Proceedings of the 6th WSEAS International Conference on Robotics, Control and Manufacturing Technology, Hangzhou, China, April 16-18, 2006 (pp130-135)

An Agent-Based Manufacturing System using Asynchronous Messaging

PAULO SOUSA
GECAD – Knowledge Engineering and Decision Support Group
Institute of Engineering – Polytechnic of Porto
R. Dr. António Bernardino de Almeida, 431, 4200-072 Porto
PORTUGAL

Abstract - Recent changes in the society and economy lead to the need to study new approaches to model and develop an emerging generation of manufacturing systems. Agent-based systems appear as a well suited approach for supporting distributed intelligence in manufacturing, allowing autonomy, reactive and pro-active behaviours, and social abilities support. This paper presents *Fabricare*, a prototype system for handling the problem of dynamic scheduling of manufacturing orders. We assume the holonic paradigm as a "vision" and overall guiding structure, while the agent paradigm is used as a development or implementation technology. In this implementation we use a main stream programming environment (.net) and the message passing paradigm to convert the existing prototype system developed in Prolog.

Keywords: agent-based manufacturing, dynamic scheduling, asynchronous messaging.

1 Introduction

Manufacturing has changed (and will continue to change) [1]; there is a shift from mechanization and mass production to flexible manufacturing and product customization as well as customized "digital" services. Innovation is no longer neglectable and knowledge has become the primary growth factor as opposed to capital and labor. The manufacturing company of the future will use [1] [6] intelligent processes and flexible tools to achieve new dimensions of flexibility and reactivity; will support its decisions with knowledge based systems; and will operate on world-wide networks of plants, suppliers, delivery and service centers; taking attention to change and discontinuity in order to achieve competitive advantage.

As observed by [13], rigid, static and hierarchic manufacturing systems are expected to be replaced by adaptable and reconfigurable distributed manufacturing systems where autonomous and flexible manufacturing entities cooperate in a coherent and coordinated manner [9] [12]. In order to deal with the identified problems with current manufacturing systems and prepare them for the expected future scenarios, the new generation of manufacturing systems must support attributes as decentralization, distribution, autonomy, adaptability, and incomplete information handling [12].

Agent-based systems are "suited for modular, decentralized, dynamic, complex and ill-applications" [7], showing "a large number of interactions among components" [5].

Holonic Systems [4] are based on the concept of holon with its dual nature representing the whole and part, allowing for a holon to be part of another holons at the same time that itself is made up by other holons. This enables the construction of complex systems that are efficient resource managers and highly resilient to changes and disturbances. Holons are by definition autonomous and cooperative, meaning they have the ability (and responsibility) to create and execute its own plans, but also engage in the execution of mutually accepted plans with other holons.

In the manufacturing arena, *Holonic Manufacturing Systems* [13] apply the holonic concept to the manufacturing enterprise, allowing the existence of a dynamic and decentralized manufacturing process where changes are applied dynamically and continuously. Agent-based manufacturing systems and holonic manufacturing systems are at the same time overlapping and complementing each other, mainly by using agents as a development tool for the holonic concept.

To the scope of this work, a *holon* is understood as a logical design entity in the system architecture, and is implemented as an *agent*. Thus, conceptually, a design entity in the architecture is a Holon while the software application that models that entity is an Agent, i.e., the holonic paradigm provides the vision and "glue" for the architecture while the agent paradigm provides the implementation.

2 Work Description

2.1 Agent's description

A *task holon* represents a manufacturing order to execute a certain quantity of a specific product on the shop floor. This kind of holon has as its objective to schedule the order and to monitor its execution. Its life cycle begins when the manufacturing order is created (either to fulfil a customer order or to balance stocks). During its existence the task holon will negotiate with resource holons the execution of the operations needed to perform the ordered product. It will then monitor the execution of the task and renegotiate if necessary. The holon will cease existing when the order is fulfilled or cancelled.

A *resource holon* represents the current state of a physical resource on the shop floor. The resource's list of activities is called agenda, stating what to do and when. The resource is able to perform operations necessary to execute products (e.g. drill). A resource holon can represent a single resource or a work cell composed of several resources. The objective of a resource holon is to control the physical equipment, providing information about its abilities and status to the system and managing the scheduled activities. Its life cycle is very long, since it is expected that a resource is fully operational for long periods of time. During its existence, the resource holon executes the commands sent by the resource controller and negotiates with task holons the scheduling of manufacturing orders.

For a more detailed description of each agent's knowledge base and incomplete information handling see [10] and for a description of operational behaviour see [11].

2.2 Interaction

For the scheduling of task's sub-operations, the Task holon will negotiate with Resource holons, using an extension of the Contract Net Protocol [8] with a cooperation phase between service providers (i.e. resource holons). The Resource holons will use constraint propagation in order to guarantee the relationships among different operations that aim at the same task. This new protocol is called *Contract Net with Constraint Propagation Protocol* (CNCPP).

This protocol has six steps (it uses a scheduling procedure based on agendas and due dates):

1 when a new task arrives at the system (via *task launcher*), it will obtain information from the process planning holon about the product's alternative plans and will choose one based on a set of criteria given by the scheduling holon based on the plant current status.

2 the task holon will then contact the resource holons able to perform each needed operation according to the selected plan, informing the resource holon of the requested operation and its parameters, as well as informing it about resource holons contacted by predecessor and successor operations.

3 the resource holons will then begin the forward influence phase by exchanging messages with their free agendas constrained with the agenda from predecessor resources in order to determine the lower limit of each time interval.

4 at the end of the forward influence phase, the backward influence phase begins. The resources will then exchange messages with their free agendas further constrained with the agendas from successor resources in order to determine the effective upper limit of each time interval.

5 each resource is now able to make a final bid (feasible in its own agenda and respecting the constraints) to the task holon. If there are alternative resources a bid is made for every combination.

6 after receiving all ther bids from the resource holons, the task holon will analyze them and decide on the combination of resources and time intervals to use (this selection is based on heuristics, e.g., greatest slack, and the total cost of the solution). The task holon will then inform the resource holons with a contract or cancel message.

Since multiple tasks can be negotiated at the same time, conflicts may arise if some resources are used in the same time interval for different tasks. To overcome this problem there is a pre-negotiation step in the protocol where each task holon will ask for authorization from the *scheduling holon*, which maintains a list of negotiating resources and respective time windows. Only in case of non-overlapping a "green light" will be given to the negotiation.

Each negotiation uses the set of holons that are present and available at that time. The runing agents are all registered in a special agent that acts as a *Directory Service*. Before entering a negotiation, a task agent must query the directory service for running resource agents.

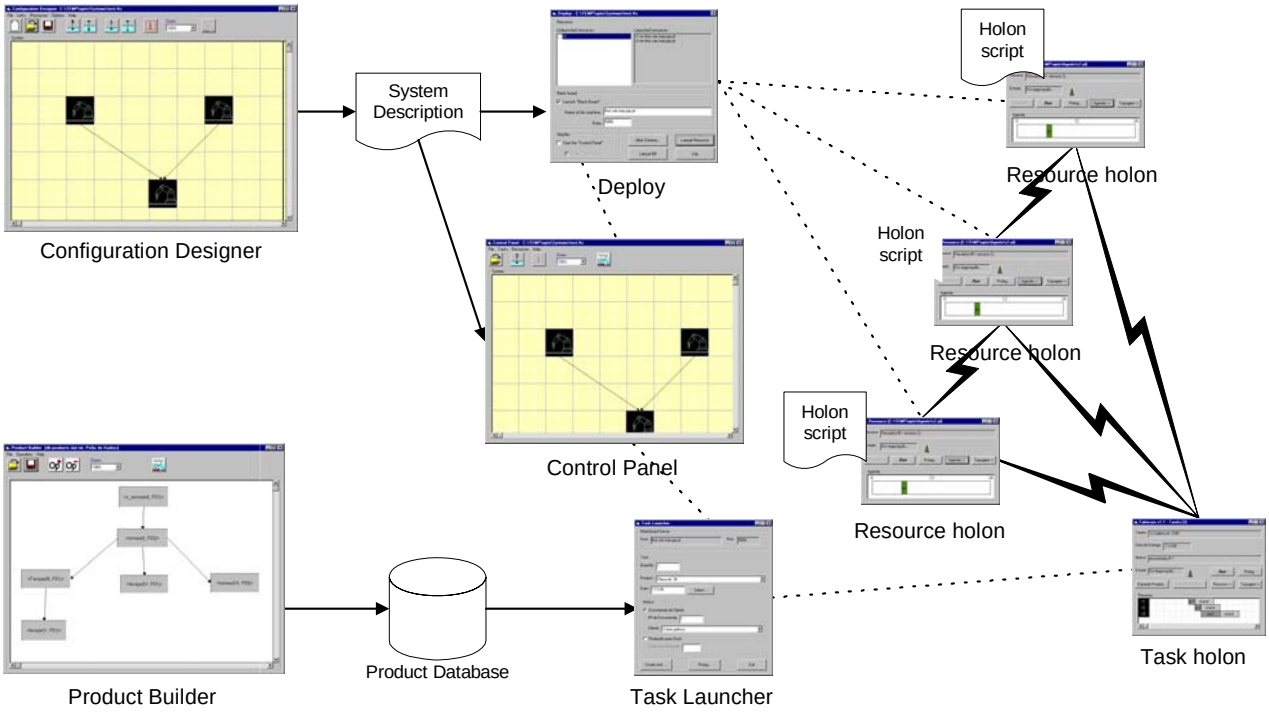

Figure 1 - Fabricare prototype

2.3 The existing prototype

Figure 1 presents the *Fabricare* prototype suite for scheduling of manufacturing orders, composed of several applications.

The *Configuration Designer* allows specifying the resource agents in the factory plant, and to some extent, represent graphically the physical layout of the resources. The system description is read by the *Deployment* tool, which launches *Resource Holons* on the desired machines. Each resource holon is composed of a common kernel and a specific script (both written in Prolog) that comprise the holon's 'mental' state (e.g., name, resource's agenda).

The *Control Panel* is the interface to the system's operation, monitoring and controlling running holons. This tool also allows the user to launch tasks (manufacturing orders) in the system by evoking the *Task Launcher* tool, which prompts the user for data about the order and dynamically creates a *Task Holon* for that order. One last tool in the suite is the *Product Builder*, which allows to generate graphically a product's process plan. The several operations in the plan are the abilities of the physical resources (modelled in the resource holons).

3 The New Prototype system

3.1 General structure

We are currently migrating the *Fabricare* system to a new platform (Microsoft .Net 2.0) and using the Microsoft Message Queuing (MSMQ) component of the windows operating system for asynchronous communications. The main rationale for this migration is the use of a mainstream development environment instead of Prolog and Linda. Furthermore, the current version of the system uses synchronous communications which involved some tricks and timer based pooling of messages, as well as a single centralized message board (tuple space) for all the messages exchanged in the system.

The new implementation offers a more natural way to develop each agent, since each agent owns its own queue (which may be distributed) and the message handling is now done in a event driven way, allowing the agent's main thread to execute other operations and be interrupted only when a new message arrives.

We have divided the system in seven major projects (Figure 2):

- *Agents* – contains the agent's object model and interfaces for all the agents in the system (i.e., Task, Resource, Directory Service, Process Planning and Scheduling)
- *AgentsImplementation* – contains the implementation of each service defined in an agent interface. These implementations are independent of the communication mechanism.
- *Messages* – contains the message's data structures.
- *MessageHandling* – contains auxiliary classes for building and translating message contracts to agent contracts and vice-versa.
- *Communications* – contains generic interfaces for abstracting the communication mechanism and al-

lowing for the independent evolution/substitution of the agent's implementation and communication mechanism.

- *Communications.Messaging* – a concrete implementation of a communication mechanism. In this case using the MSMQ component of the windows operation system.

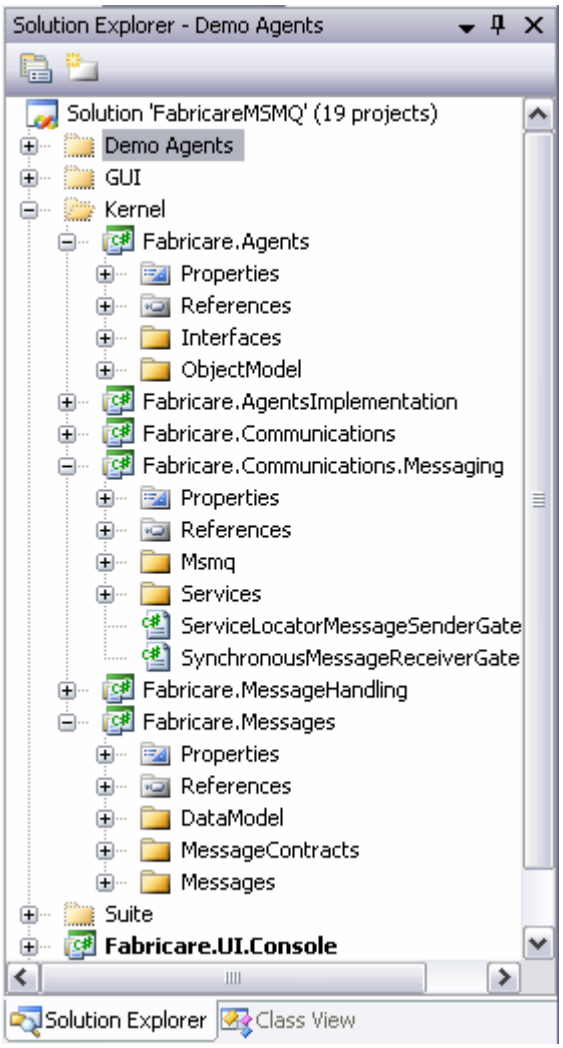

Figure 2 – implemented projects

3.2 Messages

We have defined a well know contract in the form on C# interfaces for each agent. In fact there are two contracts for each agent. What we call the *message contract* and the *agent contract*. Message contracts are used for the exchange of messages using the MSMQ component as XML data structures. Internally however, the agent uses a rich object model that is different from the message plain structures.

For example, the message contract for registering an agent in the directory service is as follows:

```
public struct RegisterRequest {
      public ID AgentID;
      public Location Where;
      public ID[] Capabilities;
}

public struct RegisterResponse {
      public bool Suceeded;
}
```

The agent interface for this operation is:

```
bool Register(ID sender, Location loc,
              IList<ID> capabilities);
```

The `ID` class referred in the message contract is different from the `ID` class referred in the agent contract. In the case of the message contract it is a simple data structure with public fields while in the case of the agent contract it is a true object-oriented class with rich functionality.

3.3 Message handling

As a goal, it was decided upfront that the system should be flexible to accommodate different communication mechanisms (or at least allow for a easy substation of the underlying communication mechanism). For the moment, we decided to use the MSMQ component of the windows operating system as it provides an asynchronous and reliable way for transmitting messages. The choice of using messaging instead of remoting was due (1) to the ease of use for asynchronous operation and (2) to the fact that the future Windows Communication Foundation subsystem of the .Net Framework will be based on the messaging paradigm (which makes messaging a more secure bet for future evolutions).

We factored out the functionality for low level MSMQ interaction and the actual message processing in the `MessageHandling` project. A set of classes called message processors is defined responsible for the translation of the message contract and agent contract and the routing of the message request to the actual object implementing the agent algorithm. These message processor objects are created by the MSMQ Service objects (e.g., `ResourceHolonService`). This allows for a reuse of this message processing functionality independent of the actual communication mechanism in use.

Each message processor implements the following interface:

```
public interface IMessageProcessor {
      Type[] GetRequestBodyTypes();
      bool ProcessMessage(object inBody,
                          ref object outBody);
}
```

A typical implementation looks like this:

```
public bool ProcessMessage(object inBody,
                           ref object outBody)
{
MessageToAgentModelTranslator tr =
         new MessageToAgentModelTranslator();

if (inBody is RegisterRequest)
{
   RegisterRequest req =
                  (RegisterRequest)inBody;

   bool ret = impl.Register(
               tr.Translate(req.AgentID),
               tr.Translate(req.Where),
               tr.Translate(req.Capabilities)
               );

   DirServiceResponseMessageBuilder builder =
      new DirServiceResponseMessageBuilder();
   RegisterResponse resp =
      builder.CreateRegisterResponse(ret);

   outBody = resp;
   return true;
}
else if (inBody is CanDoRequest)
      ...
...
else
      return false;
}
```

Since a resource holon is in fact a specialization of an agent and a specialization of an holon, several message processors can be combined for handling each of the recognized set of messages an agent/holon can answer. The combination of these message processors follows the Chain of Responsibility pattern [2].

3.4 Communications

In order to decouple the implementation of the agent's service and the communication mechanism, there are two special interfaces in the `Communications` project, `IAttachableToCommunicationsLayer` and `IMessageSender`

```
public interface IAttachableToCommLayer
{
      void AttachToCommunicationsLayer(
                  IMessageSender gtw,
                  Agents.Location loc);
}

public interface IMessageSender
{
      void SendMessage(ID destination,
                           object messageBody);
      void SendMessage(IList<ID> destinations,
                           object messageBody);
      object SendAndReceive(ID destination,
                  object messageBody,
                  Type[] expectedRespTypes);
}
```

Each agent implementation derives from a base class that implements the `IAttachableToCommunicationsLayer` interface. This interface allows the agent to receive an object that acts a message gateway that the agent uses to send messages to other agents. The objects implementing `IMessageSender` are created by the specific implementation of the communication mechanism being used; in this case MSMQ.

For example, to create a resource holon, one must create an object of type `ResourceHolonService` declared in the `Communications.Messaging` project. This object will read a configuration file, create a `ResourceHolon` object (the object responsible for the implementation of the resource holon algorithm declared in the `AgentsImplementation` project), create a specific message queue for that agent, attach the `ResourceHolon` object to the queue object, and start listening for messages.

The code for handling MSMQ is derived from the sample code available in [3]. We have extended the provided code with the ability to handle more than one type of message for each message queue. we also introduced a new class `ServiceLocatorMessageSenderGateway` that is used to send messages to a specific agent by querying the directory service to obtain the agent's queue address.

Whenever a message arrives at a MSMQ queue, the following code is executed (in the base `MQService` class):

```
protected virtual void OnMessage(Message inMsg)
{
inMsg.Formatter = GetFormatter();
object inBody = GetTypedMessageBody(inMsg);

if (inBody != null) {
     bool tratada = false;
     object outBody = null;
     foreach (IMessageProcessor p in procs)
     {
        if(p.ProcessMessage(inBody, ref outBody))
        {
           if (outBody != null)
              SendReply(outBody, inMsg);
           break;
        }
     }
}
}
```

This code will call the registered message processors until it finds a suitable one.

3.5 Implemented holons

The current version of the system has complete kernel functionality of the following holons:

- Directory service
- Process planning
- Scheduling holon
- Task holon

- Resource holon

For the time being, only a console user interface is available for each of these holons. The development of a new graphical user interface or the adaptation of the existing GUI applications to call the new kernel is still being considered. We think the later approach will provably be taken since the GUI is not the main focus of our work.

4 Summary

In this paper we addressed the ability to build and maintain computer-supported manufacturing systems able to cope with recent (and expected future) requirements. The *Fabricare* system resembles the distributed nature of manufacturing, thus allowing for a natural modelling of the real system. The *Fabricare* system combines resource-based holons with task-based holons, offering an easy way to access task activities that are supported in task-based holons as well as high adaptability to the dynamic nature of resource conditions and availability.

The presented negotiation protocol to regulate the interaction among the several agents in the system, the Contract Net with Constraint Propagation Protocol, has as a main characteristic the explicit cooperation phase between service providers (i.e. resources) motivated by the need to coordinate temporal relations of task's operations. The protocol also allows for dynamic participants, using the information of running agents stored in the directory service; and conflicts are avoided by serializing overlapping negotiation (concurrent time-windows for different resources/tasks).

The present migration of the existing prototype consists only of console applications (however, a migration of the original GUI applications done in VB 6.0 to VB.net was already made). We have developed the core holons of the system: directory service, process planning, scheduling holon, Task and resource holons.

The use of a mainstream programming environment makes it easy for undergrad students to enter the project and contribute with evolutions as part of their final course project/thesis.

One point of action for future work is the development of a GUI for each holon (or the adaptation of the existing GUI to call the new agent kernel).

Acknowledgments

The authors would like to acknowledge FCT, FEDER, POCTI, POSI, POCI and POSC for their support to R&D Projects and GECAD Unit.

References

[1] CVM, 1999, *Visionary Manufacturing Challenges for 2020*. (Washington: National Academic Press.)

[2] Gamma, E., Helm, R., Johnson, R. and Vissides, J. (1995) *Design patterns : elements of reusable object-oriented software*. Addison-Wesley

[3] Hophe, G. And Woolf, B. (2005) *Enterprise Integration Patterns : Designing, Building, and Deploying Messaging Solutions*. Addison-Wesley

[4] Koestler, A., 1967, *The Ghost in the Machine.* Hutchinson & Co: London.

[5] Kouiss, K.; pierreval, H. and mebarki, N., 1997, "Using Multi-Agent Architecture in FMS for Dynamic Scheduling". *Journal of Intelligent Manufacturing*, vol. 8, pp.41-47. Chapman & Hall.

[6] NGM, 1997, Next Generation Manufacturing – A Framework for Action. Next Generation Project Report, Agility Forum.

[7] Parunak, H., 1998, "What Can Agents Do in Industry and Why?". *Proc. of the Second International Conference on Co-operative Information Agents*. Paris, France. 3-8 July 1998.

[8] Smith, R., 1980, "The Contract Net Protocol". *IEEE Transactions on Computers*, vol. **C**-29(12).

[9] Solberg, J., and Kashyap, R., 1993, ERC Research in Intelligent Manufacturing Systems, *Proceedings of the IEEE*, 81(1), pp.25-41.

[10] Sousa, P., Ramos, C., and Neves, J., 2003, "The Fabricare Scheduling Prototype Suite: Agent interaction and knowledge base". *Journal of Intelligent Manufacturing*, 14(5), pp.441-455. October 2003. Kluwer Academic Publishers.

[11] Sousa, P., Ramos, C., and Neves, J., 2004, "The Fabricare System". *Production Planning & Control*, 15(2), pp.156-165. Taylor & Francis.

[12] Sousa, P., Silva, N., Heikkila, T., Kallingbaum, M., and Valcknears, P., 2000, Aspects of Co-operation in Distributed Manufacturing Systems, *Studies in Informatics and Control Journal*, 9(2), pp.89-110.

[13] Van brussel, H.; wyns, J.; valckenaers, P.; bongaerts, L.; and peeters, P., 1998, "Reference Architecture for Holonic Manufacturing Systems: PROSA". *Computers in Industry*, vol. 31(3), pp.255-276. 1998. Elsevier Science B.V.