Message-Passing Architecture and its Construction in Object-Oriented Rapid Modeling for Automated Manufacturing System Simulation

Moon Ho Hwang and Byoung Kyu Choi
Dept. of Industrial Engineering
Korea Advanced Institute of Science and Technology
373-1, Kusung-Dong, Yusung-Gu, Taejon, 305-701, South Korea

Correspondence: Moon Ho Hwang
IE Dept., KAIST,
373-1, Kusung-dong Yusung-gu, Taejon 305-701, South Korea
FAX: +82-42-861-9195
Tel: +82-42-869-5155
E-mail: s_hmh@cais.kaist.ac.kr
Abstract
To develop a simulator for automated manufacturing systems using an object-oriented modeling paradigm, we have researched the message-passing architecture and methodology of rapid model construction. For the message-passing methodology, we propose a modified hierarchical control architecture, modified to allow signal paths between adjacent job flow-connected resources. Model libraries using the proposed architecture have been described with DEVS formalism and have been implemented based on DEVSim++ so that they support the hierarchical and modular concept. In order to support the rapid modeling concept, we have designed a prototype simulator in which the entire message-passing architecture is constructed from limited and partial information in order to reduce modeling overload for users. By showing a sample modeling procedure of a system, we demonstrate the interesting features of the developed simulator.
Key Words: object-oriented modeling paradigm, message-passing architecture, modified hierarchical control architecture, rapid model construction, DEVS
1. Introduction

The object-oriented modeling paradigm has been applied to manufacturing system simulations based on discrete event modeling [1-4]. One of the principles supported in the paradigm is *encapsulation*. Encapsulation consists of the separation of the external aspects of the object. These external aspects are accessible to other objects from the internal implementation details of the object, which are hidden from other objects [5]. Encapsulated objects have *methods* which can perform operations on the object states and invoke each other in a manner described as *message passing*. The message-passing paradigm differs from subroutine invocation, which is a command sent by an omniscient master to a completely subservient slave; a message is a polite request by one peer to another to perform an action that the latter may or may not choose or be able to do [6].

Depending on the simulation objective, various objects and their message-passing methods have been proposed. Judd et al. [7] introduced a modeling view in which objects in the real control architecture can be mapped onto objects in the simulation model. Narayanan et al. [8] developed the controller simulation model so that the control logic can be easily adapted to the real system. In order to increase the reusability of simulation models, Pratt et al. [9] separated control logic, plant behavior, parts and information. However, if the main simulation objective is performance evaluation (which is one of the traditional objectives of simulation studies), “entire” modeling of a message-passing architecture can be considered overcomplicated by the user. Moreover, in “rapid modeling” for simulation, it is not necessary for the message-passing architecture to be the same as that of the physical target system. Therefore, it is a recommended that the user inputs the minimum information while the simulator generates the internal message-passing architecture.

The main objective of this paper is the development of a simulator which supports rapid modeling of automated manufacturing systems. To achieve the main objective, this paper will treat two of the main topics considered by automated manufacturing system simulator developers. One of these topics will be the proposal of an message-passing architecture of rapid modeling for performance simulation. The other will be the introduction of a generation method for its architecture, in which input requirements of users are simpler than the entire message-passing architecture so that it can support rapid modeling. In addition, to develop object-oriented simulation libraries, we use the discrete event system specification (DEVS) formalism, which supports specification of discrete-event models in a hierarchical, modular manner [10], utilizing DEVSim++, which realizes DEVS formalism for modeling and associated abstract simulator concepts for simulation, all in C++ [11][12].

The rest of the paper is organized as follows. In section 2, the motivation for this research is described briefly, some typical control architectures are surveyed, and a
modified hierarchical control architecture is proposed for message-passing architecture in the developed simulator. Section 3 describes the model libraries in the order of class hierarchies, the structure and dynamics of the proposed message-passing architecture, and a method for the generation of the entire message-passing architecture from user-input partial information. An application example utilizing the proposed prototype simulator is included in section 4. Section 5 gives concluding remarks.

In addition, the acronyms used in modeling of manufacturing systems are summarized in Table 1. The detailed dynamics of two of developed model libraries are explained in the Appendix.

<table>
<thead>
<tr>
<th>No</th>
<th>Acronym</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>AGVS</td>
<td>Automated Guided Vehicle System</td>
</tr>
<tr>
<td>2</td>
<td>CLP</td>
<td>Control Load Point</td>
</tr>
<tr>
<td>3</td>
<td>FMS</td>
<td>Flexible Manufacturing System</td>
</tr>
<tr>
<td>4</td>
<td>IB</td>
<td>Input Buffer</td>
</tr>
<tr>
<td>5</td>
<td>LT</td>
<td>Loading Table</td>
</tr>
<tr>
<td>6</td>
<td>MT</td>
<td>Machining Table</td>
</tr>
<tr>
<td>7</td>
<td>MC</td>
<td>Machining Center</td>
</tr>
<tr>
<td>8</td>
<td>MCS</td>
<td>Machining Center Station</td>
</tr>
<tr>
<td>9</td>
<td>MHC</td>
<td>Modified Hierarchical Control</td>
</tr>
<tr>
<td>10</td>
<td>LUS</td>
<td>Loading Unloading Station</td>
</tr>
<tr>
<td>11</td>
<td>OB</td>
<td>Output Buffer</td>
</tr>
<tr>
<td>12</td>
<td>UT</td>
<td>Unloading Table</td>
</tr>
</tbody>
</table>

2. Overview of Message-Passing Architecture

2.1 Needs of Message-Passing Architecture

Consider a flexible manufacturing system (FMS) as shown in Figure 1(a). The system consists of two loading tables (LT1 and LT2), two unloading tables (UT1 and UT2), an automated guided vehicle system (AGVS), and two machining centers (MC1 and MC2), both consisting of: an input buffer (IB); a machining table (MT); and an output buffer (OB). With respect to the flow lines of jobs (which means volumed moving objects such as parts, pallets, or vehicles), the FMS behaves as follows:

A part entered in the system is loaded on either LT1 or LT2. Via AGVS, where two vehicles circulate without conflict, the part is transported to one of the two machining centers’ input buffers. After machining, it is unloaded from the output buffers and moved
to either UT1 or UT2 by the AGVS. Finally, the part disappears from the system.

![Figure 1. A FMS Layout](image)

From the viewpoint of systems operation, it is not difficult to see that decision-making requires the involvement of more than one object. In an event-driven system, the situations usually occur when the states of senders and receivers change. We use the term "push" if the situation is triggered when a job arrives at a resource (sender), and we say "pull" if it is triggered when a receiver’s state changes because of a job’s departure.

Simulators on the market traditionally support the modeling methodology of the push-pull situation. In SIMAN [13], a “SCAN” block that checks conditions and a SIGNAL block which can activate the “WAIT” block are designed. The AutoMod simulator [14], which is based on a push system in default mode, recommends modeling the pull situation using an “ORDER LIST” in which jobs wait for other conditions to be fulfilled. These simulators, however, don’t use any message passing among objects but “global variables” which can be directly accessed in any simulation flow. In these paradigms, models (or, to be precise, functions or blocks) can become so interdependent that a small change has a massive ripple effect [5]. MODSIM II [15], one of the object-oriented simulation languages available on the market, recommends using a statement “WAIT FOR TriggerObj FIRE,” in which the process will wait until the special object’s “Trigger” method is invoked by some other method. As the message-passing methodology, MODSIM II recommends using the statement “TELL an_object TO a_method IN duration,” which means “execute the method a_method of an_object after time duration has elapsed.” In the simulator,
however, since all parts of message-passing architecture should be designed by the user, the construction of its models can be somewhat overloaded.

### 2.2 Typical Message-Passing Architecture

To employ the internal message-passing architecture in a rapid modeling for simulation, we have surveyed some of the real architectures that have been used in most physical manufacturing systems. For the object-oriented modeling paradigm that mimics real target systems, surveying control architecture in physical systems gives plenty of design insight into the internal architecture of simulation models.

The control architectures have expanded and each has varying potential with respect to the modifiability/extensibility and reconfigurability/adaptability of the control system. Figure 2 illustrates three typical control architectures (a circle denotes a controller and a rectangle a plant component). The earliest control architecture employed a centralized approach as shown in Figure 2 (a). Even though the centralized architecture is one of the simplest, to overcome its overhead management of the whole control, many researchers have tried to develop it in the direction of a more distributed architecture [16]. To support distributed decision-making, there are two main directions: one is vertically distributed, and the other horizontally. Hierarchical control architectures like Figure 2 (b), in which systems can be implemented in a hierarchical and gradual manner [16][17][18], have been developed. However, as the number of hierarchical layers increases, signal traffic between them becomes overloaded. As shown in Figure 2 (c), the heterarchical architecture, which is generally based on field bus topology, is fully distributed horizontally, with no master/slave relationships. However, this heterarchical one has a high likelihood of only local optimization, and requires a significant amount of inter-process communication [16].

![Figure 2. Typical Control Structures](image-url)
2.3 A Modified Hierarchical Control Architecture

To design the internal message-passing architecture, two main criteria that we have considered are:

(1) Correspondence with the hierarchical modeling view;
(2) The efficiency of message passing.

As Sargent et al. have addressed [19], the hierarchical modeling capability has desirable features such as modeling ease (e.g. reducing the time and effort required to develop models), allowing for model reuse (a topic of long-standing interest), reducing the number of specific models required, allowing for the use of a database of models, and aiding in model validation. To satisfy the first criterion, we have chosen the hierarchical control architecture. However, it is not difficult to see that adjacent objects connected by job flow should transmit their state to each other, which could lead to traffic overload in proper hierarchical control architecture. To ensure the second criterion, we have modified the proper hierarchical control architecture so that the signal flows between components connected by a job flow are allowed. Figure 3 illustrates the structure of the proposed modified hierarchical control (MHC) architecture.

![Figure 3. Basic Concept of MHC Structure](image)

In terms of components, the functional units of the MHC architecture are *plant* and *controller*. A plant acts as a *sensor*, which handles such events as job arrival, job departure and end-of-processing. In addition the plant acts as an *actuator*, which executes commands received from outside. Figure 4 (a) illustrates an example of a plant relationship where CLPi and CLPj are plant components of ResA, which is a plant component of ResB, which is a plant component of ResC, recursively. A controller performs the role of a *signal distributor* that distributes the signals received from other models to their destinations, as well as that of a *command generator (commander)* which monitors the plant states and generates commands when certain conditions are satisfied. In this architecture, *resource* is defined as the model consisting of one controller and more than one plant. Therefore, ResA, ResB and RecC in Figure 4(a) are resources but CLPi and CLPj are not.
Figure 4. Components of MHC Architecture
In the hierarchical structure, there are pairs of *vertical couplings* which can be broken down into an upward and a downward coupling. An *upward* coupling from a plant component to a controller is used for sensed signal flows; conversely, a *downward* coupling from a controller to a plant component is used for command signal flows as shown in Figure 4 (a). For each of the *job flow couplings*, there is a *horizontal coupling* consisting of a pair of *forward* and *backward* channels. Forward and backward couplings are used for the sending of "ready-to-send" and "ready-to-receive" signals respectively. Figure 4 (b) shows a horizontal structure consisting of job flow couplings (solid arrows) and signal flow couplings (dashed arrows).

3. Development of Rapid Modeling System

3.1 Class Hierarchies

From the viewpoint of the user-modeling view as well as the internal message-passing structure, it is not difficult to know that the hierarchical and modular modeling features are required in our rapid modeling system. Since the DEVS formalism [10] provide inherently the desired modelling features, the DEVSim++ environment that is a DEVS implementation in C++ language [11][12], have been utilized to develop the proposed model library.

![Derived Classes for the AMS Simulator](image)

Figure 5. Developed Class Hierarchies

From the DEVSim++ classes, we have derived the domain-specific classes shown in Figure 5. From the Coupled Model Class, which contains the structure information, the Resource Class is derived in which there must be one controller and more than one plant component. We subdivide the Resource Class into the Class Conveyor, the AGVS (Automated Guided Vehicle System) Class and the SRMS (Storage and Retrieval Machine
System) Class. Besides these predefined resource classes, the user can make customized models using the Resource Class by incorporating plant components into it. From the Atomic Model Class, the CLP Class for the control load point is derived so that it works as a sensor and as an actuator. From the CLP Class, there are derived classes: the MT (performing a machining process), the Palletizer (performing a palletizing process) and Depalletizer classes (performing a depalletizing process). On the other hand, the Controller Class is designed to work as a signal distributor and/or a command generator.

3.2 Structure of the MHC Architecture

In this section, we discuss which interfaces are designed for each model, how models influence each other through couplings, and how to construct the MHC structure from user input models.

3.2.1 Interfaces

Figure 6 shows the interfaces of a controller and a plant. In Figure 6, the input ports are denoted by hollow triangles and output ports by shaded triangles. Ports of job flows are indicated by solid lines, and ports of signal flows by broken lines.

![Interface of Controller](image)

(a) Interface of Controller

![Interface of Plant Components](image)

(b) Interface of Plant Components

Figure 6. Interfaces of Controller and Plant

In terms of interfaces, the Controller Class is designed to have at least one input/output port to communicate with sub-plant components, and one input/output port to communicate
with a parent controller. In Figure 6 (a) for example, a Controller has FromChild_1,…, FromChild_n input ports for upward signals and ToChild_1,…,ToChild_n output ports for downward signals. On the other hand, a Controller has one FromParent input port and one ToParent output port.

All classes that can be plant components such as CLP Class (and its derived classes) and Resource Class (and its derived classes), have input and/or output ports for upward/downward signals, jobs, and forward/backward signals. In Figure 6 (b) for example, a plant component has one ToParent output port for upward signals, and one FromParent input port for downward signals. The plant has RECEIVE_1,…,RECEIVE_n input ports for jobs while has SEND_1,…,SEND_n output ports for jobs. For each RECEIVE_i input port, there must be a ToSender_i output port and FromSender_i input port. Similarly, for each SEND_i output port, there must be a ToReceiver_i output port and a FromSender_i input port.

3.2.2 Couplings

The couplings are constructed in all resource classes derived from the Coupled Model Class. As mentioned before, all resources have vertical couplings and horizontal couplings. Along the vertical structure, a controller working as a signal distributor and/or command generator exist in the central point of a resource. A resource, therefore, has upward/downward signal couplings between itself and its controller as well as its controller and all of its plant components. From the viewpoint of horizontal structure, each job flow coupling has forward/backward signal couplings by connection of related interfaces. Figure 7 shows couplings of the machining center introduced in Figure 1.

3.3 Dynamics of the MHC Architecture

In this section, we discuss the behavior of the Controller Class and the CLP Class, which are derived from the Atomic Model of DEVS representing the generic behavior of all their subclasses. In this paper, we are describing their dynamics from the viewpoint of message description (port, flow-entity) which means flow-entity is inputted or outputted through a port. Based on DEVS formalism, the dynamics of Controller and CLP are explained in more detail in the Appendix.

3.3.1 Flow-Entities

First of all, the flow-entity is categorized into JOB and SIGNAL as follows:

\[ \text{JOB}=\langle \text{name, type}_j \rangle \text{ where name: instance name; } type_j \in \{\text{part, pallet, vehicle}\}; \]

\[ \text{SIGNAL}=\langle \text{source, dest, job, type}_s \rangle \text{ where source: source name; dest: destination name; job: job information; and } type_s \in ust \cup dst \cup fst \cup bst : \text{the signal types as} \]
illustrated in Table 2.

Table 2. Nomenclature for Signal Types

<table>
<thead>
<tr>
<th>No</th>
<th>Notation</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>UST</td>
<td>Upward signals types = {occupied(O), empty(E), end-of-processing(EOP), end-of-assembly(EOA), end-of-disassembly(EOD)}</td>
</tr>
<tr>
<td>3</td>
<td>DST</td>
<td>Downward signals types = {transfer(T), process(P), assembly(A), disassembly(D)}</td>
</tr>
<tr>
<td>4</td>
<td>FST</td>
<td>Forward signals types = {ready-to-send(RTS)}</td>
</tr>
<tr>
<td>5</td>
<td>BST</td>
<td>Backward signals types = {ready-to-receive(RTR)}</td>
</tr>
</tbody>
</table>

For example, a signal <IB, CTRL_MCT, <part1, partA>, occupied> means that a message is detected by the IB and sent to the controller of MC, provided IB is occupied by a job having the name “part1” name and type “partA.” In this paper, if we don’t specify a field value, we use an “*” instead of a specific value such that <*, *, <*, *>, occupied> means the “occupied” typed signal.
3.3.2 Controller

As mentioned before, all controllers’ functionality might be categorized into a signal distributor (please see Figure 8 (a)) and a command generator (please see Figure 8 (b)). As shown in the left side of Figure 8 (a), if a controller receives an upward sensed signal that its destination is not aimed at the controller, the controller sends the signal to a parent. As shown in the right side of Figure 8 (a), if a controller receives a downward command signal, the controller distributes the command proper sub-plant components. These two cases (left and right side of Figure 8 (a)) are not related to the Control Model input by the user while the remaining two cases (Figure 8 (b)) have very much to do with Control Model. If a controller receives an upward sensed signal such that its destination is aimed at the controller, the controller updates its Control Model. And then if there are satisfactory conditions to control plant components, commands are generated and sent to the related sub-plant components as shown on the left side of Figure 8 (b). Otherwise no commands are generated and only the Control Model is updated as shown on the right side of Figure 8 (b). In our simulator, the Control Model that is one of the user-input models, is further explained in section 3.5

![Figure 8. Dynamics of Controller](image_url)

(a) CTRL\textsubscript{RESi} is not destination of a signal

(b) CTRL\textsubscript{RESi} is destination of a signal
3.3.3 Control Load Point (CLP)

Figure 9 illustrates the dynamics of the CLP Class. Initially, if there are some controllers that are monitoring the CLP’s “empty” state, the CLP sends the empty typed signal to the controllers. Figure 9 (a) shows that the CLP sends the empty typed signal to a monitoring controller whose name is RESj. When the CLP receives a job, if there are some controllers that are monitoring the CLP’s “occupied” state, then the CLP sends the occupied typed signal to the controllers as shown in Figure 9 (b). If the CLP has a job to be processed and the CLP receives a command of “process,” after the processing time, the CLP informs a waiting controller of the state by transmission of the “end-of-processing” typed signal if it exists as shown in Figure 9 (c). If the CLP has a job and the CLP receives a command to transfer a job to a connected plant component, the CLP sends a “ready-to-send” typed signal to the receiver as shown in Figure (d). If the CLP has a job to be transferred and receives a “ready-to-receive” typed signal from the receiver, it transfers the job to the receiver as shown in Figure 9 (e). If the CLP is empty and receives a “ready-to-send” signal from a sender component, the CLP will reply with “ready-to-receives” to the sender.

Notice that in the initialization step before the simulation is run, the state marking information relative to which state should send which controller(s) their state information, is generated at the control load point, based on the user input dynamic model. The generation method is further explained in the next section.

3.4 User Inputs and Generation of MHC Architecture for Rapid Modeling

In order to develop a rapid modeling system for simulation, we have proposed the MHC architecture, and have designed an algorithm to generate the architecture from partial and limited information taken from them.

Remember that the characteristics of the proposed architecture are that every resource has the following properties:

(1) One controller and vertical signal couplings are determined by the hierarchical structure of the plant components; and forward and backward signal couplings must exist for each job flow;

(2) The objective of the control model in each controller is to control plant behavior as intended.

Therefore, in our proposed scheme, the input information is the plant model structure and the plant model behavior.
3.4.1 Construction of MHC Structure

The user inputs are a partial Resource which includes only “plant structure,” that is, plant components and job-flow couplings, from which a controller and signal-flow couplings are excepted. Therefore, to construct the total MHC structure, a “controller” and “signal-flow couplings” should be added. Following are the procedures to construct the whole MHC structure from the input model.

Construction Procedures of MHC Structure
Step 1: Construction of Vertical Structure

1.1 Add input/output ports for vertical signal flows to the Resource;
1.2 Add a controller to the Resource;
1.3 For each of the plant components, construct its vertical structure, construct the vertical
signal-flow couplings between the added controller and the Resource, and between the
added controller and plant components; --- (I)

Step 2: Construction of Horizontal Structure
2.1 Add input/output ports for horizontal signal flows to the Resource;
2.2 For each of the plant components, construct its horizontal structure; --- (II)
2.3 For each of the job-flow couplings, construct additional forward/backward signal-flow
couplings between the models connected by the job flows;

Note that, depending on the type of the plant component, step 1.3 has different meanings.
If a plant component is for instance of Resource Class (or an instance of its derived classes),
statement (I) means step 1 recursively, otherwise, if it is an instance of CLP Class (or an
instance of its derived classes), the statement denotes step 1.1 only. Similarly, at step 2.2,
if a plant component is an instance of Resource Class, statement (II) means the whole of
step 2, recursively, or else if a plant component is an instance of CLP Class, only step 2.1 is
performed.

Figure 10 shows the MHC structure construction procedures for the MC example
introduced in section 2. If the user input is as shown in Figure 10 (a), the vertical structure,
including the addition of the MC controller and vertical couplings, is added as shown in
Figure 10 (b). Then, using job flow relations, the horizontal control structure is
constructed as in Figure 10 (c).

3.4.2 Construction of MHC Dynamic Model
To supply users with a simple modeling view, we define the dynamic user-input model
using “behavior of plant components” such that

\[
DYNAMIC_{USER} = \{IF < CONDITION > THEN < ACTION > \\
[ELSEIF < CONDITION > THEN < ACTION >] \\
[ELSE < ACTION >] \\
END; \}
\]

where \( < CONDITION > \) constitues the parts
describing conditions of sub-plant models in which \( clp_c \) is the name of a control load point
and \( state_c \) is one of UST (see Table 2); \( < ACTION > \) are the parts describing action of sub-plant models in which \( clp_s \) is
the source name of a control load point, \( state_a \) is one of DST (see Table 2), and \( clp_d \) is
the destination name if \( state_a = \text{“transfer.”} \) All of the parts enclosed by the square
brackets “[ ]” are optional; all of the parts enclosed by the brace brackets “{ }” can be used repeatedly.

If one of the behavior requirements of the MC is “If a job arrives on the input buffer IB and the machining table MT has no jobs, the job on the IB is sent to the MT,” then the requirement might be modeled as (R-1):

\[ \text{IF (IB, occupied) AND (MT, empty) THEN (IB, transfer, MT) END; } \text{---- (R-1)} \]

![Diagram of MHC structure](image)

**Construction Procedures of MHC Dynamics**

1. Set \( DYNAMIC_{USER} \) to the Control Model of the RES’ controller;
2. For each \( clp \) described in \( CONDITION_{PLANT} \), mark the sensing requirement on the \( clp \) so that when \( clp \)’s state becomes \( state_c \), the \( clp \) will send its state to the RES’ controller.

In modeling of MHC dynamic, if the user inputs the (R-1) requirement into the MC layer, the following construction procedures are performed:
the (R-1) requirement is set to control model of the CTRL\textsubscript{MC} controller;
- the input buffer IB is marked to send the “occupied” signal to the CTRL\textsubscript{MC};
- the machining table MT is marked to send “empty” signals to the CTRL\textsubscript{MC}.

Figure 11 shows the message-passing sequences to perform the requirement (R-1). During a simulation run, the IB and the MT send the “occupied” signal and the “empty” signal to the CTRL\textsubscript{MC} controller, respectively. If the condition of (R-1) is hold, the CTRL\textsubscript{MC} send the “transfer” command to the IB. After sending and receiving of “ready-to-send” and “ready-to-receive” signals, the IB transfers a job to the MC.

4. Examples

In this chapter, the modeling procedure of a FMS in gradual and modular manner is illustrated. In addition, some modeling issues related modeling of dynamics are addressed.

Firstly, to construct the modular and reusable model for the machining center, the Resource Class is used. The developed machining center consists of an input buffer (IB), a machining table (MT) and an output buffer (OB). The job flows are tandem-connected as shown in Figure 12 (a). Also, the job-flow requirements of the machining center are such that: (1) no blocking and starvation between the plant components are connected; and (2)
MT processes the job. Figure 13 (a) shows the user-input statements defined at MC, which make MC behave as the behavioral requirements.

IF (IB, occupied) AND (MT, empty) THEN (IB, transfer, MT) END;
IF (MT, occupied) THEN (MT, process) END;
IF (MT, end-of-processing) AND (OB, empty) THEN (MT, transfer, OB) END;

IF (LUS.LT1, occupied) AND (MCS.MC1.IB, empty) THEN (LUS.LT1, transfer, MCS.MC1.IB)
ELSEIF (LUS.LT1, occupied) AND (MCS.MC2.IB, empty) THEN (LUS.LT1, transfer, MCS.MC2.IB)
ELSEIF (LUS.LT2, occupied) AND (MCS.MC1.IB, empty) THEN (LUS.LT2, transfer, MCS.MC1.IB)
ELSEIF (LUS.LT2, occupied) AND (MCS.MC2.IB, empty) THEN (LUS.LT2, transfer, MCS.MC2.IB)
END;
IF (MCS.MC1.IB, occupied) AND (LUS.UT1, empty) THEN (MCS.MC1.IB, transfer, LUS.UT1)
ELSEIF (MCS.MC1.IB, occupied) AND (LUS.UT2, empty) THEN (MCS.MC1.IB, transfer, LUS.UT2)
ELSEIF (MCS.MC2.IB, occupied) AND (LUS.UT1, empty) THEN (MCS.MC2.IB, transfer, LUS.UT1)
ELSEIF (MCS.MC2.IB, occupied) AND (LUS.UT2, empty) THEN (MCS.MC2.IB, transfer, LUS.UT2)
END;

IF (IB, occupied) AND (MT, empty) THEN (IB, transfer, MT) END;
Secondly, using the MC model base, we try to model the simple flexible manufacturing system introduced in section 2. The input structure model of plant components and job flow connections is shown in Figure 12 (b). The machining center station (MCS) consists of two identical machining centers (MC1 and MC2) constructed from the MC model base previously mentioned. At the highest system level, the behavior requirements are to prevent blocking and starving of job flows which are from either LT1 or LT2 to either MC1 or MC2, and from either MC1 or MC2 to either UT1 or UT2. The user model for the behavioral requirements is input at the FMS layer as shown in Figure 13 (b).

One particular interest is the coverage of control which can be seen as sensor/actuator placement problem. The controller at a certain resource layer can control all of the plant components that hierarchically belong to the resource layer. For example in Figure 12(b), MC, MCS, and FMS include IB, MT, and OB, all controllers located in MC, MCS, and FMS can control IB, MT, and OB. On the contrary, LUS and AGVS cannot control any behavior related with IB, MC, and OB because the components are not in place. Therefore, the first behavior requirement in Figure 13 (a) were able to be inputted in the MCS layer or the FMS layer with a conventional hierarchical naming such as:

\[
\text{IF (MC1.IB, occupied) AND (MC1.MT, empty) THEN (MC1.IB, transfer, MC1.MT) END; or} \\
\text{IF (MCS.MC1.IB, occupied) AND (MCS.MC1.MT, empty) THEN} \\
\text{(MCS.MC1.IB, transfer, MCS.MC1.MT) END;}
\]

In terms of selection of the layer based on the modular and distributed control, however, it is recommended that the users define behavior at the “lowest” layered resource rather than at higher resources. Therefore, to control the behaviors described in Figure 13 (a), MC is the lowest resource rather than the MCS station or the FMS.

Based on the user input of structure models and dynamic models, the generated MHC structures are Figure 14 (a) and Figure 14 (b). In this example, the controllers of LUS and MCS work as only signal distributors, the controller of FMS only work as a command generator, and the controllers of MC1 and MC2 serve as signal distributors as well as command generators. Finally, Figure 15 shows the animation of the FMS simulation using a developed simulator.

5. Conclusions and Discussion

In order to develop a rapid modeling system for an automated manufacturing system simulation, we tried to solve two problems.

1. To find a robust and efficient message-passing architecture corresponding to the hierarchical and modular user view;
2. To generate the internal message-passing architecture from a limited amount of
user-input information.

Firstly, we have surveyed the control architecture of physical manufacturing systems. Then we proposed a hierarchical control architecture in which every resource has one controller with plant components and, between the job flow connected components, there exist short cut signal couplings to resolve the traffic overload. In spite of its robustness and efficiency, the modified hierarchical control architecture is somewhat overcomplicated for the user, so we sought to reduce the user model overload.

Secondly, from the coupled model of DEVS and DEVSim++, we have specified the structure of the Resource Class and its derived classes in which the modified hierarchical control is embedded. From the atomic model, we have designed and implemented some
Figure 15. Animation of the FMS

classes working as sensors and actuators, together with controller classes working as signal distributors and command generators. In order to construct a complete MHC architecture from user input information, we proposed an algorithm in which the input requirements are limited to the structure and dynamic information of plant components, not including the controller structure and controller dynamics.

Finally, in order to illustrate the modeling procedure of a system, we used a flexible manufacturing system. In the procedure, the gradual and modular modeling approach supported by the developed rapid modeling system are introduced. And some modeling issues related to alternatives of behavior definition in multi-hierarchical layers are addressed.

Even though the rapid modeling system has been applied to a various type of automated manufacturing system, more research is needed in order to establish it as a general modeling system. Following is a list of possible research directions related to the proposed rapid modeling system:

(1) Open architecture that supports the users to be able to extend the signal set and
related plant dynamics;
(2) Extending the user-input dynamic description method (described in section 3.4.2) to cover a wide variety of control strategies.

Appendix: Detailed Dynamic Models

Shown in Figure A1 is the structure of a phase-transition diagram representing the dynamics of an atomic model. The graphical notation is similar to that used in the literature [20][21].

A phase is either an active type or a passive type. An active node represents an active phase with a finite lifetime \( t_a \). After the time advance by \( t_a \), an atomic model outputs something through an output port and an internal transition takes place simultaneously. On the other hand, a passive node represents a passive phase which can be awakened only by external events (i.e., infinite life time and no outputs).

An arc denotes that the phase is changed from its source node to its destination, whereas a transition function on the arc describes the changes of the various state variables. An arc is either external or internal. An external transition is triggered by externally inputted events. On the other hand, an internal transition is not triggered by any external events but by its schedule.

An input port is denoted by a hollow triangle and an output port by a shaded triangle. The initial node in a phase-transition diagram is indicated by a thick arrow (node \( p_i \) in Figure A1). A passive phase node is denoted by a hollow circle and an active phase node is denoted by a shaded circle. An active node \( p_j \) generates an output \( \lambda(p_j) \) after a time.
period specified by the lifetime $ta(p)$, which is expressed at the node as $ta(p); \lambda(p)$. Since all passive nodes have infinite $ta(p)$, we omit the $ta(p)$ and $\lambda(p)$.

An internal transition is denoted by a dashed arrow, and an external transition by a solid arrow. Here, to prevent diagrams becoming too complicated, special external transitions (self-looped solid arcs) in which there are no changes with the phase value are omitted. The input event and output event are expressed as $x?msg$, meaning that the signal "msg" is received from the input port "x," while $y!msg$ means the signal "msg" is sent to the output port "y." For each transition function, its condition COND and operation OPR are expressed as if(COND) {OPR;}. Moreover, in {}, any condition form (i.e. if(COND)) can be used. We mark only a label "( )," and add the extra description using something like pseudo-codes in which there should be a matched label. In COND and OPR, operators used are & (and), || (or), = (equal), != (not equal), ~ (not), := (assignment).

Based on the phase-transition diagram, Figure A2 shows behavior of the Controller Class. Since the lifetime and output of all active phases are illustrated completely in Figure A2 (a), the Controller’s time advance and output functions are omitted in this paper. On the other hand, since the internal transition function and the external transition function of the Controller are not described in Figure A2 (a), Figure A2 (b) and Figure A2 (c) explains the internal and external transition function, respectively. And Figure A2 (d) shows the related functions called by the internal and external functions of the Controller.

Figure A3 illustrates the dynamics of the control load point. In Figure A3 (a), the lifetime and output of all active states and internal transitions are completely explained, except for the external transitions. So Figure A3 (a) is the additional pseudo code of the external transition function as shown in A3 (b).
Figure A2. Dynamics of Controller
References


Cybernetics, pp.1392-1397, 1994


