CONSIDERATION REGARDING A WORKCELL AND RESOURCE MODEL IMPLEMENTATION IN FMS

Ilie Octavian POPP
"*Lucian Blaga*" University of Sibiu, Faculty of Engineering "*Hermann Oberth*"
Departaments of Machine and Machinery, Emil Cioran St. no. 4, Sibiu-2400
E-mail: **ilie.popp@ulbsibiu.ro**, Tel. 0269-216062/454

Key words: FMS, workcell, reference architecture model.

Abstract: In this paper, we briefly explain how to configure the different resource modules that actually control the tasks of the physical device and put together a work cell consisting of other FMS resources. Finally, an example on building a FMS using a hierarchical approach is presented.

1. INTRODUCTION

An implementation based on the architecture proposed in the earlier section has been made. Although different cell configurations seem to have different operational rules at a lower level, it is possible to view them uniformly at a higher level of abstraction. An object-oriented approach is used to model the various resources of the FMS. Using multiple inheritance and function overloading, a uniform operational logic can be adopted for different cell configurations. Each resource would, however, also support a device specific interface that might be applicable to other application objects in the system but play no role in the operation of the work cell. The program consists of object models for the transportation units, the storage units, the processor unit and the I/O ports. It also consists of a control module that is based on the supervisory control theory developed in [3], [4].

From the implementation point of view, we require communication software that would handle message passing between the different resources of a work cell. Using the principles of distributed object technology, we could implement each resource in the workcell as a distributed object. A resource, modeled as a distributed object, has a well-defined interface, describing the data and the methods that it supports. A distributed object can execute either on the same computer or another networked computer. CORBA is an accepted middle-ware standard for building distributed applications. It provides a robust, heterogeneous, inter-operable, multi-platform environment and hence we propose to use a CORBA based architecture for the implementation of out architecture.

Common Object Request Broker Architecture (CORBA) CORBA is an industry middle-ware standard for building distributed, heterogeneous, object-oriented applications. It details the interfaces and characteristics of the object request broker (ORB) of the Object Management Architecture (OMA). The ORB is mainly responsible for facilitating communication between clients and objects. The ORB delivers requests to objects and returns any responses to the clients making the requests. The client does not know where the target object resides, how the target object is implemented, whether the object is currently activated, and the communications mechanisms used. It allows for client-server and peer-to-peer communication as well. Any distributed object, under the CORBA standard, has to support a list of methods called an interface. The interface has to be defined in OMG Interface Definition Language (OMG IDL) and these definitions would be

mapped to a higher level programming language like C++ or Java. This enables us to build clients and servers using different programming languages.

OMG Interface Definition Language (OMG IDL)

An object's interface specifies the operations and types that the object supports and thus defines the requests that can be made on the object. Interfaces for objects are defined in the OMG Interface Definition Language (OMG IDL). Interfaces are similar to classes in C++ and interfaces in Java. An example interface definition is given below:

```
OMG IDL
interface machine
{
boolean isMachineFree();
boolean Start (in o_token identifier();
boolean JobFinished (in string identifier();
boolean Stop();
};
```

This definition specifies an interface named Machine that supports a set of operations.

The Start operation takes an object reference of type untaken. Given an object reference of type Machine, a client could invoke (say) Start operation on it. This interface might be supported by one of the machine objects mentioned above. An important feature of OMG IDL is its language independence. It forces interfaces to be defined separately from object implementations. This allows objects to be constructed using different programming languages and yet communicate with one another. OMG IDL provides a set of types that are similar to those found in most high level programming languages. It provides basic types such as long, double, Boolean, constructed types such as structure union. Types are used to specify the parameter types and return types for operations. The types might also be user-defined in IDL. To define exceptional conditions that may arise during the course of an operation, OMG IDL provides exception definitions. The OMG IDL module construct allows for scoping of definition names to prevent name clashes. An important feature of OMG IDL interfaces is that they can inherit from one or more other interfaces. This makes it possible to reuse existing interfaces when defining new services.

2. PROGRAM STRUCTURE

We employ a CORBA based framework for the implementation of the single processor workcell. Resource is an abstract base class from which all the other objects have been derived. An object relationship diagram clarifies the relationship between the different objects used in the architecture. It makes it clear, for e.g., that all the resources contain a port in their class definition. Each resource module, in essence, behaves like an object server and responds to queries from other application objects. The focus, when developing the resource modules, has been on identifying a set of interface functions that it should support. Recall that the FMS resources have been classified under one of the four categories listed below:

1. Transportation unit
2. Storage Unit
3. I/O Port
4. Processor Unit

The interface of all the different resource modules is defined using OMG IDL. The work cell class is also derived from the resource class and it presents a similar control interface for upward compatibility. The principal difference is that the work cell contains a

supervisory control module. Figures 1 and 2 shows the class hierarchy and the object relationship diagrams for the workcell architecture.

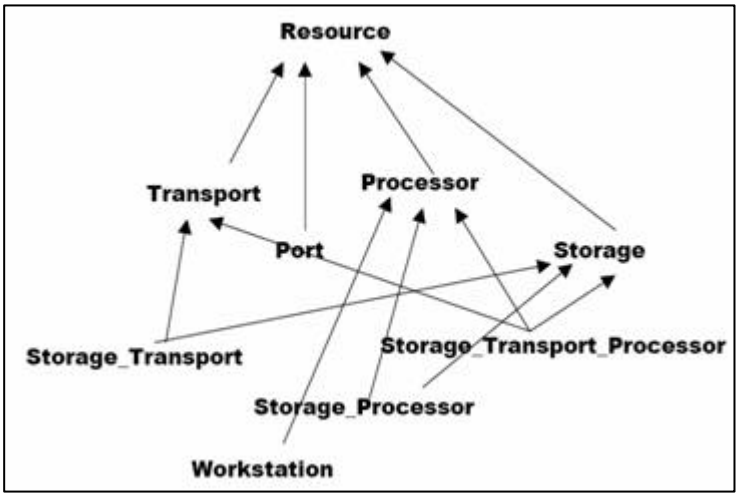

Figure 1: Class hierarchy diagram for the workcell/station architecture

The control logic used by the module to ensure deadlock free behavior is based on the supervisory control theory developed in [3] and [4]. This makes it easier to build a manufacturing system using a hierarchical approach.

3. RESOURCES

The software modules that are used to control the physical devices in an FMS are called the resource modules or the virtual manufacturing devices (VMD). These are divided into the generic and the equipment specific part, also called the device driver.
The controller module knows the status of every resource under it so that correctness of operation is ensured. The work cell module presents an interface similar to the resource module for upward compatibility but in addition it has a dispatcher that schedules the flow of events in it and a controller module that supervises the set of allowable events.

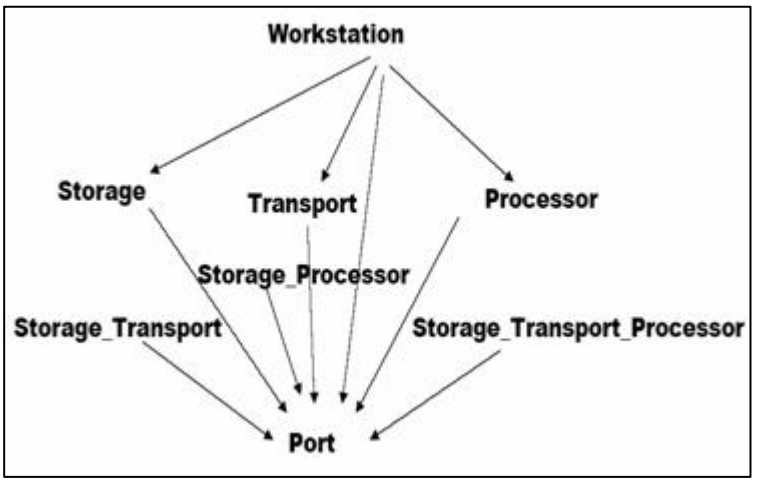

Figure 2: Object Relationship Diagram for the workcell architecture

Message Passing: The workcell module has a standardized sequence for sending a handshake message. The handshake message between any two resources is handled through the controller. The sequence of messages that are exchanged during the

unloading of a job from a processor (Pri), at port P1, by a transportation device (Tri) is listed below. The transportation device loads the job into port P1 as well

- Pri->prepareRemoveJob(tid,P1);
- Tri->prepareAcceptJob(tid,P1);
- Pri->send_Message(message1);
- Tri->send_Message(message1);
- Pri->removeJob(tid,P1);
- Pri->sendMessage(message2);
- Tri->acceptJob(Jobid,P1);
- Tri->sendMessage(tid,P1);

This sequence of handshake messages completes the unloading of a job from a processor Pri to a transportation unit Tri. Note that the entire message passing scheme is asynchronous and so we have a provision for the resources to signal the controller when it has completed a specified operation. The equipment specific part or the device driver of a resource contains the code for controlling the physical unit. The different codes to be run are read in by the device driver from a configuration file. The resource modules for the different class of devices (See Section on Resources in Section 3) have some fundamental differences. For e.g., the generic resource module for a storage unit has an internal data structure to keep track of the free buffer locations and the length of time occupied by the jobs in each buffer space. When a new job is loaded into the storage unit, the module assigns a free buffer location to the incoming job, makes the appropriate call to the device driver of the storage unit and updates it own data structure. A module for a processor needs to support a function to run a NC program. It sends back a message to the controller on completing the processing of a job. This stage is absent in other resource modules. Thus, the dispatcher in the work cell makes sure that this stage is called on all processors but is bypassed on other resources.

Further, the controller also ensures that conflicting operations on a resource are never issued simultaneously. Let us consider a case where a resource is prepared to accept a job at one of its input ports (say) P1, and then the resource cannot prepare to accept another job even if it has additional free capacity until the first job has been accepted by the resource. Thus, atomicity of a certain set of operations is ensured at the resource level.

Resource Configuration:

As per our architecture:

- Every resource in the workcell consists of a port(s).
- Jobs enter/exit a resource only through ports.
- Every port in the device is defined as interfacing with another device in the workcell.

Thus, these ports are the material transfer points. Since every physical piece of equipment in the system need not provide this functionality, the device driver (or the specific part of the VMD) holds the information about these ports and provides the cell controller with this functionality. The connectivity information is defined in the configuration file of the work cell controller. Whenever a job route is defined and uploaded to the system, it is verified to make sure that it is compatible with the connectivity information.

The device driver of each resource is configured so that it would know the actual programs (NC programs) it needs to execute on the physical device on receiving an instruction from the cell controller. The device driver would have a mapping function that would translate a call (say) prepare_to_acceptJob at port P1 into a corresponding NC program. These translation maps would be read in from corresponding configuration files of the resource modules.

A sample configuration is given in Figure 3. As shown in the figure, every device in the system has a set of ports associated with it. The work cell has a list of input ports and output ports associated with each device. In the above example, P1 is both an input port and output port for MT1. Similarly, P1 is an output port and P3 is an input port for PC1. P_2 is both an input and output port for PC1.

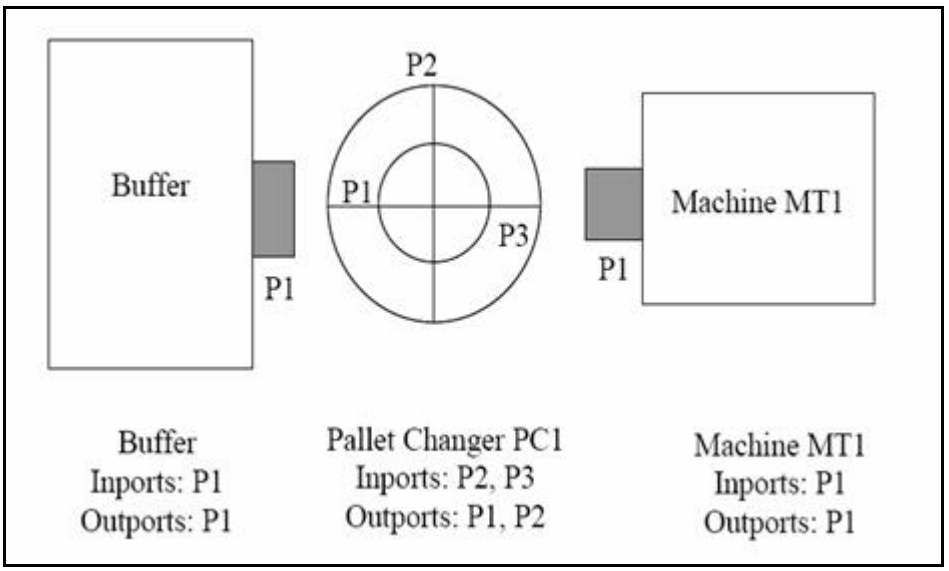

Figure 3: Resource configuration in a workcell with two devices

The various programs that can be supported by the resource are specified in the configuration file. It is also possible to dynamically add programs to the resource through the control interface of the module.

4. WORKCELL CONFIGURATION

The workcell can be setup to support different manufacturing environments. A workcell can be configured to have multiple buffers, separate I/O ports, combined storage/transportation and so on. The ports associated with each device and the connectivity information is defined in the workcell configuration file. The job routes supported at the workcell level are also specified in the configuration file. The operational logic of the workcell controller in such architecture is relatively independent of its configuration. The appropriate procedure calls (handshake messages) are made on the devices sequentially as given in the job definition. The cell controller waits on an acknowledgement from the resource module before making the next call as explained in [3], [4].

On accepting a job, each device runs an appropriate program. These programs are device dependent. The workcell controller has no control over this. This way the operational logic is separated from the processing logic. When the job in a device is ready to be sent out (processing complete), an appropriate signal is sent to the cell controller. The cell controller schedules the next event after ensuring that this event results in a deadlock free operation. Two different scenarios of a workcell configuration are presented below. The steps followed by the controller during the execution of a job are listed here.

5. CONCLUSIONS

The use of automatic synthesis of supervisory controllers allows a high degree of flexibility in the system. Whenever there has been a significant change in the system configuration (when new job routes have been defined or when resources have been added/removed), the control-laws are recalculated and re-synthesized. We have suggested hierarchical synthesis as a strategy for rapidly configuring large systems. In this paper, a methodology for formally modeling hierarchical resource allocation systems is developed. A distributed hierarchical control policy for ensuring deadlock free behavior in such a system has been proposed. In paper, we apply this methodology to model a FMS setup under the framework of our architecture.

Furthermore, the use of distributed object technology to implement the system enables us to run each resource module as a distributed object/server on a computer node. It is possible to access the control panels associated with each resource from a separate computer and this allows the operator to access the system at different control levels (the resource or the workcell).

The software module we have implemented based on this architecture is highly configurable to suit the needs of a variety of manufacturing environments. A CORBA based framework has been used to develop the various object modules. This gives us the added benefit of being able to run the application across multi-platforms (operating systems).

References:

[1] Adlemo A., and S.-A., *Models for Specification and Control of Flexible Manufacturing Systems*, Technical Report, School of Electrical and Computer Engineering, Chalmers University of Technology, Goteborg, Sweden, 1997.

[2] Lawley M., *Structural Analysis and Control of Flexible Manufacturing System*, PhD Thesis, University of Illinois at Urbana-Champaign, 1995.

[3] Popp, I., *Consideration regarding model Architecture for Scalable Flexibility in Manufacturing*, International Conference on Manufacturing System, Buletinul Inst. Politehnic din Iasi, publicat de Universitatea Tehnica “Gh. Asachi”, Tomul LI, sectia Constructii de masini, Iasi, 2005.

[4] Popp, I., *Consideration regarding a Resource Models and Job Models for Scalable Flexibility in Manufacturing*, Acta Universitatis Cibiniensis, Buletin stiintific al Univ. “Lucian Blaga” din Sibiu, seria Tehnica, vol. LII, ISSN 1583-7149, p. 77-80, Sibiu, 2005.

[5] Reveliotis S., *Structural Analysis and Control of Flexible Manufacturing Systems with a Performance Perspective*, PhD Thesis, Univ. of Illinois at Urbana-Champaign, 1996.

[6] Wyns, J., Brussel, H., Valckenaers, L., *WorkStation Architecture in Holonic Manufacturing Systems*, Cirp Journal on Manufacturing Systems, Vol. 26, 220-231, (1996).