Generation, Control, and Simulation of Task Level Actions Based on Discrete Event Models

J.M. Couretas **J.W. Rozenblit**

Department of Electrical and Computer Engineering
The University of Arizona
Tucson, Arizona 85721
{couretas,jr}@ece.arizona.edu

Abstract

A model based method for task level command generation is used here to simulate a pipeline process. Using a discrete event systems formalism, a method for describing manufacturing systems is reviewed prior to constructing the sequential assembly line simulation. This precedes a study of the entire assembly line model, its coordination, and performance metrics. An example system is then simulated and analyzed for language generation and system performance.

1 Introduction

Expanding computer power is causing increased interest in modeling and simulation. This is especially true in engineering where physical prototypes can be costly and time consuming. One step in this direction was taken by Zeigler [3] in implementing the Discrete Event System Specification (DEVS) in a software environment called DEVS-Scheme [2]. It facilitates the construction of modular, hierarchical models and their organization.

Jacak and Rozenblit [1] introduce the exact form robot and workstation models take to work together in an assembly line simulation. Their method deals with a series of workcells, each of which has its own processing program. Moving parts between the workstations is done by robot pick and place actions. A program of robot and workstation instructions is then expressed in a task-oriented robot programming language [4] [5] (TORPL). This program controls a discrete event system simulating the sequential manufacturing process that is constructed using this methodology. It is then monitored for performance.

An additional layer is introduced for motion planning to their manufacturing system representation. This provides collision free geometrical path planning and optimal trajectory planning.

2 Background

2.1 Discrete Event Systems Representation

With the overall goal of achieving a means for rapid modeling and simulation of the entire technological line, a representational formalism is required. In this case, we employ the Discrete Event System Specification (DEVS) formalism [3].

DEVS specification consists of external inputs, X, external outputs, Y, and a set of states, S. States transitions are due to either an external event or the time limit, t_a(s), elapsing. A state change due to an elapsed time limit is called an internal transition, δ_{int}. Similarly, state changes due to external events are called external transitions, δ_{ext}. All of these transitions occur over the system's state set, S.

Interpretation of the DEVS and a full explication of the semantics of the DEVS are in [3].

2.2 Manufacturing Environment Representation

The system presently under consideration is a sequential technological process. This consists of robots and workstations where robots do pick and place actions between workstations. Robot actions in this system occur so that each step has an associated set of instructions in the task oriented robot programming language [1], hereafter referred to as TORPL.

The basic macroinstructions of TORPL are:

$$MOVE \left\{ \begin{array}{l} EMPTY \\ HOLDING \end{array} \right. \quad TO \text{“}position\text{”}$$

0-8186-6440-1/94 $04.00

PICKUP "part" AT "position"
PLACE "part" AT "position"
WAIT FOR "sensor input signal"
START "output signal"

The above instructions synthesize the robot's action program. This process requires an introduction of conditional instructions that depend on the states of each device of the assembly line. Thus, defining a program simulator requires modeling conditions that enable each program instruction. An example assembly line is shown in the Figure 1.

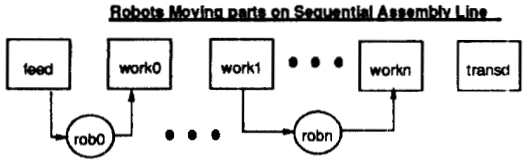

Figure 1: Assembly Line

2.3 Experimental Frame

The experimental frame executes and monitors a model. It consists of a generator sending inputs to a model and a transducer that observes the resultant model behavior. This concept is implemented on the example assembly line by making the generator a parts feeder and the last workstation on the assembly line a transducer. Feed rate of incoming parts is modulated with the internal transition time of the feeder, and assembly line performance is observed with the transducer.

The feeder, as shown in Figure 2, feeds parts to the assembly line at given time intervals. Upon sending a part to the assembly line, the feeder waits in a passive state for the specified time period before sending another part.

The transducer is positioned as the last workstation in the assembly line. Here, it takes in parts as they are completed and performs user specified calculations. The transducer is also responsible for maintaining the observation time of the overall simulation. Once this observation time is exceeded, all models stop work. The transducer is shown in Figure 3.

2.3.1 Robot Definition

As explained in [1], robot states reflect position and action. Robot position designates where it is on the

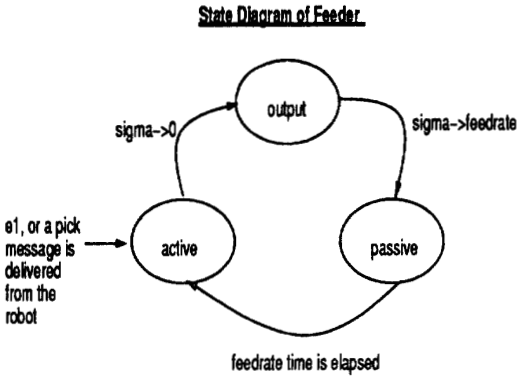

Figure 2: State Behavior of Feeder

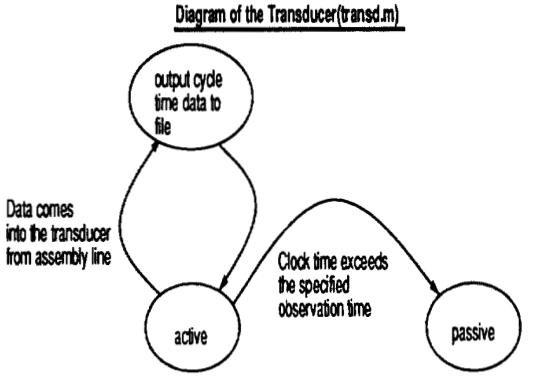

Figure 3: State Behavior of transducer

simulated assembly line. Robot actions include picking up a part, placing a part, or doing nothing.

2.3.2 Robot Operation

Zeigler [2] introduces event-based control as a method of monitoring an operation. This architecture consists of two models, a sender and receiver. The sender has a time window in which it expects the receiver to confirm the sent command's completion. Should the receiving model's confirmation not return within this time window, the sender assumes failure.

Event based control is used here to interpret robot tasks as a *state-space* [6] representation of actions needed to complete the task. The event based controller, hereafter referred to as EBC, controls the robot by sending subsequent actions, upon confirmation that the previous action is done, until the assigned task is complete. This is shown in Figure 4.

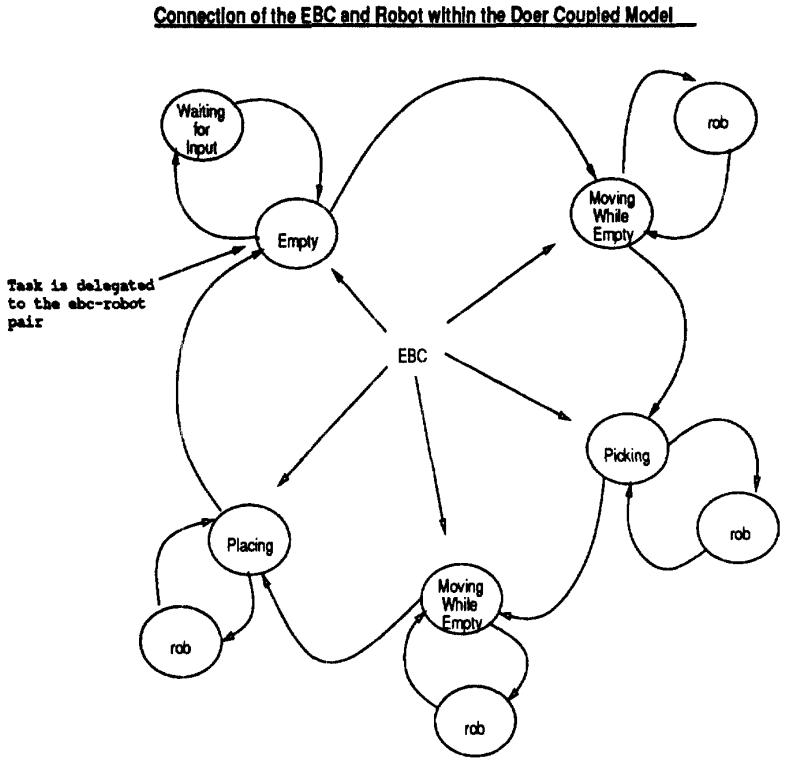

Figure 4: Internal Robot Behavior

2.3.3 Workstation Definition

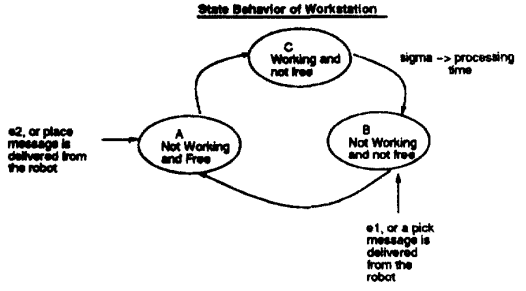

Figure 5: Internal Workstation Behavior

Workstations have three states. These states are "not working and free," "not working and not free," and "working and not free." This is defined in [1].

The workstation transitions between these states by interacting with the robot. For example, a workstation in state "not working and free" which gets a part "placed" on it by a robot goes to state "working and not free." The workstation will automatically transition to state "not working and not free" upon completing the part. Robot "picking" is then required to return the workstation to the state "not working and free". This is shown is Figure 5.

3 Modeling Effort

3.1 System Construction

Having decided on the methodology and model forms, the next step is building the sequential assembly line simulation. This includes constructing and testing each model independently, organizing models into an overall structure, coupling them, and deciding on performance metrics.

Model construction was software implementation of the sequential assembly line. With the models developed, the next step was aggregating them into one overall structure and coupling them. Communication between the models was achieved through name di-

rected coupling. This consists of sending communications directly to the model via its name.

3.2 System Architecture

The system architecture is centralized. This means that one central module, the controller, processes all of the assembly line entity states and positions, and then decides on work allocation. The system simulator's architecture is shown in Figure 6.

3.3 Task Formation

The controller's goal is to complete all tasks presently in the system. Tasks needing attention are determined by the states of sequential workstations. Robot and workstation states are communicated to the controller upon every change.

The controller is a central data repository into which state updates for all assembly line entities are directed and stored. Workstation states are tracked by a service list. It is called a service list because the workstations are continually monitored for potential requirement of service. Other lists used are the task list, robot list, and job list.

The service list contains the states of all workstations. The service list maintains a state for every workstation on the assembly line, all the time. When state updates come in from the assembly line, the service list is updated to the new assembly line representation. An example of the service list is shown below:

service-list = ((0 b)(1 a)(2 c)(3 a) ...)

A task is formed whenever two workstations are sequentially in states "B" and "A". When this happens, the two addresses are combined to make a task as shown below:

States: (1 b)(2 a)
Task: (1 2)

A task list could have any number of tasks, and would take the following form:

Task-list: ((1 2)(3 4) ...)

An example robot list has the following form:

robot-list = ((rob1 move-while-holding) ...)

Multiple tasks require multiple robots to maximize system performance. This need is met by allowing any number of robots to service the assembly line.

Once tasks are formed, a search is made for the closest robot to perform the task. Proximity is measured both by physical distance and by an estimation of time to completion from the present task. Physical distance is the distance from the present location of the robot to the first workstation in the given task. This is what is used for free robots - robots that are empty and waiting for an assignment. If a robot is still completing an action, distance equivalent approximations are interpolated from the present robot state. This approximation is then added to the actual distance between addresses in order to find the total distance between the present robot and the first workstation in the task.

Robots presently working on a task are also evaluated for their proximity to tasks awaiting assignment. The main reason for this is that a robot close to a task to be assigned could be nearly finished with its present work when the nearby task becomes available. If only idle robots were considered, an idle robot farther away might be chosen for the task, resulting in a high travel time. Choosing the robot already working amounts to not choosing a robot, and waiting until that robot reports that it is free before assigning the task.

When a task is assigned to a robot, this becomes a "job." Jobs are sent to the robot designated as the first element of the job. An example of a job is shown below:

Available Robot: (rob1)
Task: (1 2)
Job: (rob1 (1 2))

3.4 Robot Application Strategy

An example of control modification is assigning different location prioritization algorithms to the robots. They are assigned to prioritize service to the beginning of the assembly line, the end, and the closest available task. This exercise monitors the system performance differences of alternative prioritization schemes.

Priority on the beginning and end of the assembly line started off as an algorithmic exercise. Over time, however, we saw the merits of either of these in application. Prioritizing the beginning of the assembly line might occur in situations where the line is, for one reason or another, perpetually starved for parts. Similarly, priority on the end of the assembly line, on getting parts out, might occur in a situation where there is a bottleneck at the end of the assembly line. Additionally, on a multi-robot assembly line, different robots could be assigned different operational

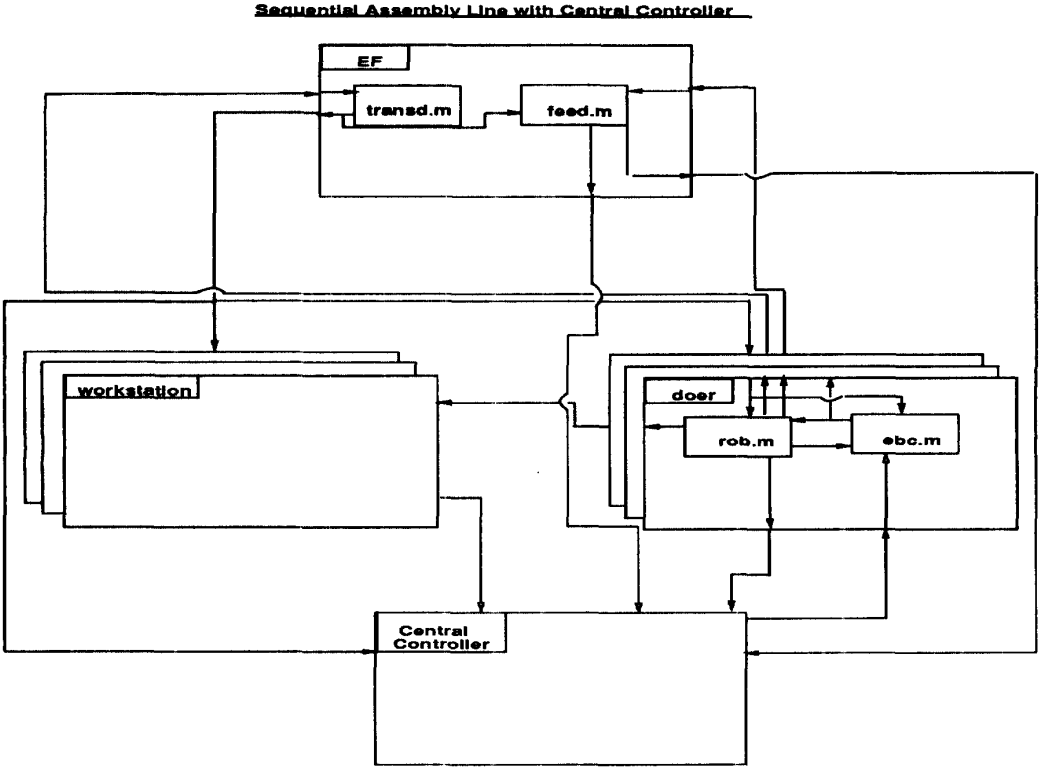

Figure 6: Assembly Line Architecture

algorithms for part processing.

Simulating the robots with different prioritization schemes gives us a trace of the robot movements on the sequential assembly line. Assuming the simulation accurately depicts the assembly line, this "trace" is validated robot code that could be used to control the actual robots working on an assembly line.

3.5 TORPL Code

TORPL code is sequentially output for each step taken by the robot model. A task, (0 1), assigned to $robot_1$ generates the following TORPL code:

$robot_1$ **move while empty**	$robot_1$ moves to position 0.
$robot_1$ **pickup widget at feed**	$robot_1$ picks up the part at position 0, the feeder.
$robot_1$ **move while holding**	$robot_1$ moves the part between workstations 0 and 1.
$robot_1$ **place widget at 1**	$robot_1$ places the widget at workstation 1.

This process is carried out in detail for each task performed. Completing the tasks leads up to completing entire jobs, and job completion leads us to look at system performance.

4 Assembly Line Performance Evaluation

In this simulation model, assembly line throughput was selected as the performance measure. It was calculated on-line by a transducer that totals the number

of parts completed and divides this by the total time that the line is operational.

This measurement is similar to what one finds on actual assembly lines. The system does an on-line update of throughput, and this allows any internal control methods that rely on throughput to automatically evaluate the present state of the system and take action.

4.1 Simulation Setup

Setting up the simulation requires deciding the feed rate of parts coming into the process, the mean processing time per workstation, and the robot execution times. The feedrate, controlled by the feeder, is one new part into the process every 5 time units. Workstation processing time is a uniform distribution from 1 to 10 time units. The robot takes 2 time units to pick up or place a part, while moving takes the same number of time units as the distance traversed. In this system, 1 distance unit equals 1 time unit.

4.2 Analysis of Model Results

Looking at throughput data for the 5000 time unit production run in Figure 7, we see the following throughput rates:

Robot focusing on the beginning	0.0492
Robot focusing on closest task	0.0494
Robot focusing on the end	0.0455

While the above numbers are relatively close, the robots prioritizing the closest available task and on the beginning of the assembly line exceed the performance of the robot prioritized to the end of the assembly line.

This performance discrepancy by the robot focusing on the end of the assembly line is due to its increased travel time between priority workstations at the end of the line and jobs earlier in the line. The robot focusing on the closest task has minimal travel time. The robot focusing on the beginning of the assembly line benefits from each workstation in the assembly line holding a completed job before it moves to the end of the line.

4.3 Data Analysis

At 0.0494 jobs per time unit, the robot prioritized to the closest task was slightly better than the robot prioritized to the beginning of the assembly line at 0.0492 jobs per time unit. A less intuitive merit of the robot prioritized to the beginning is that every workstation sits with a completed job before it starts completing jobs. So many jobs at the end of the line results in "spurts" of job completions, exemplified by throughput oscillations as shown in Figure 7.

The slowest throughput, 0.0455 jobs per time unit, came from the robot prioritized to the assembly line's end. Decreased performance with this algorithm results from its priority on getting jobs to the assembly line's end. It uses too much travel time going from end of the line jobs to those earlier in the process.

Algorithms focusing on the beginning of the assembly line, or simply on the closest task at hand, outperform the algorithm whose only goal is output. A focus on the process, instead of only the goal, turned out to be the winning method.

5 Future Work and Conclusions

Model operation exemplified how TORPL code can be generated and verified before implementation in an actual manufacturing system. The example here requires both task formation, and assignment of the task to the robot most suited. Real world scenarios are rarely so simple. Future systems require planning to deal with unprecedented difficulties.

Planning is commonly approached in two ways, on-line and off-line. Off-line planning has merit in that computation time does not impose upon the present process. On-line planning benefits the user in its autonomy. Decisions are made and executed on the spot.

ElMaraghy and Rondeau [7] propose "a new environment for off-line programming of robot tasks, including a feature-based geometric database, an off-line programming system with a knowledge base, an expert task and motion planner, and a run-time monitoring system." This comprehensive planning methodology would substantially extend control capability in our system.

Extending task formation in our system to include the benefits of planning is a natural extension. Similarly, extending individual or group robot capability would take us from pick-and-place robotization to a myriad of possibilities.

Decentralization of this system would involve moving task and job formation to the robot level. This is similar to Jacak and Rozenblit's [1] original conception of attaching an acceptor to each robot in order to allow it an immediate state representation of the assembly line. The difference will be transforming robots into endomorphic agents as conceived by Zeigler [2].

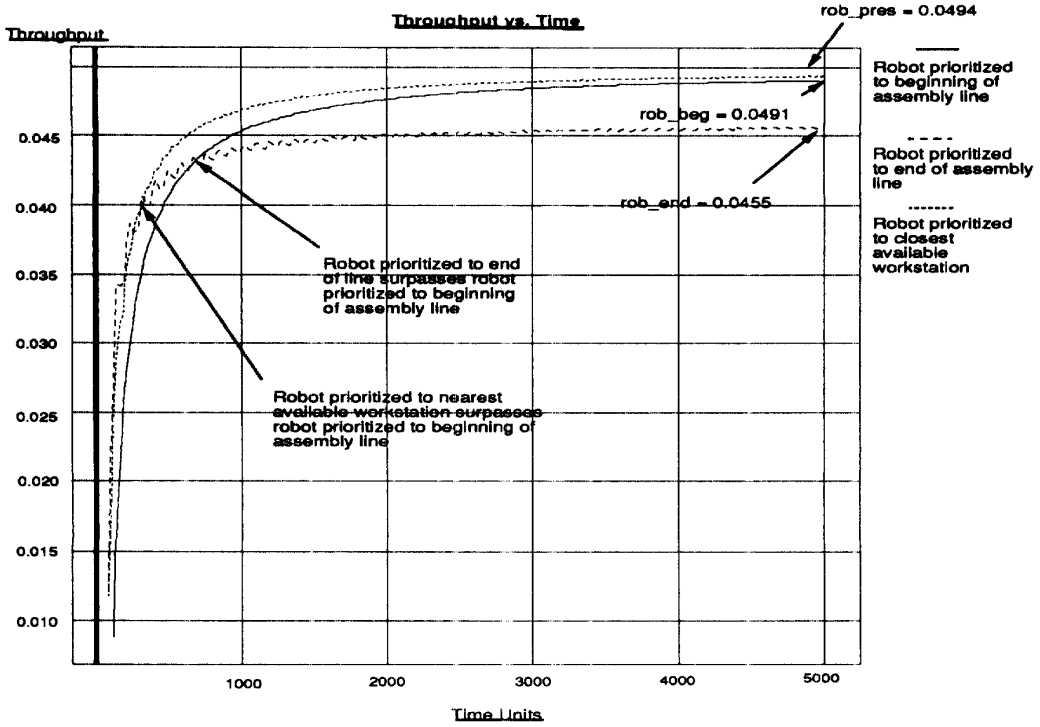

Figure 7: Assembly Line Throughput Performance

Adding planning and decentralization to assembly line entities opens up a whole new world of robustness for the system. Changing robots from simple slaves of the centralized controller to autonomous operators would significantly increase this system's versatility.

References

[1] W. Jacak and J.W. Rozenblit, "Automatic Simulation of a Robot Program for a Sequential Manufacturing Process" *Robotica*, Vol. 10, pp. 45-56, 1992.

[2] B.P. Zeigler, *Object-Oriented Simulation with Hierarchical, Modular Models,* Academic Press, London, 1990.

[3] B.P. Zeigler, *Multifacetted Modelling and Discrete Event Simulation* Academic Press, London, 1984.

[4] B. Faverjon, "Object Level Programming of Industrial Robots" *IEEE Int. Conf. on Robotics and Automation*, Vol. 2, pp. 1406-1411, 1986

[5] R. Speed, "Off-line Programming for Industrial Robots" *Proc. of ISIR 87*, pp. 2110-2123, 1987.

[6] N.J. Nilsson, *Principles of Artificial Inteligence*, Tioga, Palo Alto, CA, 1980.

[7] H.A. ElMaraghy and J.M. Rondeau, "Automated Planning and Programming Environments for Robots" *Robotica*, vol. 10, pp. 75-82, 1992.