JADE Multi-Agent System Holonic Control Implementation for a Manufacturing Cell

Karel Kruger [a,*] and Anton Basson [a]

[a] Dept of Mechanical and Mechatronic Eng, Stellenbosch Univ, Stellenbosch 7600, South Africa
[*] Corresponding author. Tel: +27 21 808 4258; Email: kkruger@sun.ac.za

February 2018

Abstract

Multi-Agents Systems (MASs) is a popular approach for the implementation of holonic control architectures in manufacturing systems. Software agents and holons share several similarities, allowing for the exploitation of the advantages that is offered by holonic systems. The Java Agent Development (JADE) framework is the tool most often used in implementations of holonic control. This paper describes a JADE MAS implementation of the Product-Resource-Order-Staff Architecture (PROSA) for holonic control of a manufacturing cell. The mapping of the holonic and MAS architectures is explained and the communication and functionality of the individual agents in the MAS is detailed.

Keywords: Multi-Agent System (MAS), Java Agent Development framework (JADE), Holonic manufacturing system (HMS), Reconfigurable manufacturing system (RMS)

1 Introduction

Modern manufacturing systems require short lead times for the introduction of new products into the system, the ability to produce a larger number of product variants and the ability to handle fluctuating production volumes (Bi *et al*, 2008). The concept of Reconfigurable Manufacturing Systems (RMSs) is aimed at addressing these requirements.

RMSs aim to switch between members of a family of products, through the addition or removal of functional elements (hardware or software), with minimal delay and effort (Martinsen, 2007; Vyatkin, 2007). RMSs can rapidly adjust the production capacity and functionality in response to sudden changes, by reconfiguring hardware and control resources (Bi *et al*, 2008; Bi, Wang, and Lang, 2007). RMSs are characterised by (Mehrabi, Ulsoy, and Koren, 2000; ElMaraghy, 2006): modularity of system components, integrability with other technologies, convertibility to other products, diagnosability of system errors, customizability for specific applications and scalability of system capacity.

Holonic control architectures is a popular approach for enabling control reconfiguration in RMSs. The term holon (first introduced by Koestler in 1967) comes from the Greek words "holos" (meaning "the whole") and "on" (meaning "the particle"). Holons are "any component of a complex system that, even when contributing to the function of the system as a whole, demonstrates autonomous, stable and self-contained behaviour or function" (Paolucci and Sacile, 2005). When this concept is applied to manufacturing systems, holons are autonomous and cooperative building blocks for transforming, transporting, storing or validating the information of physical objects. A Holonic Manufacturing System (HMS) is a system of holons that can cooperate to integrate the entire range of manufacturing activities (Paolucci and Sacile, 2005).

The use of holonic control for RMSs holds many advantages: holonic systems are resilient to disturbances and adaptable in response to faults (Vyatkin, 2007); have the ability to organise production activities in a way that they meet the requirements of scalability, robustness and fault tolerance (Kotak *et al*, 2003); and lead to reduced system complexity, reduced software development costs and improved maintainability and reliability (Scholz-Reiter and Freitag, 2007).

The application of the holonic concept to manufacturing control systems has been a popular field of research since the early 1990's. The most popular approach to implementing holonic control architectures has been Multi-Agent Systems (MASs). The main motivation for this approach is the similarities between holons and software agents – both must exhibit autonomy and provide interfaces to facilitate cooperation. Several experimental implementations have been reported, such as Leitao and Restivo (2006) and Giret and Botti (2009).

Several tools exist for the development of MASs – of these tools, the Java Agent Development (JADE) framework is most commonly used in the control of manufacturing systems. JADE was developed by Telecom Italia and has been distributed under an open source license since 2000. The JADE framework provides the middleware to facilitate distributed applications that exploit the software agent abstraction (Bellifemine *et al*, 2007). JADE provides tools that simplify the development, testing and operation of MASs, such as the Agent Management System (AMS) and the Directory Facilitator (DF). The AMS includes all the functionality to manage the agents in the MAS, from the creation of agents, to the migration and termination of agents. The DF provides a mechanism for the registration and discovery of resources by agents in the MAS. JADE also provides special Java classes, called *behaviours*, for implementing common functionality of agents – this includes behaviours for communication protocols that comply with the Foundation for Intelligent, Physical Agents (FIPA) specifications for agent communication.

This paper presents a JADE MAS implementation of a holonic reference architecture for a manufacturing cell. The implemented PROSA holonic architecture is discussed in section 2 and the case study, on which the implementation is based, is presented in section 3. The MAS holonic control implementation is described in section 4 and the paper concludes with a discussion of related and future work.

2 Holonic Reference Architecture

The exploitation of the advantages of holonic control, as mentioned in section 1, relies on the holonic system's architecture. Several reference architectures, which specify the mapping of manufacturing resources and information to holons and to structure the holarchy, have been proposed (e.g. Chirn and McFarlane (2000) and Leitao and Restivo (2006)), but the most prominent is the Product-Resource-Order-Staff Architecture (PROSA), as developed by Van Brussel *et al* (1998).

PROSA defines four holon classes: Product, Resource, Order and Staff. The first three classes of holons can be classified as basic holons, because, respectively, they represent three independent manufacturing concerns: product-related technological aspects (Product holons), resource aspects (Resource holons) and logistical aspects (Order holons).

The basic holons can interact with each other by means of knowledge exchange, as is shown in Figure 1. The process knowledge, which is exchanged between the product and resource holons, is the information and methods describing how a certain process can be achieved through a certain resource. The production knowledge is the information concerning the

production of a certain product by using certain resources – this knowledge is exchanged between the order and product holons. The order and resource holons exchange process execution knowledge, which is the information regarding the progress of executing processes on resources.

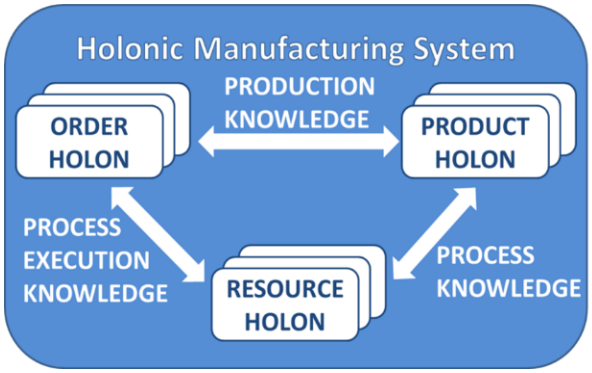

Figure 1: Knowledge exchange between the PROSA holons.

Staff holons are considered to be special holons as they are added to the holarchy to operate in an advisory role to basic holons. The addition of staff holons aim to reduce work load and decision complexity for basic holons, by providing them with expert knowledge.

The holonic characteristics of PROSA contribute to the different aspects of reconfigurability mentioned in section 1. The ability to decouple the control algorithm from the system structure, and the logistical aspects from the technical aspects, aids integrability and modularity. Modularity is also provided by the similarity that is shared by holons of the same type.

3 Case Study

The case study used for the presented implementation is a manufacturing cell for the assembly and testing of electrical circuit breakers. The layout of the cell is shown in Figure 2. The cell consists of the following workstations:

- Manual assembly station – the sub-components of circuit breakers are assembled and placed on empty carriers on the conveyor.
- Inspection station – a machine vision inspection is performed on the circuit breakers as the carriers are moved by the conveyor.
- Electrical test station – circuit breakers are picked up by a robot and placed into testing machines. The testing machines perform the necessary performance and safety tests on every breaker. When the testing is completed for a breaker, it is removed from the testing machine by the robot and placed on an empty carrier on the conveyor.
- Stacking station – multiple circuit breakers are stacked to produce multi-pole circuit breakers. The breakers are removed, stacked and placed on empty carriers by a robot.
- Riveting station – the casings of the circuit breakers are manually riveted shut.
- Removal station – the completed circuit breakers are removed from carriers. The breakers are then moved to the next cell for packaging.

The conveyor moves product carriers between the various workstations. The conveyor is equipped with stop gates and lifting stations at every workstation. The carriers are fitted with Radio Frequency Identification (RFID) tags and RFID readers are placed at multiple positions along the conveyor, to provide feedback of carrier location.

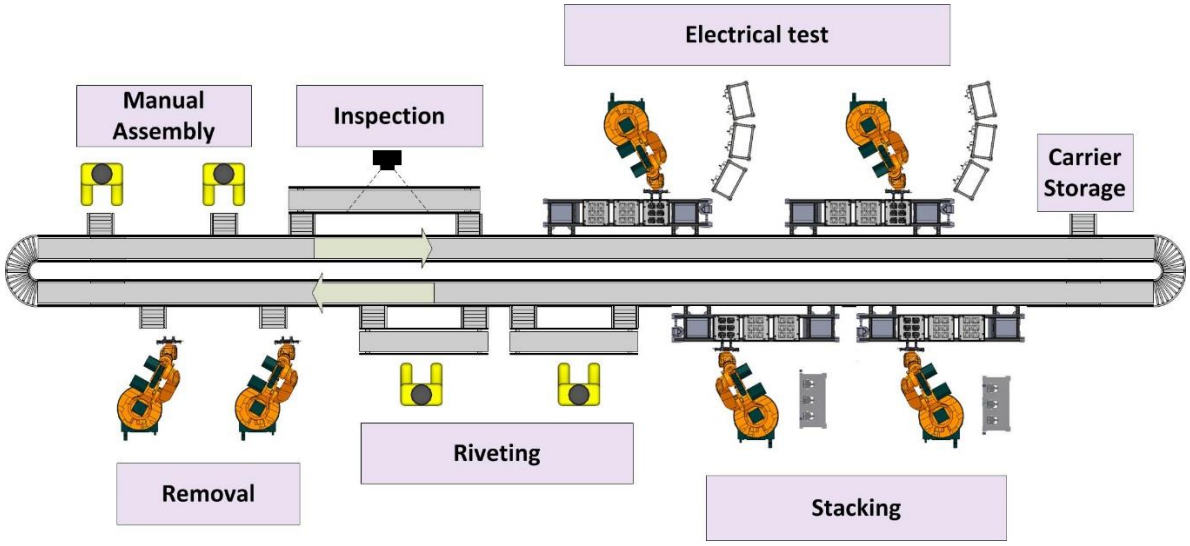

Figure 2: Layout of the electrical circuit breaker assembly and testing cell.

4 Holonic Control Implementation

This section presents the JADE MAS implementation of holonic control, based on PROSA, for a manufacturing cell. The embodiment of the holonic architecture through a MAS is explained and the communication between system agents is discussed. Finally, the functionality and implementation of the individual agent types are described.

4.1 Holonic Architecture

In accordance with PROSA, the various functional components of the manufacturing cell are embodied as Product, Resource, Order or Staff holons. All of the holons are represented in the high level control implementation by software agents. The cooperation of the agents within the MAS implementation provide all the necessary functionality to drive the production of the manufacturing cell.

The information that pertains to the production of every product that is to be manufactured by the cell is contained within Product holons. Since these holons exist purely as information within the control implementation, the holons are wholly represented as Product agents within the MAS.

The Order holons should exhibit the functionality to utilize the product information to produce a product of a specific type. Order holons encapsulate the logic and information needed for production, and thus only exist within the high level control implementation, where Order holons are represented as Order agents.

In the presented architecture, it is only the Resource holons that include both physical and software functional components. A Resource holon contains the resource hardware (as present on the factory floor), the low level control component (that control the actuators of the hardware and receives feedback from sensors) and the high level control component. The high level control components is implemented as a Resource agent in the MAS control implementation. Resource agents must provide the functionality to communicate with the other agents in the MAS, manage the agenda of the resource (i.e. the schedule of the execution of the resource's services), control the service execution tasks and sequences and maintain a communication interface with the low level control components of the Resource holon. The internal architecture of the Resource agent is presented in Figure 3.

The implementation includes one special Resource agent – the Transport agent. The Transport agent is responsible for the high level control of the conveyor system, which moves the product carriers between the different workstations. The implementation makes use of conveyor controller that was previously developed using Erlang (see Kruger and Basson (2016) for details). The Transport agent included in this implementation acts as a wrapper, i.e. to provide an agent interface to the Erlang controller. This interface allows the agents in the MAS to communicate with the Erlang controller as if it was just another Resource agent.

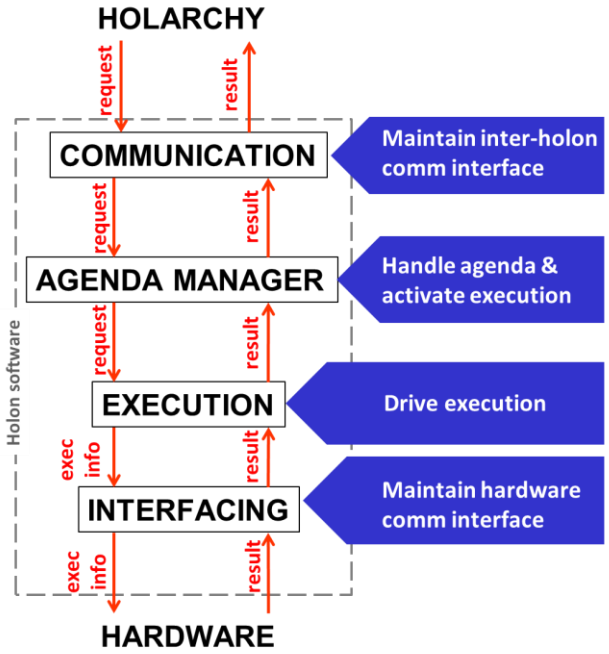

Figure 3: Internal architecture for the Resource agent.

The Staff holons for the manufacturing cell are implemented as different agents in the MAS. Some of the Staff holons functionality are provided by JADE, such as the AMS and DF. Two other Staff agents are included: the Order Manager agent (to manage the creation and monitoring of Order agents within the MAS) and the Performance Logger agent (to record the performance of Resource and Order agents for diagnostic purposes).

4.2 Agent Communication

The cooperation of agents within the MAS is achieved through communication – information is passed as messages between agents. The implementation aimed to make use of the communication protocols and accompanying functionality provided by JADE – specifically, the FIPA Rational-Effect protocol and the contract net protocol. To supplement the communication a messaging ontology is defined and is applied to the construction of the content information that is added to the various message instances. This ontology and the formation of customized communication protocols using the JADE protocols are described in the following sections.

4.2.1 Messaging ontology

The implementation makes use of eXtensible Markup Language (XML) ontology for structuring the information exchanged during communication. The XML ontology specifies the information that must accompany a specific message type, as is determined by the elements that comprise the XML document.

The templates of the XML documents for the various message types are included in every agent. When a message is composed, the template is used and the required information is added to the elements. The constructed XML document is then converted to a *string* data type, so that the data can be added to the content slot of a normal JADE ACL message. On the receiver side, the template is used to determine the elements of the received message from where data must be extracted. The string obtained from the content slot is converted back to an XML document, from which point it can be parsed and the required information can be extracted. An example of the content of a *start* request message, as would be sent from an Order agent to a Resource agent, is shown below:

```
<?xml version="1.0" encoding="UTF-8"?>
<message>
      <initiator>OrderAgent_O24</initiator>
      <responder>ResourceAgent_R06</responder>
      <msg>
            <message_type>start</message_type>
            <service_type>test</service_type>
            <conversation_ID>C021</conversation_ID>
            <result>undefined</result>
            <info>
                  <product_ID>P02</product_ID>
            </info>
      </msg>
</message>
```

4.2.2 Service booking, confirmation and execution

Order agents, as embodiments of Order holons, are responsible for driving production – each Order holon exhibits the functionality coordinate the resources necessary to produce their specific product. The Order agents then follow a protocol for the booking of resource services, according to the tasks specified by the product information. When the part is ready for the next service to be performed on it, the Order agent must first confirm the service booking and then start the execution of the service. This interaction is illustrated in Figure 4.

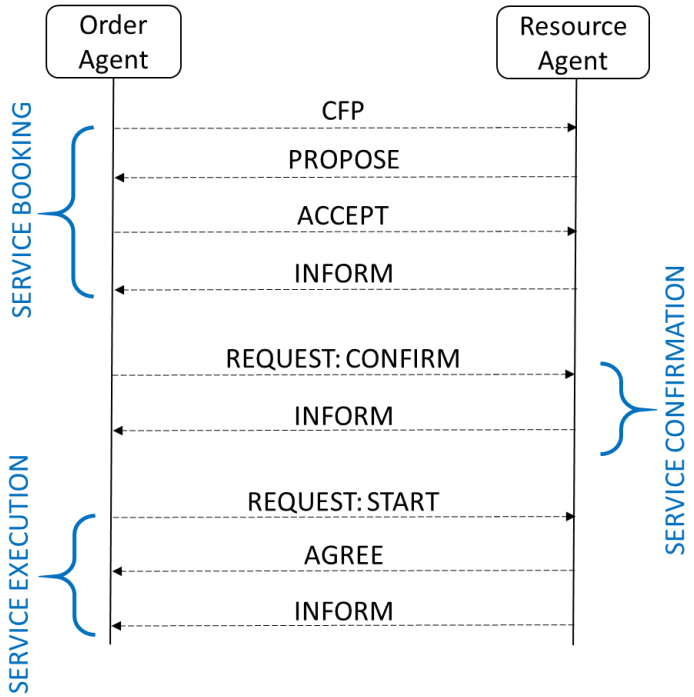

Figure 4: Communication between an Order agent and a Resource agent.

The booking of services is accomplished through a contract net protocol between the Order agent and the Resource agents. After the Order agent obtains the agent identifiers of all the Resource agents capable of performing a specific service, it initiates the communication protocol with each Resource agent. The protocol commences with the sending of Call For Proposal (CFP) messages to each Resource agent. The Resource agents reply to this CFP message with a proposal. The received proposals are compared and the Resource agent that sent the best proposal is sent an Accept Proposal message. Once the selected Resource agent replies with an Inform message, the booking is completed.

When the product that is controlled by the Order agent is ready for the next booked service to be performed on it, the Order agent must first confirm that the service booking is still valid – this is done by using the simple FIPA Rational-Effect (RE) protocol. The Order agent initiates the protocol by sending a Request message to the booked Resource agent. This request message contains a XML string in its content slot, which contains the "confirm" string in the element holding data for the message type. The Resource agent parses the XML string content of the request message and identifies it as a confirmation message. If the details of the Order agent are present in the bookings list of the resource agent, it replies with an *inform* message (if not, a *failure* message is sent – this is an indication of a fault in the execution of the Order agent). The confirmation step is included in the communication protocol as an additional check.

Upon receiving confirmation, the Order agent again initiates a simple RE protocol – in this case, the content slot of the request message is similar to the string version of the XML document presented in section 4.2.1. The Resource agent identifies the request as a Start message and immediately replies to the Order agent with an Agree message and start the execution of the service. The Agree message provides an indication to the Order agent that execution of the service has started on the product – this indication can be used to start a timer, which can indicate when an error has occurred in the Resource agent. Upon completion of the service, the Resource agent sends an Inform message to the Order agent.

4.2.3 Interaction with the Transport Agent

Most of the services performed at the workstations involves physical interaction with the carriers of the conveyor – e.g. at the input of the Electrical Test Station (ETS) products are removed from carriers for testing and, upon completion, are placed back on empty carriers available at the output of the station. This physical interaction between resources and the conveyor at the workstations is replicated in the virtual interaction, i.e. in the communication between the various Resource agents and the Transport agent.

The architecture of this MAS dictates that the coordination of services is done by Order agents – e.g. an Order agent will trigger the execution of a transportation service and, once completed, will thereafter trigger the execution of a testing service. The Order is blind to the interaction between the ETS and Transport agent necessary for the testing service to be executed – this interaction is completed through Resource-to-Resource communication.

From the physical system, two types of interaction between resources and the conveyor are identified: the placing of products on empty carriers and the removal of products from carriers. These physical interactions are represented by two AchieveRE protocols – one performing a *binding_request* and the other a *release_request*. The *binding_request* is used to initiate the placement of a product on a carrier, i.e. the binding of a product to a carrier. Alternatively, the *release_request* initiates the removal of a product from a carrier, so that a previously bound product is released from a carrier. The sequence of communication between an Order, ETS and Transport agent for the execution of a testing service is illustrated in Figure 5.

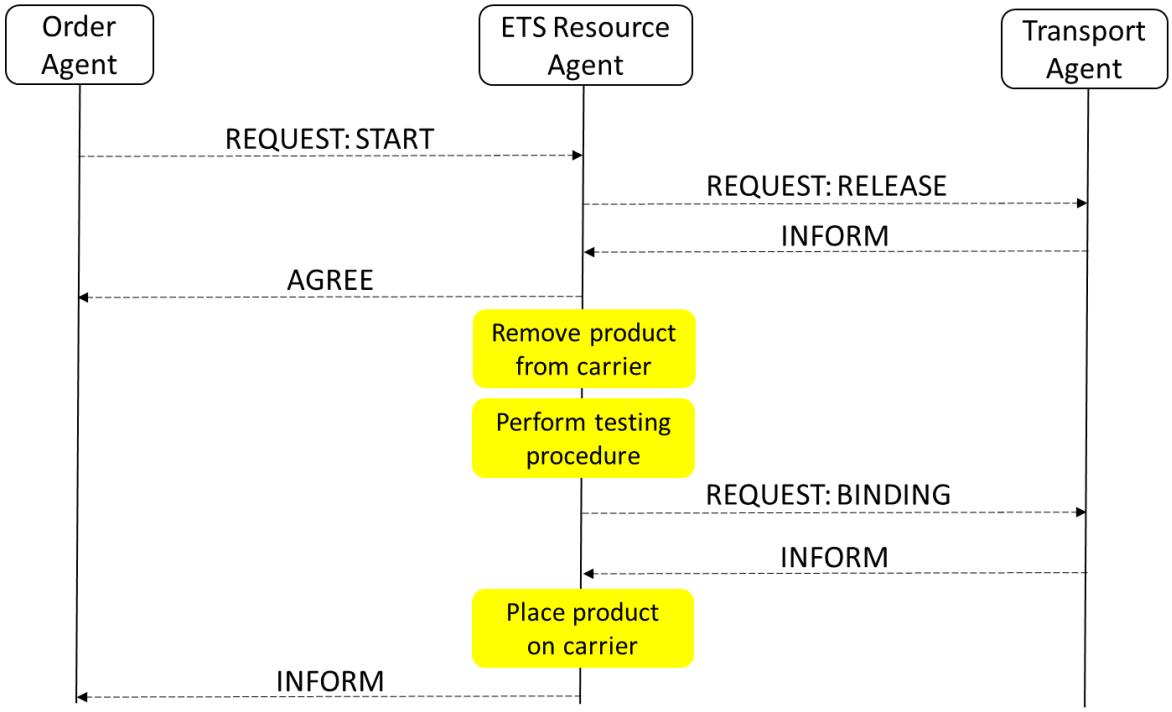

Figure 5: Communication sequence between the Order, ETS Resource and Transport agents.

Each type of request is accompanied by the exchange of important information. With a *binding_request* message the Resource agent must include information regarding the type of product that it wants to place on a carrier – since carriers might be fitted with fixtures that are specifically designed for certain product types, this information is used by the Transport agent to determine if a suitable carrier is available at the workstation. The Transport agent will reply with the result of the request – either an *inform* or *failure*. If the result is true, the specific position on the carrier may be specified, as carriers can be fitted with multiple fixtures and are thus capable of carrying more than one product at a time.

```
<?xml version="1.0" encoding="UTF-8"?>
<message>
	<initiator>ResourceAgent_R06</initiator>
	<responder>TransportAgent</responder>
	<msg>
		<message_type>binding_request</message_type>
		<conversation_ID>C021</conversation_ID>
		<result>undefined</result>
		<info>
			<order_ID>OA43</order_ID>
			<product_ID>P02</product_ID>
		</info>
	</msg>
</message>
```

The Transport agent will reply with the result of the request – either an *inform* or *failure*. If the result is true, the specific position on the carrier may be specified, as carriers can be fitted with multiple fixtures and are thus capable of carrying more than one product at a time.

```
<?xml version="1.0" encoding="UTF-8"?>
<message>
	<initiator>ResourceAgent_R06</initiator>
	<responder>TransportAgent</responder>
	<msg>
		<message_type>binding_request</message_type>
		<conversation_ID>C021</conversation_ID>
		<result>true</result>
		<info>
			<place_coords>
				<x>0.0</x>
				<y>200.0</y>
				<z>10.0</z>
				<ang>0.0</ang>
			</place_coords>
		</info>
	</msg>
</message>
```

For a *release_request*, the Resource agent must specify the product to be released, based on the Order agent that governs it. For a *release_request* message, the content slot of the FIPA RE Request message will contain the following XML string:

```
<?xml version="1.0" encoding="UTF-8"?>
<message>
      <initiator>ResourceAgent_R06</initiator>
      <responder>TransportAgent</responder>
      <msg>
            <message_type>release_request</message_type>
            <conversation_ID>C027</conversation_ID>
            <result>undefined</result>
            <info>
                  <order_ID>OA43</order_ID>
            </info>
      </msg>
</message>
```

Should the requested product be available on the carrier at the workstation, the Transport agent will reply with an *inform* message. If multiple products are present on the carrier, the Transport agent must also specify the position of the product on the carrier – the information is structured as follows:

```
<?xml version="1.0" encoding="UTF-8"?>
<message>
      <initiator>ResourceAgent_R06</initiator>
      <responder>TransportAgent</responder>
      <msg>
            <message_type>release_request</message_type>
            <conversation_ID>C027</conversation_ID>
            <result>true</result>
            <info>
                  <pick_coords>
                        <x>0.0</x>
                        <y>200.0</y>
                        <z>10.0</z>
                        <ang>0.0</ang>
                  </pick_coords>
            </info>
      </msg>
</message>
```

4.3 Agents

The MAS implementation contains agents of four types, as prescribed by PROSA, namely Product, Resource, Order and Staff agents. The functionality of each agent type is described in this section.

4.3.1 Product Agent

The Product agent exhibits the behaviour of a simple server, only replying to received messages requesting the information for a specified product. The agent employs an AchieveREResponder behaviour to receive and handle request messages from Order agents. These request messages specify the product type in the content of the request message. The product information is then retrieved from the product information XML file and is converted

to an XML string. The product information string is then added to the content slot of the inform ACL message that is replied to the requesting agent.

An extract from the product information XML file is shown below:

```
<?xml version="1.0" encoding="UTF-8"?>
<product_information>
     <product id="p01">
          <task_info>
               <task type="feed">
                         <place_coords>
                              <x>0.0</x>
                              <y>0.0</y>
                              <z>0.0</z>
                              <ang>0.0</ang>
                         </place_coords>
               </task>
               <task type="transport">
                    <origin>"undefined"</origin>
                    <destination>"undefined"</destination>
               </task>
               •
               •
               •
          </task_info>
     </product>
```

4.3.2 Resource Agent

The Resource agents employ behaviours to negotiate service bookings via the contract net protocol, handle confirmation and start requests and the execution of the resource's service.

To negotiate service bookings, Resource agents employ the ContractNetResponder behaviour. In the handleCFP() method, the agent creates a proposal – this proposal contains a value that indicates the length of the resource's booking list. If a proposal is successful and an accept_proposal message is received, the information of the booking Order agent is added to the bookings list.

The confirmation of service bookings by an Order agent, as discussed in section 4.2.2, is handled with an AchieveREResponder behaviour added to the execution of Resource agents. The behaviour matches every incoming message to a message template – the message template uses a regular expression to evaluate the XML string content of the received message.

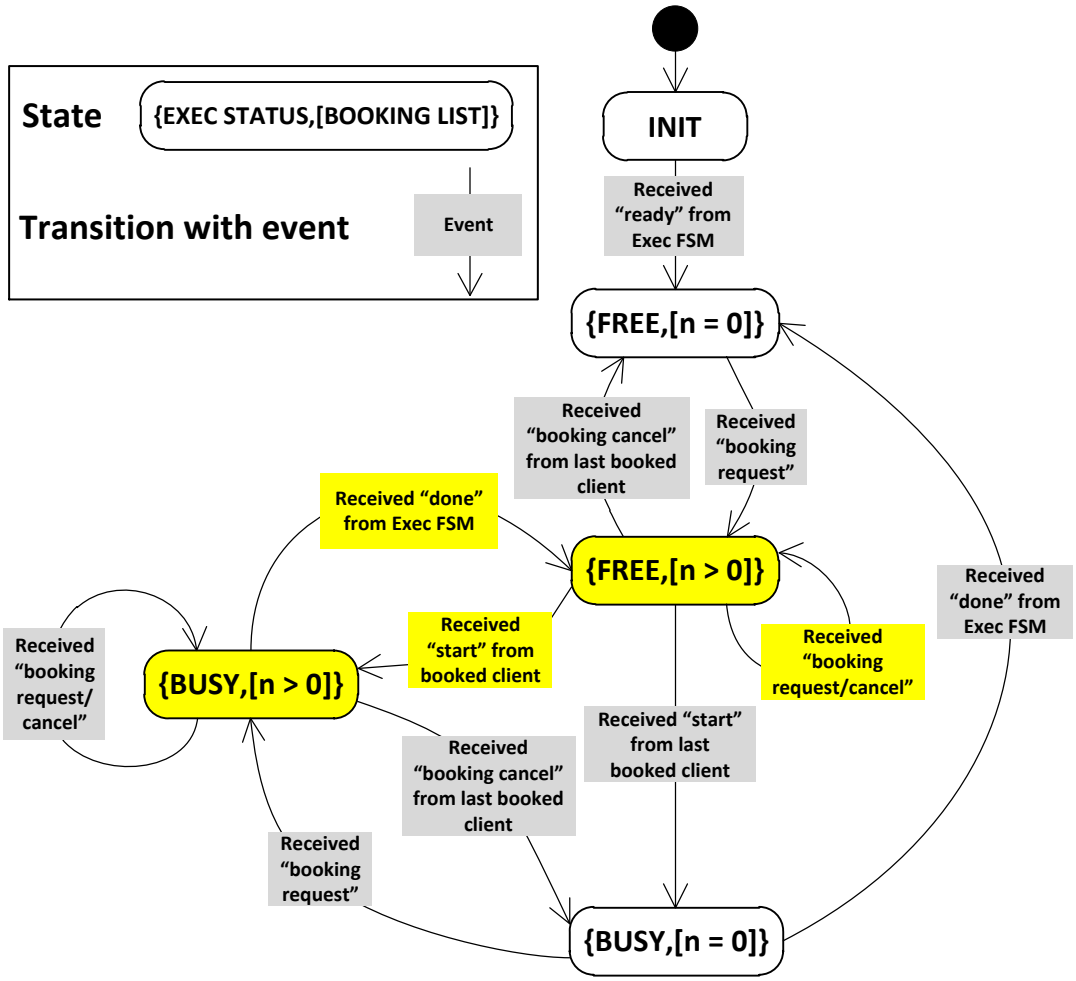

Figure 6: State diagram for the Agenda Management component of the Resource agent.

The agent behaviour, as described by the state diagram in Figure 6, is constructed through three concurrently active JADE behaviours. Concurrency within the execution of the agent is needed to ensure that the Resource agent remains available for communication even when it is performing its designated service. One behaviour is responsible for handle service booking requests using the contract net protocol and another behaviours handles the confirmation protocol for service bookings. The third behaviour is responsible for the execution of the service as initiated by an Order agent that previously booked the Resource agent's service.

To handle service bookings from Order agents a ContractNetResponder behaviour is added to the execution of the Resource agent. The ContractNetResponder behaviour is built on the JADE finite state machine behaviour – the behaviour is constructed with the necessary states to participate in a CNP negotiation. The states provide the necessary methods to handle the communication with the CNP initiator agent.

An AchieveREResponder is added to the agent execution to handle the confirmation of service bookings by Order agents. Similar to the ContractNetResponder behaviour described above, the AchieveREResponder behaviour embodies a finite state machine that is configured to handle the communication of the FIPA RE protocol. The behaviour compares a received message with a defined message template – in this case, the content of the message is matched to a template specifying a confirmation message.

To handle the communication intended to start the execution of a booked service, a SSResponderDispatcher behaviour is added. The SSResponderDispatcher behaviour launches a behaviour that is dedicated to handle the communication with one specific agent, for a single communication session only. The createResponder() method of this behaviour allows the developer to specify which behaviour to handle the session. Here, a SSIteratedAchieveREResponder is utilised to handle the communication involved with the execution of the Resource's service. The SSIteratedAchieveREResponder is similar to the AchieveREResponder behaviour discussed earlier, but is different in the sense that the behaviour terminates after a single communication session.

For the SSIteratedAchieveREResponder behaviour that is launched to handle the start message, the standard handleRequest() method is overwritten. Instead, by using the registerHandleRequest() method, the actions that occur when a start request is received can be specified by the developer. This method is then used to add an FSMBehaviour that describes the execution of the Resource's service.

The behaviours described above, up to the service execution FSMBehviour, are generic for all Resource agents. Each Resource agent adds a FSMBehaviour that is specific to the service(s) that it can perform. The behaviour executes all the actions that are necessary to perform the booked service. Upon completion, the FSMBehaviour returns the result of the execution to the SSIteratedAchieveREResponder behaviour, which in turn replies to the Order agent with an *inform* or *failure* message.

4.3.3 Order Agent

Order agents must book and trigger the execution of the services, provided by Resource agents, to complete all the tasks specified in the product information of a certain product type. The finite state machine behaviour to implement this functionality and facilitate the necessary communication (as explained in section 4.2.2) is described in this section.

The Order agent firstly adds a behaviour to request and receive the product information from the Product agent – this is done by an AchieveREInitiator behaviour. Thereafter, the Order agent executes a FSMBehaviour until the product that it is responsible for is completed. The FSMBehaviour embodies the state diagram shown in Figure 7.

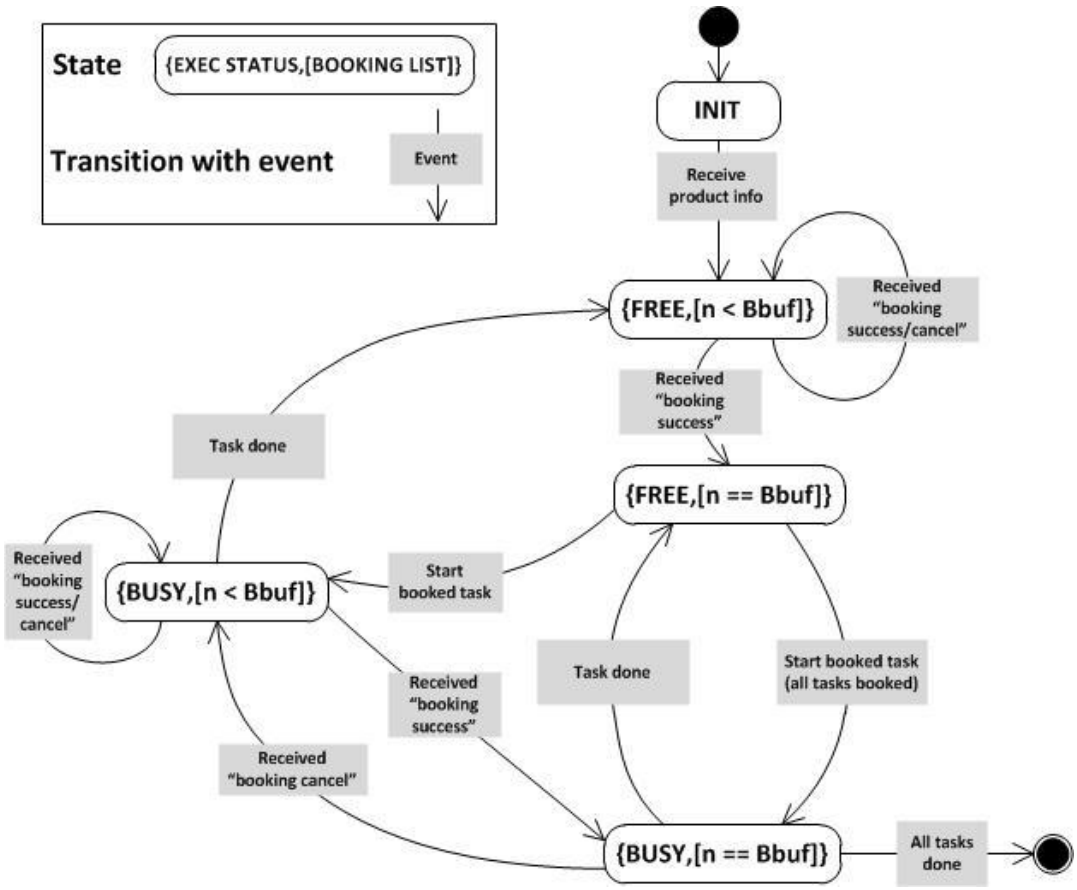

Figure 7: State diagram for the behaviour of an Order agent.

The execution in the initial state, `free_booking`, is implemented using a TickerBehaviour. The function of this behaviour is to perform service bookings sequentially for the services specified in the product task list. This behaviour is executed periodically, adding a new ContractNetInitiator behaviour for service booking every time. The number of service bookings to be made in advance is determined by the user-defined booking buffer variable (`Bbuf` in Figure 7) – when the number of bookings made (`n` in Figure 7) is equal to the booking buffer, the FSMBehaviour transitions to the next state.

In the `free_booked` state a OneShotBehaviour is added that triggers the execution of the first booked service in the bookings list of the Order agent. The execution, which includes the confirmation and starting of the service via communication with the booked Resource agent, is performed by a SequentialBehaviour (discussed at the end of this section). The SequentialBehaviour is started in a separate thread to simplify concurrency of the Order agent behaviours. The service that is being executed is removed from the bookings list, meaning that the number of entries is less than the specified booking buffer – the FSMBehaviour now transitions to the `busy_booking` state.

The behaviour of the `busy_booking` state is similar to the `free_booking` state. A TickerBehaviour adds ContractNetInitiator behaviours until the booking buffer is reached. When all required bookings have been made, a state transition to the `busy_booked` state occurs.

The `busy_booked` state also implements a TickerBehaviour, but here the behaviour just periodically checks the status of the booking list and service execution. During the first

execution of the TickerBehaviour an AchieveREResponder behaviour is added. This behaviour will receive any booking cacellations from booked Resource agents (which can occur when the Resource agent either fails or is manually shut down) – in which case the canceled booking is removed from the bookings list and a state transition is triggered back to the `busy_booking` state. Also, at every execution, the variable indicating the status of service completion is checked. If the service is completed, the state transitions to the `free_booked` status again so that the execution of the next booked service can be started. If the service is completed and it was the last service required for the product, the FSMBehaviour transitions to the `done` state to terminate execution of the Order agent.

When an Order agent has made enough service bookings to fill the booking buffer, the first service in the bookings list (which corresponds to the next service to be performed according to the product information) can be started. The confirmation of the service bookings and the starting of the service execution, through the protocols discussed in section 4.2.2, are done in a separate behaviour to the FSMBehaviour described above, and in a dedicated thread. The use of a dedicated thread, instead of adding concurrency through behaviours, was selected due to the simplicity of implementation. The thread is again terminated once the service is completed by the Resource agent.

The thread implements a JADE SequentialBehaviour to sequentially execute two AchieveREInitaitor behaviours – one for confirming the service booking and the other to start the execution. Once the Resource agent indicates the successful completion of the service execution through an *inform* message, the necessary updates are made to the agent variables and the behaviour, and thereafter the thread, terminates.

4.3.4 Staff Agents

Staff agents are included in the MAS implementation to provide the functionality that is not exhibited by the Product, Order and Resource agents. Apart from the Staff agents included in JADE (such as the Directory Facilitator), two Staff agents were added to the implementation: an Order Management agent and a Performance Logger agent.

4.3.4.1 Order Management Agent

As the name suggests, the Order Manager (OM) agent is responsible for the management of the Order agents within the MAS. The OM agent maintains a Graphical User Interface (GUI) to receive input from the user concerning the creation of Order agents. The user can specify the number of Order agents and the type of products they must produce – this information can be entered manually in the GUI fields, or a XML production schedule filename can be specified.

From the input information, the OM agent launches Order agents by sending requests to the Agent Management System. The OM agent displays the number of active Order agent in the MAS in the GUI – this number is incremented with each launched agent. Once Order agents have completed all required tasks, a *done* message is sent to the OM agent before the agent terminates – the number of Order agents is decremented when a *done* message is received.

4.3.4.2 Performance Logger Agent

To gather diagnostic information on the performance of Resource and Order agents, a Performance Logger (PL) agent is added to the MAS. The PL agent records the number of times a Resource agent performs its service, the duration of each service execution and the total time that the Resource agent spends in service execution – all data required to calculate the utilization of the resource during a period of production. The agent also records the start and

end times of the execution of Order agent, in order to provide data for the calculation of the time-in-system of each product and the overall production throughput.

Resource agents send a *start* message to the PL agent every time a service execution is started and a *done* message when the execution is completed. The PL agent starts a timer for every *start* message received from a Resource agent and stops the timer when the *done* message is received. The information is stored in an ArrayList data structure – the entries of the ArrayList are of a custom class type, with fields for the Resource agent's name, its activity status, the total number of services performed and the total time that the Resource has been active. Similarly, Order agents send corresponding messages upon their instantiation and termination.

5 Conclusion

Holonic control architectures have been frequently used in manufacturing systems to reduce the complexity of the control system, simplify reconfiguration and improve robustness. Multi-Agent Systems (MASs) have often been used to facilitate the implementation of holonic control due to the similarities between holons and software agents. Of the MAS development tools used to implement holonic control in manufacturing systems, the Java Agent Development (JADE) framework is the most popular choice.

This paper presented a JADE MAS implementation of a reference holonic control architecture for a manufacturing cell. The implementation used an electric circuit breaker assembly and testing cell as case study, of which the various functional components were mapped to the holon types as prescribed by the PROSA reference architecture. The high level control components of the holons are implemented as agents in the MAS. The communication between the agents is described and the implementation of the functionality for each agent type is discussed.

The implementation of holonic control using JADE MASs has become the *status quo* within the field of holonic and reconfigurable manufacturing systems. For this reason, the presented MAS implementation is used as a baseline for a comparison with an equivalent holonic control implementation that is based on the Erlang programming language (details on this implementation can be found in Kruger and Basson (2015; 2017a)). The evaluation criteria and the comparison are presented in Kruger and Basson (2017b; 2017c).

6 References

Bi, Z.M., Wang, L., and Lang, S.Y.T., 2007. Current Status of Reconfigurable Assembly Systems. *International Journal of Manufacturing Research*, Inderscience. Vol. 2, No. 3: 303 - 328.

Bi, Z.M., Lang, S.Y.T., Shen, W., and Wang, L., 2008. Reconfigurable Manufacturing Systems: The State of the Art. *International Journal of Production Research*. Vol. 46, No. 4: 967 - 992.

Bellifemine, F., Caire, G. and Greenwood, G., 2007. *Developing Multi-Agent Systems with JADE*. John Wiley & Sons, Ltd.

Chirn, J.L. and McFarlane, D., 2000. A Holonic Component-based Approach to Reconfigurable Manufacturing Control Architecture. *Proceedings of the International Workshop on Industrial Applications of Holonic and Multi-Agent Systems.* pp. 219–223.

ElMaraghy, H., 2006. Flexible and Reconfigurable Manufacturing System Paradigms. *International Journal of Flexible Manufacturing System*. Vol. 17: 61-276.

Giret, A. and Botti, V., 2009. Engineering Holonic Manufacturing Systems. *Computers in Industry*. Vol. 60:428-440.

Kotak, D., Wu, S., Fleetwood, M., and Tamoto, H., 2003. Agent-Based Holonic Design and Operations Environment for Distributed Manufacturing. *Computers in Industry*. Vol. 52: 95–108.

Kruger, K. and Basson, A.H., 2015. Implementation of an Erlang-Based resource Holon for a Holonic Manufacturing Cell. *Service Orientation in Holonic and Multi-Agent Manufacturing*, Studies in Computational Intelligence, Springer International Publishing.

Kruger, K. and Basson, A.H., 2016. Erlang-based Holonic Controller for a Modular Conveyor System. *6th Workshop on Service Orientation in Holonic and Multi-Agent Manufacturing*. Lisbon, Portugal (October 2016).

Kruger, K. and Basson, A.H., 2017a. Erlang-Based Control Implementation for a Holonic Manufacturing Cell. *International Journal of Computer Integrated Manufacturing*. Vol. 30, No. 6:641-652.

Kruger, K. and Basson, A.H., 2017b. Evaluation Criteria for Holonic Control Implementations in Manufacturing Systems. *Submitted to the International Journal of Computer Integrated Manufacturing, September 2017.*

Kruger, K. and Basson, A.H., 2017c. Comparison of Multi-Agent System and Erlang Holonic Control Implementations for a Manufacturing Cell. *Submitted to the International Journal of Computer Integrated Manufacturing, September 2017.*

Leitao, P. and Restivo, F.J., 2006. ADACOR: A Holonic Architecture for Agile and Adaptive Manufacturing Control. *Computers in Industry*. Vol. 57, No. 2: 121-130.

Martinsen, K., Haga, E., Dransfeld, S., and Watterwald, L.E., 2007. Robust, Flexible and Fast Reconfigurable Assembly System for Automotive Air-brake Couplings. *Intelligent Computation in Manufacturing Engineering*. Vol. 6.

Mehrabi, M.G., Ulsoy, A.G., and Koren, Y., 2000. Reconfigurable Manufacturing Systems: Key to Future Manufacturing. *Journal of Intelligent Manufacturing*. Vol. 13: 135-146.

Paolucci, M. and Sacile, R., 2005. *Agent-Based Manufacturing and Control Systems*. London: CRC Press.

Scholz-Reiter, B. and Freitag, M., 2007. Autonomous Processes in Assembly Systems. *Annals of the CIRP*. Vol. 56: 712–730.

Van Brussel, H., Wyns, J., Valckenaers, P., Bongaerts, L., and Peeters, P., 1998. Reference Architecture for Holonic Manufacturing Systems: PROSA. *Computers in Industry*. Vol. 37: 255–274.

Vyatkin, V., 2007. IEC 61499 Function Blocks for Embedded and Distributed Control Systems Design. *North Carolina: Instrumentation, Systems and Automation Society, ISA.*