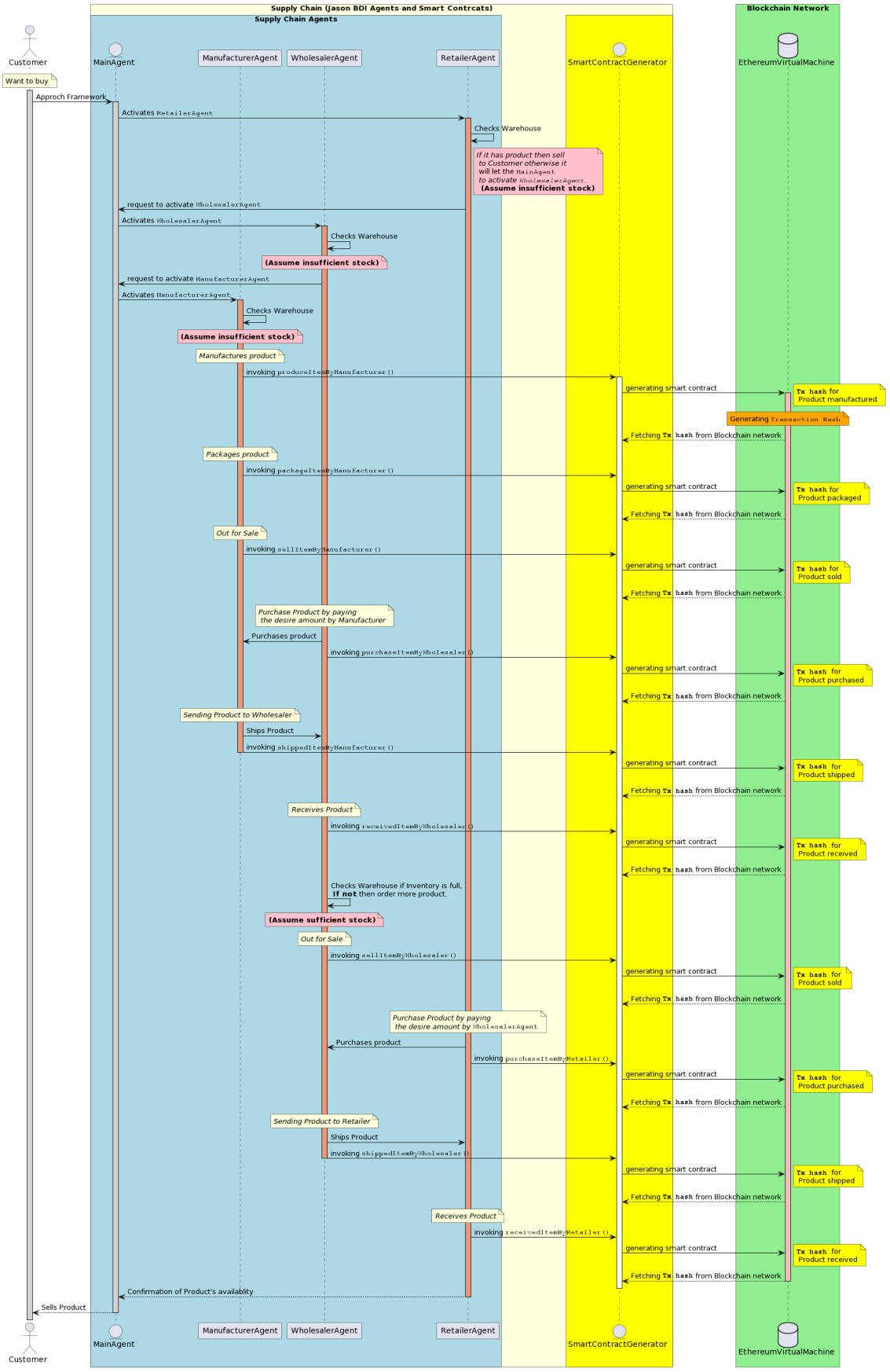


Figure 4.8: Sequence Diagram for BDI agents and Smart Contracts

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 framework 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.

The following excerpts demonstrate how to utilize smart contracts with previously deployed contract addresses and link to the Web3 library.

```
from web3 import Web3
web3 = Web3(Web3.HTTPProvider(ganache_url))
contract_address = web3.toChecksumAddress("0xc9f78D73aCAf603Fe2319682316268A39Cc5CBB7")
contract = web3.eth.contract(address=contract_address, abi=abi)
print(f'Deployed_Contract_Address:_{contract.address}')
```

The code snippet below demonstrates how to define a specific action regarding product manufacturing i.e., `produceItemByManufacturer` and store the action in blockchain network when performed by the agent. BDI agents created using the Jason framework are capable of performing a variety of actions, and it is also possible to create the custom actions that help us to integrate the smart contract functions with the agents.

```
actions = agentspeak.Actions(agentspeak.stdlib.actions)
@actions.add_function(".produceItemByManufacturer", (int, ))
def produceItemByManufacturer(upc):
    tx1 = contract.functions.produceItemByManufacturer(upc, "manufacturer_name", "product", 10).
        transact({
         'from': manufacturer_id
        })
    tx_receipt = web3.eth.wait_for_transaction_receipt(tx1)
    hash = tx_receipt.transactionHash.hex()
    return hash
```

The code sample below demonstrates how specific agents' `.asl` files can call the functions specified in the actions declared for those agents in the main file to execute a plan and accomplish goals.

```
+!manufacture: true
<-  .print("Manufacturing Product");
    .produceItemByManufacturer(upc, X);
    .print("Tx produceItemByManufacturer successful with hash:", X);
    !package;
    .wait(1000).
```

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 the each agent were written with belief sets and plans with goals and a MAS was produced using Python with AgentSpeak as shown in figure 4.7, 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. With the exception of a few common modules Jython applications 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 constrBeforeapplicationa. Before integrating the technologies, each technique is evaluated independently.

5.1 Multiple Agent System 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 of the tables. Then, 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 a 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-basa ed interpreter with `web3` package, we immediately switched to ASTRA and Jason with a Pythonased interpreter. However, all of them produced the same outcome as described above, despite the fact that the scripting of agents and the MAS enviroSUPPLY CHAINred.

ASTRA agents, unlike Jason, are not written in the `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 expended through `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 agent's 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.

5.2 Smart Contract Implementation

Solidity-based smart contracts were created to integrate 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. Each test case examines a variety of factors to determine whether the smart contracts are being executed by the correct contract owner. The example of verifying one smart contract is shown in the code fragment below.

```
        assert.equal(resultBufferOne[0], sku, 'Error:_Invalid_item_SKU')
        assert.equal(resultBufferOne[1], upc, 'Error:_Invalid_item_UPC')
        assert.equal(resultBufferOne[2], ownerID, 'Error:_Missing_or_Invalid_ownerID')
        assert.equal(resultBufferTwo[2], productID,'Error:_Missing_or_Invalid_productID')
        assert.equal(resultBufferOne[3], originManufacturerID, 'Error:_Missing_or_Invalid_originManufacturerID')
        assert.equal(resultBufferOne[4], originManufacturerName, 'Error:_Missing_or_Invalid_originManufacturerName')
        assert.equal(resultBufferTwo[6], itemState, 'Error:_Invalid_item_State')
        assert.equal(eventEmitted, true, 'Invalid_event_emitted')
```

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, 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 'LocalNetwork'.
Compiling your contracts...
===========================
> Compiling ./contracts/SupplyChain.sol
> Artifacts written to /tmp/test--842330-COAk5X3aAlas
> Compiled successfully using:
   - solc: 0.8.13+commit.abaa5c0e.Emscripten.clang
<----------------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

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.

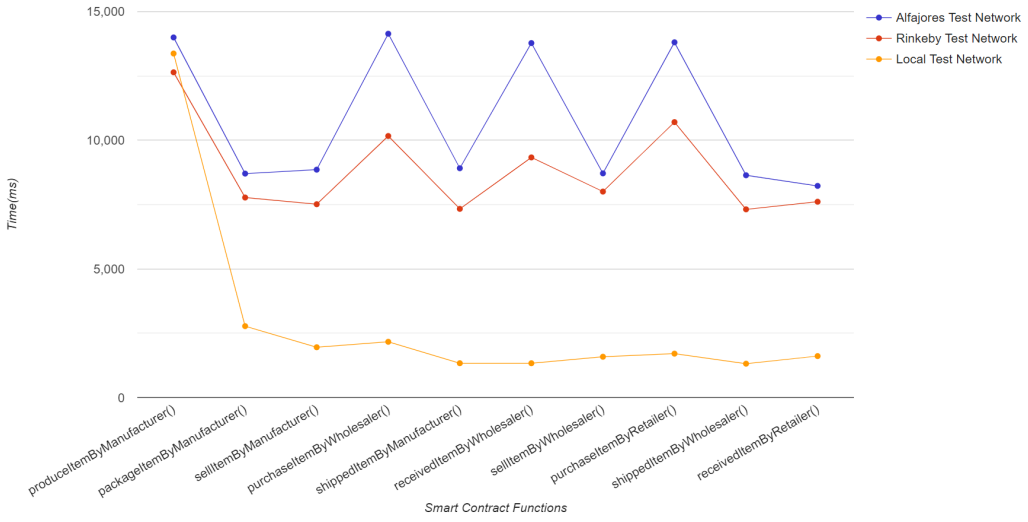


Figure 5.1: Performance across various networks

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.

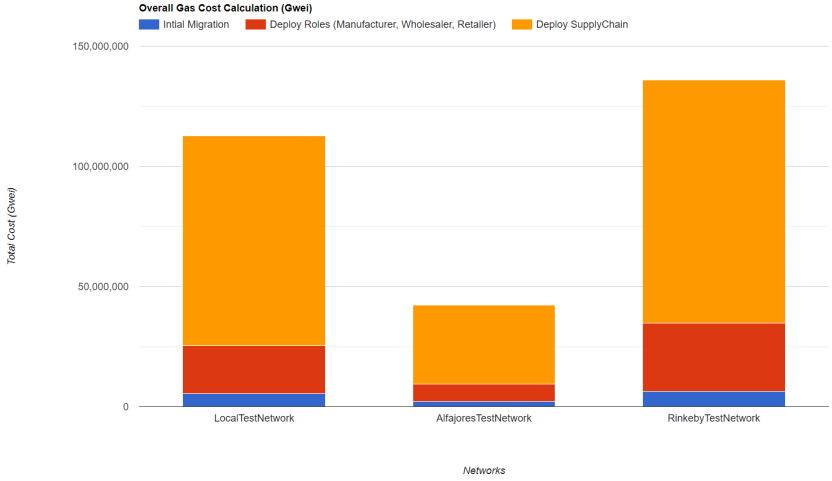


Figure 5.2: Overall deployment gas cost calculation

The entire amount of gas utilized on a public testnet, such as alfajores and rinkeby, will depend on the network's present condition and resource demand. Due to the greater volume of transactions and contracts being executed on the public testnet, the overall gas consumption can be higher than on a local test network. In light of the fact that gas prices change on the public testnet as well, the total amount of gas utilized will also rely on those prices at the time of deployment and transaction. In figure 5.2, The most expensive test network is rinkeby, however even though its gas consumption was same as alfajores test network's, alfajores' overall cost decreased to approximately one-third of rinkeby test network's cost due to low gas prices which was just 7.5 Gwei or 7500000000 Wei compared to rinkeby test network's gas price which was 20 Gwei or 20000000000 Wei.

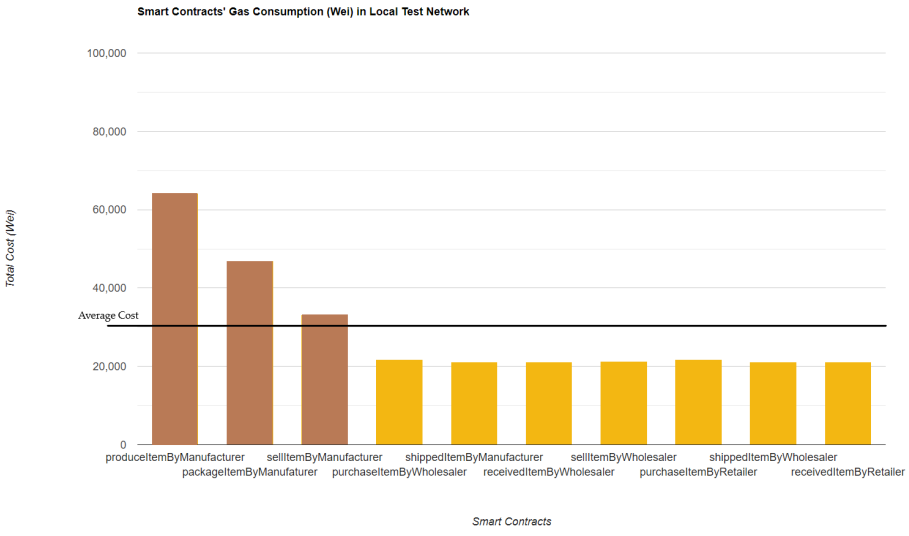


Figure 5.3: Gas consumed by each smart contract

The intricacy of the code, the volume of data being processed, and the present status of the network are some of the variables that affect how much gas will be required overall for a transaction or the deployment of a smart contract on a blockchain network. Because a local test network is less busy and has fewer resource demands than a public testnet like alfajores or rinkeby, the overall amount of gas utilized there is probably lower. Therefore, we selected the local test network to determine how much gas was used to generate each of the smart contracts as shown in figure 5.3. We determined that the overall quantity of gas utilized might still vary depending on how much data is handled and how complicated the code is.

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.

5.3 Infuse Blockchain Technology in Multiple Agent System

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.

Also, regarding the quantity of stock in the warehouse and the batch order for the product, there are several scenarios that might occur as shown in figure 4.5 and figure 4.6. The results of each scenario are displayed below:

- Retailer's warehouse has sufficient product, so `RetailerAgent` doesn't need to order from `WholesalerAgent`.

```
<----------------------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---------------------->
mainAgent           : Starting SupplyChain with SmartContracts
mainAgent           : Hi, I am the owner of Contract, with account: 0xad0BC114B5CF3F0797346fF1Fb1Daf1Cf5123395
mainAgent           : Creating RetailerAgent
retailerAgent       : Hi, I am here, with account: 0xc7D1C50D87B82E85b959DBC2cD9959bfc0480A5E
retailerAgent       : Checking Warehouse, and no need to order
retailerAgent       : Giving products to supplyChainAgent
mainAgent           : Selling to customers
mainAgent           : SUPPLYCHAIN COMPLETE
```

- Retailer's warehouse has insufficient product, so `RetailerAgent` need to order from `WholesalerAgent` and Wholesaler's warehouse has sufficient product, so `WholesalerAgent` doesn't need to order from `ManufactureAgent`.

```
<----------------------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---------------------->
mainAgent           : Starting SupplyChain with SmartContracts
mainAgent           : Hi, I am the owner of Contract, with account: 0xad0BC114B5CF3F0797346fF1Fb1Daf1Cf5123395
mainAgent           : Creating RetailerAgent
retailerAgent       : Hi, I am here, with account: 0xc7D1C50D87B82E85b959DBC2cD9959bfc0480A5E
retailerAgent       : Checking Warehouse
retailerAgent       : INSUFFICIENT INVENTORY!! Ordering Product..
retailerAgent       : Ordering to wholesalerAgent
mainAgent           : Creating WholesalerAgent
wholesalerAgent     : Hi, I am here, with account: 0x5fB0Cd136C7A19E8E12F062548002B4460B0dC0d
wholesalerAgent     : Checking Warehouse, and no need to order
wholesalerAgent     : Selling product to retailerAgent
wholesalerAgent     : Tx sellItemByWholesaler successful with hash: 0
      x421c5ed80a71376a4e8e4e7b6a4083bd8e83f734ed43c67130847eab0f2c7fdf
retailerAgent       : Purchasing product from wholesalerAgent
retailerAgent       : Tx purchaseItemByRetailer successful with hash: 0
      xc07aba5f53bb5d74ce1aea3d788d0c65cb34a036a1459750e9d2db6d9e81cda0
wholesalerAgent     : Shipping product to retailerAgent
wholesalerAgent     : Tx shippedItemByWholesaler successful with hash: 0
      x33e0b25c8d44b837b573ed00e1fe34315c6de613fcacfa2856a4f1cb10671be3
retailerAgent       : Received product from wholesalerAgent and Inventory full!!
retailerAgent       : Tx receivedItemByRetailer successful with hash: 0
      xcc84c9fe3cd74b803419e4dde53f71f0d01243ea0d4ec1873b5aadb95cba23e2
retailerAgent       : Checking Warehouse, and no need to order
retailerAgent       : Giving products to supplyChainAgent
mainAgent           : Selling to customers
mainAgent           : SUPPLYCHAIN COMPLETE
```

- Retailer's warehouse has insufficient product, so `RetailerAgent` need to order from `WholesalerAgent`, Wholesaler's warehouse has also insufficient product, so `WholesalerAgent` need to order from `ManufactureAgent` and Manufacture's warehouse has sufficient product, so `ManufactureAgent` doesn't need to manufacture product.

```
<----------------------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---------------------->
mainAgent            : Starting SupplyChain with SmartContracts
mainAgent            : Hi, I am the owner of Contract, with account: 0xad0BC114B5CF3F0797346fF1Fb1Daf1Cf5123395
mainAgent            : Creating RetailerAgent
retailerAgent        : Hi, I am here, with account: 0xc7D1C50D87B82E85b959DBC2cD9959bfc0480A5E
retailerAgent        : Checking Warehouse
retailerAgent        : INSUFFICIENT INVENTORY!! Ordering Product..
retailerAgent        : Ordering to wholesalerAgent
mainAgent            : Creating WholesalerAgent
wholesalerAgent      : Hi, I am here, with account: 0x5fB0Cd136C7A19E8E12F062548002B4460B0dC0d
wholesalerAgent      : Checking Warehouse
wholesalerAgent      : INSUFFICIENT INVENTORY!! Ordering Product..
wholesalerAgent      : Ordering to manufacturerAgent
mainAgent            : Creating ManufacturerAgent
manufacturerAgent    : Hi, I am here, with account: 0x4A9fe326Edc88F1f22940DC9F70BD391fB4218f8
manufacturerAgent    : Checking Warehouse
manufacturerAgent    : Quantity alread exist in inventory, dont need to manufacture!
manufacturerAgent    : Selling product to wholesalerAgent
manufacturerAgent    : Tx sellItemByManufacturer successful with hash: 0
      x95c053089529888fb8c399355e666e6827935cc1b3e45b4bce90ccbc07eccd6b
wholesalerAgent      : Purchasing product from manufacturerAgent
wholesalerAgent      : Tx purchaseItemByWholesaler successful with hash: 0
      x28719d94c2f40383a74c25d39859dc4a1f26b06d0adb6d39f5e20ab7a5f6f499
manufacturerAgent    : Shipping product to wholesalerAgent
manufacturerAgent    : Tx shippedItemByManufacturer successful with hash: 0
      x5cbf427497b18c441dc7d44808cfd522c1f62e41a0c7c8524acb63a8aa8b4461
wholesalerAgent      : Received product from manufacturerAgent, and added to the inventory and Inventory FULL!!
wholesalerAgent      : Tx receivedItemByWholesaler successful with hash: 0
      x3ff03e21cd1b42eea4a55504dca701cd16484408acd9831f7f6a5df216f46e2e
wholesalerAgent      : Checking Warehouse, and no need to order
wholesalerAgent      : Selling product to retailerAgent
wholesalerAgent      : Tx sellItemByWholesaler successful with hash: 0
      x5a786f97f9ff516c319e1473c8fff3b850c68010ea37ea32778f8f53e6704090
retailerAgent        : Purchasing product from wholesalerAgent
retailerAgent        : Tx purchaseItemByRetailer successful with hash: 0
      xb2a75b53a5f4ee86551a6a31d8324fd807376c2bc03a0a6b3d0585f5819be626
wholesalerAgent      : Shipping product to retailerAgent
wholesalerAgent      : Tx shippedItemByWholesaler successful with hash: 0
      x3fc8c3f81c033257ab58c5f9ca09197e7eafe7a61a6df4ba9b60e054ed7fb42a
retailerAgent        : Received product from wholesalerAgent
retailerAgent        : Tx receivedItemByRetailer successful with hash: 0
      x70660ae51b082f39a551e8c0132058c1bbc557267826ecd81ed59006025b5722
retailerAgent        : Checking Warehouse, and no need to order
retailerAgent        : Giving products to supplyChainAgent
mainAgent            : Selling to customers
mainAgent            : SUPPLYCHAIN COMPLETE
```

- Retailer's warehouse has insufficient product, so `RetailerAgent` need to order from `WholesalerAgent`, Wholesaler's warehouse has also insufficient product, so `WholesalerAgent` need to order from `ManufactureAgent` and Manufacture's warehouse has also insufficient product, so `ManufactureAgent` needs to manufacture product. In this scenario, `WholesalerAgent` ordered again because it wanted to make its inventory full.

```
<----------------------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---------------------->
mainAgent            : Starting SupplyChain with SmartContracts
mainAgent            : Hi, I am the owner of Contract, with account: 0xad0BC114B5CF3F0797346fF1Fb1Daf1Cf5123395
mainAgent            : Creating RetailerAgent
retailerAgent        : Hi, I am here, with account: 0xc7D1C50D87B82E85b959DBC2cD9959bfc0480A5E
retailerAgent        : Checking Warehouse
retailerAgent        : INSUFFICIENT INVENTORY!! Ordering Product..
retailerAgent        : Ordering to wholesalerAgent
mainAgent            : Creating WholesalerAgent
wholesalerAgent      : Hi, I am here, with account: 0x5fB0Cd136C7A19E8E12F062548002B4460B0dC0d
wholesalerAgent      : Checking Warehouse
wholesalerAgent      : INSUFFICIENT INVENTORY!! Ordering Product..
wholesalerAgent      : Ordering to manufacturerAgent
mainAgent            : Creating ManufacturerAgent
```

```
manufacturerAgent    : Hi, I am here, with account: 0x4A9fe326Edc88F1f22940DC9F70BD391fB4218f8
manufacturerAgent    : Checking Warehouse
manufacturerAgent    : INSUFFICIENT INVENTORY!! Manufacturing Product..
manufacturerAgent    : Tx produceItemByManufacturer successful with hash: 0
      xde3468f453048739395cfe770a2be70654ac05d7fe006fd77ad2e91e55cd0fb5
manufacturerAgent    : Packaging Product
manufacturerAgent    : Tx packageItemByManufacturer successful with hash: 0
      xac339f4c9d1bd18c206c4fefa2da1578255ab184cfb56a19aa5f0c749691c90f
manufacturerAgent    : Checking Warehouse
manufacturerAgent    : Quantity alread exist in inventory, dont need to manufacture!
manufacturerAgent    : Selling product to wholesalerAgent
manufacturerAgent    : Tx sellItemByManufacturer successful with hash: 0
      x70d3e37c5cae73f9e3721dd1cec079b2329f241d035bc3c0d8d5435c9980291a
wholesalerAgent      : Purchasing product from manufacturerAgent
wholesalerAgent      : Tx purchaseItemByWholesaler successful with hash: 0
      x45057874c25a62ad149fc3e984a6856fc8ca3865fe8f1faa558e237b84437a87
manufacturerAgent    : Shipping product to wholesalerAgent
manufacturerAgent    : Tx shippedItemByManufacturer successful with hash: 0
      x9ae2c0454a62266658d252827a852d8c4f2636ce106e5fe122c0a0fce1f49518
wholesalerAgent      : Received product from manufacturerAgent, and added to the inventory and and Inventory not full yet!!
wholesalerAgent      : Tx receivedItemByWholesaler successful with hash: 0
      x7b2b32121504e0c424faf0721aba7dde1152781d8e6ca099146bca2a99321863
manufacturerAgent    : Selling product to wholesalerAgent Again
manufacturerAgent    : Tx sellItemByManufacturer successful with hash: 0
      xb5b7ed54756b9123522d942c131f52a79450ed8a61b5b5f17b5e762c4dfd2c96
wholesalerAgent      : Purchasing product from manufacturerAgent Again
wholesalerAgent      : Tx purchaseItemByWholesaler successful with hash: 0
      x0000ce3707589a17982b27b0efb13872ffc8b2c8ff509e3783f19fa2e81a2263
manufacturerAgent    : Shipping product to wholesalerAgent
manufacturerAgent    : Tx shippedItemByManufacturer successful with hash: 0
      xdc4e1db2675c4a85b20eca1ce2e94ce9064ae8ec4d7160c23f5dc3e682e714b1
wholesalerAgent      : Received product from manufacturerAgent, and added to the inventory and Inventory FULL!!
wholesalerAgent      : Tx receivedItemByWholesaler successful with hash: 0
      x130f6ac86210088f05687b9d769615856bc0da2459c5ff68640330c2e8ffcf66
wholesalerAgent      : Tx receivedItemByWholesaler successful with hash: 0
      xe37eb45b7918a377f96daaf95be30432d2c8902f3033c0809fc3a1ec0970bcf5
wholesalerAgent      : Checking Warehouse, and no need to order
wholesalerAgent      : Selling product to retailerAgent
wholesalerAgent      : Tx sellItemByWholesaler successful with hash: 0
      x6aaf996c1a2347612b38350cc31c20320d25d21da80878be1776cbd1f8138258
retailerAgent        : Purchasing product from wholesalerAgent
retailerAgent        : Tx purchaseItemByRetailer successful with hash: 0
      x18677b7134c6ee338ac749c090e4121f2fa80308837da9f1c8f13955841e411a
wholesalerAgent      : Shipping product to retailerAgent
wholesalerAgent      : Tx shippedItemByWholesaler successful with hash: 0
      x45e406258e946a2eee2969dbf5461c799d036190598677a3b787b3e93bbf9ddd
retailerAgent        : Received product from wholesalerAgent
retailerAgent        : Tx receivedItemByRetailer successful with hash: 0
      x24d53dcdb1b44e2b86938daa9df29ad8efd7f40718a73d0f3a4aff9437c2f68f
retailerAgent        : Checking Warehouse, and no need to order
retailerAgent        : Giving products to supplyChainAgent
mainAgent            : Selling to customers
mainAgent            : SUPPLYCHAIN COMPLETE
```

The output above demonstrates how the various supply chain roles interact with one another to complete the process of planning, manufacturing, and delivering a product or service. It demonstrates that each role has been given an `account number` and that whenever they interact or complete a procedure, a `transaction hash` is issued to verify its authenticity and for future use.

After a successful implementation, we remained adamant about 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, but it was unable to import the package `org.python`, which is necessary to start the program.

6 Discussion

The chapter is dedicated to the self-discussion of the main concepts discovered when exploring the research topics, as well as the understanding of research, implementation, and findings discovered during the thesis.

Beginning with the fundamentals, we investigated the distinctions between the AgentSpeak(L), AgentSpeak model, and BDI model. AgentSpeak(L) (a high-level programming language) is created on top of the AgentSpeak model, which is itself based on the BDI model. Following the BDI model, which is a theoretical framework, that defines how agents choose, reason, and behave by their beliefs, desires, and intentions. The AgentSpeak model extends the BDI model by including a collection of libraries and tools for dealing with events and triggers. AgentSpeak(L) provides a collection of libraries and tools for building agents based on the AgentSpeak model.

Our study throughout the literature review assisted us in exploring various frameworks that can be utilized to create BDI-based agents. The table 6.1 compares Jason, ASTRA, and various BDI model frameworks. From table 6.1, it's important to note that ASTRA is a research initiative rather than a programming language to create a research agenda for the creation of agent-based systems and technologies. GOAL framework can be used to create MAS as well as a multi-agent programming environment. The initial difference between both terminologies is that a MAS is a system composed of numerous autonomous agents that collaborate to achieve mutual goals. A MAS's agents might be software, hardware, or a combination of the two, and a multi-agent programming environment, on the other hand, is a software framework or toolkit that makes MAS creation, testing, and deployment easier. It includes a collection of tools and libraries that allow programmers to construct and control the behavior of numerous agents, as well as the communication and coordination mechanisms that allow them to communicate and coordinate with one another. In summary, a MAS is the finished outcome, whereas a multi-agent programming environment is a tool used to create it.

Developers can use LightJason to construct and manage large and sophisticated systems of autonomous agents that can collaborate to deal with challenges, make decisions, and achieve goals. Message passing, broadcasting, and a centralized management system are

among the framework's sophisticated features for controlling and coordinating the activity of various agents. LightJason also offers parallel and distributed computing capability, making it simple to create scalable and efficient multi-agent systems.

Table 6.1: BDI model frameworks

Framework	Language	Features
Jason (Java)	AgentSpeak, Java	Based on Concurrent programming, includes speech-act based inter-agent communication, first fully-fledged interpreter for a much enhanced version of AgentSpeak
Jason (Python)	AgentSpeak, Python	Based on Event-based programming, includes speech-act based inter-agent communication, first fully-fledged interpreter for a much enhanced version of AgentSpeak
Agent Factory	ASTRA	Based on Jason, logic-based agent programming language, combines AgentSpeak(L) with teleo-reactive functions
GOAL	Prolog as a knowledge representation language	Based on basic practical reasoning and common sense notions, design and implement cognitive agents
JADEX	Java	Extension of JADE to build rational agents, focuses on web services in a new version, called ActiveComponents implemented using Java
JACK	JACK Agent Language, a super-set of Java	Provides graphical planning tool and its own java based plan language
BRAHMS	Graphical user interface, custom syntax to define the agents and the environment	Simulate and develops multiagent model of machine and human behaviour
2APL	2APL (logic-based, similar to Prolog)	2APL interpreter is built on JADE (collection of tools and libraries to develop, deploy and execute MAS)
BDI4JADE	Java	Provide infrastructure like message exchange and agent modularity
JaCaMo	AgentSpeak, Jason	Combining CartAgO (writing environment artifacts), Moise (programming multi-agent organisation) and Jason (creating agents)
LightJason	AgentSpeak(L++), Java	Includes Lambda-expression, multi-plan and multi-rule definition, explicit repair actions, multiple variable assignments parallel execution and thread-safe variable

Jason with Python is a port of the original Java-based Jason Framework. It enables Python programmers to create MAS and agents that are BDI-based. It offers an event-based programming model, which enables programmers to create reactive code that reacts to outside events, as well as a high-level, user-friendly, expressive programming model for creating MAS. Jason with Java is the Jason Framework's initial implementation. The Java programming language is used to create BDI-based agents and MAS, and it offers a concurrent programming approach that enables developers to create code that can manage numerous tasks at once. Debugging and visualization tools are among the several libraries and tools it offers for creating and deploying MAS. There are more BDI frameworks; this table is not all-inclusive.

We also explored and delved deeply into the subject of smart contracts. To create the smart contracts, we explored numerous languages and had to choose between Solidity and Vyper. The contrasts between Solidity and Vyper were clearly illustrated in table 6.2. As a result

of the comparison, we came to the realization that Solidity is a more feature-rich language that is better suited for complex contract construction, while Vyper is more streamlined and security-focused, making it more appropriate for auditing and basic contract writing. In addition to the language used, as we shown in the previous chapter, the kind of network, the complexity of the code, and the amount of data being processed all affect how much gas is consumed for deployment and contract execution.

Table 6.2: Solidity v/s Vyper

Feature	Solidity	Vyper
Syntax	Similar to JavaScript and Python	Similar to Python
Feature Set	More extensive, including inheritance and libraries	More minimalistic, focused on security
Audibility	Less auditable	More auditable
Gas consumption	Can consume more gas	Can consume less gas
Support for complex user-defined types	Yes	No
Support for libraries	Yes	No

Our study and the results from our implementation led us to the evidence that agents can use smart contracts to coordinate their beliefs, desires and intentions with rules, which provides the explanation to our RQ1. Smart contracts must have particular characteristics or qualities in order to coordinate with agents' ideas, desires, and intentions. Smart contracts should be adaptable enough to accept changes in agents' beliefs, wants, and intentions over time. Smart contracts should be transparent, allowing all parties to see the agreement's rules, conditions, and terms. Smart contracts should be self-executing, which means that they automatically enforce the contract's terms and conditions. Smart contracts should also be accessible and clear to all parties involved. This can be accomplished by using basic, unambiguous language and enabling tools and resources to help participants comprehend the agreement's provisions. The rules that control how supply chain participants interact can be referred to as smart contracts. These regulations may specify the terms and circumstances for carrying out certain activities, such as when a retailer may buy products from a wholesaler or when a manufacturer may send goods to a store. By using the smart contract's features and providing the necessary data and settings, agents can communicate with it. The smart contract may then put the rules into action and determine how the interaction will turn out based on the state of the supply chain at the moment and the data provided by the agents.

We came to the conclusion that roles for retailer, wholesaler, and manufacturer can be represented in a Solidity-based supply chain application using BDI model agent library in reference to our RQ2 since it was addressed by our implementation. The BDI agent library and the web3 library can be used to implement MAS on the blockchain in addition to implementing the beliefs, desires, and intents of the agents, which can involve a variety of actions and behaviors in the supply chain.

Additionally, smart contracts can be used to create consensus and dispute resolution processes, such as when a contract or agreement is not fulfilled by all parties or when there

are disagreements. Further, the smart contract may be designed to encourage automated, transparent communication and coordination amongst supply chain agents. All parties in the supply chain may maintain tabs on the movement of goods and adjust their operations accordingly by using smart contracts, for instance, to automatically update inventory levels and shipment status of products.

We also determined that synchronizing AgentSpeak tactics and goals with on-chain smart contract transaction payloads is achievable, albeit the precise implementation would rely on the AgentSpeak library and the blockchain platform employed. The technique we adopted was to utilize the AgentSpeak library to specify the agents' decision-making strategies and goals, and then use smart contracts to manage transaction execution based on those strategies and goals. It ought to be noted that synchronizing AgentSpeak tactics and goals with on-chain smart contract transaction payloads is a difficult operation that would need a thorough grasp of both AgentSpeak and blockchain technology.

7 Conclusion

The proposed application of this research was to build a decentralized supply chain network with various agents interacting with one another and smart contracts regulating the exchange of information and products. As a result, the supply chain process would be more transparent, traceable, and effective.

According to the research described in this thesis, the merge of both technologies can be achieved by combining a library for interfacing with the Ethereum blockchain and a BDI-based framework. Combining these two technologies enables the development of intelligent agents that can communicate and store the interactions in the form of transactions using smart contracts and perform steps depending on data stored on the blockchain.

In conclusion, this thesis has shown the possibility of merging BDI-based multi-agent systems with blockchain technology, specifically smart contracts. The BDI model provides a strong foundation for developing intelligent agents capable of making decisions based on their beliefs, desires, and intentions, while blockchain technology and smart contracts provide a safe and decentralized platform on which these agents can interact. The findings of this study might pave the way for the creation of new decentralized applications and multi-agent systems in various domains.

8 Further Work

As a part of our ongoing effort, it will amazingly enhance the functionality of our application and build an interactive user interface. Another area of future research may be looking at how different BDI frameworks handle agent coordination and communication in a MAS, and how they can be connected with blockchain technology and smart contracts. This could entail investigating various architectures and protocols for agent communication and assessing their influence on the system's overall performance and scalability.

One can experiment with BDI frameworks built using Java such as LightJason, JACK, or 2APL for developing agents in MAS and connecting them with blockchain technology like smart contracts. We also anticipate having a successful implementation of our program using the web3j library, maybe after a significant upgrade to Jason or ASTRA, so that the application can operate using these Java-based frameworks too.

Furthermore, investigating the influence of various BDI frameworks on system security and the capacity to deal with malicious agents would be an essential subject of future research. It would also be interesting to examine the usage of BDI-based multi-agent systems for additional blockchain technology use cases, such as decentralized banking and prediction markets. Future studies may also look at other blockchains outside Ethereum, such as Hyperledger, and Corda, to see how well the method works with various blockchains.

Finally, it would be intriguing to examine the integration of other future technologies such as IoT with the proposed system to improve the functionality and performance of the supply chain network.

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.

[JW95] Nicholas R Jennings and Michael Wooldridge. "Applying agent technology". In: *Applied Artificial Intelligence an International Journal* 9.4 (1995), pp. 357–369.

[Ing+97] James Ingham et al. "What is an Agent". In: *Centre for Software Maintenance University of Durham* (1997).

[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.

[FG03] John France and Ali A Ghorbani. "A multiagent system for optimizing urban traffic". In: *IEEE/WIC International Conference on Intelligent Agent Technology, 2003. IAT 2003.* IEEE. 2003, pp. 411–414.

[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.

[BPL05] Lars Braubach, Alexander Pokahr, and Winfried Lamersdorf. "Jadex: A BDI-Agent System Combining Middleware and Reasoning". In: *Software Agent-Based Applications, Platforms and Development Kits*. Ed. by Rainer Unland, Monique Calisti, and Matthias Klusch. Basel: Birkhäuser Basel, 2005, pp. 143–168. ISBN: 978-3-7643-7348-1.

[MDA05] Viviana Mascardi, Daniela Demergasso, and Davide Ancona. "Languages for Programming BDI-style Agents: an Overview." In: *WOA*. Vol. 2005. Citeseer. 2005, pp. 9–15.

[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.

[SCV07] Maarten Sierhuis, William J Clancey, and Ron JJ Van Hoof. "Brahms: a multi-agent modelling environment for simulating work processes and practices". In: *International Journal of Simulation and Process Modelling* 3.3 (2007), pp. 134–152.

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

[Boi+13] Olivier Boissier et al. "Multi-agent oriented programming with JaCaMo". In: *Science of Computer Programming* 78.6 (2013), pp. 747–761.

[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.

[KB15] Kalliopi Kravari and Nick Bassiliades. "A survey of agent platforms". In: *Journal of Artificial Societies and Social Simulation* 18.1 (2015), p. 11.

[LPG15] Florin Leon, Marcin Paprzycki, and Maria Ganzha. "A review of agent platforms". In: *Multi-Paradigm Modelling for Cyber-Physical Systems (MPM4CPS), ICT COST Action IC1404* (2015), pp. 1–15.

[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.

[Abb+18] Omid R Abbasi et al. "Location Recommendation in Geo-Social Networks: A Human-Centric Agent-Based Approach". In: *Adjunct Proceedings of the 14th International Conference on Location Based Services*. ETH Zurich. 2018, pp. 189–193.

[Asc+18] Malte Aschermann et al. "LightJason, a Highly Scalable and Concurrent Agent Framework: Overview and Application." In: *AAMAS*. 2018, pp. 1794–1796.

[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).

[Nor18] Alex Norta. “Self-aware smart contracts with legal relevance”. In: *2018 International Joint Conference on Neural Networks (IJCNN)*. IEEE. 2018, pp. 1–8.

[RAT18] Yara Rizk, Mariette Awad, and Edward W Tunstel. “Decision making in multiagent systems: A survey”. In: *IEEE Transactions on Cognitive and Developmental Systems* 10.3 (2018), pp. 514–529.

[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.

[SKM19] Sana Sabah Sabry, Nada Mahdi Kaittan, and Israa Majeed. “The road to the blockchain technology: Concept and types”. In: *Periodicals of Engineering and Natural Sciences* 7.4 (2019), pp. 1821–1832.

[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.

[ZL20] Shijie Zhang and Jong-Hyouk Lee. “Analysis of the main consensus protocols of blockchain”. In: *ICT express* 6.2 (2020), pp. 93–97.

[AC21] Rifat Shahriyar Amiangshu Bosu Anindya Iqbal and Partha Chakraborty. *Understanding the motivations, challenges and needs of Blockchain software developers: a survey*. 2021. URL: https://www.researchgate.net/figure/Simplified-diagram-of-a-blockchain_fig1_332715800.

[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.