OBJECT ORIENTED MODELLING OF TASK ORIENTED MACHINES

Y C Tan, D A Sanders, S Onuh and J Graham-Jones

Systems Engineering Research Group, Mechanical and Design Engineering Department,
Faculty of Technology, University of Portsmouth, Portsmouth, PO1 3DJ, UK.

Email: yong.tan@port.ac.uk

Abstract

New methods are presented for controlling, programming and automating advanced production machines. Object Oriented Programming is used to model Task Machines. A brief review is provided of machine, communication and machine programming classifications. Task Machines are created from Functional Machines and some benefits of using this method are described through a comparison with conventional Imperative and Functional Programming methods. Using the new methods can improve efficiency and flexibility of machine programming systems.

Keywords: Object Oriented Programming, Control and Automation, Robotics, CAD, Task Machine

Introduction

In this paper, several models of Task Machines are presented and developed. Drilling, Threading, Milling and Conveyor Task Machines are considered. These are modeled using an Object Oriented approach.

A result is that control and programming of the machines is simplified because jobs no longer need to be expressed "explicitly" or by using step-by-step teaching methods in order to instruct a machine to perform a task.

The new methods mean that machine operators need less programming skill or knowledge of the task to operate a machine. Much research has been undertaken to improve or simplify the control and programming of a machine and some is included in [1, 2, 3 & 4].

In the work described in this paper, machines were considered in two different categories based on how they were used. Functional machines such as SCARA Robots or Cartesian machines were built and designed to be multi-functional for multiple purposes; structures and kinematics were not designed for a specific product or task.

With advances in technology, a machine may possess functionality that is similar to some human abilities. However, the multi-purpose functions a machine possesses may not always be beneficial. Instead, it may make a machine more expensive and more complex and extensive training may be needed for machine operators.

A Functional Machine does not possess knowledge of a task it will perform. A machine operator of this classification will need to be well versed, skilled and knowledgeable in both the programming language used to communicate with the machine and the task to be performed.

Task Oriented Machines were first proposed by Strickland to overcome some drawbacks of a Functional Machine [5]. Strickland defined a Task Oriented Machine as a machine that was constrained or built specifically for a task and not product dependent. A Task Machine is knowledgeable in the area of its predefined task. The concept was later developed further and described in more detail by Tewkesbury [6]. Task Machines were classified into three different categories: True Task Machine, Surrogate Task Machine and Virtual Task Machine.

A True Task Machine is built specifically for a task. Whereas, a Surrogate Task Machine is built from modular parts specifically for a task and the controller is replaced by a distributed controller. Lastly, a Virtual Task Machine is a general-purpose machine that has been constrained in software for a particular or specific task.

Advantages of the Task Oriented approach are that machine operators will not be burdened with low-level functionality or programming of a machine. Instead, they can concentrate and focus on improving the production task rather than exploring the intricacies of the machine.

In addition, operating a Task Machine on a specific task does not require operators to be knowledgeable in the task. Operating a machine is no longer restricted to only a highly skilled operator; instead an operator without any programming knowledge can operate it.

Human / Machine Interface

The machine and human interface can be classified into two categories based on the type of instructions needed by a machine in order to perform a task: Non-Intelligent Communication for functional machines and Intelligent Communication for Task Machines.

Interfacing with a Functional Machine using Non-Intelligent Communication means that communication is not possible in ways that a human would communicate with another human. It needs to be given with "How-to-do" instructions in order for it to perform a task. In other words, a task such as a pick and place task needs to be expressed "explicitly" or using step-by-step teaching in terms of speeds, motions, directions and positions etc. Non-Intelligent Task Communication is Sequential Procedures + Data where Data = Speeds + Motions + Directions + Positions + etc…

In contrast, communication with a Task Machine is Intelligent Communication. A Task Machine possesses similar intelligence to that of a human operator in the area of a predefined task. Hence, communication to perform a task would be by using "What to Do" instructions. A user operating this classification of machine does not need to have programming skills or knowledge of the task to be performed. An Intelligent Task Communication = Final Output + Object Description, where the Object Description = Parts Geometry + Parts Location.

Benefits of Intelligent Communication are that communication with a machine no longer needs to be expressed "explicitly" and machines possess similar levels of intelligence compared to a human in terms of a specific bounded and predefined task.

Programming Machines

Machine Programming Systems have been classified based on levels of abstraction, syntax, generation of machine program and generation of geometrical information, [1, 2, 3, 4, 7, 8, 9, 10 & 11]. The most popular way to classify a Machine Programming System has been based on the level of abstraction. This could be explained as the level of sophistication of language used to program a machine (for example, Machine Code; Assembly; High-Level or Object-Oriented Languages) to accomplish a task [1, 2, 4 & 11]. Typical levels of abstraction are:

- Joint Level.
- Manipulator Level.
- Task / Object Level.
- Objective Level.

Programming at a Joint level was achieved by specifying movements and actions in terms of joint coordinates. A machine was programmed by manually moving to each desired position and then recording the internal joint coordinates. An advantage was the simplicity of implementation. It did not require a general-purpose computer. Disadvantages are that it is impossible to program a task off-line, the system cannot be integrated with sensors and it is difficult to forecast a complete machine simulation when all the drives are in motion.

Manipulator level programming was a level above the Joint level. Programming in this level allowed programmers to concentrate on the motions of a machine end effectors (arm positions). A machine was guided to a desired position using a teach-pendant. An advantage of this level of programming was that it allowed simple integration with on-line sensor information. However, it still required a programmer to "explicitly" specify every movement of a machine instead of simply stating what actions have to be performed in order to accomplish a task. Therefore languages used in this level were also known as "explicit languages".

Task level or Object level systems were developed to improve the problems faced during manipulator level programming. The systems in this level operated in virtual environments based on objects existing in a workspace. A programmer only needed to inform the system about objects to be transferred and a task to be accomplished [10]. Languages used at this level were defined as "implicit languages". A problem with programming at this higher level of abstraction was that it sacrificed the simplicity of programming used by joint or manipulator levels. Another problem was that the programmer at this level still required the planning of the order in which subtasks were performed.

Objective level is the highest level defined in the Machine Programming System classifications. At this level, a programmer only needed to describe the parts to be used, their general layout and the final assembly. The system plans and performs the subtasks needed to accomplish the goal.

Machine Programming Languages can be classified as NC Languages, languages with specific machine syntax, (for example VAL), general-purpose languages for machines (for example KAREL) and general-purpose computer languages (for example VB, Java, C++, Fortran).

Conventionally, Machine Programming Languages were based on existing NC Languages. The advantage of using NC Languages was their ease of integrating an industrial machine into a NC-production cell. However, NC Languages lacked program structure and on-line sensing capabilities for assembly tasks. This led to a limitation in their flexibility and expandability.

Languages with specific machine syntax were specifically designed with easier syntax to adjust to usual machine terminology. This was also the main advantage of these languages (VAL was the first commercially available language using this method). A disadvantage of this category was that it also lacked structuring capabilities when they would have been useful in more complex applications. General-Purpose Programming Languages for machines were developed with additional machine-specific commands added that provided an easier integration with Computer Aided Manufacturing (CAM) systems, [4]. The advantage of this category was that it was more capable compared to languages with specific machine syntax, which tended to have better logic testing capabilities (for example, Fanuc's KAREL language). Different machine manufacturers developed different Machine Programming Languages for their Machine Programming Systems. The main reason for developing different Machine Programming Languages was to raise the level of abstraction of the machine programming system, from "explicit Machine Programming Languages" to "implicit Machine Programming Languages" [1]. However, this led to another problem for machine programmers when they had to program machines from different manufacturers.

General-Purpose Computer Languages were Machine Programming Languages created as extensions of existing Computer Programming Languages such as Basic and C. A library of procedures that handled the interface with the machine and external sensor was developed as a supplement to the General-Purpose Computer Language. An advantage of this level was that it provided structuring capabilities, an important factor for programming efficiency. This category gained favour in the machine research communities [8] and led to a system that is less limited in flexibility and expandability.

There were mainly four programming paradigms used for expressing a computation: Imperative; Functional; Logic and Object-Oriented Programming. Imperative Languages include Pascal, Cobol and Fortran, Functional Languages include LISP, Logic Languages include Prolog and Object-Oriented Languages include VB .NET, Java and C++.

Imperative Programming Languages used stepwise or sequential methods for data computation. The algorithm for the computation was expressed explicitly in terms of instructions such as assignments, tests, branching and so on. The drawback of Imperative Programming Languages was that a program written in terms of "How To" carried out a task and its design entailed every function accessing one another without boundaries. Therefore, the programs written using Imperative Programming Languages were difficult to modify or reuse if the system needed to be upgraded.

Functional Programming Languages used mathematical "lambda calculus" as a computation method. The concept of a variable was not used. A user defined a description of a problem and the language interpreter applied logical reasoning to find an answer for the problem.

Logic Programming Languages were similar to Functional Programming and also took a mathematical approach but through "formal logic". Both Logic and Functional Programming Languages could be classified as Artificial Intelligence Languages. Artificial Intelligence (AI) programming only required a programmer to define the question and the program would find out the answers for the question using logic and functional reasoning methods.

Object-Oriented Programming was a programming paradigm based on the idea of objects and classes. The idea came from Ole Dahl and Kristen Nygaard in Norway and dated back to the mid-1960s when they created Simula Language for simulating physical processes. It could be explained as an extension developed from Imperative Languages and the computation method used was similar; data was manipulated in a stepwise or sequential method. However, it could be distinguished from Imperative Languages because of the object boundary idea. Object-Oriented Design used the separation of data and functionality into object classes. In summary, a system was created from object instances. The advantages of programming using Object-Oriented Programming Languages were that design of the software is easier compared to other paradigms because modelling is based on real-world objects; hence it is more natural and easier to understand, development risks for complex systems can be reduced, maintenance and upgrading is easier and Classes could be reused by other software systems.

Defining and Modelling Task Machines

In this paper, a machine is only considered as a Task Machine when it is constrained to perform only a specific task. A Functional Machine can be converted into a Task Machine by constraining the multiple functionalities to convert it into a Task Machine that can only carry out a single type of operation.

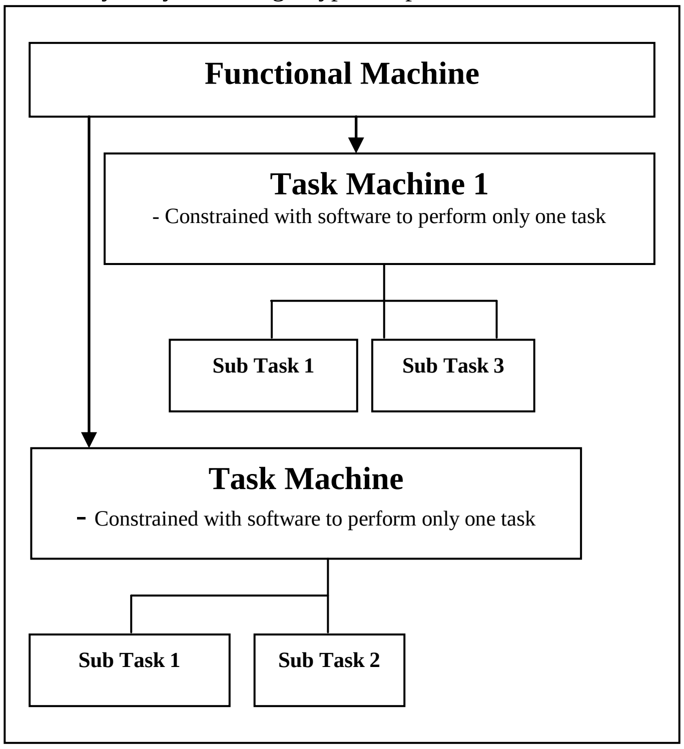
Functional Machine
Task Machine 1
- Constrained with software to perform only one task
Sub Task 1
Sub Task 3
Task Machine
- Constrained with software to perform only one task
Sub Task 1
Sub Task 2

Figure 1 – EMCO PC Turn 55-II Functional Machine

Figure 2 - Creation of a Task Machine from a General-Purpose Functional Machine

An example is the Functional Machine - EMCO PC Turn 55-II shown in Figure 1. It has multiple functions needed to perform operations such as drilling, threading, boring etc. In order for it to be converted into a Drilling Task Machine, its functionality has to be constrained to drilling operations only. The same applied when it was converted into a Threading Task Machine, its functionality had to be constrained to perform only threading operations. Figure 3 shows the creation of Drilling and Threading Task Machines.

Software is created to constrain the functionality of a machine. When it was constrained to perform only drilling task, it was then called “Drilling Task Machine” after the constrained task. A Drilling Task Machine only had the knowledge and rules required for drilling operations. The knowledge and intelligence needed were distributed among its sub tasks such as coolant control, select tool, drill hole etc.

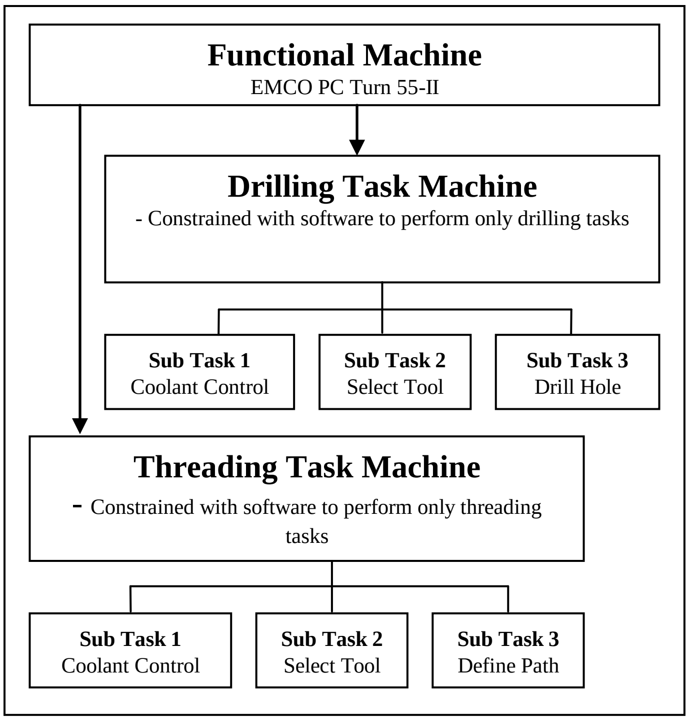

Figure 3 - Creation of a Drilling Task Machine and a Threading Task Machine from a Specific Functional Machine [Reproduced from Tan, Sanders & Tewkesbury (2004a)]

To perform a drilling task using a Drilling Task Machine, a machine operator only needed to input information such as hole size (12 mm), work piece material and the drilling position on the work piece. The coolant control sub task had the intelligence to determine for itself whether coolant was needed or not by analyzing the material information provided. The select tool sub task then determined a suitable (12 mm) drill bit to be used. The drill hole sub task generated the drilling sequence.

In the case where no suitable (12 mm) drill bit size was available from its tools collection, the Task Machine will then feedback to the machine operator that it was unable to perform the task and a suggestion may be given. A Drilling Task Machine will not have intelligence beyond the knowledge and rules needed for a drilling operation. For example, a Drilling Task Machine tools library is constrained with only drilling tools. It does not include threading tools in its library and the define path sub task needed to generate a threading path. Therefore, the intelligent communication cannot be used to perform the threading operation even though its physical structure has the capability to perform a threading operation.

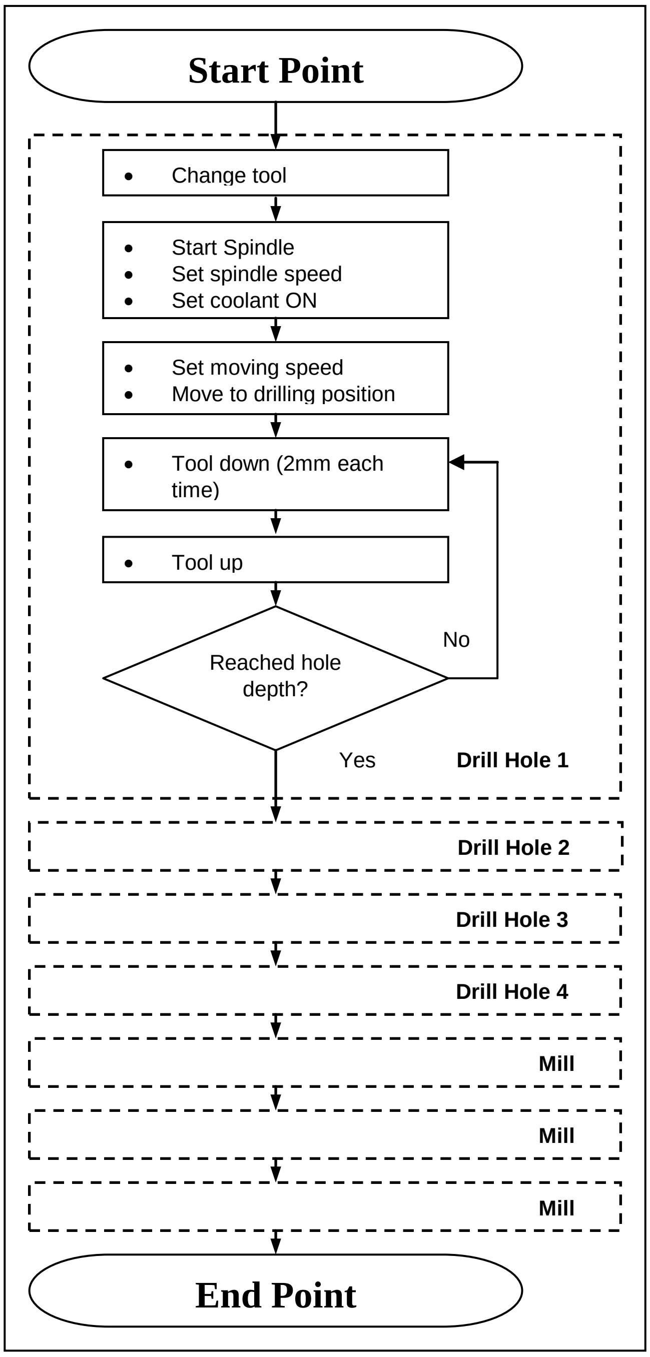

Figure 4 - NC Programming Approach Flowchart
[Reproduced from Tan, Sanders & Tewkesbury (2004a)].

The same principle is used to create a Threading Task Machine by constraining the same Functional Machine using software to perform only threading tasks. A Threading Task Machine only has the intelligence and functionality for threading operations.

A machine operator did not need to tell the Threading Task Machine what threading tool size or threading sequence was needed. The only information needed was work piece information (material and geometry) and final output (threading pitch size, location and length). The define path sub task automatically generated the appropriate path needed for the threading operation. This approach simplified and improved the efficiency of controlling and programming the machine during a particular task.

Programming

An example of system modeling using Imperative Programming and a Functional Oriented approach is described and compared with the new method.

As an example, a NC Programming modelling using EMCO PC Mill 55-II Functional Machine is described. The NC Programming Language used imperative methods for data computation. The algorithm for the computation was expressed explicitly in terms of instructions such as assignments, tests, branching and so on. Figure 4 shows a NC Programming modelling flowchart for an operation of drilling 4 holes, 2 pockets and a surface milling. Tools selection, coolant and drilling sequences all needed to be explicitly programmed by a machine operator.

A drawback of NC Language Programming was that a program written in terms of “How to do” carried out a task or operation and its design entailed every function accessing one another without boundaries. Programs written using NC Programming Languages were difficult to modify or reuse if the operations needed to be rearranged.

In Imperative Programming modelling, a programmer would need to explicitly describe procedures in detail. The drawback of using Imperative Programming is that program length is proportional to the number of workstations.

The program will be long when modelling a complex system with many workstations and thus difficult for programmer if any debugging is necessary.

Another drawback of Imperative Programming was that once the system was created, it was difficult to make any modifications as this design method entailed every function accessing one another without boundaries. For example, if a Conveyor System needed to be

Country Code 21.

reconstructed with sensors or workstations reallocated. Often programmers would choose to rewrite a whole program rather than modifying the old program for the new system.

Modelling using Object-Oriented techniques

A Conveyor System shown in Figure 5 is used to describe modelling using Object Oriented techniques.

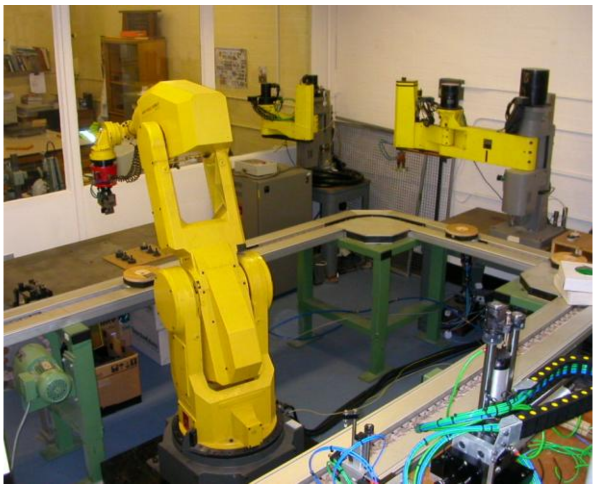

Figure 5 – Conveyor System

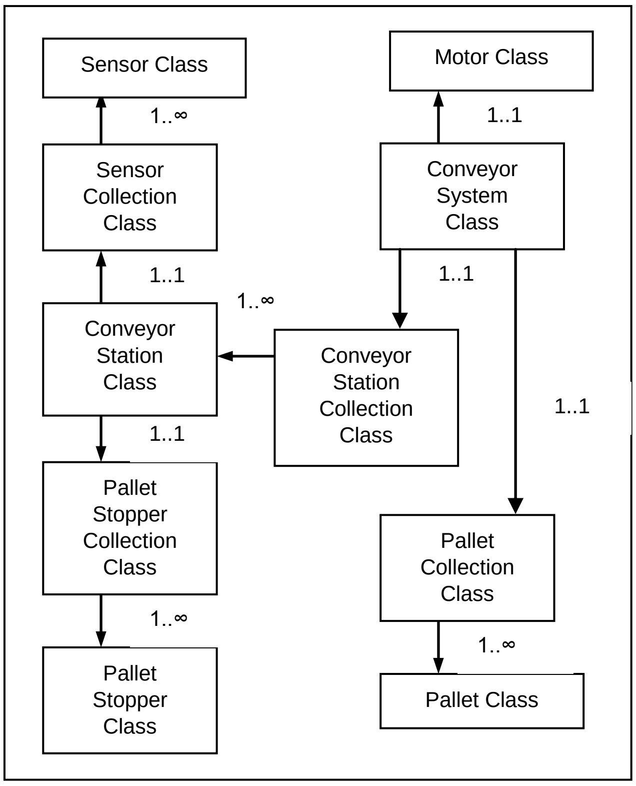

Figure 6 - Object Oriented Approach Diagram
[Reproduced from Tan, Sanders & Tewkesbury (2004b)]

Figure 6 shows the modelling of a Conveyor System using Object Oriented Programming and a Task Oriented approach. Objects within the system were identified in the first stage of Object Oriented Design. Objects identified from the real-world system consisted of tangible or intangible objects such as sensor, group of sensors, pallet stopper, group of pallet stoppers, conveyor station, group of conveyor stations, pallet and conveyor system. Classes of objects were then created and their relationships were defined. A system computation was based on object interaction. Every object was an instance of a class. A class simply represented a template for a group of similar objects. The relationship between each of the objects is shown with the arrows.

The idea of an object boundary is shown by defining individual attributes, operations and properties for each object. This is the reason why a system could be modified easily using an Object Oriented approach. An example of the details of an object's properties is described using Universal Modelling Language (UML).

The Conveyor Machine was converted into a Conveyor Task Machine using both Object Oriented Programming and a Task approach. An object instance was easily created from its template class so that the length of a program modelling a complex system was kept short and simple. Debugging and modifying in the future is easier and more efficient compared to Imperative Programming. Even if the system needed to be modified in the future, a programmer would no longer need to rewrite the whole program but could reuse classes to create a new system.

When a Conveyor Machine was converted to a Conveyor Task Machine, it possessed the knowledge and intelligence required for a specific conveyor task. The knowledge and intelligence needed were distributed among its sub tasks such as Assembly Workstation Sub Task, QC Workstation Sub Task and Reject Workstation Sub Task etc.

$$OOP\ Length \neq No.\ of\ Workstation\ (1..n)$$

A Conveyor Task Machine only had the knowledge and rules required for the specific predefined conveyor task.

To perform a conveyor task, for example to transfer a part from a start point (Assembly Workstation) to an end point (Reject Workstation), a machine operator only needed to input information such as number of parts to be transfer and its final destination. The Assembly Workstation Sub Task had the intelligence to move a pallet to Assembly Workstation and determine when to release the pallet automatically. Then QC Workstation Sub Task would move a pallet to QC Workstation and release it when the job is done. Reject Workstation Sub Task determines if a part assembly completed successfully or if not completed then should be rejected.

Discussions and Conclusions

An Objective Level was defined as the highest level to be achieved among all the Machine Programming System classifications. This level of Machine Programming System could be achieved using the Task Oriented approach so that machine operators would not be burdened by low-level functionality of a machine. They no longer need to be well versed in the programming language used by a machine or be knowledgeable in the task to be performed. Instead they could concentrate and focus on improving the production task. This approach suggested that operating a machine on a specific task could be easier and more efficient [12].

Object Oriented Programming Languages provided a better design paradigm to model a Task Machine compared to other Computer Language classifications because the whole Conveyor Task Machine System could be described as a main task made up from many other sub tasks. All these tasks were easier to model when treated as individual objects. The Conveyor Task Machine System shown in Figure 6 is an example of a system suited to a description using objects and classes. The system created using this programming paradigm could be easily modified, upgraded and debugged.

The Object Oriented approach provides an easy and efficient solution for program modification and debugging. Programs created could be reused even if the system needed to be modified in the future [13].

There are still issues for future work, such as integration of Task Machines with CAD systems to provide information and advice to designers and the use of intelligent agents.

References

[1] Aken, L.V.; & Brussel H.V. 1988. Robot programming languages: the statement of a problem. *Robotica* Volume 6: 141-148.

[2] Bonner, S. & Shin, K.G. 1982. A comparative study of robot languages. *Computer* 15(12): 82-96.

[3] Lapham, J. 1999. RobotScriptTM: the introduction of a universal robot programming language. *Industrial Robot* 26(1): 17-25.

[4] Zielinski, C. 1995. *Robot programming methods*. Warsaw.: Publishing House of Warsaw University of Technology, ISSN 0137-2319.

[5] Strickland, P. 1992. Task Orientated Robotics. Ph.D. diss., University of Portsmouth.

[6] Tewkesbury, G. 1994. Design using Distribution Intelligenge within Advanced Production Machinery. Ph.D. diss., University of Portsmouth.

[7] Gruver, W.A., Soroka, B.I., Craig, J.J., & Turner, T.L. (1984). Industrial robot programming languages: A comparative evaluation. *IEEE Transactions on Systems, Man and Cybernetics, SMC-14(4)*, 565-570.

[8]Latombe, J.C. (1983). Survey of advanced general-purpose software for robot manipulators. *Computerss in industry , 4(3)*, 227-242.

[9] Tewkesbury, G.E. & Sanders, D.A. (1999a). A new robot command library which includes simulation. *Industrial Robot, 26(1)*, 39-48.

[10] Tewkesbury, G.E. & Sanders, D.A. (1999b). A new simulation based robot command library applied to three robots. *Journal of Robotics System, 16(8)*, 461-469.

[11] Zielinski, C. (1997). Object-oriented robot programming. *Robotica, 15*, 41-48.

[12] Tan, Y.C., Sanders, D.A., & Tewkesbury, G.E. 2004a. Control, programming and automation of intelligent production machines using a task oriented approach. *Proceedings of the 2nd Int' Conf' on AI in Engineering and Technology, Vo 2*, 162-166. ISBN: 9832643384.

[13] Tan, Y.C., Sanders, D.A., & Tewkesbury, G.E. 2004b. Creating a new smart workplace using intelligent task machines. *Proc' 7th Int Conf - Work with Computing Syst*, 120-125. ISBN: 9834174209.