Int J Adv Manuf Technol (2001) 17:850–859
© 2001 Springer-Verlag London Limited

The International Journal of
Advanced Manufacturing Technology

An Object-Oriented Approach to an Agile Manufacturing Control System Design

K. W. Young, R. Piggin and P. Rachitrangsan
Warwick Manufacturing Group, International Manufacturing Centre, University of Warwick, Coventry, UK

Modern manufacturing control systems must respond quickly to continuous change in an agile manufacturing environment. With traditional manufacturing control system programming, it is time-consuming to make changes as a result of separate databases for the programmable logic controller (PLC), man–machine interface (MMI), and supervisory control and data acquisition (SCADA) packages. To resolve this, an object-oriented model of the control system using the unified modelling language (UML) is proposed, which then provides control and diagnostic code, enabling the network architecture, data mapping and control and diagnostic system to be designed within a single tool. The model is defined in accordance with the virtual machine concept which decomposes the complex machine into smaller elements. Based on the virtual machine concept, the model consists of the conceptual (or normal perception for the system) and control view (the control perception of a control engineer). A case study demonstrates the concept with both control and diagnostic code consistently derived from a single model.

Keywords: Agile manufacturing control system design; Manufacturing systems modelling; MMI; Object-oriented control system; PLC programming; SCADA; UML

1. Introduction

The emergence of the agile manufacturing paradigm reflects a requirement of engineering enterprises to respond rapidly to unpredictable customer demands. In response to this trend, it is necessary for modern manufacturing control systems to provide the capability to cope with continuous change. A manufacturing control system programmed in a traditional manner takes time for modifications. A change in one sensor, for example, results in the need to update three different databases in the programmable logic controller (PLC), operator panel, and SCADA package. The control code (ladder logic programming, for instance) is normally complex and difficult to understand as it represents a high-level problem domain with low-level signals (inputs from sensors and outputs to solenoid valves). Originating in the computer science community, object-oriented analysis and design methods are considered as an approach to enable a single tool to support a modular design philosophy (encompassing network architecture, data mapping, control and diagnostic system design). It provides reusable software, a single tool to generate PLC programs, SCADA programs and MMIs, and an ability to program a fully distributed control system.

This work applies an object-oriented approach to create a model of the control system represented in the unified modelling language (UML) and then translates it into control and diagnostic code. The design of the object-oriented model is based on the virtual machine concept, recognising that the complex machine consists of its subcomponents. As a result of applying the virtual machine concept, the object-oriented model comprises two different views: the conceptual view (a normal perception of the system) and the control view (the perception of a control engineer).

The demonstrator for this research employs facilities provided by the agile manufacturing system shown in Fig. 1. The manufacturing system consists of three robotic cells (which are the welding, riveting, and adhesive cells) operating within a single conveyor system and the tooling.

The architecture of the control system can be classified into three layers: an information layer (Ethernet), a control layer (ControlNet), and a device layer (DeviceNet). The programmable logic controller employed is a Rockwell Automation PLC-5® processor for which the ladder logic programming is developed using Rockwell Software RSLogix™. For diagnostics, Rockwell Software RSView32™ (SCADA) is used. Rockwell Automation Panel View is used as an operator panel.

2. Object-Oriented Methods in the Manufacturing Arena

Applications of object-oriented design in areas of manufacturing software development given by Usher [1] in 1996 include product modelling and design, process planning, manufacturing

Correspondence and offprint requests to: Dr K. W. Young, Warwick Manufacturing Group, International Manufacturing Centre, University of Warwick, Coventry, UK.

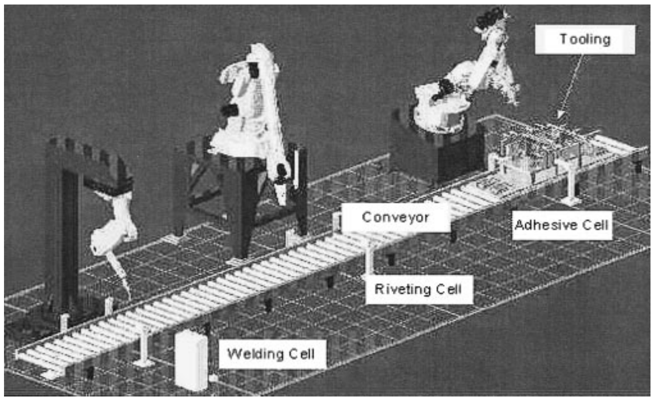


Fig. 1. The agile manufacturing system employed as a demonstrator.

system modelling and simulation, planning and control, and manufacturing databases. This section focuses on manufacturing system modelling in the context that supports the development of control software, encompassing the representation of devices in manufacturing systems as object groups and classes as well as the relationship between them.

In 1992, Mize et al. [2] applied the object-oriented concept to model manufacturing systems [3–5] by classifying objects within manufacturing systems into three categories: physical, informational, and control/decision objects. As a result, the system can be independently modelled based on these three groups of system objects in a more natural environment [1]. A further example of a similar classification was given by Bodner et al. [6] in 1994; objects could be categorised into plant, control, and database objects [4].

In 1993, Bellorin and Fishbourne [7] used object-oriented analysis (OOA) – based upon the principles proposed by Shlaer and Mellor [8,9], and Coad and Yourdon [10] – to address the problem of controlling a flexible system for batch production. The complete analysis consists of three orthogonal viewpoints: the information model (entity/relationship diagrams), the state machines and the data flow diagrams, and the object communication model. Objects, their relationships, and their attributes are defined in the informational model. The active objects are determined from the information model and their behaviours are then defined in the state machines. All active objects in the system and the messages exchanged between them are described in the object communication model, providing the chance to correct any inconsistencies or impossibilities modelled in the individual state machines. The dynamic aspect of the messages exchanged within the system can be further defined into the object communication model viewpoint by employing Petri nets to fulfil the needs of control engineering.

In 1994, Wu introduced a new approach for the analysis of manufacturing systems called hierarchical and object-oriented manufacturing systems analysis and definition (HOOMA) [11]. The method is a combination of two general object-oriented analysis methods (object-oriented analysis by Coad and Yourdon [10], and hierarchical object-oriented design [12] by the HOOD work group) that assists the analysis and definition within the manufacturing context. The aim of HOOMA is to connect the difference between the function-based approach and the pure object-oriented approaches by making use of existing tools in conventional approaches. Unlike traditional modelling methods, HOOMA allows the time-dependent logic to be encapsulated in the system objects. Because of this, it makes it possible to present dynamic processes through object interactions. The majority of graphical notations used in HOOMA are based on those in object-oriented analysis [10], whereas some new notations are additionally introduced. Procedures for an analysis are carried out in an iterative manner.

In 1997, Chen and Lu [3] established the Petri-net and entity-relationship diagram based object-oriented design method (PEBOOD), which is an object-oriented design method for the control software design. The method employs three modelling tools: IDEFO, an entity relationship diagram, and a Petri net. PEBOOD consists of four steps:

1. Analyse system requirements by constructing an IDEFO model. Classes, their attributes and operations are identified.
2. Identify entities in the manufacturing, system by constructing an entity relationship diagram. Classes, their relationships, operations and static attributes are identified.
3. Construct Petri-net models of manufacturing resources and controllers. Dynamic attributes and operations of physical classes are described.
4. Use transformation rules to transfer all the classes, their attributes and operations from IDEFO, which is an entity relationship diagram, and a Petri net to the object-oriented model.

Also, in the same year, Park et al. [4] proposed the job resource relation-net modelling framework (an object-oriented modelling framework) for flexible manufacturing systems. Similar to the object modeling technique (OMT) by Rumbaugh et al. [13], the job resource relation-net modelling framework consists of three models: static layout model (object model), job flow model (functional JR-net model), and supervisory control model (dynamic model). The development of a control program generator was suggested to convert the job resource relation-net model into supervisory control programs.

In 1999, Duran and Batocchio [14] established a methodology called the object-oriented specification technique (OOST) to specify the control logic via the use of a collection of objects representing machines and devices existing within a manufacturing system. The technique can be applied to the software development processes (as introduced by Booch [15] in 1986), from the requirement description phase to the implementation phase. Misunderstanding and excessive documentation can be eliminated when applying the methodology in the requirement analysis phase, since all the people involved (including manufacturing engineers, programmers, and analysts) can communicate effectively via a common natural language. Furthermore, the common language can be used to specify the input language for an automatic PLC software generator developed in Arity Prolog.

3. The Architectural Design of the Manufacturing Control System Model

Previous work undertaken for modelling the manufacturing system was considered in Section 2. This section introduces the fundamental concepts which have a significant effect on the architectural design of the object-oriented model. These crucial concepts are the virtual machine concept, and the conceptual and control view concept, respectively. The process of creating the model (generalised from various authors) is then explained.

3.1 The Concept of a Virtual Machine

Originating from experience gained during the previous development of the procedural man–machine interfaces (MMIs) of the demonstration manufacturing system, the concept of a virtual machine is defined to deal with the complexity of the manufacturing control system. It decomposes a complex machine into smaller elements. Each single element contains the information (for example, the name of the device, the vendor, the last failure) and status (such as "failed" or "working properly").

From the diagnostic point of view, the MMI implementation based on the virtual machine concept reduces human errors stemming from misunderstanding or misinterpretation. The complex diagnostic information obtained from the manufacturing system is reorganised and displayed according to the element of the virtual machine of interest. This aspect is recognised as "information hiding" in object-oriented communities.

Theoretically, the virtual machine is an abstraction, representing essential behaviours and attributes of the devices in the manufacturing system. It makes use of a hierachy to arrange the order of abstractions when decomposing the complex machine into smaller elements. Two types of hierarchy are used to define the relationship between virtual machines: the composition ("part-of") and inheritance ("type-of").

Figure 2 shows the representation of the manufacturing system when applying the virtual machine concept. As can be seen from Fig. 2, manufacturing cells, conveyor, and tooling are parts of the manufacturing system, which is the highest level of the virtual machine in the hierarchy. Their relationships are defined by using composition. The welding cell, riveting cell, and adhesive cell in this context are a type of manufacturing cell. The relationship between these three cells and the manufacturing cell is the inheritance. These three cells inherit the characteristics and behaviour of the manufacturing cell while possessing special characteristics and behaviours of their own. Each element for instance, the tooling, can then be broken down into the lowest level of the virtual machine which comprises sensors and solenoid valves.

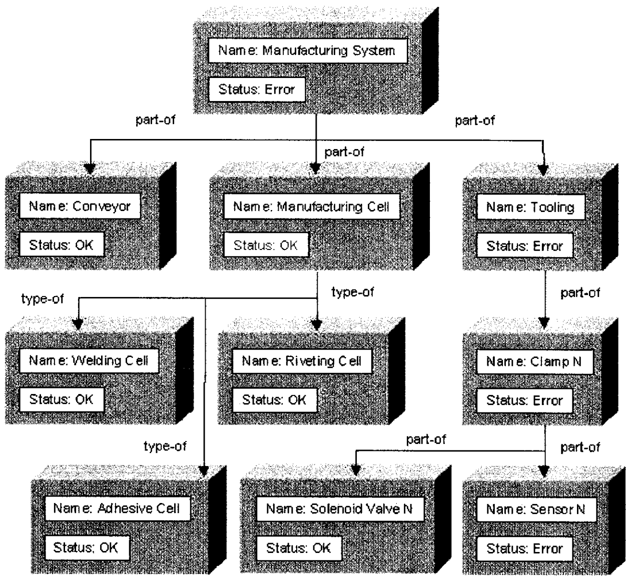


Fig. 2. The manufacturing system virtual machine.

3.2 The Concept of "Conceptual" and "Control" View

Having considered the concept of a virtual machine, its method of decomposition described previously helps people to understand the system. For example, it is easier to understand that the manufacturing system consists of the manufacturing cells, the conveyor, and the tooling. When there is a failure, it is normal to consider, for example, that a failure occurred in the manufacturing system, and specifically, that, the tooling caused the problem. The fault may be traced further through the hierarchy to find that clamp 1 failed to operate.

Alternatively, the control engineer may analyse the cause of this problem in a different manner. Sensors indicating the position of clamp 1 may fail to operate, or there may be a failure of the solenoid valve that causes clamp 1 to close or open. As the information of the failure is passed through the layers of the control system, the I/O block connecting to the sensors of clamp 1 or the valve manifold connecting to the solenoid valve of clamp 1 may fail to operate.

Taking these two perceptions into account, the former leads to the conceptual view whereas the latter can be represented by the control view, as shown in Fig. 3. It can be seen from the model that the lowest level of devices, such as sensors, and solenoid valves, is the boundary between these two views.

3.3 Selection of a Modelling Language

The manufacturing control system is modelled using the unified modelling language (UML) for two reasons. First, it combines many useful concepts from various authors [16]. The method was mentioned in the majority of the literature concerning UML [16–18], reflecting the situation during selection of an appropriate object-oriented technique for the research. The emergence of the UML puts an end to the debate over the most suitable modelling tool. Secondly, it has the potential to be an industry standard [19–21]. The UML is proposed for international standardisation by the Object Management Group which is its current developer [21]. As a consequence, UML-based applications are likely to be widely recognised and accepted in the future. The software used for visual modelling is Rational Software's Rational Rose.

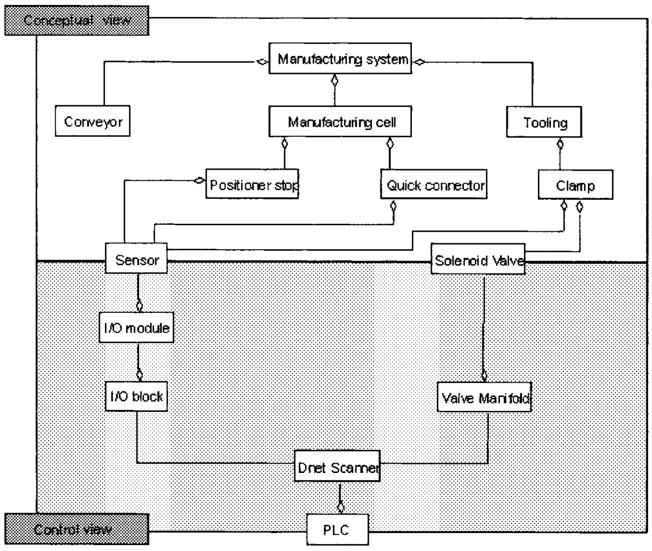


Fig. 3. An example of a model created. (This is a simplified version for the purposes of explanation.)

3.4 A Generalisation of Useful Steps from Other Methods

As the UML is process independent, the work indicates the steps to be taken to model the system by considering the generic approaches from various methods. Each object-oriented method, in principle [18], has its own modelling language (mainly graphical notations for expressing designs) and processes (advice on steps taken to achieve a design) which have various strengths and weaknesses [22]. Despite the difference in notations and procedures, all the brief examples of object-oriented methods given previously reflect some of their common goals, including static structure and dynamic structure definition.

When constructing a new subsection of the model, tasks are always accomplished in order within the following process (mainly generalised from an overview of various methods provided by [23] and [12]):

1. Objects existing in the system can be initially determined by considering real physical objects and listing them in a class diagram. Finding classes and objects is the first step suggested in object-oriented analysis (OOA/Coad–Yourdon [10]) and object-oriented design (OOD [24]).
2. The static relationships among objects listed can be defined according to the concept of the virtual machine. Although defining the static structure is a part of every method, the virtual machine concept results in a unique static architectural design.
3. Once the main virtual machine is created (a virtual device consisting of its subcomponents), its functional requirements are captured in a use case diagram. A use case concept is first introduced using object-oriented software engineering (OOSE [23]) before its integration into the UML.
4. The behaviour of the main virtual machine (and its components) with a single functional requirement, is modelled using interaction diagrams (a collaboration diagram and sequence diagram). The collaboration diagram is mainly used in conjunction with the sequence diagram.
5. All the required operations of each element of the virtual machine are determined from those messages passing between objects. A further technique to determine operations is presented in every object-oriented method, for example, the object modeling technique (OMT [13]).
6. All required attributes of each element of the virtual machine are determined from those operations defined in step 5. A further technique to determine attributes is presented in every object-oriented method, for example, the object modeling technique (OMT [13]).
7. The rest of the functional requirements are examined individually in steps 4 to 6.
8. After defining interactions between components of the virtual machine, the internal behaviour of the virtual machine and its components is analysed using state diagrams.

4. A Case Study in Applying the Concept to a Self Drill Drive Screw tool

To demonstrate that these concepts are practical, a self drill drive screw (SDDS), a tool for blind fixing, is selected for a trial run, by creating a model and then translating it to the control and diagnostic code. The SDDS system performs its function as a result of two subsystems, as shown in Fig. 4,

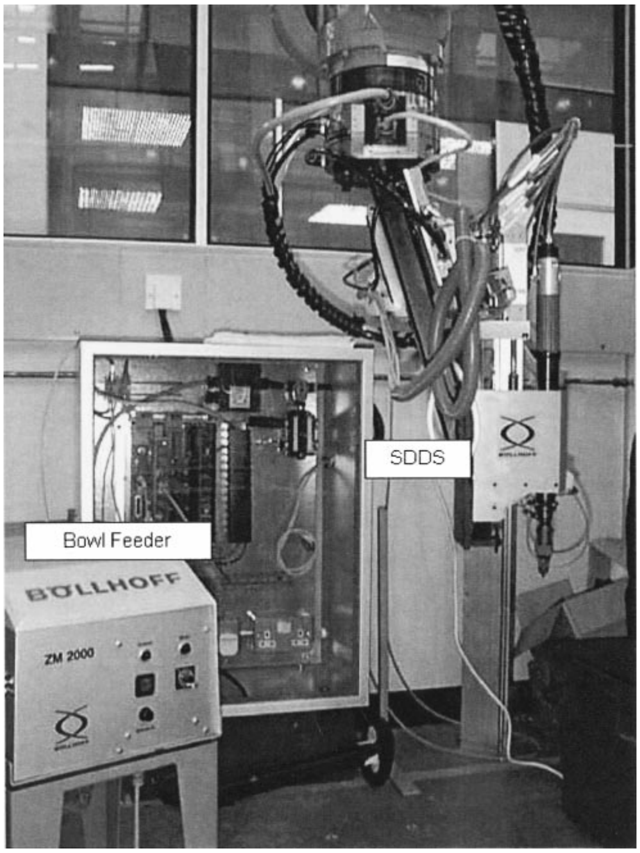


Fig. 4. The self drill drive screw (SDDS) system.

that work in conjunction with each other: the SDDS tool and the bowl feeder. The SDDS tool obtains the screw to be fired from the bowl feeder.

In order to create the SDDS virtual machine, the steps generalised from other methods are taken as an example which is described in the following subsections.

4.1 Constructing the Main Virtual Machine and its Components

To simplify matters, the case study considers the portion of the SDDS system from the conceptual point of view excluding the manufacturing system, manufacturing cell, sensors, and actuators. The case study concentrates on the tool and its main components for ease of understanding.

1. As shown in Fig. 5, the SDDS tool is decomposed into three main components: main feed cylinder, screw feed cylinder, and screwdriver. The bowl feeder is decomposed into two main components: the separator valve and the air feeder.
2. Based on the concept of a virtual machine, the relationships between the SDDS tool and its three main components can be defined using aggregation. Similarly, the relationships between the bowl feeder and its two main components are aggregated. The relationship between the SDDS tool and the bowl feeder is association.

4.2 Analysing the Functional Requirements of the Virtual Machine Created

3. The functional requirements of the SDDS system are captured. In this case study, an operator uses the SDDS tool to drive a screw, or clear it when screws are occasionally stuck in the head of the gun.

4.3 Defining how the Components Work in Conjunction with Each Other to Fulfil Each Single Functional Requirement Analysed

4. Interactions between the components of the SDDS tool (its behaviour) when it is in use to drive the screw are analysed, as shown in Fig. 6.

   The purpose of this stage is to determine the messages between the components to fulfil the functional requirement for the drive screw function. The concept behind the design is that the component is permitted to perform any particular task only when receiving a message for activation. At a particular layer of the virtual machine hierarchy, the main component acts as a project manager for a particular task. After receiving the message, the main virtual machine makes a decision about how to command its subcomponents in order to perform an assigned task. This includes how to interpret an external message into internal messages which are only for the component itself. After finishing a particular task, the component reports the result (an achievement or a failure) back to the original source of the message in the virtual machine hierarchy. Accordingly, the message can be reset when the task is finished. After receiving the drive screw message from the riveting cell, the SDDS virtual machine (in Fig. 6) translates the message for driving the screw into the "get screw" message. The get screw message is an internal message generating the "send screw" message. The send screw message causes the bowl feeder to activate its components to send a screw to the SDDS. Noticeably, the pattern with which the bowl feeder manages its resources to achieve a requested task is the same as for the SDDS. The SDDS resets the get screw message by generating the "reset get screw" internal message, when it is acknowledged via the "send screw done" report from the bowl feeder. After that, the SDDS generates the "drive screw" internal message allowing the SDDS to command its components (the main feed cylinder, the screw feed cylinder, and the screw driver) to achieve the task. When the screw is driven in the manufacturing process, the SDDS reports the success to the riveting cell so that the drive screw message can be legally reset.
5. To find out which operations are required for each component of the virtual machine, the messages considered before are used as a guideline. The message sending from the source component to the target component will be the operation of the target component.

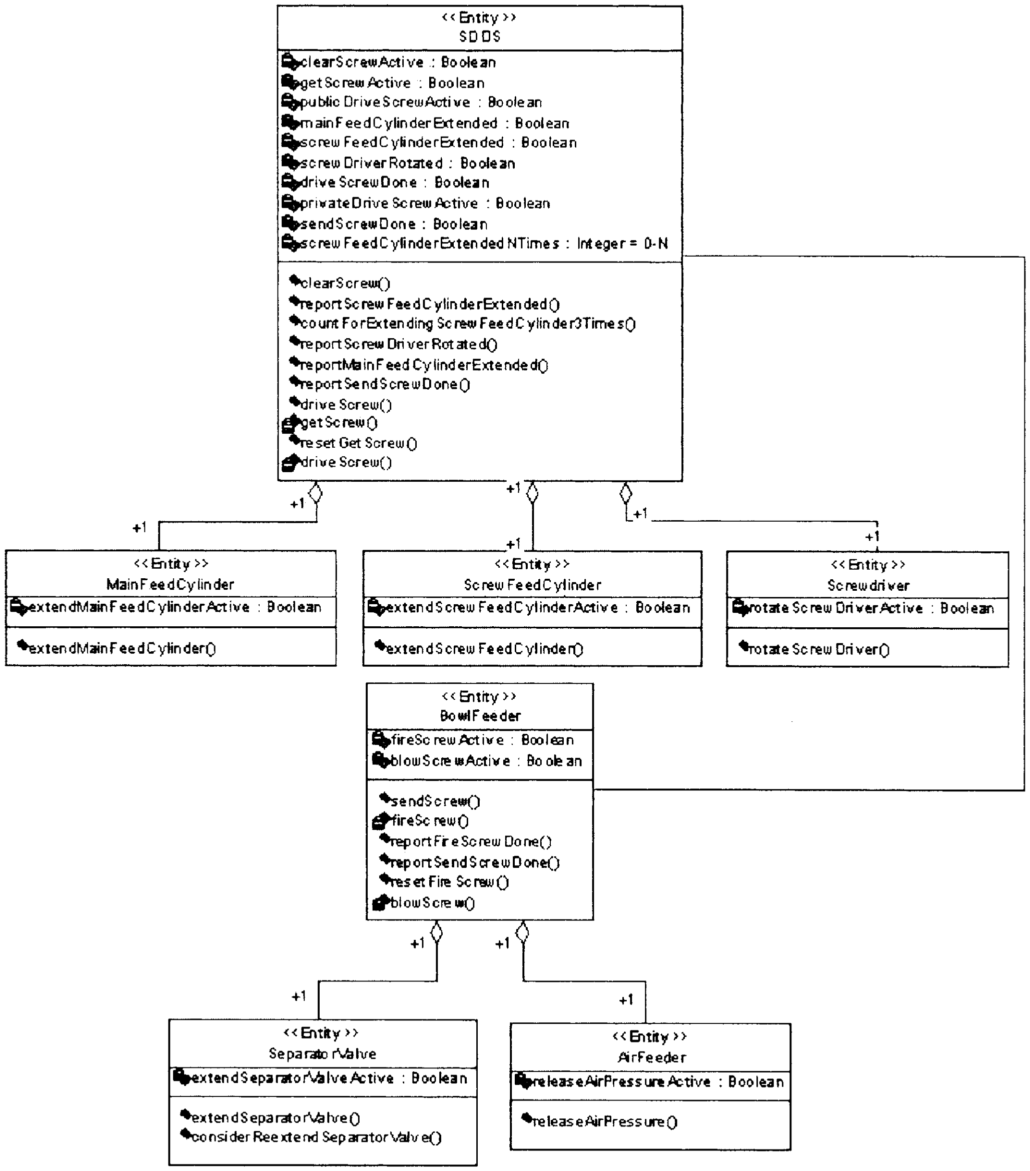
   For example, the drive screw message sent from the riveting cell to the SDDS (shown in the collaboration diagram in Fig. 6) becomes the drive screw operation of the SDDS (shown in the class diagram in Fig. 5).
6. To determine attributes required for each component of the virtual machine, those operations derived in the previous step are used as a guideline. The work lists the required attributes for each component to reflect two facts for diagnostic purposes: the particular operation being executed, and when it is finished. For example, the drive screw message operation of the SDDS can be used initially to derive the two attributes shown in Fig. 5. These attributes are the public drive screw active attribute, indicating that the operation is being executed, and the drive screw done attribute, monitoring that the operation is finished.
7. With regard to the real analysis, the rest of the use cases are analysed individually in the manner described previously.

4.4 Defining the Dynamic Behaviour of the Virtual Machine and its Components

8. An example given in Fig. 7 shows the internal behaviour of the "riveting cell" defined in a state diagram. The state diagram shown defines how the "riveting cell" reacts with two input messages.

4.5 A Translation of the Virtual Machine and its Components Modelled to PLC Programs

The virtual machine and its modelled components are mapped to the SFC as shown in Fig. 8 by making use of a simultaneous branch. As a result, the component are active at the same time and their operations are triggered by input messages from other components.



<<Entity>>
SDDS
clearScrewActive : Boolean
getScrewActive : Boolean
publicDriveScrewActive : Boolean
mainFeedCylinderExtended : Boolean
screwFeedCylinderExtended : Boolean
screwDriverRotated : Boolean
driveScrewDone : Boolean
privateDriveScrewActive : Boolean
sendScrewDone : Boolean
screwFeedCylinderExtendedNTimes : Integer = 0-N
clearScrew()
reportScrewFeedCylinderExtended()
countForExtendingScrewFeedCylinder3Times()
reportScrewDriverRotated()
reportMainFeedCylinderExtended()
reportSendScrewDone()
driveScrew()
getScrew()
resetGetScrew()
driveScrew()
+1
+1
+1
+1
+1
+1
<<Entity>>
MainFeedCylinder
extendMainFeedCylinderActive : Boolean
extendMainFeedCylinder()
<<Entity>>
ScrewFeedCylinder
extendScrewFeedCylinderActive : Boolean
extendScrewFeedCylinder()
<<Entity>>
Screwdriver
rotateScrewDriverActive : Boolean
rotateScrewDriver()
<<Entity>>
BowlFeeder
fireScrewActive : Boolean
blowScrewActive : Boolean
sendScrew()
fireScrew()
reportFireScrewDone()
reportSendScrewDone()
resetFireScrew()
blowScrew()
+1
+1
+1
+1
<<Entity>>
SeparatorValve
extendSeparatorValveActive : Boolean
extendSeparatorValve()
considerReextendSeparatorValve()
<<Entity>>
AirFeeder
releaseAirPressureActive : Boolean
releaseAirPressure()

Fig. 5. The SDDS system virtual machine (generated by Rational Rose, a visual modelling tool from Rational Software).

Figure 9 shows the "riveting cell" mapped to a ladder program. It is derived directly from the internal behaviour previously modelled in the state diagram shown in Fig. 7. The structure of the program file is defined by using a pair of master control reset (MCR) instructions in the PLC to represent a single operation. Each pair of MCR instructions creates a program zone that turns off all the non-retentive outputs in the zone when the MCR rung starting the zone is false [25]. The example given in Fig. 9 illustrates two public operations of the riveting cell: one for creating a drive screw message and the other for creating a clear screw message. The public operation for creating a drive screw message, for example, is active when all conditions at the beginning of the zone are true (including drive screw command or drive screw message, clear screw command, and drive screw done).

4.6 A Translation of the Virtual Machine and its Components Modelled to Diagnostic Software

The conceptual view of the SDDS tool is translated to the SCADA package as shown in Fig. 10. Each object on the MMI, including the SDDS and sensors, could also be kept in the object library in RSView32™ (SCADA package) for future use. It would be possible for all objects to exist on separate pages opened up by cursor clicks on the relevant part of the object inheriting them.

Figure 11 shows the representation of interactions between components and their internal behaviours. It demonstrates that it is possible to track those operations being executed for a particular functional requirement, and the messages passing between components externally and internally. The MMI rep-

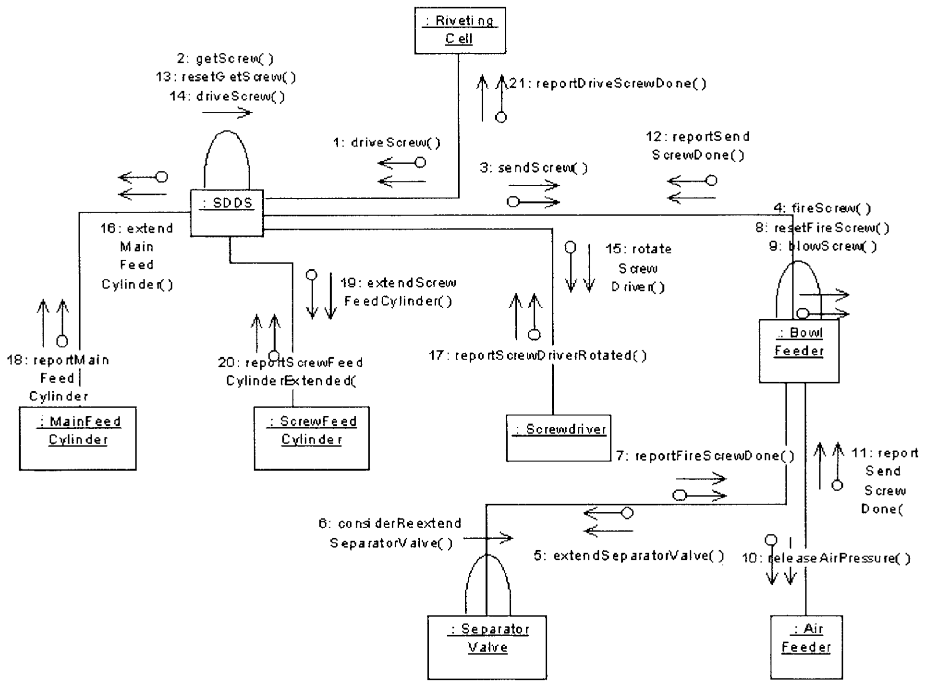

: Riveting Cell
2: getScrew()
13: resetGetScrew()
14: driveScrew()
21: reportDriveScrewDone()
1: driveScrew()
12: reportSend ScrewDone()
3: sendScrew()
: SDDS
4: fireScrew()
8: resetFireScrew()
9: blowScrew()
16: extend Main Feed Cylinder()
15: rotate Screw Driver()
19: extendScrew FeedCylinder()
: Bowl Feeder
18: reportMain Feed Cylinder
20: reportScrewFeed CylinderExtended(
17: reportScrewDriverRotated()
: MainFeed Cylinder
: ScrewFeed Cylinder
: Screwdriver
7: reportFireScrewDone()
11: report Send Screw Done(
6: considerReextend SeparatorValve()
5: extendSeparatorValve()
10: releaseAirPressure()
: Separator Valve
: Air Feeder

Fig. 6. An example of a collaboration diagram defining interactions between objects in the model for the drive screw task (generated by Rational Rose, a visual modelling tool from Rational Software).

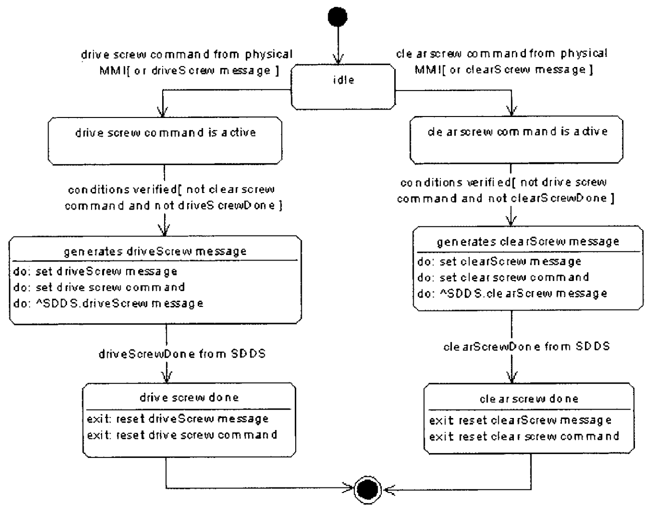

drive screw command from physical MMI[or driveScrew message]
idle
clearscrew command from physical MMI[or clearScrew message]
drive screw command is active
clear screw command is active
conditions verified[not clear screw command and not driveScrewDone]
conditions verified[not drive screw command and not clearScrewDone]
generates driveScrew message
do: set driveScrew message
do: set drive screw command
do: ^SDDS.driveScrew message
generates clearScrew message
do: set clearScrew message
do: set clear screw command
do: ^SDDS.clearScrew message
driveScrewDone from SDDS
clearScrewDone from SDDS
drive screw done
exit: reset driveScrew message
exit: reset drive screw command
clear screw done
exit: reset clearScrew message
exit: reset clear screw command

Fig. 7. Internal behaviour of the riveting cell object is defined in its state diagram (generated by Rational Rose, a visual modelling tool from Rational Software).

resents each virtual machine with its public operations (the upper compartment) and private operations (the lower compartment). It also shows the route of communication (how messages flow) and the rule that one virtual machine can communicate only with others via their public operations.

For example, the SDDS can publicly command the bowl feeder to send a screw. It is not possible for other virtual machines to directly access “fire screw” and “blow screw” internal operations. In other words, the bowl feeder and the SDDS, as illustrated in Fig. 11, hide the details of their

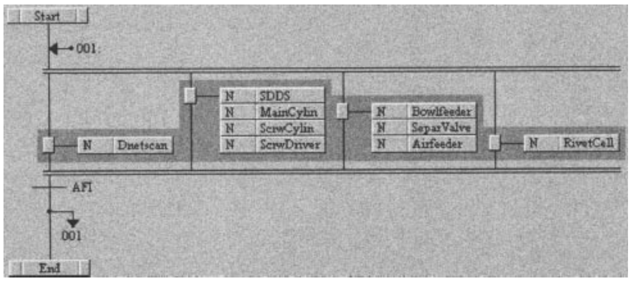


Fig. 8. Mapping the SDDS model to an SFC (generated by RSLogix™, a ladder logic programming package from Rockwell Software).

implementation for a particular task and show only public operations to other virtual machines.

5. Benefits, Issues and Discussion

The implementation benefits and weaknesses found are described below:

5.1 Benefits

The diagnostic and control code is created from a single model of the control system. This results in consistency between the diagnostic and control programs when the system is changed or modified.

Compared to complex diagnostic information obtained from the procedural MMIs, the virtual machine concept provides simple information based on the level of the virtual machine of interest in the hierarchy.

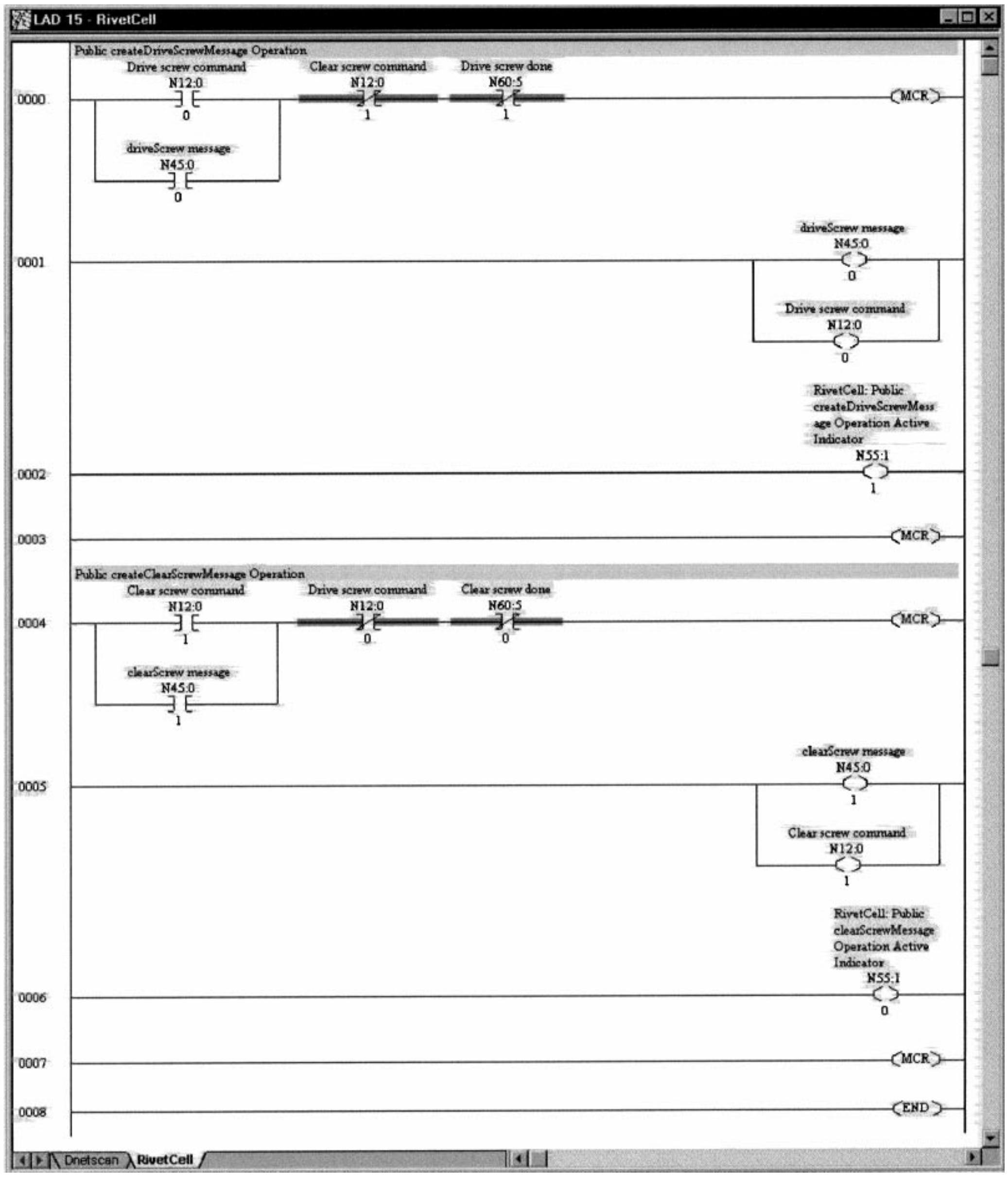

LAD 15 - RivetCell
Public createDriveScrewMessage Operation
Drive screw command N12:0 0
Clear screw command N12:0 1
Drive screw done N60:5 1
driveScrew message N45:0 0
MCR
driveScrew message N45:0 0
Drive screw command N12:0 0
RivetCell: Public createDriveScrewMessage Operation Active Indicator N55:1 1
MCR
Public createClearScrewMessage Operation
Clear screw command N12:0 1
Drive screw command N12:0 0
Clear screw done N60:5 0
clearScrew message N45:0 1
MCR
clearScrew message N45:0 1
Clear screw command N12:0 1
RivetCell: Public clearScrewMessage Operation Active Indicator N55:1 0
MCR
END
0000
0001
0002
0003
0004
0005
0006
0007
0008
Dnetscan
RivetCell

Fig. 9. Mapping the riveting object to a ladder diagram (generated by RSLogix™, a ladder logic programming package from Rockwell Software).

Reusable software, for example, components kept in the object library.

Encapsulation, mainly achieved though information hiding as shown in Fig. 11, reduces the complexity dealing with a particular virtual machine and prevents any accidental changes through inappropriate access to the internal mechanism of a particular virtual machine.

5.2 Issues

Defining the interaction between virtual machines for some complex activities as previously described in the case study in Fig. 6, is sometimes considered as a time-consuming task. However, relying on the design concept demonstrated can accelerate the process and avoid program errors.

As all objects are active simultaneously in the SFC, debugging PLC programs is difficult, compared to employing the step and transition technology. The only way to debug the PLC programs correctly is to consider the interaction between virtual machines. This relies on the MMI implementation as shown in Fig. 11.

As the control and diagnostic software currently used is not fully object-oriented, the benefit of inheritance cannot be exploited when mapping to RSLogix™ and RSView32™.

5.3 Discussion

To construct the virtual machine, the first three steps suggested in Section 3.4 can perhaps be carried out simultaneously. The main functional requirements of the real device assist the analysis of appropriate classes and objects as well as their

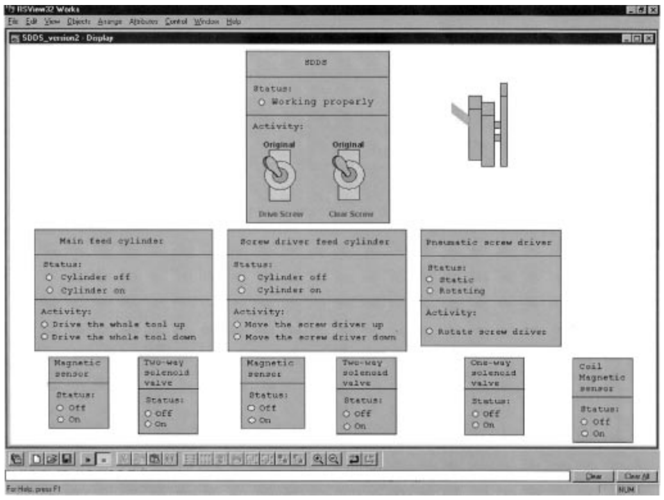


Fig. 10. MMI for the conceptual view of the SDDS tool (generated by RSView32™, a SCADA package from Rockwell Software).

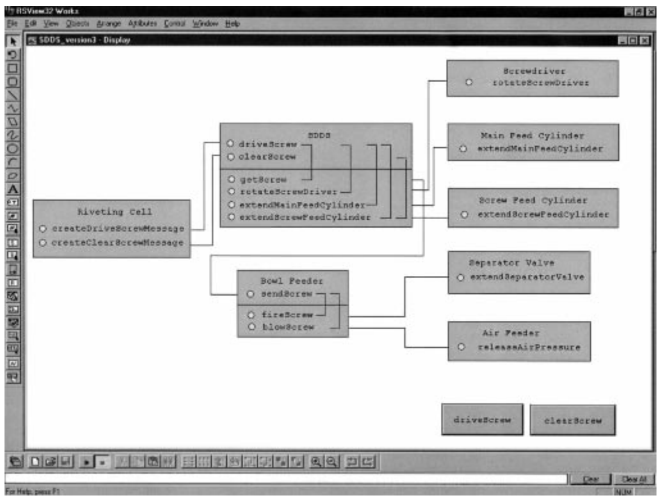


Fig. 11. Interaction between virtual machines and their components (generated by RSView32™, a SCADA package from Rockwell Software).

relationships. At the same time, studying the virtual machine and its components gives a good understanding of the mechanism. This, in turn, leads to a better requirement analysis.

Although it is recommended in the UML that only the objects with significant dynamic behaviour need state diagrams, each object requires a state diagram for mapping to its PLC program.

Compared to the idea of grouping manufacturing objects and classes as suggested by Mize et al. [1–5] and Bodner et al. [4,6] in Section 2, we initially classified objects and classes into three groups (entity, control, and interface objects) as first suggested by Jacobson et al. [23]. When the complexity of the model becomes an issue, further consideration for using the package is required, for example, a sensor or I/O block package.

The use of the UML eliminates the need to learn specific notations which are different in each method, as mentioned in Section 2, for example, between the hierarchical and object-oriented manufacturing systems analysis and definition (HOOMA) by Wu [11]. The model created using the UML is used as a common language for different groups of people involved in the development process for communicating with each other, compared to the object-oriented specification technique (OOST) by Duran and Batocchio [14]. Furthermore, it is possible to translate the model created into both control and diagnostic code, whereas OOST or the further work suggested by Park et al. [4] concerned only the control side. Lastly, as the construction of the model is object-oriented from the beginning, no transformation rules are required as mentioned in the Petri-net and entity-relationship diagram based object-oriented design method (PEBOOD) by Chen and Lu [3].

Whatever the methods used, the model created should not be considered as right or wrong. Instead, it should be viewed as more or less useful [18]. There is never a best or optimal design [12]. Different models created by different designers may be equally suitable for their purpose.

6. Conclusion

The virtual machine concept (the way to decompose the complex machine into smaller elements) is introduced. When the concept is applied to decompose the manufacturing system according to the normal perception, it is the conceptual view. While applying the concept based on the need of a control engineer, it is the control view. These two views comprise the object-oriented model of the manufacturing system. The case study shows how to translate the model created into diagnostic and control code. Although the analysis stage is sometimes time-consuming, the translated software provides many long-term advantages including robustness and reusability.

Further work includes the automation of the translation from the object-oriented model to the control code. This comprises:

Formalise UML model construction.

Develop the UML to ASCII compiler.

Investigate automation of MMI creation in Internet browser (XML)/RSView32™.

References

1. J. M. Usher, "A tutorial and review of object-oriented design of manufacturing software systems", Computers and Industrial Engineering, 30(4), pp. 781–798, 1996.
2. J. H. Mize, H. C. Bhuskute, D. B. Pratt and M. Kamath, "Modeling of integrated manufacturing systems using an object-oriented approach", IIE Transactions, 24(3), pp. 14–26, 1992.
3. K. Y. Chen and S. S. Lu, "A Petri-net and entity-relationship diagram based object-oriented design method for manufacturing systems control", International Journal of Computer Integrated Manufacturing, 10(1), pp. 17–28, 1997.
4. T. Y. Park, K. H. Han and B. K. Choi, "An object-oriented modelling framework for automated manufacturing system", International Journal of Computer Integrated Manufacturing, 10(5), pp. 324–334, 1997.
5. C. H. Kuo, H. P. Huang and M. C. Yeh, "Object-oriented approach of MCTPN for modelling flexible manufacturing systems", International Journal of Advanced Manufacturing Technology, 14(10), pp. 737–749, 1998.

6. D. A. Bodner, S. Narayanan, U. Sreekanth, T. Govindaraj, L. F. McGinnis and C. M Mitchell, "Analysis of discrete manufacturing systems for developing object-oriented simulation models", 3rd Industrial Engineering Research Conference Proceedings, Atlanta, GA, pp. 154–159, 1994.
7. J. Bellorin and C. Fishbourne, "Object-oriented analysis of a flexible batch production system", Computing & Control Engineering Journal, 4(5), pp. 233–238, 1993.
8. S. Shlaer and S. J. Mellor, Object-Oriented Systems Analysis: Modeling the World in Data, Yourdon Press, USA, 1988.
9. S. Shlaer and S. J. Mellor, Object Lifestyles: Modeling the World in States, Yourdon Press, USA, 1991.
10. P. Coad and E. Yourdon, Object-Oriented Analysis, 2nd edn, Prentice-Hall, USA, 1991.
11. B. Wu, Manufacturing Systems Design and Analysis: Context and Techniques, 2nd edn, Alden Press, UK, 1994.
12. I. Graham, Object-Oriented Methods (2nd edn), Addison-Wesley, UK, 1994.
13. J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy and W. Lorensen, Object-Oriented Modeling and Design, Simon & Schuster, USA, 1991.
14. O. Duran and A. Batocchio, "Automatic PLC software generator with a natural interface", International Journal of Production Research, 37(4), pp. 805–819, 1999.
15. G. Booch, "Object-oriented development", IEEE Transactions on Software Engineering, 12(2), pp. 211–221, 1986.
16. T. Quatrani, Visual Modeling with Rational Rose and UML, Addison-Wesley-Longman, USA, 1998.
17. H.-E. Eriksson and M. Penker, UML Toolkit, John Wiley, USA, 1998.
18. M. Fowler and K. Scott, UML Distilled, Addison-Wesley-Longman, USA, 1997.
19. Rational Software, Microsoft and Hewlett-Packard et al., UML Summary: Version 1.1, document ad970803_UML11_Summary, Rational Software, September 1997.
20. UML Revision Task Force, OMG Unified Modeling Language Specification: Version 1.3, document ad/99–06–08, Object Management Group, June 1999.
21. C. Kobryn, "UML 2001: A standardization odyssey", Communications of the ACM, 42(10), pp. 29–37, October 1999.
22. G. Booch, J. Rumbaugh and I. Jacobson, The United Modeling Language User Guide, Addison-Wesley-Longman, 1999.
23. I. Jacobson, M. Christerson, P. Jonsson and G. Overgaard, Object-Oriented Software Engineering: A Use Case Driven Approach, Addison-Wesley, USA, 1992.
24. G. Booch, Object-Oriented Analysis and Design with Applications, 2nd edn, Benjamin/Cummings, USA, 1994.
25. Allen-Bradley, PLC-5 Programming Software Release 4.4: Instruction Set Reference, Allen-Bradley, USA, 1993.