TU WIEN Informatics

Multi Agent Systems in Discrete Manufacturing – Use Case: Optimising the Tool Life Cycle

BACHELOR'S THESIS

submitted in partial fulfillment of the requirements for the degree of

Bachelor of Science

in

Software and Information Engineering

by

Shahin Mahmody
Registration Number 0926487

to the Faculty of Informatics

at the TU Wien

Advisor: Ao.Univ.Prof.Dr. Wolfgang Kastner

Vienna, 5th March, 2021

Shahin Mahmody	Ao.Univ.Prof.Dr. Wolfgang Kastner

Erklärung zur Verfassung der Arbeit

Shahin Mahmody

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit – einschließlich Tabellen, Karten und Abbildungen –, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

Wien, 5. März 2021

Shahin Mahmody

Kurzfassung

Die Vernetzung von Smart Devices in verschiedensten Bereichen des Lebens schreitet weiter voran. In der Fertigung hat diese Entwicklung zu hohen Erwartungen an eine bevorstehende „vierte Industrielle Revolution“ geführt, welche zu großen Fortschritten in der Automatisierung von industriellen Prozessen führen soll. Indem menschliche Planung und Intervention möglichst durch autonome Steuerung, Überwachung und Korrektur ersetzt wird, soll damit ein höheres Niveau an Optimierung erreicht werden können. Viel theoretische Arbeit ist bereits in mögliche Standards, Potentiale und Konsequenzen dieser Entwicklung geflossen, allerdings gab es bisher nur beschränkte konkrete Umsetzungen in industriellen Prozessen.

Dies ist bei Multiagentensystemen (MAS) der Fall. Dabei befinden sich autonome Softwareagenten in kommunikativem Austausch, womit sie basierend auf ihrer jeweiligen internen Logik und Zielsetzung emergente Lösungen finden. Diese Arbeit implementiert einen vereinfachten industriellen Anwendungsfall, worin eine CNC-Maschine eingehende Aufträge sowie die für deren Durchführung notwendige Werkzeugbeschaffung verhandelt. Damit sollen aktuell verfügbare Tools für die agentenorientierte Programmierung bewertet werden sowie die Frage, welche Tauglichkeit dieses Paradigma in der diskreten Fertigung hat.

Zu diesem Zweck wurde ein simples MAS mit Hilfe des JADE Frameworks in Java implementiert. Mit kurz gehaltener Entwicklungszeit und ohne Vorerfahrungen mit JADE wurde ein besonderes Augenmerk auf die Einstiegskosten für MAS Umsetzungen gelegt. Trotz ihrer Limitierungen hat die Implementierung gezeigt, dass JADE an sich und besonders auch der FIPA-Standard, auf dem es zu weiten Teilen beruht, verständlich und ausgereift genug ist, um zumindest bei Testprojekten und Prototypen hilfreich zu sein. Mit diesem Befund sowie in der Literatur verzeichneten ersten Erfolgen in der Umsetzung erscheinen hohe Erwartungen an diesen Forschungsbereich gerechtfertigt.

Abstract

Interconnectivity of smart devices has been increasing in many distinct areas of life. In manufacturing, this has led to much enthusiasm about a so-called 'fourth industrial revolution', which is expected to result in large gains in automation. Self-management, self-diagnosis and self-correction, among other things, are supposed to lead to greater optimisation than had been possible using high levels of human intervention. A lot of theorising has been done about the standards, potential and consequences of this development, but actual adoption within industrial processes has been slow for some of its technologies.

This has been the case for Multi Agent Systems (MAS), which equip software agents with reasoned autonomy, thus relying on emergent communicative interaction between parts of a process to solve situations. This work implements a simplified industrial use-case, a CNC machine managing incoming orders and procuring tools for their completion, in order to inspect current tools for programming MAS and perhaps assess the paradigm's suitability to the discrete manufacturing domain as a whole.

For this task, a simple MAS has been implemented using the JADE framework for Java. Development time has been kept short and no prior experience with the framework existed, putting emphasis on the initial cost of adopting an MAS project. Though limited in scope, the design has shown that JADE and the FIPA standard it makes heavy use of are useful, mostly straightforward tools that are at least sufficiently mature for test projects and prototypes. With this conclusion and the literature offering preliminary findings of success, high expectations of MAS in manufacturing seem warranted.

Contents

CHAPTER 1

Introduction

1.1 Problem Statement

Technological advancements within the last decades have led to a number of emerging ideas that are promising to transform large parts of the industrial sector, ranging from changes in organisational policies between enterprises all the way down to real-time considerations on the production floor. The still developing technologies that are supposed to enable this radical transformation are chiefly concerned with increasing the informational connectedness between various organisational layers, geographical locations, individual machines and points in time during a product's or process' life cycle. [dAI13] At the same time, new and increasing challenges have grown that necessitate new approaches. The increased distribution of industrial processes and resources have led to more complex and dense communication requirements. In addition, the sheer amount of information produced by sometimes globally distributed supply and value chains presents a problem in its own right. All these changes are happening on the background of the aforementioned technologies, generally summed up by the term 'Industry 4.0'. The term is a reference to the hope that a paradigm shift within industry, a fourth Industrial Revolution, will be the result of further developing the associated technologies. Cyber-physical systems (CPS) are at the core of that shift, a CPS being a system that integrates a network of virtual abstractions and services with physical components that perceive and act upon the physical world via sensors and actuators. Forming the basis of CPS are key concepts such as the Internet of Things (IoT), connecting physical entities via their virtual representation in a shared virtual space. Those abstracted entities may offer and call upon services in that shared space, sometimes referred to as a 'cloud', thus resulting in an open and flexible system. [dAI13] Within the paradigm of Industry 4.0, the connectedness of resources and services is supposed to increase further and decentralise decision-making. Information and systems will not only be integrated within the organisational hierarchy (vertical integration) but also across the entire life cycle of a product (horizontal integration).

While the ubiquitous connectedness offers an increase of information exchange, it is also coupled with an increase in complexity in all affected systems. [dAI13] A particularly interesting part of industry to look at through the lens of Industry 4.0 is manufacturing. In discrete manufacturing especially, products yield to new models more frequently and are often subject to customer desired customisation, thus also changing production steps more frequently and increasing overall process complexity. [BJW04] Since demand changes more rapidly, orders coming in in an unpredictable fashion, both batch sizes and content need to be able to change in an adaptive manner. Internally, resources are also subject to constant change, including the availability of equipment and materials, shifting timetables and new technology that needs to be integrated into existing processes without causing too much of a disturbance to running procedures and allotted budget. [CC04] [LF13] In the past, in order to meet tight monetary and temporal restrictions and reduce susceptibility to disturbances, production control in manufacturing operated in a way that was highly hierarchical and dependent on exact scheduling. With changing circumstances, these solutions are no longer ideal, especially considering the inherently highly distributed nature of manufacturing resources. [BJW04] One technology that promises decentralised decision-making, high flexibility and faster reaction time to changing circumstances is the Multi Agent System (MAS). Within an MAS, autonomous software agents communicate with each other in pursuit of their own goals. They are the virtual representation of some physical or organisational real-world unit, equipped with knowledge over their own state. While they may stand in cooperative or competitive relation to each other, agents are usually primarily motivated to try to achieve some state which aligns with their goals rather than follow some overarching hierarchical plan. Since individual parts of the distributed whole can adapt to local or external changes on their own and reorganise without outside interference MAS appear well-suited to deal with the current challenges in manufacturing. Resources becoming available or vanishing no longer need to be the concern of some central authority when their agents simply communicate their status to whoever needs to know. Differences in interfaces are hidden between abstracted services, making heterogeneous systems more easily manageable. [MVK06] Despite these apparent advantages and some development by established industrial companies, [MP11] MAS are not currently in widespread use in manufacturing. [LF13] This work will attempt to explore some of the strengths and weaknesses of MAS in that field, along with the requirements of existing tools for implementation.

1.2 Approach

To assess the viability of MAS in a manufacturing context is a massive undertaking in its entirety. For that reason this work's scope is limited to examining the capabilities of existing frameworks for programming agents. In particular one framework will be picked among them to have its suitability examined through an example implementation. The suitability will be judged with regard to the demands of manufacturing control. The example implementation is an MAS managing part of the tool life cycle of the manufacturing process. It was provided by Dr. Solmaz Mansour-Duschet in the course of

her doctoral thesis (an updated version can be found here: [MD20]), intending to explore possible improvements to the process through the introduction of agents. Details of the model are discussed in Section 3. For the framework, JADE [JAD] was requested. With these parameters set the first step will be to study existing literature concerning the implementation of MAS. Following established guidelines will ensure that the technology's potential is used properly. The next step will be to establish which requirements exist for manufacturing generally and the example design specifically. Keeping that set of requirements in mind the example will be implemented next. Finally, the process of implementation using JADE will be retrospectively evaluated, putting the emphasis on its suitability for real world manufacturing applications.

CHAPTER 2

State of the Art

2.1 Definition of Terms

2.1.1 RAMI

Reference Architectural Model Industrie 4.0 (RAMI) is an attempt by 'Plattform Industrie 4.0', a work group comprised of representatives of public and private German institutions and enterprises, to establish a standardised reference model for Industry 4.0 topics. The model has three axes describing a three-dimensional space in which topics can be classified. Along one axis there are layered views analogous to the interoperability layers of the Smart Grid Architecture Model [Gro12]. This is the Layers axis. These views show on which layer or between which layers of abstraction information is exchanged. The Asset Layer, for example, is at the bottom and contains actual devices as well as human actors, while the Business Layer at the very top concerns business strategy and processes, legal circumstances and so on. Another axis is called Hierarchy Levels, sorting by where the object in question belongs functionally. Is it related to a product, work stations or the enterprise as a whole? The final axis, Value Stream, maps to points in time within a product's life. It is split between a product's type, meaning its model and the model's instances. For types, their phases are development, usage and maintenance. Instances also have usage and maintenance on the axis, their life cycle begins with production though. The advantage of these axes is that together they are able to somewhat capture the complex relations that arise in an Industry 4.0 context with increased distribution and communication. Smart Products [dAI13] that may be able to track their own history and states have also expanded the timeline during which information is generated and consideration is necessary in novel ways. Having a common basis for discussion in the form of a reference model should help transition to the more complex world of Industry 4.0. [ABD+15]

2.1.2 Agents

The concept of computational agents arose at least in part following research in distributed artificial intelligence. [MVK06] The current understanding of an agent is an autonomous piece of software that operates within an environment in pursuit of its own goals. They are autonomous in the sense that they make their own decisions without explicit orders from outside. Instead they have control over both their internal state and how they act within their environment. An agent is considered rational if that behaviour aims to maximise some value aligning with its goals with the information it has. Intelligent agents choose their behaviour through a reasoning process and adapt their understanding of the environment through learning mechanisms. They make plans to act to further their goals. One model addressing this kind of intelligent agent is the BDI model. In it, agents have beliefs, desires and intentions. Beliefs are the assumptions the agent has about its environment. They must be constantly updated and revised for the agent to operate successfully. Desires are long-term goals that the agent wants to achieve and sustain. Intentions are the agent's current and past commitments, which are used to inform future decisions. Outside input and past actions form a loop of adjustment of knowledge and plans that are supposed to make the agent act in an informed and coherent way. When agents act, they can act both reactively and proactively. Reactive behaviour is triggered by some change in the environment or inner state of the agent. Proactive behaviour is the deliberate execution of actions in accordance with previously laid plans. The environment is usually not entirely known or knowable to the agent, so an agent's model of their environment can be incomplete or outdated. This makes it particularly important for them to be able to adapt to unforeseen internal and external states, or at least to not fail in their execution completely. [MVK06] What makes adaptability even more important is the existence of other agents within the same environment. A system that has more than one agent operating at the same time is called a multi-agent system (MAS).

2.1.3 MAS

Within an MAS, multiple agents exist within a shared environment. Within that environment agents interact directly and indirectly. Acting upon the shared environment would be interacting indirectly. One agent's actions can thereby influence other agents' behaviours without intent or even being aware of one another. Direct interaction is achieved through messaging, one agent sending data to another over a medium. Since agent design philosophy strongly favours openness and flexibility, making heterogeneous agents implemented without knowledge of other agents' internal workings very likely, there is a need for standardised communication channels within the environment. They need to be able to send messages in a shared syntax, as well as how to semantically interpret them. Such a framework is called an agent communication language (ACL).

One example for a standard of communications is FIPA-ACL developed by the Foundation for Intelligent Physical Agents which is part of the IEEE Computer Society as a standards organisation. It defines standards for messaging between agents with both the

formal structure and operations of messages as well as how to define semantic content being covered. At the heart of FIPA-ACL are the messages themselves, which are also called Communicative Acts. FIPA-ACL messages can be very lean, strictly speaking only requiring a `performative` parameter. `Performative` denotes what kind of Communicative Act is being executed. Simple examples include `agree`, which signifies readiness to perform some action, and `confirm`, which confirms an uncertain proposition asked by another. More complex performatives include `cfp` (call for proposal) which requests proposals for some action from one or more agents. A noteworthy performative is `not-understood`, which has to be sent whenever an unknown, unexpected or malformed message is received. There are a few additional parameters which are not required but expected in the vast majority of cases. The sender and receiver parameter are filled to identify the participants of the message. In the case that a message has multiple recipients, a set of names can be written into the receiver parameter. The other highly important parameter is `content`, in which the actual message content is contained. FIPA-ACL messages also support custom parameters, which are supposed to start with the `'x-'` prefix. MAS generally propose openness to the degree that agents might interact that have been designed with no knowledge of each other. Thus, in order to reduce the risk of information being misinterpreted, the content needs to be put in syntactic and semantic context. The `language` parameter contains the formal language the content is written in, such as FIPA-ACL's own FIPA Semantic Language [Fou02c]. The way the message is encoded is specified in the `encoding` parameter. With the syntax covered by these parameters the agent interprets the meaning contained within the message with the help of an ontology, which is referenced by an `ontology` parameter. Simply put, an ontology gives meaning to symbols. Using an ontology ensures that agents in the same domain have a common basis of understanding for the symbols in their messages. On the other hand it also ensures that agents in different domains never falsely interpret incoming information due to a coincidental similarity in used symbols across those domains. FIPA has released recommendations on how ontologies should be managed in an MAS for agents to be able to share knowledge effectively. [Fou01] To help the agent with associating a given incoming message with a predefined pattern of communication or previously received messages, the `protocol` and `conversation-id` parameters can be used. `Protocol` removes the burden of having to dynamically infer much of the context and intent of a message by its timing and content. Instead, agents know to engage in a particular pattern of expression without the significant cognitive power required to identify its appropriateness on the fly. `Conversation-id` is mandatory at the start of a protocol initiation and highly encouraged with any messages following, but can be used with any sent message. It is useful for distinguishing between multiple conversations, both with different agents and with the same agent. An example would be two agents starting an extended protocol twice in close succession, resulting in overlapping messages being sent. Naturally it should be ensured that the assigned conversation ids are globally unique. [Fou02b] [Fou02a] [Pos07] To be able to send and receive messages to other agents, they require a set of services. Other than the already mentioned communication medium, agents need to be able to look up other agents. Checking availability is one of

the simplest applications of such a mechanism. In a system where other agents and their traits aren't known a priori a service-based lookup is an appealing option. Agents register what services they offer and others find them by looking for the specific service they require. The system offering these functionalities is generally called the MAS middleware or MAS platform. In addition to lookup and communication channels the middleware may offer a large number of other services. Among them are agent life cycle management, data persistence and diagnostics. The middleware needs not be explicitly designed with agents in mind as long as it offers adequate services. Identifying which design goals to strive for when designing a system to serve as an MAS middleware is an ongoing field of study. It is important to note, in particular given the context of Industry 4.0 and its extremely high connectedness, that MAS middleware and MAS themselves aren't necessarily closed environments. Rather they can communicate with other MAS in arbitrarily complex organisations. An MAS may be a part of another MAS for example. It is additionally sometimes even possible for individual agents to migrate from one system to another. This trait is called mobility. [MVK06] Given these tools agents can engage in what can be called social behaviour. They may coordinate, cooperate and negotiate. This is usually done through the employ of protocols which define possible exchanges of messages in predefined patterns. Through them it is possible for agents that were never explicitly designed to work together to cooperate. The result is emergent behaviour that can solve problems as they arise within the system without designer intervention. A prominent example of a protocol is the contract net protocol, which implements a bidding mechanism. Here one agent wishes for a specific task to be executed by one of multiple potential others. They request a bid from the potentials, choosing a favourite from among them by evaluating the bid against a measurable criterion. All participants get informed about their acceptance or rejection and the winning agent then proceeds to execute the task. [MVK06]

2.1.4 JADE

JADE (Java Agent DEvelopment Framework) is an MAS framework for Java that puts emphasis on compliance with FIPA specifications. In addition to the previously discussed FIPA-ACL, those specifications also outline the key features of an agent platform. The reference model contains several agents of importance to running such middleware. It names the Agent Management System, handling registration and authentication of agents on the platform, as well as the Agent Communication Channel, which routes messages within the platform and both to and from outside, and finally the Directory Facilitator, which offers lookup or yellow page services. The motivation of FIPA is to act as a standard, empowering designers to build MAS with some fundamentals already covered. FIPA does not specify how to implement the systems it describes. The internal workings of components are left open to the needs of the developer, only external behaviours being standardised. This is supposed to leave a large margin of flexibility while hastening the development process and aiding interoperability. JADE offers the described standard agents as well as other fundamental services (such as a communication channel) to form a middleware. Since the resulting agent platform is executed within a Java Virtual Machine,

developers can easily test and deploy their agents across a multitude of hardware. If necessary the platform can be distributed over several machines, given that no firewall prevents their communication, via Java RMI. In that case each machine has a container with its own communication and lifecycle services. An attempt has been made to keep communications as lightweight as the particular case permits, with container internal messaging of course causing vastly less overhead than messaging an agent on a wholly different platform. As a programming framework, JADE includes a large amount of pre-made interfaces and functionality up to complete FIPA-compliant communication protocols. Also included is a set of diagnostic tools to assist testing and a GUI to manage the running MAS. JADE expresses its agents' actions through Behaviours. A Behaviour is a set of instructions that can be called through internal and external triggers. They may be explicitly started by another Behaviour, triggered by a repeating timer or waiting to be awakened by messages from another agent. They can be added, removed, suspended and duplicated. Behaviours can be very simple and short-lived or highly complex. JADE includes several predefined types of Behaviours that can be nested to achieve the desired pattern of activity. Broadly speaking, JADE differentiates between simple and complex Behaviours. Simple Behaviours execute atomic actions, so they are not interrupted by the scheduler until finished. At the end of their activity, they check whether their job is done and, if the answer is false, they queue up to be executed again from the beginning. Complex Behaviours are made up of sub-behaviours which give control back to the scheduler between their executions. Important to note is that every agent only gets a single thread, so truly executing Behaviours in parallel is impossible. [JAD]

ACL-based messages are the core communicative tool inside the MAS. They contain information expressed in accordance with a content language. The expressions must be semantically meaningful, meaning they must be legal constructs within the rule set of a specific ontology. Constructing messages that adhere to these rules is not done by hand but delegated to a content manager. That object is fed an expression in the form of a Java Object, as well as which content language and ontology to use, and attempts to build a valid message. The various elements that can make up a message are categorised by the type of meaning they convey. Their structures and relationships are defined in ontologies. One such type of element (i.e. predicates) makes a statement about the environment and may be true or false. They can be used to ask whether a statement is true and, on the flipside, to assert that a state is or isn't the case. Here ACL's performatives show their importance in adding contextual meaning to a message. As an example using natural language the sentence 'Machine 123 is online' will be used as a question with the QUERY-IF performative, but as a statement of fact (or perhaps it would be better to say belief) when INFORM is used. The actual enforcement of this standard is, however, left to the designers and developers. The subjects of predicates are formed by terms, also called entities, which signify both concrete objects, such as agents or resources, and abstract ones like numbers or truth-values. The simplest predicates are primitives like numbers, strings and boolean values. More complex are aggregates, which are groups, such as lists or maps, of other terms, and concepts, which have structures of any complexity. Concepts can be compared to objects in object oriented paradigms, being blueprints of

entities with named slots of expected content. A special type of concept is the agent action. Agent actions are used in messages that aren't simple queries or statements of fact, indicating any other actions an agent takes and containing whatever information is required. Predicates and terms (other than the predefined primitives and aggregates) are defined as very simple Java classes. They implement the respective interfaces, adding required fields as well as getter and setter methods. The objects are then registered in the ontology class they will be a part of and the ontology gets informed about the names and expected content of their fields. The ontology can then enforce the semantic validity of incoming or outgoing messages according to its vocabulary. Agents are not bound to a single semantic domain, able to use ontologies as needed in conversations. [Fou01] The extent to which JADE delivers on its promises of smooth development through patterns and FIPA conformity is discussed in the conclusion.

2.2 MAS in Manufacturing

This section introduces an implemented MAS solution in the manufacturing domain and summarises its mechanisms.

In 1996, DaimlerChrysler started an experimental project with the goal of producing a production system that offered high availability, tolerance to disturbances and flexibility while maintaining or improving both the quality and throughput that more traditional systems offered. The project, dubbed Production 2000+ (P2P), was supposed to design and construct the prototype of an engine cylinder head manufacturing system. Previously, cylinder heads were produced in traditional transfer lines, where different stations performed their particular work step in linear fashion. While availability of individual stations has reached high levels, disturbances can cause the whole process to be shut down in the worst case. Additionally, changing to a new kind of workpiece means having to reprogram the transfer line. As mentioned in the introduction, the demand for customised products or new models has risen, making these costly switches an unappealing prospect. [BSA00] The physical design was therefore changed from the rigid transfer lines to a series of standardised modules. These modules consist of a CNC machine, several conveyors as well as switches to move workpieces between them. The conveyors can move workpieces forwards or backwards along the series of modules, allowing the workpieces to be moved from a machine to any other machine for further processing. Alternatively they can be kept in a loop or moved out of the way if there is no machine available to service them at the moment. Individual machines are set up so that multiple machines can perform any given work step. This overlap makes the system more robust, ensuring that a single machine failing does not hold up the entire manufacturing process. On the virtualised side machines, switches and workpieces got their own agents. Workpiece agents know what sequence of work steps are required and which ones have already been performed. They initiate communications with machines and switches as needed. Whenever a workpiece needs to move to another machine or out of the system, it messages the next switch agent. The switch agent checks whether it can move the workpiece along the faster route to its goal. If the way is blocked it increases the workpiece's priority

(making it more likely to win out when competing for the switch agents' service) and checks the alternate route, if one exists.

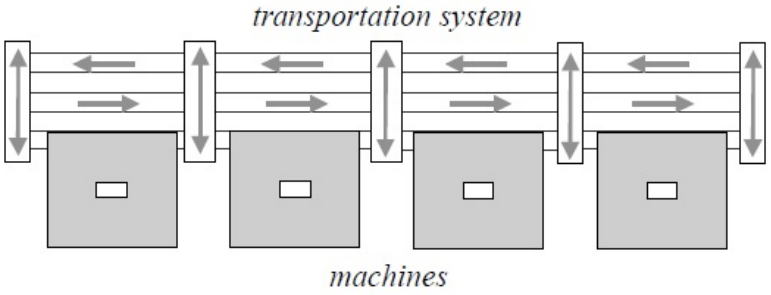

Figure 2.1: P2P's setup of conveyors, switches and machines. Source: [BS01]

When a workpiece agent has at least one work step left to be done it calls the machine agents in an auction for what is left. Machine agents know which tasks they can perform and additionally manage what is called the virtual buffer. The virtual buffer consists of the input buffer, containing a list of workpieces in the queue to be processed, and the output buffer. The output buffer remembers those workpieces that have already been processed by this machine but have not found a machine to execute their next step, yet. When receiving a call to bid in an auction a machine agent will check whether it has the capability to execute at least the first remaining step, otherwise it sends no answer. The capability is based on the machine's ability to execute the step at all and also whether its virtual buffer is full, as compared to a predefined maximum value. A machine that decides to post a bid includes the size of its virtual buffer and the biggest continuous sequence of work steps required by the workpiece it can execute, starting with the next required step. The workpiece agent compares the incoming bids, prioritising machines with smaller virtual buffer primarily, and those with more work steps in sequence secondarily. When a winner is decided the awarded machine agent is informed and puts the workpiece into its input buffer while the workpiece agent initiates routing. If the workpiece agent receives no bids it repeats its call for auction until it succeeds. This system was tested in two stages. First, a series of simulations was run, fed with real data typical of that kind of production. Both the physical configuration setup and disturbance rates were also kept authentic. In these circumstances, the virtual system performed extremely well, achieving workpiece throughput in excess of 99% of the theoretically achievable optimum. [BSA00] The next stage of tests was to build the physical setup in one of DaimlerChrysler's plants, confirming the positive results of the simulations. During the tests in the plant the agent based control system also turned out to be the least error prone part of the process. [SB01] In addition to typical loads, an additional test was conducted with different products and tiny lot sizes. Despite the unusually more difficult conditions, the system was deemed to perform well. Production 2000+ has shown the potential for agent based systems in manufacturing outside of theoretical musings, not only matching but exceeding previous solutions in several metrics. [BSA00]

CHAPTER 3

Design

3.1 Use Case

The provided use case chiefly concerns what we will call the tool life cycle. This work builds heavily on [MFTP98] for its theoretical assumptions. The tool life cycle in a manufacturing process encompasses all steps of tool usage from requisitioning tools from storage, assembly, to eventual disposal or return to storage. Figure 3.1 shows a simplified view of the process, expanded to also show orders coming in at the top. Received orders include the tools required for fulfilling the order, including the durations of usage per tool. From this order, a tool requirement can be calculated, based on the total usage required to produce the entire order. Based on this document, components for the assembly of the required tools are gathered. The tools are then assembled, measured and possibly adjusted before being sent to the machine for setup. After the production process has finished, the tools are removed from the machine. Then, they are inspected for their further viability and either stored for future use or disassembled, worn components being discarded and the rest returned to the storehouse. Many of these steps are happening in parallel for different orders to guarantee high machine utilisation. The goal of the use case is to model this expanded tool cycle via agents, focusing especially on the effective reuse of material. While the figure shows either a return to storage or salvaging intact components upon tool removal, tools of low individual price are often discarded immediately after unmounting if post-usage inspection is deemed too costly. [MFTP98] Tracking the work times of individual tools, much less their constituent components, is tricky. The information is generated at the machine the tool is mounted on but may not be communicated to other parts of the system for record keeping. An issue causing this is that simply keeping track of tools is not entirely trivial. Tools pass many physical stations during their lifetime, from storage racks to assembly tables, measuring devices and tool machines. The transport to these stations and many of the operations there are still usually done by human operators. Even disregarding potential sources of error

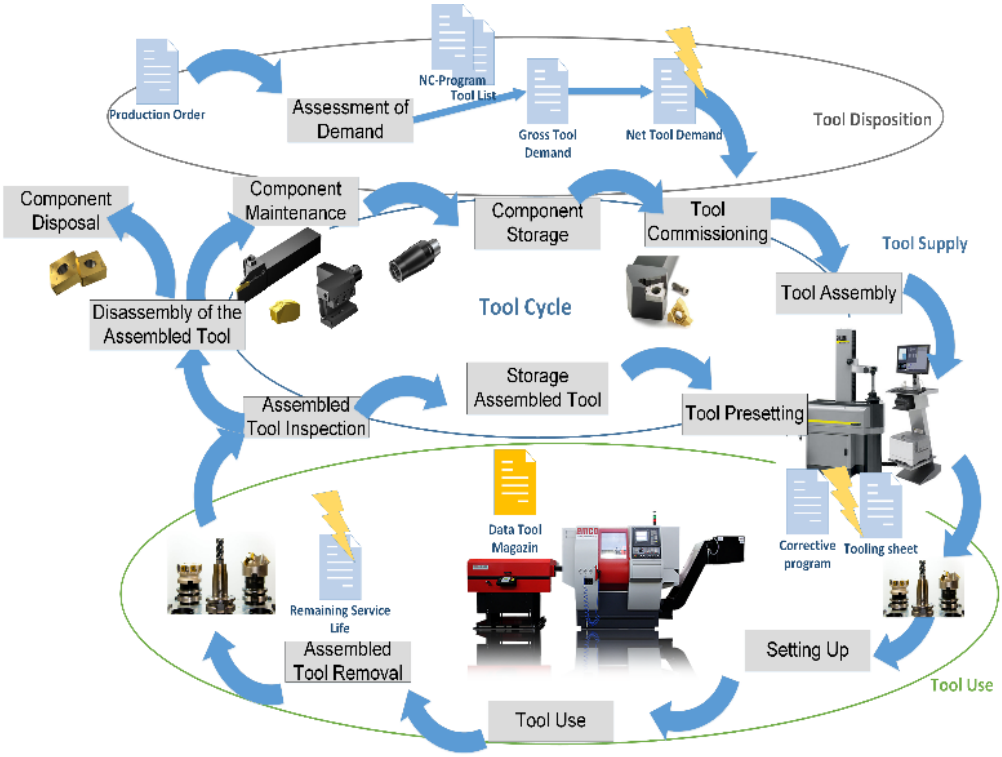

Figure 3.1: The Tool Cycle. Source: [MFTP98]

that arise from this, such as misplacement or misidentification, any additional time that arise from a worker ascertaining the identity of a tool and entering it into the computer system may be costly if it means the tool machines are idle. An estimated 20% - 30% of time in tool preparation is spent on locating the correct tool. [SAM17] Such tracking is done in different ways such as with Data Matrix Codes and RFID tags. RFID tags in particular are useful due to being readable automatically and from a distance. By expanding tracking via RFID or similar technologies to all physical and procedural steps of the tool life cycle these numbers may be reduced. Additionally, by more closely coupling a tool with its virtual representation gains in terms of information retention can be achieved. Of particular interest here is tracking the duration of a tool's previous usage. By having an accurate and reliable way of adhering to producer-given lifetime recommendations across multiple uses different positive effects appear. First, fewer tools may be discarded prematurely, a significant gain given that up to a third of production costs is spent on them. Inventory sizes may also be shrunk as a result of more accurate planning possibilities. Also significant, having an exact measure of a tool's previous use lowers the chance of tool breakage during use, preventing machine idle time. [SAM17]

The provided use case attempts to show the possibilities of improvements to the mentioned issues as well as the potential of agents in themselves. It aims to simulate a system wherein tools' traceability along with information sharing across the whole tool life-cycle

allows exact tracking of tools' lifetimes. To facilitate this, machines are expected to update and propagate information about equipped tools immediately after use. It's important for this to occur prior to some of the tools' removal from the magazine, since at this point only the machine holds the information on the reduced lifetime of the tools it just used. The information on tools' remaining lifetime even after they are returned to storage makes it possible to more safely reuse them at a later date. Building on this information, the use case makes use of a net tool list as described in [MFTP98] With the knowledge of remaining lifetimes both in the machine as well as on used and fresh tools in storage the tool requirement sent to storage for requisition can be altered to a net tool list. That list contains an adjusted number of tools, both newly assembled and previously used. The savings gained through this process extend further than just reduced waste of tools; assembling new tools can be time consuming and encouraging the reuse of previously assembled tools saves time during setup.

In addition to the purely technical aspects the use case wishes to address, there is also the greater context of exploring the use of MAS within the field of manufacturing. With the changes brought to industrial processes by Industrie 4.0 pushing towards greater distribution of decision-making, this is surely an opportune time to examine the possibilities afforded by its emerging technologies.

With all these motivations in mind, let's look at the concrete steps to be implemented, independent of the agents they are assigned to. Figure 3.2 shows one cycle of processing in detail. The process starts with an order arriving. These machining orders are assumed to already be split up in such a way that a single machine can execute the whole order without a change of tools during processing. Any orders too large to be processed with a single fully loaded magazine are rejected. The use case is made with CNC machines in mind that can be filled with a multitude of different tools to execute multiple different work steps. Their execution is controlled by an NC-programme including the concrete actions and durations per tool. One such programme is included in each order and a list of tools can be extracted from it. Along with the NC-programme identifier and the quantity of workpieces to produce the order contains the maximum accepted costs in terms of time and money, as seen in Table 3.1. There is also a tolerance factor, which controls how much additional time can be added if money would be saved overall (or the order has no spot in the queue, yet).

The tools' data is queried from a database, most importantly concerning their maximum lifetimes. Tables 6.1 and 6.2 in the appendix show an example of how that data looks, per tool and aggregated as a setting sheet. The setting sheet generated to satisfy the NC-programme contains the utilisation time of tools in total for a single workpiece. Using this information, a gross tool requirement list is calculated. Importantly, one cannot simply multiply the time per workpiece from the setting sheet by the number of workpieces and divide by the tools' real lifetimes (i.e. the tool's lifetime minus its so-called critical time – the amount of lifetime that should be left when the tool is discarded to reduce the risk of unplanned breakage). Different processes and workpiece materials put different amounts and kinds of stress on tools, which is accounted for in the setting sheet

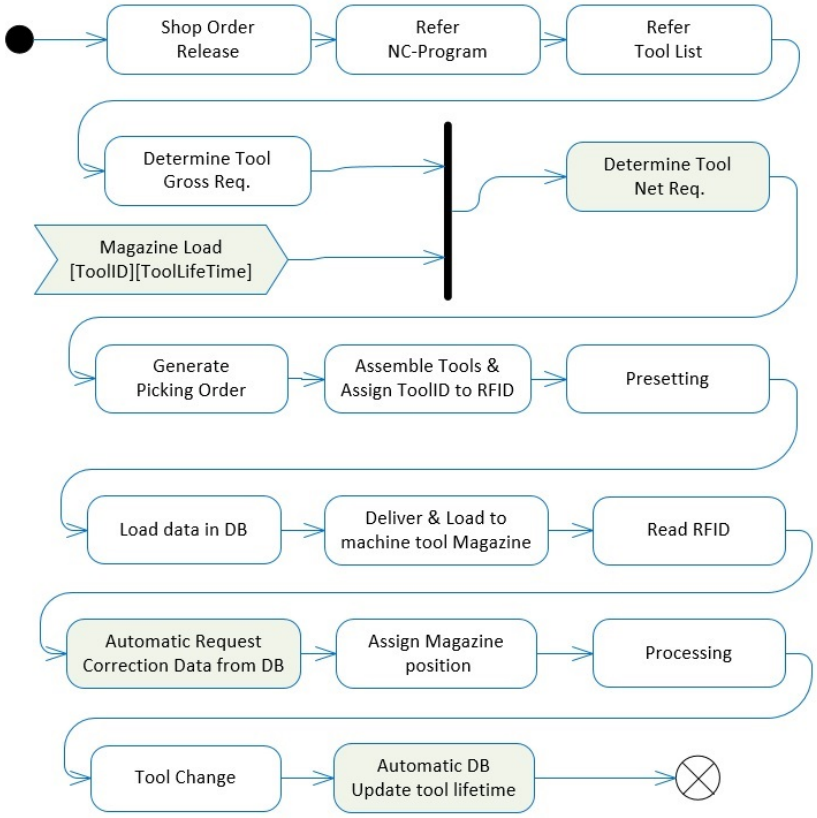

Figure 3.2: The Process in individual steps. Source: [MFTP98]

as an effective lifetime which may differ from the reference data of the tool type. That programme-specific lifetime is how long a new tool is expected to last if it performs only that programme.

The tool machine's current magazine is then used to modify that list so that only the net requirement of additional tools remains. That list is then turned into a concrete picking order to be fulfilled by storage. While not explicitly shown in the figure, there is the possibility of a small order not requiring any tools in addition to the ones already mounted on a machine. In that case, any steps until 'processing' are skipped. Otherwise the storage looks up any stored reusable tools to involve, then the new tools are assembled and given a trackable ID. Tools bound for the machine, whether newly assembled or previously returned to storage, can't just be mounted without being measured and possibly adjusted. This happens during the presetting step. After the tools have been assembled, given an ID and measured for deviation their info is saved. While relevant data can be delivered directly to the machine when the tools arrive, having it persistently accessible helps in the event of tools being misplaced or similar mishaps. Next, the requested tools get delivered to the machine and placed in its magazine. Realistically

Order ID	Quantity	Nc Programme ID	Max Cost	Max Time	Time Factor
OA	10	NC12072018	2000	200	1.45

Table 3.1: A shop order as will be used to trigger production in the MAS

speaking, this is also when some old tools get unloaded to make space or because they are worn out, but unloading is given its own step and will be discussed shortly. When the tools are loaded their IDs are read and further data loaded from the database.

Lifetimes and deviations are particularly important here to assure error-free processing. Before processing can begin the magazine slots need to be associated with the tools they contain. This like most of the setup steps up to this point is done manually, thus getting its own step. Once everything is in place processing commences, reducing the lifetimes of mounted tools according to the order's parameters. Now, we come back to the unloading of the magazine. Tools are removed based on calculations for the next order's tool requirements. This occurs simultaneously to the next mounting, if there is another order queued up. In fact multiple processes can run in parallel, although it is prudent to not let successive orders progress too far along the setup phase, since changing circumstances may invalidate that work. Decisions should be made and implemented relatively close to the execution of the next steps to prevent inefficiencies. What was optimal several cycles ago may not be optimal, or even possible, now. The final step of the process is to return the tools to storage (or discard worn out tools) and update persistent information about them in the database. At this point, the machine is already being loaded, new tools are being created for following orders and so on.

3.2 MAS Design

To simulate all the described behaviour in an MAS, we need several agents with distinct and well-defined competences. The required steps and behaviours are clustered around the scope and interests of the shop floor real-life analogues. In the following sections, agents' names will be space separated when the conceptual agent is meant and camel cased when talking about the corresponding Java class.

First there's the Order Agent, representing the interface between the ERP layer and the shop floor. Its job is to negotiate favourable conditions for the incoming orders' completion. Those orders come with a number of constraints. Most important is the type and quantity of product to be made, with the type of product expressed as an associated NC programme. That programme provides the precise instructions for the tool machines. Apart from the literal order itself, there are two more limiting factors to the execution: time and money. The Order Agent will only accept bids during negotiations that project the monetary and temporal costs to be below a provided maximum. If multiple offers satisfy that condition, cheaper (referring to their monetary cost) alternatives are preferred. Apart from the initial negotiation when the order is first created, the Order Agent may

have to renegotiate at a later point in time. An order's place in the queue is not set in stone until setup commences, so only the order being worked by a particular machine and the very next one in line (which is being set up) are immutable. Others' placement is subject to change if a new order comes in. For example, the new order may prove to be more economical somewhere in the middle of the queue because the tool magazine configuration at that point allows instant reuse of some tools, shortening setup time. Order Agents will have to consider whether rearranging the queue still satisfies their restrictions, but the orders are seen in a greater context in this step. Each order has a factor which expresses the leeway that can be given to the deadline to save money in the grander scheme. Should a different sorting to the one initially accepted end up offering cheaper production, the order's maximum throughput time can be extended by that much. So, for example, an order with a maximum time of 100 minutes but a time factor of 1.2 will accept up to 20 minutes of additional delay if it reduces costs. With new orders expecting a placement within the queue, the question arises as to what happens in the case of conflicting interests between orders. The model keeps things relatively simple in this regard. Orders that have already been given a spot in the queue before will stay in the queue unless they somehow turn out to be unfulfillable. That means that any change (or addition) to the queue needs the agreement of each order already in it as well as the newcomer's. If no configuration can be found in which all orders meet their conditions, the new order is cancelled. In the case of multiple possible configurations the greatest cumulative savings win out.

Machine Agents represent individual CNC machines. The machine has a tool magazine to manage. The lifetimes need to be checked before use and updated after use. Tracking and predicting the magazine's state is what allows us to give accurate estimates of costs to Order Agents. Without knowledge of future states the net tool list is an educated guess at best. Speaking of orders, the order queue is the second major internal structure maintained by the Machine Agent. Here the machine remembers the commitments made to various orders in terms of position, costs and waiting time. More than the magazine the queue can be very volatile, potentially subject to massive change with every incoming order. The position and estimated costs are the smaller parts of this. Rather, the preliminary information gleaned from predictions must be updated when changes occur further ahead in the queue. Of course this need is predicated on such predictions being saved in the first place rather than sending them to the order and only checking again once setup begins. In its role as the second part of negotiations with Order Agents, the Machine Agent acts with relatively little self-interest. It only cares about whether orders can be completed without idle time. Other than that it is relegated to calculating the information for Order Agents and evaluating their responses. In the current use case, the Machine Agent only checks which proposals have been accepted by all Order Agents and picks one arrangement, as mentioned above, based on their cumulative lowest cost.

Setup procedures and general tool handling outside of the machine is handled by the Supply Agent. It models the responsibilities of tool storage in real life. As such it offers services necessary to the setup process. This is where the duration of the setup process

is predicted which the Machine Agent uses to give the Order Agent an estimate. It is also the interface through which persistent tool information is accessed and updated. For calculations and query results to not be inaccurate, it is important that it is also records whether tools are currently in storage or elsewhere, where their info may not be up to date. When performing the preliminary checks for a requested tool setup, the Supply Agent doesn't enter into any negotiations with the machine. It simply states its capabilities given the parameters, although it does have the authority to reject or cancel a setup order if internal constraints are violated.

While the Supply Agent is in charge of automatic information management for tools there are some points in the tool cycle where human interaction beyond simple scanning, transport and mounting is required. This is modelled by the Visual Agent, so called because it has the only GUI for humans to interact with (orders can also be entered by a human but aren't necessarily). There are three points where human processing is required: when tools are assembled, measured and mounted. These points in time are also used as breakpoints within the execution of the use case, which does not process in real time, but progresses only when human input furthers it along. The Visual Agent thus serves a dual purpose as both the virtual model of some of the work done in the tool cycle and also the principal point of interaction to advance the test case currently running. Its value to the theoretical model is that components requiring intensive human interaction may very well be part of a production system.

The final part of the model is the Database Agent. It was decided that rather than complicating the system by having data interfaces for each agent they would instead access a central database service. It represents a layer of persistent data that changes less frequently than some of the other agents' internal structures. The Database Agent is therefore only updated when a permanent change to tools occurs. It is completely reactive and could easily be modelled as any arbitrary service other than an agent. However, reasons exist why one would want to design a system in a way where database access is addressable over communication channels shared with other agents. For instance, an open system designed for extensibility may want to offer read and write access to a central repository as a service like any other, simplifying the job of future developers to implementing the same kind of protocols they use for every other communicative act within the MAS.

Figure 3.3 shows an overview of the used agents and their planned interaction.

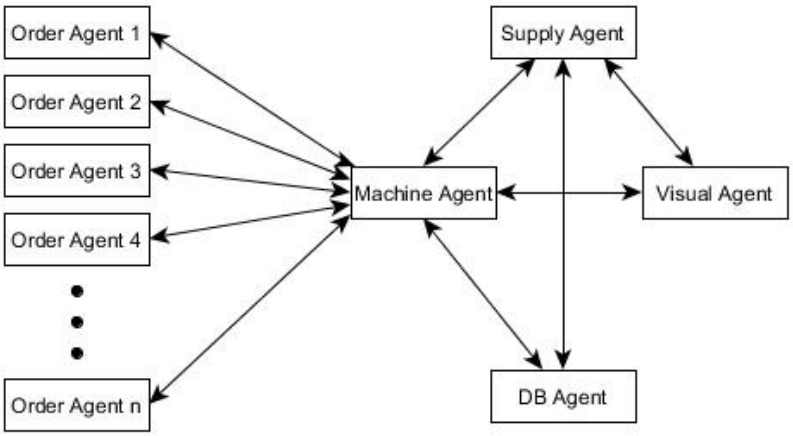

Figure 3.3: Outline of the agents' communication

CHAPTER 4

Implementation and Evaluation

4.1 Implementation

4.1.1 Constraints

A number of constraints were established before work on a concrete implementation plan began. Some of them exist to add more realistic working conditions, while others are concessions in detail for the sake of rapid development – the prototype implementation was limited to a development time, from the first diagrams to testing, of about 3 months. They are listed here but may be brought up again where appropriate.

1. No changes are made to orders in position one and two. They are either already set up or in the process of being set up by the Supply Agent, making any changes complex to calculate, as well as difficult to accurately track if this was a real-time system.
2. Potential order spots in the queue are rejected if the setup time's length would cause the machine to be idle, i.e. not working or in the process of mounting
3. An order must be finished with a single magazine load, it is assumed larger orders have already been split up to match that criterion
4. To minimise tool exchanges, orders are rejected if a mounted viable tool would have to be replaced by a new one of the same type to be able to finish it
5. Only one machine is assumed to exist, coordination between machines and further competitive negotiations for tools would exceed both the timeframe and scope of this work
6. The projected duration and costs of an order's execution should be reliable given no unexpected changes

7. Difference in tool wear between NC-programmes is assumed to be in quantity only, as well as being accurately known beforehand. Tools of the same type can be used in different programmes consecutively with their lifetime being affected in simple, foreseeable ways. A tool's wear for a specific programme is calculated as

$$Usetime * \frac{ProgrammeSpecificLifetime}{CoreDataLifetime} \tag{4.1}$$

8. Numbers are generally rounded to three decimal points

9. The system does not take partially fulfilled orders into account. The first order in the queue is either at 0% or 100% progress to simplify calculations.

10. No attempt is made to salvage error states. Recalculation and renegotiation due to incorrect information requires whole fallback protocols outside of this work's scope.

4.1.2 Protocols

The communication has been split into four major protocols. They flow into each other following rules that will be explained. Those parts of the communication were put together as a protocol where an agent requires services from others that are progressively dependent on earlier steps and conclude where the communication chunk ends back at the triggering agent with a result. An incoming shop order triggers the first of them, the Order Negotiation Protocol, which deals with the negotiation of orders' feasibility and queue position. The main actors involved are the Order and Machine Agents. The Order Agent requests proposals for the execution of their order and ends with rejection or a place in the execution queue. The Machine Agent has the role of a service provider and is tasked with acquiring the necessary information and resources to complete the demanded tasks. Once the Machine Agent receives the call for proposals, it gathers information to calculate the order's cost. Then it goes through several checks of feasibility. First whether the order is too large for a single magazine load's worth of tools to accomplish it, then if there are any possible spots for the new order's tools to be set up without causing the machine to stand idle. Should any possibility survive those proposals are sent to each active Order Agent for evaluation. If a spot in the queue is accepted for the order, the Order Agent is guaranteed its timely completion. Only by renegotiation with the Order Agent can this be changed later. So it retains control over the handling of its order at all times. The protocol is initiated by the Order Agent sending a Call For Proposals. The Machine Agent responds with a number of proposals equal to all possible slots for the new order to occupy. The Order Agent sends back those proposals it deems acceptable or a wholesale rejection if none meet its conditions. If a position is accepted further communication is suspended until one of three things happens: the order is completed by the machine, the order is cancelled by the machine because of an unforeseen error or a new order has come in and renegotiation is triggered. When the protocol concludes due to completion, rejection or cancellation the Order Agent also terminates, having fulfilled its purpose. Orders are accepted if both time and money

constraints can be met. The Machine Agent calculates not only the absolute values for duration and cost, also including potential saved costs of various order configurations in its message to the Order Agents.

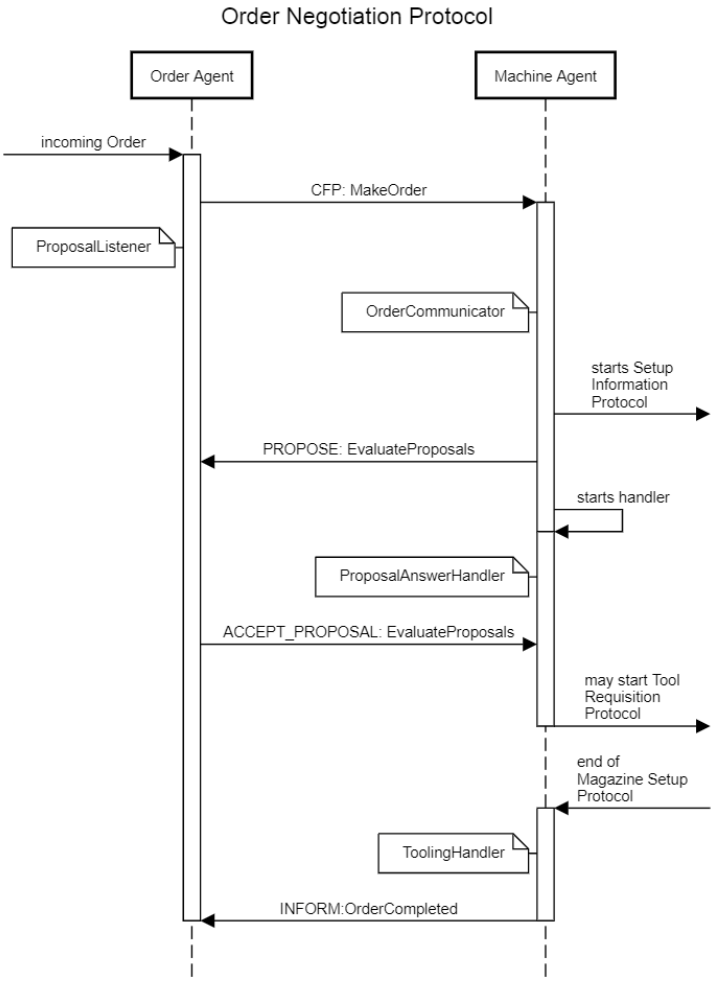

Figure 4.1: Flow of the Order Negotiation Protocol

To acquire the information necessary to make a proposal, the Machine Agent executes the Setup Information Protocol. Here, the Machine Agent utilises the Supply Agent's services to get the order's costs in terms of time and money. It starts after the Machine Agent is requested to make a proposal. It then looks up the tool requirements of the order as well as any stored tools of those types. This way it can use its knowledge of its own current (and projected future) magazine to create a net tool list which is taken as the basis of the Supply Agent's cost estimate. The Supply Agent is sent a net tool list for each possible queue position of the incoming order. To be able to create those the machine projects its magazine as well as stored and newly created tools into the future. Based on those predictions, the Machine Agent can fill the net tool list with the exact

number of required stored as well as completely new tools to be delivered per tool type. The distinction matters because of possibly varying amounts of preparation time. For example, stored tools may only have to be measured and delivered, while new tools need to be assembled first, adding to the delay. The protocol is finished with the delivery of all projected setup times and costs back to the Machine Agent to be proposed to the Order Agent.

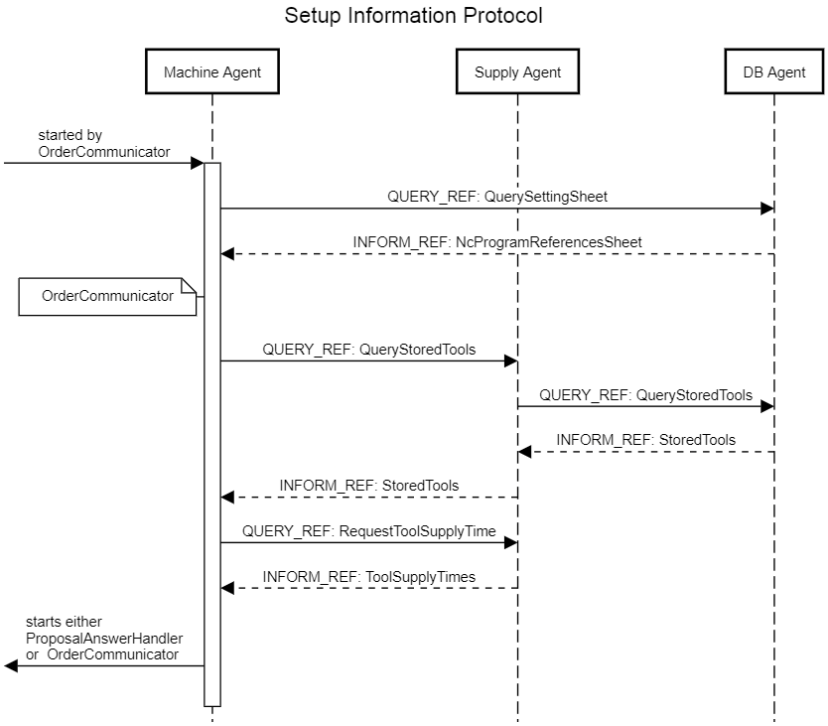

Figure 4.2: Flow of the Setup Information Protocol

In the Setup Information Protocol, time is calculated based primarily based on the used NC-programme's metadata. For reference, tables containing the metadata of programmes used in testing can be found in the appendix. The duration of the machining process itself per workpiece is the Throughput Time. This is the sum of the Main Time, which is itself the sum of all tool usages, and Nonproductive Time, which occurs during machining but during which no tools operate on the workpiece. It was decided that Setup Time, which is also included in the metadata should only apply to orders without preceding orders and that the majority of it can be done by an operator during the preceding order's runtime. Machining Time `MT` is therefore

$$MT = ThroughputTime * WorkpieceQty \tag{4.2}$$

A different time factor is tool requisition. New tools are assembled and both new and already assembled tools from storage need to be measured before use. The factors for

assembly and measurement are assumed constant and are largely arbitrarily chosen, in case of our tests `AssemblyTime = 5` and `MeasurementTime = 2`. The Machine Agent, being provided a list of available tools tries to minimise time and cost by preferring used tools in their net tool lists. The time `RT` spent on preparing tools for an order is calculated as

$$RT = AssemblyTime * NewTools + MeasurementTime * (NewTools + StoredTools) \tag{4.3}$$

As noted, `RT` needs to be lower than the previous order's `MT`. Note that in every order after the first this constraint means that the `RT`, as long as it doesn't prohibit the order entirely, does not actually factor into an order's duration since it can be performed in parallel with the previous order's `MT`. So an order's Finishing Time `FT` is

$$FT_1 = MT + RT + SetupTime \tag{4.4}$$

for the first order and

$$FT_n = (\sum_{i=1}^{n-1} FT_i) + MT_n \tag{4.5}$$

for every subsequent one.

An order's cost is calculated in three parts. First is the Machining Cost `MC`, giving a concrete monetary cost to the tool machine performing the order. It is the `MT` times a fixed cost by minute, in our case

$$MC = MT * \frac{1}{6} \tag{4.6}$$

Second is the Auxiliary Cost `AC`, which reflects the time spent changing workpieces and is set at

$$AC = Qty * 5 \tag{4.7}$$

Finally there's the Tooling Cost `TC`, which includes both new and stored tools with

$$TC = (NewTools + StoredTools) * 10 \tag{4.8}$$

The final cost is the sum of these, resulting in

$$Cost = MC + AC + TC \tag{4.9}$$

It's apparent that MC and AC are fixed costs inherent to the order, while TC is heavily dependent on the existing machine order queue. While it may seem strange that TC includes stored tools as a cost factor, it was decided that this should be the case for this model to further bias the system towards choosing order positions that reuse tools on the magazine.

Once an order in the machine's queue is available for setup, the Tool Requisition Protocol is initiated. This happens if an order is the first one to be added to the queue or when an order advances to the second place in the queue and the first one is done with its setup. During the protocol the Supply Agent is tasked with procuring the necessary tools for that order on behalf of the Machine Agent. The Supply Agent is sent the order's previously generated net tool list and starts off by requesting the picking list from the database, containing the list of components necessary for each tool's construction. After assembly the new tools are shown via the Visual Agent. After confirmation by a user, stored tools are retrieved by the Supply Agent as well. The new and retrieved tools get measured for deviation and once again displayed through the Visual Agent. Finally, the measured tools' information is sent to the Machine Agent.

The last protocol is the Magazine Setup Protocol. Its goal is to handle the communication necessary for the Machine Agent to change its magazine's tools and execute the current order. In contrast to the previous protocols the Magazine Setup Protocol behaves slightly differently. The others could be characterised as consisting of a requesting agent expecting some service and another agent going through several steps to attempt to fulfil that request. This one instead revolves around the Machine Agent contacting other agents on its own to get its work done. It gets triggered whenever an order is put into the first position of the machine queue directly after it has been delivered the necessary tools or when an order advances to the first spot right after the previous Magazine Setup Protocol has finished. Before the first message is sent, the magazine's content is swapped out. Following this exchange, the tools that are bound for storage get sent to the Visual Agent for inspection. The Visual Agent then sends a confirmation back to the Machine Agent and an update to the Database Agent to inform it of the tools returning to storage. Upon receiving confirmation the Machine Agent's setup for that order is complete and it can finally execute. After execution is finished, the Supply Agent is sent information on the tools that have to be discarded.

In attempting to follow FIPA guidelines, there are some rules that are followed across all protocols. [Fou02a] Each protocol is assigned a protocol id which is sent along with all messages that are directly involved. The exception are incidental messages sent for 'housekeeping'. Those are lookup of other agents on the middleware and some updates pushed to the persistence layer through the Database Agent. Messages also have a conversation ID to more easily differentiate between potentially multiple instances of the same protocol running in parallel. In the testing scenario presented here, this only occurs with the Order Negotiation Protocol. Any unexpected messages during a protocol – in terms of message type, performative or content – is responded to with a NOT_UNDERSTOOD performative and includes the content of the triggering message

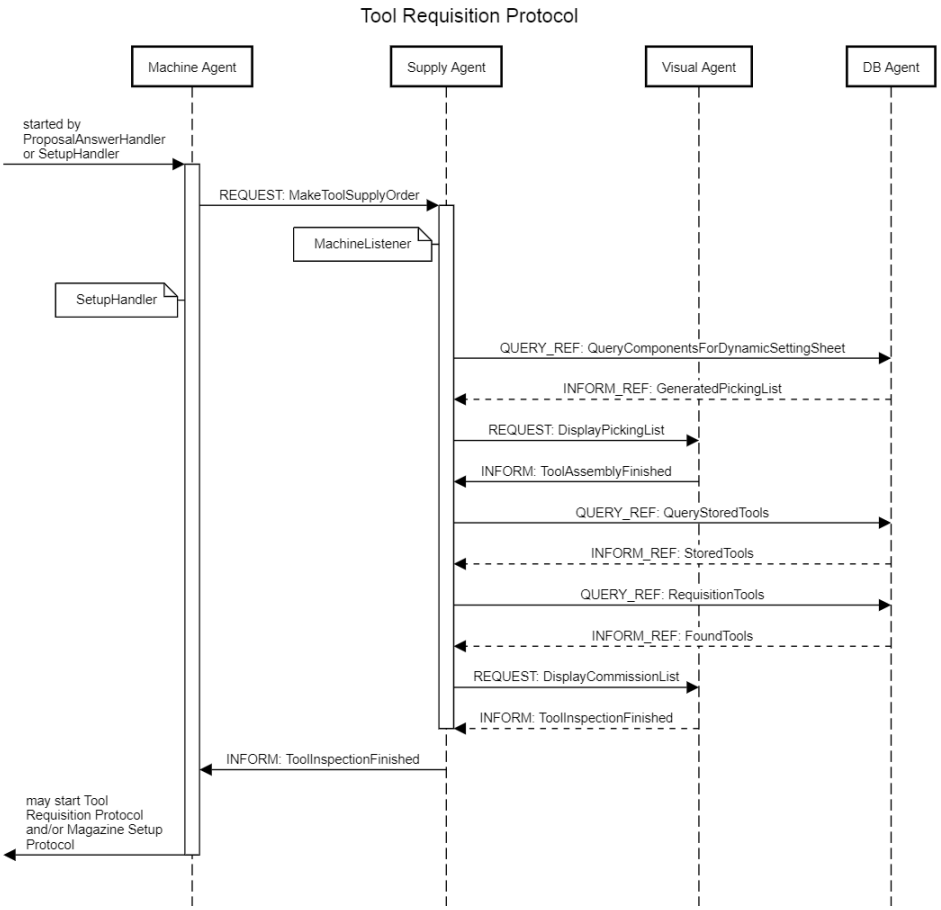

Figure 4.3: Flow of the Tool Requisition Protocol

for reference. One notable break with FIPA recommendations is the lack of use of the reply-by parameter in messages. It is theoretically supposed to impose a time limit after which the sender assumes no answer will be sent (or will no longer be accepted or processed). In this use case which is tested in a non-real time environment this temporal constraint has little use.

The resulting code (accessible here: `https://github.com/Shajinn/bakk`)for the protocols and agents within JADE is separated into two major categories: agents and ontologies. For the given use case, the semantic domains were separated into an 'order' domain, which concerns the negotiations of orders and their costs, and a 'tool' domain, which includes the technicalities of all aspects of tool usage and management that are required in the modelled tool-cycle. The Order Agent and Machine Agent both make use of the order ontology, being the negotiators of orders' execution. All agents except for the Order Agent use the tool ontology for their communication about the tool cycle. Note that the Machine Agent therefore uses two different ontologies and is a bridge between the domains. Both ontologies have been organised in the same way by splitting them

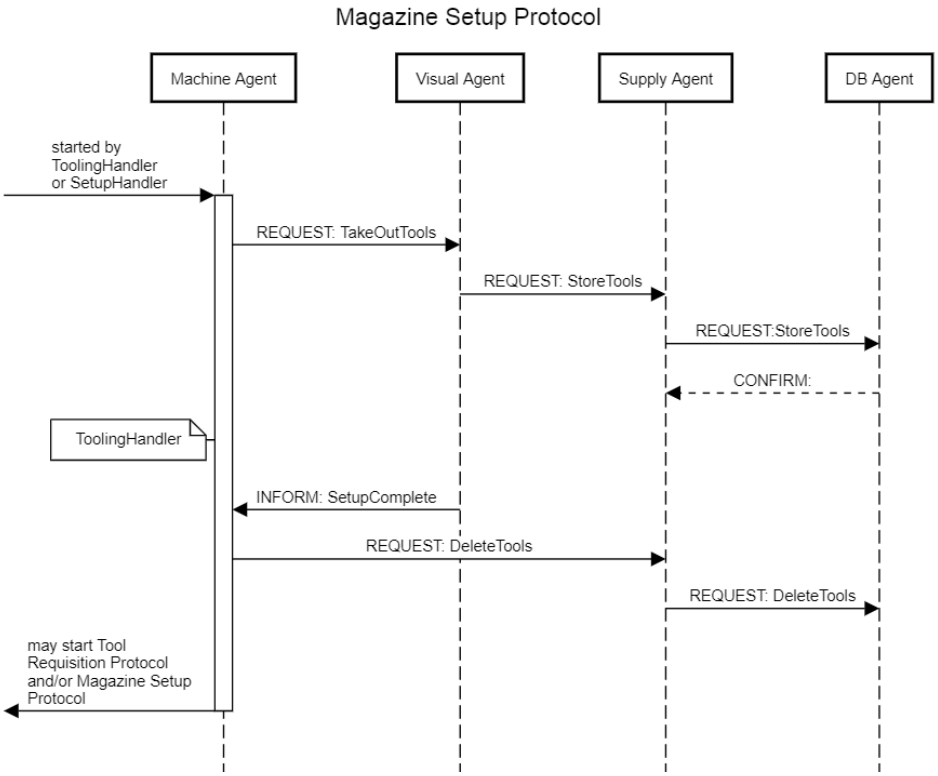

Figure 4.4: Flow of the Magazine Setup Protocol

into three types of classes: vocabularies, terms and the main ontology class.

When JADE deserialises its messages into Java objects, it uses a large number of developer-defined keywords for the task. In order to keep them together for easy reference and better maintainability, they have been defined as publicly available constants, bundled in a vocabulary class. For simplicity's sake, they are made as interfaces implemented by the main ontology class. Since the main ontology classes reference these constants a lot, this is incredibly helpful in increasing conciseness and thus readability.

`Terms` implement the appropriate JADE interfaces (`Concept`, `Predicate`, `AgentAction`) to express various semantic concepts. In their simplest form, they require only some private variables along with their getters and setters along with an empty constructor. Nearly all terms used in this implementation follow this pattern, with the vast majority of logic limited to agents. When writing these classes, it is important to precisely follow naming conventions demanded by JADE or it will not be able to use the class properly.

The main ontology class is where the terms are registered in the form of `Schemas`. With `Schemas`, the internal structure of terms is established. It has to exactly match the types and names used in the actual classes. When an ontology is assigned to a message, this main ontology object along with a particular content language is used by

```
30  private ToolDomainOntology(){
31      super(ONTOLOGY_NAME, BasicOntology.getInstance());
32
33      try{
34          //~~~~~~~~~~~~~~~~~~~~~~~~CONCEPTS~~~~~~~~~~~~~~~~~~~~~~~~~~
35          //Tool
36          add(new ConceptSchema(TOOL),Tool.class);
37          ConceptSchema cs = (ConceptSchema) getSchema(TOOL);
38          cs.add(TOOL_ID, (PrimitiveSchema) getSchema(BasicOntology.STRING));
39          cs.add(TOOL_LIFETIME, (PrimitiveSchema) getSchema(BasicOntology.FLOAT));
40          cs.add(TOOL_CRITICALTIME, (PrimitiveSchema) getSchema(BasicOntology.FLOAT));
41          cs.add(TOOL_NAME, (PrimitiveSchema) getSchema(BasicOntology.STRING));
42
43          //Component
44          add(new ConceptSchema(COMPONENT),Component.class);
45          cs = (ConceptSchema) getSchema(COMPONENT);
46          cs.add(COMPONENT_ID, (PrimitiveSchema) getSchema(BasicOntology.STRING));
47          cs.add(COMPONENT_TOOL, (PrimitiveSchema) getSchema(BasicOntology.STRING));
48          cs.add(COMPONENT_NAME, (PrimitiveSchema) getSchema(BasicOntology.STRING));
49          cs.add(COMPONENT_DESCRIPTION, (PrimitiveSchema) getSchema(BasicOntology.STRING));
50
51          //Tool Instance
52          add(new ConceptSchema(TOOLINSTANCE),ToolInstance.class);
53          cs = (ConceptSchema) getSchema(TOOLINSTANCE);
54          cs.add(TOOLINSTANCE_ID, (PrimitiveSchema) getSchema(BasicOntology.STRING));
55          cs.add(TOOLINSTANCE_TOOL, (ConceptSchema) getSchema(TOOL));
56          cs.add(TOOLINSTANCE_USEDTIME, (PrimitiveSchema) getSchema(BasicOntology.FLOAT));
57          cs.add(TOOLINSTANCE_LENGTHDEVIATION, (PrimitiveSchema) getSchema(BasicOntology.FLOAT));
58          cs.add(TOOLINSTANCE_RADIUSDEVIATION, (PrimitiveSchema) getSchema(BasicOntology.FLOAT));
```

Figure 4.5: Tool Ontology Class Snippet

the agent's `ContentManager` to serialise the message for sending. An agent receiving a message checks whether that message's ontology matches one of those registered with its `ContentManager` and then uses it to deserialise it.

While many `Terms` are specific to their part within the Agents' protocols there are some essential `Concepts` used across most communication. The most important among them are `Tool` and `ToolInstance`. `Tool` models a type of tool, including its manufacturer-provided lifetime and critical time as well as its database ID and human readable name. It is used for reference purposes across the design. `ToolInstance` is the data for a particular tangible tool. Contained within are a `Tool` to reference the type of tool it is an instance of as well as an ID to uniquely identify that particular tool. Additionally the amount of prior use is tracked to calculate the remaining life time. Deviations are included as well for completeness' sake. Figure 4.5 shows how this was modelled in the tool domain's ontology class (line 52 and following).

Agents all follow the same general structure. Their life cycle is handled by the `setup()` and `takeDown()` methods inherited from `jade.core.Agent` and are called right after the agent is started and ready, or right before it shuts down, respectively. In Figure 4.6 you can see an example of the `setup()` method in the `VisualAgent` class. Here, you can see the organisational preparations that have to be done before the agent can be fully operational. The content manager is fed the ontologies and languages it will need to understand when receiving messages (lines 56-57). Then the agent registers its services

in the MAS' yellow pages (lines 59-71). Finally all `Behaviours` that are supposed to be running from the beginning are started here (line 73). The `takeDown` method is even simpler, since the middleware can be set up to automatically deregister dead agents. All business logic is put into `Behaviours` so that an agent's actions are in flexible modules.

```
53      @Override
54      protected void setup() {
55          logger.info("Visual Agent "+ getAID().getName() +" started");
56          getContentManager().registerLanguage(codec);
57          getContentManager().registerOntology(ontology);
58
59          DFAgentDescription dfd = new DFAgentDescription();
60          dfd.setName(getAID());
61          ServiceDescription sd = new ServiceDescription();
62          sd.setType(ToolDomainVocabulary.VISUAL_SERVICE);
63          sd.setName(getLocalName()+"-Visual");
64          dfd.addServices(sd);
65          try {
66              DFService.register(this, dfd);
67          }
68          catch (FIPAException fe) {
69              logger.severe("Visual Agent "+getAID().getName()+" failed to register its service");
70              doDelete();
71          }
72
73          addBehaviour(new CommandListener());
74      }
```

Figure 4.6: Visual Agent setup method

Agents in the use case model don't act in a very deliberately autonomous fashion, usually waiting for some internal or external mechanism to finish. The agents' `Behaviours` reflect that, with all but one of them (the `OrderAgent's` call for proposals) implemented as listeners that wait for specific types of messages. Since their functions are so similar an attempt was made to keep their structure roughly uniform. `DBAgent's QueryListener`, shown abridged in Figure 4.7 can serve as a simple example.

The `action()` method is, as the name suggests, where the `Behaviour` acts when called by the scheduler. In the case of `CyclicBehaviours`, which are continuously called anew whenever they finish, the very start of every cycle is to wait for a message to arrive using the `receive()` method. While listening, control of the thread is ceded to other `Behaviours`. When multiple `Behaviours` listen simultaneously it is necessary to be more specific about which messages any particular `Behaviour` plucks from the queue. Message patterns can be given to the `receive(MessageTemplate)` method to make sure a message reaches the right destination within an agent (compare Figure 4.7 lines 524-525). Conversation IDs are particularly useful for this purpose, addressing one of possibly multiple instances of the same `Behaviour` class. After checking for `null` messages, we create a reply which is filled with most relevant information from the message it was created out of (lines 526-529). This reply will be used to send an error message of some kind if something goes wrong or to reply positively otherwise. If a message requires a different recipient, that can be changed further down the line but things like language and conversation ID are usually the same. Now the message is

```
521  private class QueryListener extends CyclicBehaviour {
522      @Override
         public void action() {
524          ACLMessage msg = myAgent.receive(
525                  MessageTemplate.not(MessageTemplate.MatchProtocol(ToolDomainVocabulary.TEST_PROTOCOL)));
526          if (msg != null) {
527              ACLMessage reply = msg.createReply();
528              reply.setPerformative(ACLMessage.NOT_UNDERSTOOD);
529              reply.setContent(msg.getContent());
530              try{
531                  AgentAction action  =
532                          (AgentAction)((Action) myAgent.getContentManager().extractContent(msg)).getAction();
533                  switch (msg.getPerformative()){
534                      case ACLMessage.NOT_UNDERSTOOD:
535                          logger.warning("DB Agent "+ getAID().getName()+
536                                                  " got a NOT_UNDERSTOOD message from " + msg.getSender().getName());
537                          return;
538                      case ACLMessage.QUERY_REF:
539                          //=========MESSAGE 11 RECEIVE===========
540                          if(action instanceof QueryComponentsForDynamicSettingSheet){
541                              ToolRequirementList toolRequirements =
542                                      ((QueryComponentsForDynamicSettingSheet) action).getToolRequirementsList();
543                              reply.setPerformative(ACLMessage.INFORM_REF);
544                              //=========MESSAGE 12 SEND===========
545                              myAgent.getContentManager()
546                                      .fillContent(reply,
547                                              new GeneratedPickingList(getPickingList(toolRequirements),++maxInvId));
548                          //=========MESSAGE 2 RECEIVE===========
549                          }else if(action instanceof QuerySettingSheet){

586              }catch (OntologyException | Codec.CodecException | ClassCastException e) {
587                  reply.setPerformative(ACLMessage.NOT_UNDERSTOOD);
588                  logger.warning("DB Agent "+ getAID().getName()+
589                                                  " could not construct a proper reply to \n"+msg.getContent());
590              }catch (SQLException e){
591                  reply.setPerformative(ACLMessage.NOT_UNDERSTOOD);
592                  logger.severe("Order Agent "+ getAID().getName()+
593                                                  " had an error occur trying to query the database for another agent");
594              }
595              myAgent.send(reply);
596          }else{
597              block();
598          }
599      }
600  }
```

Figure 4.7: Start and end of QueryListener

filtered for its content, first by performative and then by action type. The filtering by performative is useful to keep the code organised. In case the received message is `null`, which occurs if no message is actually in the queue, the `Behaviour` blocks, no longer actively polling the message queue until a new message awakens it (line 597).

Other than the basic listener-type `Behaviours` just described which serve any recognised message as it arrives there are also `Behaviours` which map the actions of a protocol. They expect a stricter sequence of messages to arrive and start the protocol by sending a message. `SupplyAgent's MachineListener` is a good example. It handles the Setup Information Protocol on its agent's behalf and is very illustrative since that protocol has the agent go through many steps rather than firing a request and getting a final result back. The `onStart()` method, seen in Figure 4.8 is called before the `Behaviour` starts for the first time. This happens when it is added to the agent rather than when it is instantiated.

It is useful to put the message initiating the protocol in there, making sure the message parameters that will be used to identify incoming messages are generated by the `Behaviour`

```
253     public void onStart() {
254         lookupAgents();
255         ToolRequirementList copiedList = new ToolRequirementList(new ArrayList(), ToolRequirementList.TYPE_QUANTITY);
256         List toolEntries = toolRequirements.getToolRequirements();
257         Iterator iterator = toolEntries.iterator();
258         while (iterator.hasNext()) {
259             ToolRequirementEntry te = (ToolRequirementEntry) iterator.next();
260             int newToolsCount = Math.max(0, (int) (Math.ceil(te.getUsetime()
261                     / (te.getTool().getLifetime() - te.getTool().getCriticaltime()))));
262             //replace tool lifetime with number of required tools
263             copiedList.getToolRequirements().add(new ToolRequirementEntry(te.getTool(), 0, newToolsCount, 0));
264         }
265
266         ACLMessage message = new ACLMessage(ACLMessage.QUERY_REF);
267         message.setOntology(ontology.getName());
268         message.setLanguage(FIPANames.ContentLanguage.FIPA_SL);
269         message.setConversationId(conversationId);
270         message.setProtocol(ToolDomainVocabulary.SETUP_INFORMATION_PROTOCOL);
271         message.setReplyWith(ToolDomainVocabulary.QUERY_COMPONENTS_FOR_DYNAMIC_SETTINGSHEET);
272         //make a request to the db
273         message.addReceiver(dbAgent);
274
275         QueryComponentsForDynamicSettingSheet queryPickingList = new QueryComponentsForDynamicSettingSheet(copiedList);
276         try {
277             //=========MESSAGE 11 SEND==========
278             myAgent.getContentManager().fillContent(message, new Action(myAgent.getAID(), queryPickingList));
279             send(message);
280         } catch (OntologyException | Codec.CodecException e) {
281             logger.warning("Supply Agent " + getAID().getName() + " could not construct an initiating message");
282             myAgent.removeBehaviour(this);
283         }
    }
```

Figure 4.8: An example of the onStart method, taken from SupplyAgent's MachineListener behaviour

itself. `MachineListener` is also a `CyclicBehaviour`, just like `QueryListener`. JADE does offer aggregates called `CompositeBehaviours` where one could model any flow of subbehaviours into one another (`FSMBehaviour` for example works as a finite state machine) but they were disregarded in favour of the simpler and less verbose `CyclicBehaviour` due to the protocols' largely linear nature. Figure 4.9 shows the beginning of `MachineListener's action()` method. Note, that the `Behaviour` will ignore all incoming messages that do not satisfy the condition given to the `receive()` method (lines 298-299). A message has to have the correct protocol and conversation ID to even be considered. Several lines further down (in 305 and 322-324), we can see how the protocol's sequentiality is handled. By making use of the `replyWith` and `inReplyTo` message parameters we can set something akin to a code word to ensure a message that arrives out of the protocol's order doesn't get processed. [Fou02a]

A message is sent with a phrase in its `replyWith` parameter. The answer then puts that same phrase into its `inReplyTo` field. When using `ACLMessage.createReply()` this is done automatically along with the other identifying message parameters. An advantage this has over e.g. a simple counter that tracks how far along a `Behaviour` has progressed is that it handles non linear protocols better and improves readability and comprehensibility when looking at messages in a vacuum. Examples would be supervision of a running system's messages or reconstruction of events from logs. Another difference to simple listener `Behaviours` is the fact that `Behaviours` mapping protocols don't continuously run while the Agent is live. Rather they are started when a necessity for

```
public void action() {
    ACLMessage msg = myAgent.receive(MessageTemplate.and(MessageTemplate.MatchConversationId(conversationId),
            MessageTemplate.MatchProtocol(ToolDomainVocabulary.SETUP_INFORMATION_PROTOCOL)));
    if (msg != null) {
        ACLMessage reply = msg.createReply();
        reply.setPerformative(ACLMessage.NOT_UNDERSTOOD);
        reply.setContent(msg.getContent());
        try {
            if (currentFilter.equals(msg.getInReplyTo())) {
                switch (msg.getPerformative()) {
                    case ACLMessage.NOT_UNDERSTOOD:
                        logger.warning("Supply Agent " + getAID().getName()
                                + " got a NOT_UNDERSTOOD message from " + msg.getSender().getName());
                        return;
                    case ACLMessage.INFORM_REF:
                        Predicate predicate = ((Predicate) myAgent.getContentManager().extractContent(msg));
                        //=========MESSAGE 12 RECEIVE==========
                        if (predicate instanceof GeneratedPickingList) {
                            PickingList pickingList = ((GeneratedPickingList) predicate).getPickingList();
                            ACLMessage message = new ACLMessage(ACLMessage.REQUEST);
                            message.setOntology(ontology.getName());
                            message.setLanguage(FIPANames.ContentLanguage.FIPA_SL);
                            message.addReceiver(visualAgent);
                            message.setConversationId(conversationId);
                            message.setProtocol(ToolDomainVocabulary.SETUP_INFORMATION_PROTOCOL);
                            message.setReplyWith(ToolDomainVocabulary.DISPLAY_PICKINGLIST);

                            currentFilter = ToolDomainVocabulary.DISPLAY_PICKINGLIST;
                            //=========MESSAGE 13 SEND==========
                            myAgent.getContentManager().fillContent(
                                    message, new Action(myAgent.getAID(), new DisplayPickingList(pickingList,
                                    ((GeneratedPickingList) predicate).getNextViableInvId())));
                            myAgent.send(message);
                            return;
```

Figure 4.9: MachineListener's action method

the protocol arises and shut down when their protocol concludes.

With the basic skeleton each agent adheres to out of the way it is time to look at some parts of the specific agents' implementation.

4.1.3 Order Agent

The `OrderAgent` is started with a set of 5 arguments corresponding to the order information already shown in Table 3.1. All interactions are handled by the `ProposalListener Behaviour` which sends a call for proposals to the `MachineAgent` on start. Following this, it listens for proposals or outright rejections.

4.1.4 Machine Agent

The `MachineAgent` is at the centre of a lot of the calculations and communication within this system. Accordingly, it has several `Behaviours` that can be active at the same time. It also has complex inner information, most of which is contained in the `MachineAdministrationEngine` class. Both the tool magazine and the order queue are handled here. The magazine is modelled as a map of `ToolInstances` referenced by numbered slot IDs. Exchange of tools on the magazine happens through the `mountTools` method which is called prior to the Magazine Setup Protocol starting.

It takes the delivered tools as input and returns outgoing tools at the end, prioritising tools that will expectedly be used the soonest to remain on the magazine. The other time the magazine is manipulated is during the actual manufacturing activity, modelled by the `executeOrder` method. It sequentially reduces the lifetimes of tools on the magazine until the order is finished. `ToolInstances` with lower remaining lifetime get used up first.

The order queue is filled with machining orders which contain both technical information such as the setting sheet, tool usage times and projected costs as well as information on the order's origin. The conversation ID, requesting agent and original `Order` object all get saved for future reference. This information is later used to reply to the correct agent with all the necessary credentials. To prevent inaccurate information during the Machine Setup Info Request Protocol, only one incoming order may be evaluated for costs at one time. That order is called the 'locking order' as it locks further orders from being processed. Additional orders coming in while an order is being considered are put into a separate processing queue. A new order is processed once the old one has been either denied or accepted into the order queue.

Estimating the costs of various potential positions of a new order in the queue requires the ability to predict the future state of the magazine, tools in storage and the order queue. The `getToolCleanup` method calculates the tools that are going to be removed after an order in a given position is finished. Its function has been briefly outlined above. `GetNetToolLists` is what the `MachineAgent` uses to predict future tool requirements. A hypothetical queue is created where the incoming order has been inserted. It then successively works through the orders from the start, simulating the tooling, execution and retooling, until the inserted order is reached. Deterministic algorithms for the choice and use of tools is crucial for this to work. In a real world environment where interferences are to be expected, exact prediction may be less important than solid self-correction mechanisms.

The `MachineAgent` is involved in almost all protocols. The `Behaviours` are set up to match them. For example, the `SetupHandler` starts and processes the Tool Requisition Protocol. The Order Negotiation Protocol which is initiated by the `OrderAgent` is handled across several `Behaviours`. The `ProposalHandler` continually listens for incoming orders and either starts their processing or puts them in a queue for later. The `ProposalAnswerHandler` sends out proposals and handles their replies. The other `Behaviours` are involved by sending rejections in case the order can't be fulfilled or a success message when it has been executed.

4.1.5 Supply Agent

The tools that are in use for production are largely handled by the `SupplyAgent`. In effect this translates to a lot of information being transformed and sent. There are three main services offered by the `SupplyAgent`. It can be queried regarding potential setup times, which is used during the Machine Setup Info Request Protocol. It is also the starting

point for initiating the Tool Requisition Protocol by the `MachineAgent`. Finally it serves as a gateway to some tool persistence services which are relayed to the `DatabaseAgent`. The messages requesting those services all go through the `SupplyBasicListener`. It then starts the `Behaviours` which handle further communication. It also calculates setup time as a function of the amount of new tools and stored tools as well as the number of new tools required for the actual setup.

`MachineListener` is the second `Behaviour` of note, handling the entirety of the Setup Information Protocol. Its main function is to gather and combine information from the other Agents involved in the protocol, matching the machine's order with tools from storage (`DatabaseAgent`) and, if necessary, new ones (`VisualAgent`). For this purpose the `MachineListener` stores temporary data on what it has already received and gradually adds to and modifies it before handing it off to the `VisualAgent` for human inspection and, ultimately, mounting on the tool magazine.

4.1.6 Visual Agent

Beyond the creation of orders all human input is done via GUIs created by the `VisualAgent`. They are created whenever the `CommandListener Behaviour` receives an appropriate message. At that point the corresponding GUI is created and a reply is only sent once the user has confirmed the data contained within it. `VisualAgent` is the only agent to implement the JADE-native `GuiAgent` class, which already contains the infrastructure for user interaction. In this case we make use of the `onGuiEvent(GuiEvent)` method to send messages with the results of the interaction. The `GuiEvent` is generated by the GUI class and can contain any amount of arbitrary information. The GUI classes themselves make use of Swing [Ora] to show simple forms.

Computationally speaking, the `VisualAgent` and its GUI classes do very little. During the tool assembly step, it generates new `ToolInstances` based on the provided picking list, giving the user the chance to rename them. On the other occasions on which this agent is called, when requested tools have been measured and when tools get unmounted from the magazine, there is no additional calculation at all.

The real value of the `VisualAgent` is that it controls the flow and timings of the running environment. It stops the protocols at critical points: when the setup is completed and after a machining order is completed. The information on tools that is displayed during these two points in particular are crucial points of comparison with expected results during testing. The timing also allows some additional nuance to the timeline of arriving orders. Since this model does not run in real-time – everything is done as fast as it can be processed by the computer – it would be practically impossible to e.g. reliably add an order during the setup process of another if it weren't for these breakpoints.

4.1.7 DB Agent

The `DBAgent` is the central interface for persistent information. Data concerning setting sheets and tools are requested here. For the test cases, a Microsoft SQL Server filled with

Test Nr.	Protocol	Description
1	Setup Information	Correct gross tool list calculation
2	Setup Information	First order immutable
3	Setup Information	First and second order immutable
4	Setup Information	Correct net tool list calculation
5	Setup Information	Cancellation when setup too long
6	Tool Requisition	Correct tool requirements calculation
7	Tool Requisition	Cancellation on unexpected tool shortage
8	Magazine Setup	Correct tool usage calculation
9	Magazine Setup	Error on unexpected tool shortage
10	Magazine Setup	Error on unexpected magazine size problem

Table 4.1: Protocol tests overview

slightly modified work floor data from a tool management software was used. In order to keep the base data consistent and have the tool data easily accessible during testing, the relevant data is kept in a simple temporary storage within the agent. It is loaded from the database on agent setup and kept up to date during execution. Additionally, the database had no way to differentiate between tools that are currently in storage and tools that are currently mounted on machines, which factors heavily into the calculations of future tool magazine states, so the `DBAgent` also keeps that information.

The only `Behaviour` it has is the `QueryListener` which waits for either requests for information or updates concerning the state of tools. The agent updates its information on tools when tools get stored, destroyed or built. Changes to tools that remain in the machine's magazine do not register here until they are removed.

4.2 Evaluation

The system is tested primarily on the protocol level. Testing the Order Negotiation Protocol is equivalent to a full system test, encompassing the whole process. All protocols except the Order Negotiation Protocol rely on non-start states within one or more agents. To assess a protocol's correctness in isolation requires a way to get to a desired state outside of the standard flow. Therefore, to prepare the various Agents for protocol testing, their internal state is manipulated with a debugging agent simply called the `TestingAgent`. `TestingAgent` can be used to communicate with `MachineAgent's` and `DBAgent's` respective testing `Behaviours` (both called `TestListener`). It sets those Agents' internal states according to the test scenario and triggers a protocol execution. A simple UI is used to setup the agent states.

Below is Table 4.1, providing a quick summary of the protocol tests. More detailed information can be found in the appendix.

The protocol tests conclude with a result printed to the testing UI, as seen above. The

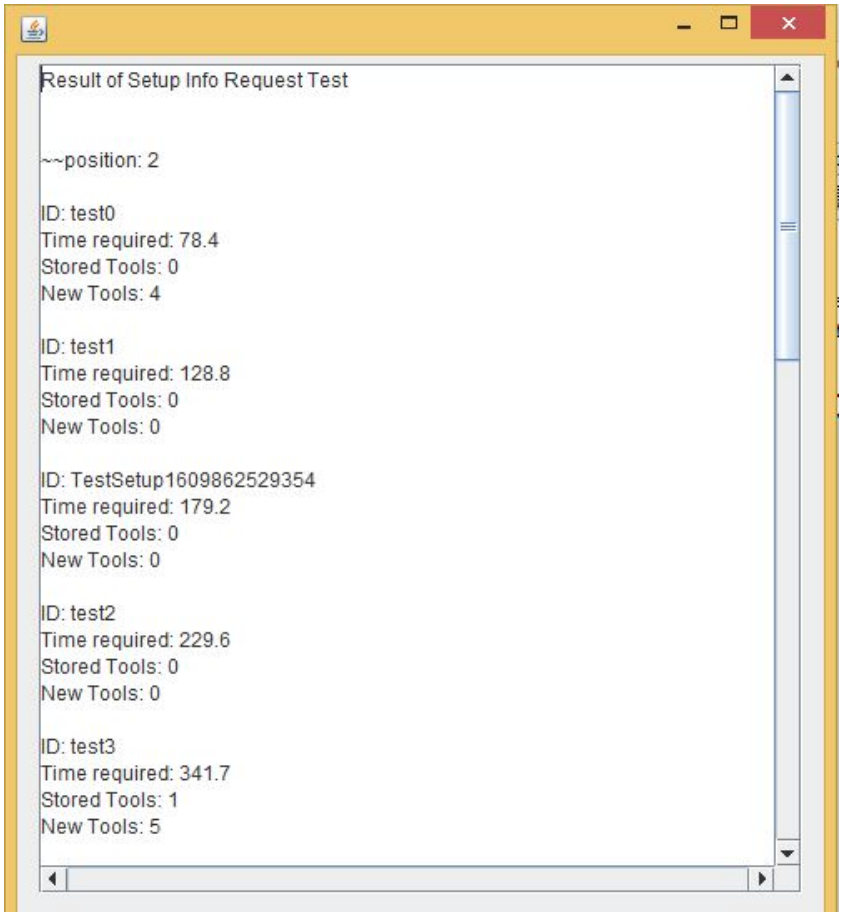

Figure 4.10: The result of Setup Info Protocol Test 4

Test Nr.	Description
1	Order Agent cancellation on exceeded time constraint
2	Time factor utilisation
3	Order Agent cancellation on exceeded cost constraint
4	Optimal order arrangement

Table 4.2: System tests overview

result was compared with the expected result. The full system tests were done via JADE's Remote Agent Management UI, creating `OrderAgents` to simulate incoming shop orders and progressing through `VisualAgent's` interfaces. Table 4.2 is a short description of the four tests. More details are in the Appendix. Results were checked by consulting the logs produced during the test and comparing them to their expected state.

CHAPTER 5

Conclusion and Outlook

In this work, a small MAS design was planned, implemented and tested in order to evaluate MAS in the manufacturing domain in general and the usefulness of the JADE framework for that task in particular. As noted before, the timeframe for development was relatively short. Despite that and the complexity of implementing an intercommunicative system as the one presented it was possible to produce at least a model of an MAS. Admittedly the model is very simplistic compared to MAS of academic or industrial use. However, this limited scope has been sufficient to evaluate how at least some elementary building blocks may be realised using JADE.

We should note that the design was worked out for the most part without intimate knowledge of the tools provided by JADE. It's not optimised specifically for JADE although it was made with FIPA in mind, which similarly is the basis for much of JADE's design. Possibly due to this match development progressed rapidly, even while in the process of familiarising with JADE's toolset. Its patterns were tremendously useful and appeared to be a decent fit for the sort of simple MAS described in this work. The `Behaviour` classes in particular deserve mention here. Even though only the most simple of JADE's `Behaviours` were utilised here, they proved both easy and versatile to use and were sufficient through various iterative reworks.

Communication between Agents is quick to implement. `ACLMessage` contains all the necessary info for a sender to enrich its communicative acts with plenty of data and metadata. Receivers can use filters to easily route messages to the intended `Behaviours`. FIPA-compliance is left for the developer to achieve but the necessary tools are there. The agent platform's `Sniffer` and `IntrospectorAgent` have been invaluable debugging tools for the small-scale communication happening in the model. The implementation of FIPA's ontology concept for serialisation-deserialisation and semantic enrichment is a both necessary and powerful tool. However, in its current state, particularly in regards to IDE integration, it was time-consuming and finicky to make use of.

Due to the very limited autonomy these agents have, it was not possible to evaluate its implementation of BDI or other sophisticated internal choice mechanisms.

As the literature research has shown, previous investigations into MAS in the discrete manufacturing domain suggest a large potential for gains in efficiency. While time and machine management are perhaps obvious areas of investigation in that regard, tool economy, which is the basis for the model explored here, may also yield worthwhile results. Discrete manufacturing, working with concrete workpieces often going through multistep processes, seems particularly well-suited to coupling with a digital twin.

This work has produced only a very limited MAS. Limited in terms not only of the agents' capabilities and autonomy but also the resources invested in its realisation. Only a single developer was involved over the course of a few months. Concerns of scalability, fallback procedures, monitoring and similar, while important in real industrial settings, couldn't be considered here. Producing desired results becomes increasingly complex with the number and autonomy of agents. While that may call the applicability of the experiences gathered in this model to larger systems into question, the ease of use of existing MAS tools even when the field is not yet fully established is a good sign for the feasibility of larger projects.

With sufficient manpower and more time, this model's use case could probably be implemented at real-life conditions right now, although some serious improvements beyond the scale alone are necessary for a functioning system. The prediction mechanisms need to work with concrete tools rather than times alone, which will necessitate more extensive tool tracking in the MAS. It may be necessary for tools to be tracked, possibly even to have self-interested agents, from the moment their future use has been determined, even if they aren't assembled at that point. Self-correction and fallbacks are other time-intensive but crucial parts that have been neglected here but would require many additional communication protocols and algorithms to add.

With increasing volume of information and efforts toward optimisation throughout industry a system organising itself autonomously as much as possible is appealing. Developing standards and best practices alongside experimental implementations and theoretical scholarship is still necessary to lessen what might be considered MAS' biggest obstacle, a perceived high financial barrier to entry for development of an as of yet unproven technology. Once overcome, however, MAS' promise of dealing with rapid changes and unforeseen complications through self-organisation, faster and with better results than a human planner, has the potential to become vital in a world of just-in-time production and shrinking inventories. Particularly the potential of extending autonomous communication along the supply chain is promising in regards to both optimisation and resilience, or at least damage mitigation, when faced with disruptions.

CHAPTER 6

Appendix: Test Details

6.1 General Information

The tables below show the testing data used during validation. Table 6.1 shows the used tools' data, excluding their components, which have no influence on any calculations. Tables 6.2, 6.3 and 6.4 show the used NC-programmes' metadata.

Tool ID	Life Time [min]	Critical Time [min]	Name
8350	80	4	StirnfräserD60
8741	70	4	ThreadMillM5X8
8750	100	2	DrillD13
8759	85	3.5	FaceMillD60
8762	83	3	FaceMillD160
8768	79.5	2.95	CylindricShaftMillD10
8777	93	2.75	ShellMillD66
8780	98.5	3	MillingCutterD10
8783	96	2.5	MillingCutterD16
8786	98	2.8	CountersinkD9
8789	70	2.5	MillingCutterD9,25
8795	59	2.9	MillingCutterD31,71
8805	58.5	1.85	RoundShankDrillD8,5
8814	57.5	1.8	DrillD12,7

Table 6.1: The tool types used in testing

NCProgram NC12072018		
NC-Program Metadata		
Time data		Time[min]
Main Time		3.95
Nonproductive Time		0.79
Setup Time		3.00
Throughput Time		4.74
Tools		
Tool	Tool Use [min]	Effective Lifetime [min]
CylindicalShaftMillD10	0.74	79.5
DrillD13	0.40	74
MillingCutterD9,25	0.85	68
ShellMillD66	1.96	81

Table 6.2: Setting Sheet information for test order 'NC12072018'

NCProgram NC12072018		
NC-Program Metadata		
Time data		Time[min]
Main Time		9.01
Nonproductive Time		1.80
Setup Time		4.00
Throughput Time		10.81
Tools		
Tool	Tool Use [min]	Effective Lifetime [min]
DrillD13	0.85	92
FaceMillD60	1.64	80
MillingCutterD10	1.62	91
MillingCutterD16	1.34	89
MillingCutterD31,71	1.45	59
ShellMillD66	1.25	89
ThreadMillM5X8	0.86	63

Table 6.3: Setting Sheet information for test order 'NC12072019'

NCProgram NC12072020		
NC-Program Metadata		
Time data		Time[min]
Main Time		5.23
Nonproductive Time		1.00
Setup Time		2.00
Throughput Time		6.23
Tools		
Tool	Tool Use [min]	Effective Lifetime [min]
CountersinkD9	0.16	98
DrillD12,7	0.48	57.5
FaceMillD60	1.24	74
FaceMillD160	0.78	73
MillingCutterD10	0.58	91
MillingCutterD16	0.89	90
RoundShankDrillD8,5	0.64	58.8
ThreadMillM5X8	0.46	61

Table 6.4: Setting Sheet information for test order 'NC12072020'

6.2 Protocol Tests

6.2.1 Setup Information Protocol

Setup Information Protocol Test 1

The first test checks whether the basic calculations of time match our expectation (Table 6.5).

Machine Orders							
Position	ID	Max Cost	Max Time	TF	Quantity	NC Program	Tooled
			–empty–				
Incoming Order							
		Max Cost	Max Time	TF	Quantity	NC Program	
		2000	200	1.45	10	NC12072018	
Expected Result							
Proposed Position		Position	ID	Time	Stored Tools	New Tools	
0		0	TestSetup...	78.4	0	4	

Table 6.5: Setup Info Protocol Scenario 1

Setup Information Protocol Test 2

In test 2, we expect the Supply Agent not to suggest replacing the first, locked in order (Table 6.6).

Machine Orders							
Position	ID	Max Cost	Max Time	TF	Quantity	NC Program	Tooled
0	test0	2000	200	1.45	10	NC12072018	no
Incoming Order							
		Max Cost	Max Time	TF	Quantity	NC Program	
		2000	200	1.45	10	NC12072018	
Expected Result							
Proposed Position		Position	ID	Time	Stored Tools	New Tools	
1		0	test0	78.4	0	4	
		1	TestSetup...	128.8	0	0	

Table 6.6: Setup Info Protocol Scenario 2

Setup Information Protocol Test 3

The same as test 2, but includes a second order that should be seen as locked (Table 6.7).

Machine Orders							
Position	ID	Max Cost	Max Time	TF	Quantity	NC Program	Tooled
0	test0	2000	200	1.45	10	NC12072018	no
1	test1	2000	200	1.45	10	NC12072018	no
Incoming Order							
		Max Cost	Max Time	TF	Quantity	NC Program	
		2000	200	1.45	10	NC12072018	
Expected Result							
Proposed Position		Position	ID	Time	Stored Tools	New Tools	
2		0	test0	78.4	0	4	
		1	test1	128.8	0	0	
		2	TestSetup...	179.2	0	0	

Table 6.7: Setup Info Protocol Scenario 3

Setup Information Protocol Test 4

Test 4 checks if more complex calculations, including predictions and net tool lists produce correct results (Table 6.8).

Machine Orders							
Position	ID	Max Cost	Max Time	TF	Quantity	NC Program	Tooled
0	test0	2000	200	1.45	10	NC12072018	no
1	test1	2000	200	1.45	10	NC12072018	no
2	test2	2000	200	1.45	10	NC12072018	no
3	test3	2000	200	1.45	10	NC12072019	no
Incoming Order							
		Max Cost	Max Time	TF	Quantity	NC Program	
		2000	200	1.45	10	NC12072018	
Expected Result							
Proposed Position		Position	ID	Time	Stored Tools	New Tools	
		0	test0	78.4	0	4	
		1	test1	128.8	0	0	
2		2	TestSetup...	179.2	0	0	
		3	test2	229.6	0	0	
		4	test3	341.7	5	1	
		0	test0	78.4	0	4	
		1	test1	128.8	0	0	
3		2	test1	179.2	0	0	
		3	TestSetup...	229.6	0	0	
		4	test3	341.7	5	1	
		0	test0	78.4	0	4	
		1	test1	128.8	0	0	
4		2	test2	179.2	0	0	
		3	test3	291.3	4	1	
		4	TestSetup...	341.7	1	0	

Table 6.8: Setup Info Protocol Scenario 4

Setup Information Protocol Test 5

The final Setup Information test predicts an empty response when a new order takes too long to set up. For this a tiny order is established in the queue, followed by a larger one (Table 6.9).

6.2.2 Tool Requisition Protocol

Tool Requisition Protocol Test 1

The first Tool Requisition test is about correct calculations (Table 6.10).

Machine Orders							
Position	ID	Max Cost	Max Time	TF	Quantity	NC Program	Tooled
0	test0	2000	200	1.45	1	NC12072018	yes
Incoming Order							
		Max Cost	Max Time	TF	Quantity	NC Program	
		2000	200	1.45	10	NC12072018	
Expected Result							
Proposed Position		Position	ID	Time	Stored Tools	New Tools	
–empty–							

Table 6.9: Setup Info Protocol Scenario 5

Machine Order							
Position	ID	Max Cost	Max Time	TF	Quantity	NC Program	Tooled
0	test0	2000	200	1.45	1	NC12072018	no
Available Tools							
Tool Type			ID			Previous Usetime	
DrillD13			353			6	
ShellMillD66			354			12	
Tool Requirements							
Tool Type			Lifetime of New Tools Required			Used Tools To Be Delivered	
DrillD13			0			1	
ShellMillD66			0			1	
CylindricShaftMillD10			7.4			0	
MillingCutterD925			8.755			0	
Expected Result							
Tool Type			ID			Previous Usetime	
DrillD13			353			6	
ShellMillD66			354			12	
CylindricShaftMillD10			NEW			0	
MillingCutterD925			NEW			0	

Table 6.10: Tool Requisition Protocol Scenario 1

Tool Requisition Protocol Test 2

Here, there are fewer tools in storage than forecast so an error is thrown (Table 6.11).

Machine Order							
Position	ID	Max Cost	Max Time	TF	Quantity	NC Program	Tooled
0	test0	2000	200	1.45	1	NC12072018	no
Available Tools							
Tool Type			ID			Previous Usetime	
ShellMillD66			354			12	
Tool Requirements							
Tool Type			Lifetime of New Tools Required			Used Tools To Be Delivered	
DrillD13			0			1	
ShellMillD66			0			1	
CylindricShaftMillD10			7.4			0	
MillingCutterD925			8.755			0	
Expected Result							
Tool Type			ID			Previous Usetime	
			–ERROR–				

Table 6.11: Tool Requisition Protocol Scenario 2

6.2.3 Magazine Setup Protocol

Magazine Setup Protocol Test 1

This test checks if post machining tool lifetimes are updated correctly (Table 6.12).

Machine Order							
Position	ID	Max Cost	Max Time	TF	Quantity	NC Program	Tooled
0	test0	2000	200	1.45	1	NC12072018	no
Magazine							
Tool Type				ID		Previous Usetime	
DrillD127				1000		0	
DrillD13				1001		95	
ShellMillD66				1002		33	
CylindricShaftMillD10				1003		0	
DrillD127				1004		0	
DrillD127				1005		0	
DrillD127				1006		0	
DrillD127				1007		0	
DrillD127				1008		0	
DrillD127				1009		0	
Brought Tools							
Tool Type				ID		Previous Usetime	
DrillD13				1011		0	
MillingCutterD925				1012		63	
MillingCutterD925				1013		63	
Expected Result							
Tool Type				ID		Usetime	
DrillD13				1011		2.4	
ShellMillD66				1002		55.54	
MillingCutterD925				-		67.255	
CylindricShaftMillD10				1011		7.4	
DrillD127				-		0	
DrillD127				-		0	
DrillD127				-		0	
DrillD127				-		0	

Table 6.12: Magazine Setup Protocol Scenario 1

Magazine Setup Protocol Test 2

An error should be thrown since the delivered tools' lifetimes are shorter than promised and needed (Table 6.13).

Machine Order							
Position	ID	Max Cost	Max Time	TF	Quantity	NC Program	Tooled
0	test0	2000	200	1.45	1	NC12072018	no
Magazine							
Tool Type			ID			Previous Usetime	
DrillD127			1000			0	
DrillD13			1001			95	
ShellMillD66			1002			33	
CylindricShaftMillD10			1003			0	
DrillD127			1004			0	
DrillD127			1005			0	
DrillD127			1006			0	
DrillD127			1007			0	
DrillD127			1008			0	
DrillD127			1009			0	
Brought Tools							
Tool Type			ID			Previous Usetime	
DrillD13			1011			0	
MillingCutterD925			1012			63	
Expected Result							
Tool Type			ID			Usetime	
–LIFETIME ERROR–							

Table 6.13: Magazine Setup Protocol Scenario 2

Magazine Setup Protocol Test 3

Here, an edge case is tested, where the tools' combined lifetimes are enough for the order but the magazine is too small to hold them all (Table 6.14).

Machine Order							
Position	ID	Max Cost	Max Time	TF	Quantity	NC Program	Tooled
0	test0	2000	200	1.45	1	NC12072018	no
Magazine							
Tool Type			ID			Previous Usetime	
DrillD127			1000			0	
DrillD13			1001			95	
ShellMillD66			1002			33	
CylindricShaftMillD10			1003			0	
DrillD127			1004			0	
DrillD127			1005			0	
DrillD127			1006			0	
DrillD127			1007			0	
DrillD127			1008			0	
DrillD127			1009			0	
Brought Tools							
Tool Type			ID			Previous Usetime	
DrillD13			1011			0	
MillingCutterD925			1012			67	
MillingCutterD925			1013			67	
MillingCutterD925			1014			67	
MillingCutterD925			1015			67	
MillingCutterD925			1016			67	
MillingCutterD925			1017			67	
MillingCutterD925			1018			67	
MillingCutterD925			1019			67	
Expected Result							
Tool Type			ID			Usetime	
–MAGAZINE SIZE ERROR–							

Table 6.14: Magazine Setup Protocol Scenario 3

6.3 System Tests

System Test 1

The first system test checks whether the system correctly cancels an incoming order if its waiting time can't be satisified. The Tolerance Factor would be used since the order is getting its initial spot, but it is also not high enough to make up for the excess (Table 6.15).

Machine Orders							
Position	ID	Max Cost	Max Time	TF	Quantity	NC Program	Status
0	OA	2000	200	1.45	10	NC12072018	machining
1	OB	3000	180	1.30	4	NC12072019	preparing
?	OC	2000	100	1.20	12	NC12072020	incoming
Expected Result							
–ORDER CANCELLED–							

Table 6.15: System Test Scenario 1

System Test 2

The same test once again, but the Tolerance Factor of the incoming order is increased. It should now be accepted (Table 6.16),

Machine Orders							
Position	ID	Max Cost	Max Time	TF	Quantity	NC Program	Status
0	OA	2000	200	1.45	10	NC12072018	machining
1	OB	3000	180	1.30	4	NC12072019	preparing
?	OC	2000	100	1.80	12	NC12072020	incoming
Expected Result							
Position	ID	Max Cost	Max Time	TF	Quantity	NC Program	Status
0	OA	2000	200	1.45	10	NC12072018	machining
1	OB	3000	180	1.30	4	NC12072019	preparing
2	OC	2000	100	1.80	12	NC12072020	open

Table 6.16: System Test Scenario 2

System Test 3

Test 3 checks whether exceeded max cost correctly causes the order to be cancelled by the Order Agents (Table 6.17).

Machine Orders							
Position	ID	Max Cost	Max Time	TF	Quantity	NC Program	Status
0	OA	59	200	1.45	10	NC12072018	machining
1	OB	28	180	1.30	4	NC12072019	preparing
?	OB1	28	180	1.30	4	NC12072019	incoming
Expected Result							
–ORDER CANCELLED–							

Table 6.17: System Test Scenario 3

System Test 4

The final system test ensures that the order queue gets reordered when profitable. Order OC comes in first and is placed on position 2, having no other option. Order OA comes in afterwards and has more tools in common with the OB orders, which saves time and money on retooling. It is therefore placed on the spot which OC occupied before it (Table 6.18).

Machine Orders							
Position	ID	Max Cost	Max Time	TF	Quantity	NC Program	Status
0	OB	59	100	1.45	8	NC12072019	machining
1	OB1	30	150	1.30	4	NC12072019	preparing
?	OC	80	200	1.30	4	NC12072020	incoming
?	OA	80	200	1.30	4	NC12072018	incoming
Expected Result							
Position	ID	Max Cost	Max Time	TF	Quantity	NC Program	Status
0	OB	59	100	1.45	8	NC12072019	machining
1	OB1	30	150	1.30	4	NC12072019	preparing
2	OA	80	200	1.30	4	NC12072018	open
3	OC	80	200	1.30	4	NC12072020	open

Table 6.18: System Test Scenario 4

List of Figures

List of Tables

Bibliography

[ABD+15] Peter Adolphs, Heinz Bedenbender, Dagmar Dirzus, Martin Ehrlich, and Ulrich Epple. Statusreport: Referenzarchitekturmodell Industrie 4.0 (RAMI4.0). *VDI/VDE-Gesellschaft Mess- und Automatisierungstechnik*, 0(April):1–32, 2015.

[BJW04] Stefan Bussmann, Nicolas R. Jennings, and M.J. Wooldridge. *Multiagent systems for manufacturing control: a design methodology.* 2004.

[BS01] Stefan Bussmann and Klaus Schild. An Agent-based Approach to the Control of Flexible Production Systems. *ETFA 2001. 8th International Conference on Emerging Technologies and Factory Automation. Proceedings (Cat. No.01TH8597)*, 2001.

[BSA00] Stefan Bussmann, Klaus Schild, and Daimlerchrysler Ag. Self-Organizing Manufacturing Control : An Industrial Application of Agent Technology. 2000.

[CC04] Maria Caridi and Sergio Cavalieri. Multi-agent systems in production planning and control: an overview. *Production Planning & Control*, 15(2):106–118, 2004.

[dAI13] Abschlussbericht des Arbeitskreises Industrie 4.0. Umsetzungsempfehlungen für das ZukunftsprojektIndustrie 4.0, 2013.

[Fou01] Foundation for Intelligent Physical Agents. FIPA Ontology Service Specification. 2001.

[Fou02a] Foundation for Intelligent Physical Agents. FIPA ACL Message Structure Specification. 2002.

[Fou02b] Foundation for Intelligent Physical Agents. FIPA Communicative Act Library Specification. 2002.

[Fou02c] Foundation for Intelligent Physical Agents. FIPA SL Content Language Specification. 2002.

[Gro12] CEN-CENELEC-ETSI Smart Grid Coordination Group. Smart Grid Reference Architecture, 2012.

[JAD] JADE. JADE – Java Agent DEvelopment Framework. https://jade.tilab.com/. Accessed on 2019-09-22.

[LF13] Arndt Lueder and Matthias Foehr. Identifikation und Umsetzung von Agenten zur Fabrikautomation unter Nutzung von mechatronischen Strukturierungskonzepten. *Agentensysteme in der Automatisierungstechnik*, (3), 2013.

[MD20] Solmaz Mansour-Duschet. A Multiagent Design Methodology for the Manufacturing Execution System Domain, 2020.

[MFTP98] Solmaz Mansour Fallah, Thomas Trautner, and Florian Pauker. Integrated tool lifecycle. *Procedia CIRP*, 79:257–262, 1998.

[MP11] Mieczyslaw Metzger and Grzegorz Polaków. A survey on applications of agent technology in industrial process control. *IEEE Transactions on Industrial Informatics*, 7(4):570–581, 2011.

[MVK06] L. Monostori, J. Váncza, and S. R.T. Kumara. Agent-based systems for manufacturing. *CIRP Annals - Manufacturing Technology*, 55(2):697–720, 2006.

[Ora] Oracle. Swing. https://docs.oracle.com/javase/8/docs/technotes/guides/swing/. Accessed on 2021-02-23.

[Pos07] Stefan Poslad. Specifying Protocols for Multi-Agent Systems Interaction Article. *ACM Transactions on Autonomous and Adaptive Systems ACM*, (November 2007), 2007.

[SAM17] Eva Schaupp, Eberhard Abele, and Joachim Metternich. Potentials of digitalization in tool management. *Procedia CIRP*, 63:144–149, 2017.

[SB01] Kurt Sundermeyer and Stefan Bussmann. Einführung der Agententechnologie in einem produzierenden Unternehmen - Ein Erfahrungsbericht. *Business and Information Systems Engineering*, 43(2):135–142, 2001.