Multi-Level Scheduling for Flexible Manufacturing[1]

Marty Humphrey and John A. Stankovic
Technical Report 95–86

UMass Computer Science Technical Report 95–86
Jan 23, 1995

Abstract

Most real-time scheduling has focused on a relatively small set of independent tasks directly invoked periodically or via interrupts. In many real-time applications such as flexible manufacturing, this system model is too simplistic. In flexible manufacturing two entirely different sets of resources must be scheduled and "connected." At the highest level, there are raw materials, robot arms, platform space, etc., and at the lowest level there are computational resources. Upon ordering products, high level resources must be scheduled, and the associated computational resources to achieve the manufacturing of those products must also be scheduled. This gives rise to the need for high level Real-Time Artificial Intelligence (RTAI) planners, low level schedulers that can handle large numbers of precedence- and resource-constrained tasks, and a suitable interface between the two schedulers. This paper presents the initial stages of the design and implementation of a flexible manufacturing testbed that incorporates all of these elements.

[1]This work is funded by the National Science Foundation under grant IRI-9208920.

1 Introduction

Scheduling in many real manufacturing sites exists at two levels of abstraction. At the higher level of abstraction, the scheduling involves objects such as materials, machines, and orders. The decisions made at this level include the selection of which orders to service and when to service them, based on the value and deadline of each order, and the availability of raw materials. The feasibility of servicing an order is determined by the scheduler at the lower level, which, as opposed to the higher level scheduler, deals with the *computational resources* needed to move robots, assemble products, etc. For example, the higher level scheduler may decide to make a certain product, but the actual manufacturing floor may not be capable of servicing this order in time. In this situation, feedback information from the lower level to the higher level allows the higher level scheduler to make more informed decisions concerning future orders, which increases system productivity as a whole. The high level will be less apt to believe incorrectly that the low level system could service the order if the high level scheduler submitted it, which means that the high level scheduler does not have as many scheduling failures from which to recover.

In a scheduling-based automated system for controlling a manufacturing site, scheduling must also exist at two levels. The key decisions to be made in the design of the two-level scheduling system are:

- How should the overall scheduling functionality be partitioned between the two levels? What objects should each scheduling level reason about?
- What information should be passed between the two levels? Should information be passed only as direct responses to queries, or can the information be passed unsolicited?
- How does the system perform as a function of the amount and frequency of interaction between the schedulers?

Because the scheduling system design is ultimately evaluated by studying the overall performance of the system, the last question is the most important. Intuitively, if too little information is passed within each message, the receiving scheduler will not be able to precisely determine what the message was intended to convey. If too few of these messages are passed, in particular, from the low level to the high level explaining the current state of the low level, decisions by the higher level scheduler that depend on an up-to-date view of the state of the lower level will be poor. Alternatively, if too much information is passed in messages, the receiving level will have to spend time computing a summary of the message that is most relevant to its processing. If too many messages are sent, both the sending level and the receiving level have wasted time that potentially could have been used on more important activities.

The goal of this paper is to explore the issues involved with selecting and implementing an appropriate level of information flow between a high level scheduler and a low level scheduler. We first introduce the system in which we test these ideas, which is the Flexible Manufacturing Testbed (FMT). The FMT is a developing system in the Center for Autonomous Real-Time Systems at the University of Massachusetts in Amherst, MA. The FMT is a realistic model of a real-world flexible manufacturing application and is a comprehensive integration of robotics, computer vision, real-time operating systems, and Artificial Intelligence (AI). In presenting the FMT, we illustrate new scheduling ideas learned from the difficulties of implementing a real-world flexible manufacturing application. After presenting the details of the FMT, we discuss the initial design and implementation of two-level scheduling in the FMT. The high level scheduler in the FMT uses AI techniques to control the submission of orders to the low level. The low level scheduler is a modified version of the Spring scheduler, which is a component of the Spring kernel [13]. The design and implementation of the two-level scheduling system in the FMT has been an iterative process, generally driven by what communication we feel must exist between the two schedulers in order to accomplish system-level goals. The work described concerning the schedulers is the state of the system to date.

While the implementation of the two schedulers and the communication between them is only in its initial stages of development, preliminary experimentation has produced situations in the FMT in which the high-level goals of two-level communication must be directly addressed. For example, we have identified that the simplicity offered by communicating at the highest level of abstraction, which is the order level, is not sufficient in certain situations for the high level scheduler to perform its intended job. This paper describes such situations and our designs to address issues issues raised by those situations. The results concerning multilevel scheduling presented in this paper are notable because they are validated in a complex model of a real-world flexible manufacturing application, and not solely a theoretical framework.

The rest of the paper is organized as follows. Section 2 discusses the FMT. Section 3 presents the high level scheduling in the FMT. This is a presentation of the implementation of AI Planner. In Section 4, the low-level scheduler of the FMT–the modified Spring scheduler–is presented. It was impossible to separately describe the AI Planner and the Spring scheduler without discussing the communication between the two systems, so some communications issues are included throughout Sections 3 and 4. Section 5 presents a high-level discussion of communication between the AI Planner and the Spring scheduler, but without being in the direct context of either scheduler. Related work is in Section 6, and the conclusions of the paper are in Section 7.

2 The Flexible Manufacturing Testbed

The Flexible Manufacturing Testbed (FMT) is a model of dynamic manufacturing systems in which raw materials and orders arrive nondeterministically, and the goal is to produce goods in some timely fashion. The value of completing the order is a function of the intrinsic value of the manufactured object and the time at which the manufactured object is produced relative to the order arrival. A single controlling process directs manufacturing processes that service orders. The controlling process decides which orders to service based upon the current raw materials and the current pool of pending, as-yet-unserviced orders. The controlling process also bases its decisions on expectations concerning the future arrival of raw materials and of new orders.

While the FMT is not intended to precisely model all the intricate details of a real-world manufacturing system, it is a comprehensive undertaking. The FMT involves an integration of different technologies:

Robotics contributes technology for coarse reaching, grasping, contact sensing, and automated assembly.

Computer Vision contributes technology for object recognition.

Automated Reasoning provides the ability to use AI techniques for process advisors, distributed factory scheduling, and concurrent engineering.

Real-Time Systems provides support for predictable and flexible response through real-time system tools, and in general provides the low level system support through which to integrate the other three components.

Figure 1 illustrates the key physical components of the testbed. These components are a rotating platform with ten bins, a linear platform with five bins, four tubes for holding completed orders, a flapper, and a robotic arm. The raw materials of products are represented by balls–either white, gray, or black–and arrive nondeterministically into bins on the rotating platform. The manufacturing of an order consists of transporting balls of the color specified in the order from the rotating platform to the linear platform, and from the linear platform to an available tube. To move a ball from the rotating platform to a tube, the ball must first be pushed into an assigned slot of the linear table. This is done by first moving the linear table so that the assigned slot is in front of the flapper, which is not mobile.

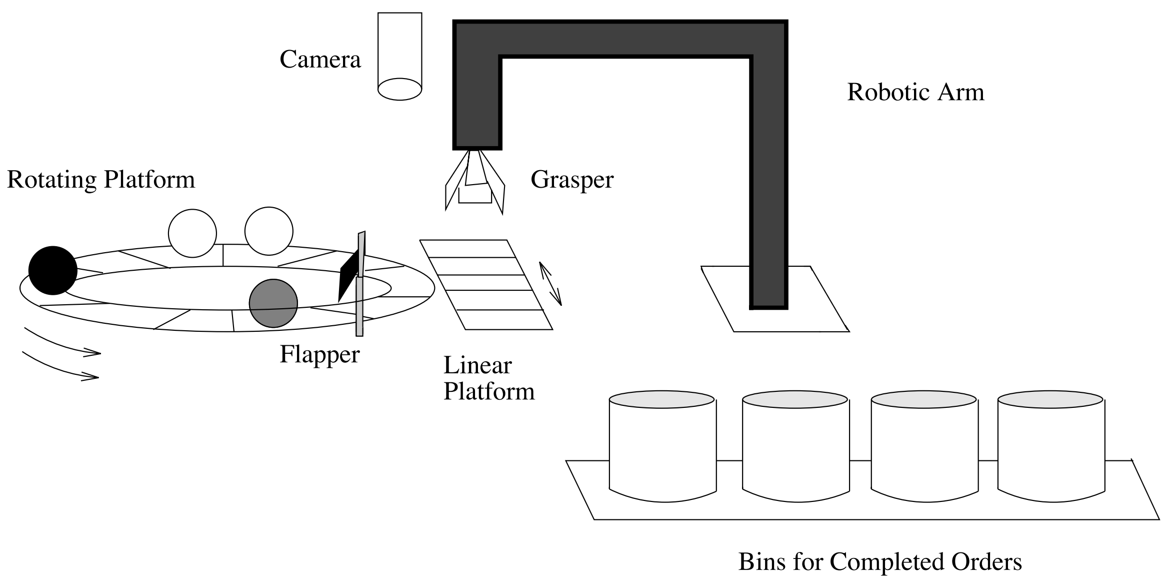


Figure 1: Schematic of the Flexible Manufacturing Testbed

When the ball has rotated around to being in front of the flapper, the flapper kicks the ball onto the linear platform. A scanner is then instructed to find the precise location of the ball within the particular linear table bin so that the robotic arm can be instructed to grasp in a precise location. Once the ball has been picked up by the grasper, it is moved to above the assigned tube, and released. Collectively, the physical components represent a set of finite-capacity stations in a manufacturing facility. A product must pass between stations in order to be machined and combined with other parts. Eventually, the part passes through the last station and is transported to a holding station until it leaves the manufacturing site.

An order has the following attributes:

- A list of balls, identified only by color
- An *ordered* flag, which indicates if the parts have to be combined in a particular ordering, or alternatively if the manufacturing system is free to opportunistically choose an ordering
- A value, in monetary terms
- An early deadline–the manufacturing system is awarded the full monetary value if the product is manufactured by this time
- A late deadline–if the product is produced after this time, no value is awarded

If the product is produced in between the two deadlines, the value awarded is proportional to the amount of time after the early deadline has passed.

The system architecture is shown in Figure 2. The high level scheduling is performed by the AI Planner, the low level scheduling is done by the Spring scheduler [10], and servicing of an order is done by the Spring kernel [13]. Orders arrive to the AI Planner from the outside world via a graphical user interface. To determine if an order can be serviced, the AI Planner consults its model of the environment, which includes the raw resources available, and the state of the manufacturing system, which includes knowledge of orders currently being serviced. When the AI Planner decides to pursue the servicing of an order, the order is passed to the Spring scheduler via an interface, which maps the AI Planner's representation of work into the Spring representation of work. The order that is passed to the

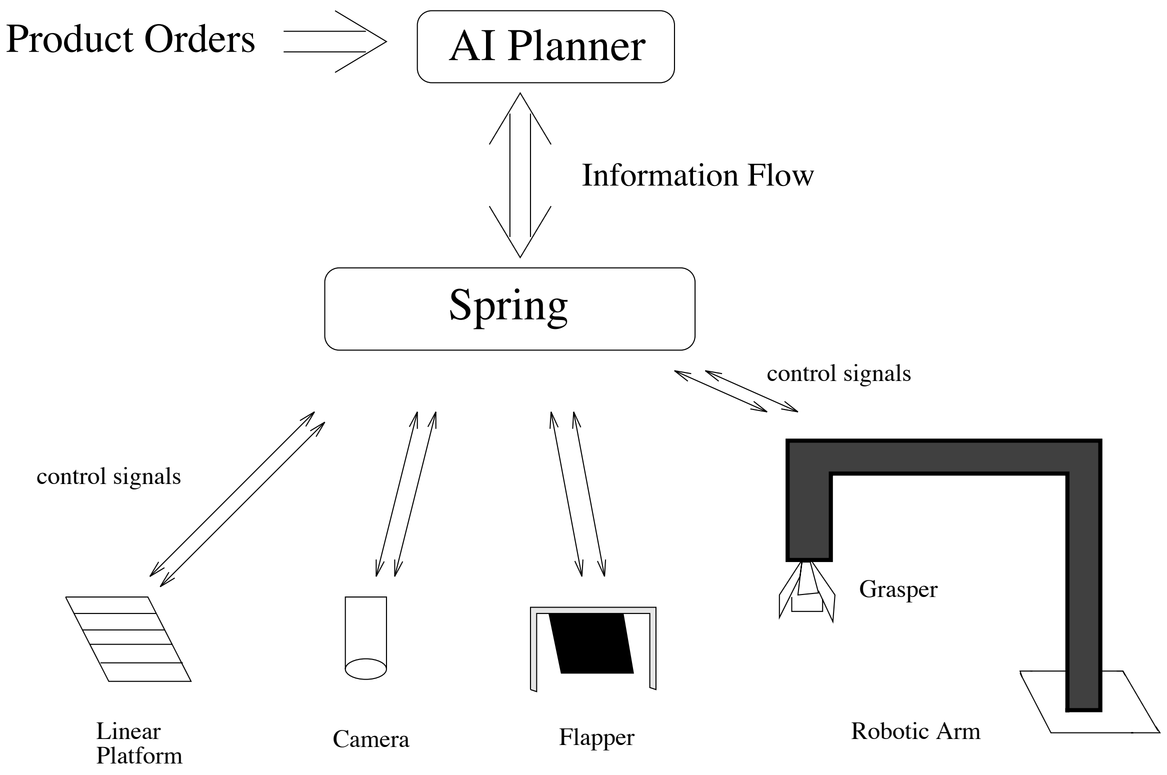


Figure 2: High Level Design of Flexible Manufacturing Testbed

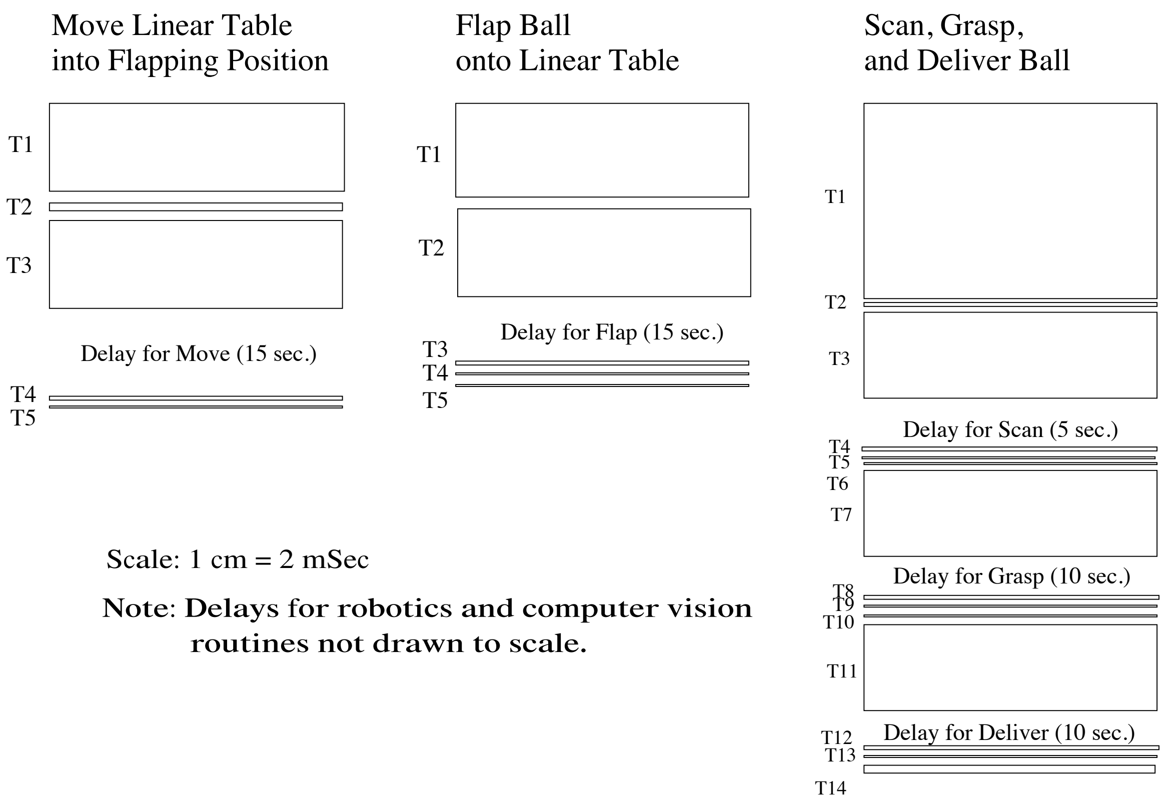
interface has a requirement for balls, and only a single deadline, because the two deadlines on the order that arrives from the graphical interface are meaningful only to the AI Planner, and not to the Spring scheduler. The first deadline on the original order is a soft deadline, while the second deadline is a hard deadline. The AI Planner determines a deadline from the values of these two deadlines, and passes it to the Spring scheduler, which treats it as a hard deadline.

The mapping from the level of the AI Planner to the Spring scheduler, or any low level real-time scheduler, can be a complicated procedure, because the two schedulers operate at different levels of abstraction and with different resources. The AI Planner is interested in managing the pool of available balls and in getting products produced by certain times. The Spring scheduler is interested in resources and actions on a much smaller scale—for example, invoking the flapper at the correct time, and activating the grasper in a slot of the linear table in which there is a ball. The obvious requirement of the interface is to map orders to the low-level actions that will service the order. There are three levels of abstraction that the interface uses to perform its mapping:

Task A task is the finest model of execution. A task has resource requirements, precedence constraints with other tasks, a deadline, an arrival time, and a worst case execution time (WCET). The task is an object that can be directly dispatched and executed on a processor by the Spring operating system. Ultimately, the Spring scheduler builds a schedule by assigning tasks to execute on processors at specific times.

Process A process is a set of tasks with precedence constraints between them. There are six processes in the FMT, five of which are related to robotics and computer vision routines: *Move Linear Table*, *Flap*, *Scan Linear Table*, *Grasp Ball* and *Deliver Ball*. The sixth process, *End Of Order*, when executed, informs the AI Planner that the order has been serviced.

Process Group A process group is set of processes with precedence constraints among them that represents a single logical action. In the FMT, there are four process groups: *Move Linear Table PG*, *Flap PG*, and *End Of Order PG* are each comprised of a single process. The fourth process group, *Scan Grasp Deliver PG*, is a sequential ordering of *Scan Linear Table*, *Grasp Ball*, and *Deliver Ball*. Figure 3 shows the individual tasks and their times for each of the robotics and visions process groups (the *End of Order* process group is a single task and is not included here). The "delays" in each process group will be discussed shortly, and are not important at this point in the discussion. It should be noted that the partitioning of processes into process groups is relatively arbitrary.

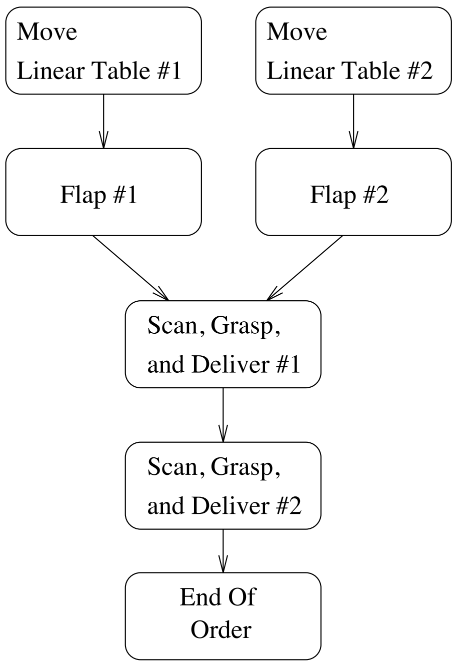


Move Linear Table into Flapping Position
T1
T2
T3
Delay for Move (15 sec.)
T4
T5
Flap Ball onto Linear Table
T1
T2
Delay for Flap (15 sec.)
T3
T4
T5
Scan, Grasp, and Deliver Ball
T1
T2
T3
Delay for Scan (5 sec.)
T4
T5
T6
T7
Delay for Grasp (10 sec.)
T8
T9
T10
T11
Delay for Deliver (10 sec.)
T12
T13
T14
Scale: 1 cm = 2 mSec
Note: Delays for robotics and computer vision routines not drawn to scale.

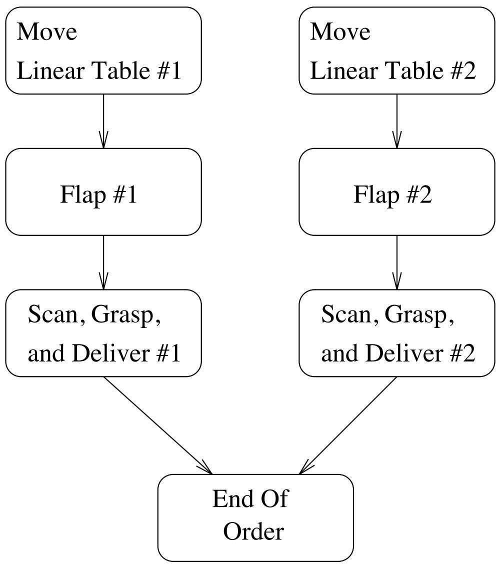
Figure 3: Tasks for Each Robotic Process Group

The first step the interface takes is to map the order to a set of process groups. The set of process groups for a two-ball order is shown in Figure 4. On the left of that figure is the structure for a two-ball order in which the first ball must be placed in the tube before the second ball, and on the right is a two-ball order in which the ordering of balls in the tube does not matter. The second step of the interface is to map each of the processes to a corresponding task graph and connect them such that precedence information is retained. The end result of the mapping that the interface performs is a potentially complex task graph that is passed to the Spring scheduler. For example, a three ball order creates a task graph of 73 tasks and hundreds of precedence constraints.

At this point, the second main requirement of the interface can be presented. When the AI Planner decides to service an order, the order is passed essentially unmodified to the interface, which constructs a representation of the work that the Spring scheduler can understand. The problem is that the order from the outside world is specified independently from its manufacturing, so the slots of the linear table that the balls of the order will be flapped into and the tube in which the balls will be placed have not been defined. Either the AI Planner or the Spring scheduler can select the particular balls, the slots on the linear table, and the tube, but there are fundamental problems with either making the selections.



2 ball order with precise ordering of balls



2 ball order with any ordering of balls

Figure 4: Process Groups Structures for Two-Ball Orders

If the AI Planner selects the bins, slots, and tubes, the manufacturing system would miss important opportunism that the Spring scheduler could provide. For example, the Spring scheduler could decide that the red ball in bin two of the rotating table should be used, because that's the nearest red ball when the flapper is scheduled to execute. Using a red ball in any other position would mean wasted time waiting for the ball to come around in front of the flapper (the flapper is invoked with a particular rotating bin number, and waits until that bin comes around before flapping). But this opportunism is very difficult to achieve, because it requires that the Spring scheduler be able to predict which ball to use based on which schedule it can build. Dynamically making this decision is complicated and error-prone. Besides, the fundamental argument against the Spring scheduler selecting the bins, slots, and tubes is that it is necessarily domain-dependent. The complications due to adding the necessary dynamic, domain-dependent hooks to the Spring scheduler might not be worth the effort. For these reasons, the intermediate level of the interface was chosen to select which balls to use, which slots of the linear table to use, and which tube to use. While hardly the optimal solution, this keeps the AI Planner away from the manufacturing details, and the Spring scheduler out of the domain. In a general sense, the interface performs resource allocation at a third level, which is between the higher level of the AI Planner, and the lower level of the Spring scheduler.

Thus, when the Spring scheduler receives the task graph that represents the order, the task graph includes the precise steps, including which physical resources to use, required to service the order. The Spring kernel [13] executes the schedule generated by the Spring scheduler on the architecture shown in Figure 5, which is a Spring node connected to a Sun Sparc. The Spring node that we are using has a single System Processor (SP), and three Application Processors (AP1, AP2, and AP3). The SP is devoted to scheduling for the APs and interacting with the AI Planner. In other words, the Spring scheduler is fixed and resident on the SP. AP1, AP2, and AP3 execute the procedures that constitute the manufacturing processes. In designing the FMT, we have recognized that we would like to allow the robotics and computer visions routines to be executable directly on the Spring node, or on special-

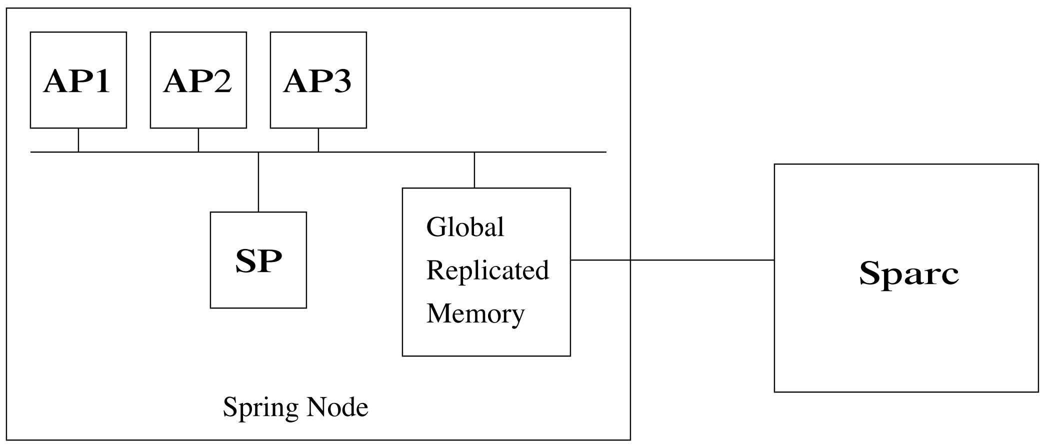


Figure 5: FMT Systems-level Architecture

purpose architectures external to the Spring node itself, as is sometimes required for robotics routines. To allow for this flexibility, we have created the *remote process invocation* (RPI). If the robotics process is executed directly on the Spring node, then all the tasks that comprise it are scheduled for execution on an AP. If the process is executed remotely, Spring packages the actual remote process in a set of Spring tasks, which are executed on the Spring node, that directly control the invocation of the remote process. The SP schedules an RPI by scheduling a task to execute on an AP that informs the remote process of its parameters and starts the remote process, and a task to execute on an AP that confirms that the remote process has terminated successfully. The length of time between the two tasks is equal to the WCET of the remote process. That way, if the remote process has not terminated in time, the SP can begin error-handling procedures immediately after the remote process was supposed to finish. By scheduling this way, real-time properties and system-level safety are ensured.

From the perspective of the Spring kernel, the Sparc in Figure 5 acts as a general architecture on which to run processes remotely. However, from a systems-integration perspective, the Sparc allows the robotic and computer vision routines to be developed independently from the development of the Spring kernel. For this reason, we currently execute the robotics and computer vision routines remotely, while anticipating eventually they will be executed directly on the Spring node. Figure 3 reflects that the robotics and computer visions routines are executed remotely. Each delay in the picture represent the worst case duration of the corresponding robotic process. Because the worst case execution times of the robotic processes are relatively long, the actual overhead due to Spring is negligible. For example, the setup tasks for the *Move Linear Table* process takes a total of 6.203 milliseconds, and the cleanup tasks take a total of 0.145 milliseconds. The combined overhead to execute the processes remotely therefore represents .04% of the duration of the actual robotic activity to move the linear table into position. The overheads for the other two process groups are similar.

The Spring task that informs the remote process of its parameters and thus indirectly starts the remote process executing uses Scramnet memory, which is global replicated memory that the SP and AP processors share within the Spring node, and that the Spring node shares with the remote cage. By using dedicated hardware to perform the message passing, the usual nondeterminism due to network traffic is avoided.

For the same benefits that arise when the robotics and computer visions routines are allowed to develop independently from work on the Spring kernel, the AI Planner and the graphical user interface are also executing on the Sparc. The difference of these two subsystems from the robotics and computer visions routines is that Spring currently views the high level scheduling and the graphical user interface as asynchronous, stand-alone processes that are not under its scheduling control. In other words, the

AI Planner and graphical user interface are executed remotely, but do not get directly scheduled by the Spring kernel. Efforts being made to design the AI Planner to run directly on the Spring node are at best preliminary, because of the inherent complications of having the Spring scheduler decide *when* to execute the AI Planner in conjunction with the manufacturing tasks.

3 High Level Scheduling in the FMT

The AI Planner ultimately decides how to run the manufacturing facility. Currently, the decisions made by the AI Planner include:

- Whether to immediately reject an order from the outside world because it's either not worth the effort or there are too many current pending orders
- Which raw components will be used for which orders, when two orders compete for resources
- When to submit an order to Spring given the AI Planner's understanding of the state of the manufacturing system
- When to re-submit an order to Spring, given that Spring has previously rejected it
- How long to retain an order before rejecting it because time currently cannot be scheduled on the machinery
- Whether to delete an order that has already been accepted by Spring, because a newer, more important order has arrived

Artificial Intelligence (AI) technology is used to design and implement the high level scheduler in the FMT. AI methods have traditionally not been used in hard real-time systems because of their inability to define a worst-case behavior, both in terms of space and time. In addition, the efficiency and simplicity often required for hard real-time systems directly oppose the nature of some AI algorithms. In the flexible manufacturing domain, however, the key is that while strict, millisecond-timing requirements will be imposed on the low level scheduler, the high level scheduler is intended to deal with time on the range of seconds. Presumably, this implies that more computationally-intensive AI methods can be explored.

The AI Planner reasons independently of the mechanisms that actually service the order. In other words, the AI Planner has no knowledge of the architecture of the Spring node. Essentially, to perform its job, the AI Planner builds a single time line that it uses to internally schedule the servicing of orders. An order is placed on the time line according to its scheduled starting time (*sst*), and its scheduled finishing time (*sft*). This time line is used to represent the the AI Planner's current knowledge of the state of the manufacturing site, as well as the AI Planner's intended use of the manufacturing site in the future. If both the *sst* and *sft* are before the current time, they represent the time in which the AI Planner believes the order was serviced by the low level system. If the *sft* is later than the current time and the *sst* is before, the *sft* is the time at which the AI Planner believes the order will complete, as determined from information provided by the Spring scheduler. If both the *sst* and *sft* are after the current time, and the order has been submitted, these values are the anticipated times that are subject to change before the order actually begins executing. If both the *sst* and *sft* are after the current time, and the order has not been submitted, the order is additionally labeled with a time at which the AI Planner intends to submit the order to Spring. By manipulating this time line as a result of information provided by Spring and as a result of the arrival of new parts and orders, the AI Planner determines which orders to submit and when to submit them.

The feedback that is necessary from the Spring scheduler in order for the AI Planner to modify the time line appropriately is the following:

- When the AI Planner submits an order to the Spring scheduler, if the order is schedulable, the Spring scheduler replies with the *sst* and *sft* of the order.

- When the AI Planner is uncertain about the status of the low level machinery, it can ask the Spring scheduler for such information. The Spring scheduler replies with a list of the *sst* and *sft* of all of the orders currently existing in the Spring schedule. In addition, the AI Planner can specifically request the *sst* and *sft* of an order.

- The AI Planner should be notified when an order completes, which the Spring scheduler does by scheduling and executing the *End of Order* process group for each scheduled order.

All of the communication discussed thus far is at the order level, and is sufficient to attain a reasonable level of competency in the AI Planner. However, in cases in which the AI Planner's internal, projected schedule does not match the reality as created by the Spring scheduler, we have found that order-level communication is not sufficient to achieve the goals of the AI Planner. In particular, when the AI Planner submits an order that the Spring scheduler cannot schedule due to resource constraints with other currently executing orders, we had originally intended to reply with the ID of the order that was preventing the new order's access to some needed resource. For example, if a three-ball order were submitted with a particular deadline, and the Spring scheduler could not schedule it because slots on the linear table were not currently free and would not become available until after its deadline had passed, the Spring scheduler was going to reply with the ID of the order that was using the linear table slots immediately prior to the time that this order would need them. Communication at this level fails to allow the AI Planner to achieve one of its goals, which is to make the most appropriate decision in the given context of scheduling failure. The problem is that the AI Planner does not know *how* the order is preventing the scheduling of the new order. The AI Planner does not know, in the example above, if a tube is not available, in which case the AI Planner should not resubmit *any* order as a response to the scheduling failure, or if, as may be true in this case, a three-ball order is not schedulable because there are not three slots available, but an equivalent two-ball order *is* schedulable. In this case, if Spring had responded that the linear table slot availability were the problem, the AI Planner might be better equipped to decide that it might be appropriate to submit an order with fewer than three balls.

In this situation, another way in which the AI Planner can make this scheduling decision is through a resource-level status report. In the above example, if the AI Planner directly queried the status of resources at a particular time, and for a particular duration into the future, it could ask the status of resources immediately preceding the deadline of the order in question. The response in this case would be that there is not three slots available, but only two. The AI Planner would then directly be able to determine that submitting a two-ball order might be appropriate.

At the present time, the AI Planner does not expect communication on the resource level, nor does it have an understanding of Spring-level resources. In future research, we will change the AI Planner to use this information. In addition, in the near future, the scope and functionality of the AI Planner will be expanded to include the following [1]:

Better planning time allocation. In addition to planning the execution of orders on the manufacturing system, the high level must realize that, because the system and environment are constantly changing, it does not have unlimited time in which to plan. That is, the AI Planner must be changed to be more cognizant of when it can *think*, and when it must *act*. Too much planning will preclude the system from actually servicing *any* orders. We intend to test these ideas by stressing the high-level system by overloading it with orders. In that situation, the AI Planner must decide if it has time to perform potentially complicated reasoning to decide among the orders, or if it must use simple and quick procedures to decide which orders to submit.

Predicting future events. In many situations, the model of the environment can not be adequately predicted. However, in more deterministic domains such as some characterizations of flexible manufacturing, the expectation of future events can be used to manage the scheduling of activities on the system. For example, the anticipation of the arrival of raw components from other factories can be used to plan the future servicing of current orders. Even if the arrival time, and the arrival at all, is uncertain, it may allow the AI Planner to postpone some job if its deadline and importance permits. The way we intend to test a scheduler that predicts future events is to make the arrival of balls (and orders) more deterministic. The graphical user interface already has the capability of having balls arrive according to well-behaved distributions. The AI Planner must be modified to use knowledge, perhaps uncertain, of these distributions.

Better dealing with low level scheduling failure. When the AI Planner internally builds a tentative schedule it intends to pursue, it is possible that the plan can not be executed. This may happen for many reasons, which include an inaccurate model of the state of the system, an inaccurate view of the environment, and simple failure of one of the tasks that comprise one of the process groups of one of the orders in the schedule. For example, the currently-scheduled use on the robotic arm may cause an order that the AI Planner has submitted to be not schedulable. The high level scheduler must be able to anticipate and react quickly from failure of this type, either by resubmitting the order with a modified deadline or by dropping the order altogether. The way we propose to test an AI Planner that better predicts and handles low level scheduling failure is to give the AI Planner a more uncertain view of the world, and evaluate its actions. In addition, the low level scheduler might be temporarily made to generate more scheduling failures, in order to test the AI Planner's reactions. In general, the AI Planner should function at a reasonable level of competence, independent from the competency of the low-level scheduler.

4 Low Level Scheduling in the FMT

Because the Spring scheduler has been described in detail in other papers, only relevant features will be presented here. The interested reader should consult [13, 14, 10] for more details. After the Spring kernel has been reviewed, modifications to the Spring scheduler that are necessary for multilevel scheduling will be presented.

The Spring kernel is a real-time reflective operating system. A key component of the kernel is the Spring scheduler, which is a planning-based scheduler that dynamically generates schedules in which every task included in the schedule is guaranteed its required resources (including a processor) for its WCET. When a new set of tasks arrive at the scheduler, the Spring scheduler attempts to assign a starting time and a processor for the new tasks and every task in its current schedule such that every task completes by its deadline and there are no resource conflicts between any tasks scheduled to execute at the same time. If such a schedule cannot be found, the system rejects the new set of tasks and reverts to its previous schedule, in which all tasks are guaranteed.

Spring recognizes that tasks often take less than their WCET, and provides a general-purpose mechanism for reclaiming and reallocating resources when they do [12]. However, in some applications, instead of using this mechanism that reclaims by essentially starting the execution of tasks in the current schedule earlier than their scheduled start times, the SP can be configured to reschedule after every task finishes or some external interrupt is generated. We have selected to do this in the FMT. This does not have a negative impact on the performance of the SP in the FMT, primarily because of the ratio it takes the Spring scheduler to scheduler compared to the length of time of an RPI, which are the most time-consuming activities of the FMT. Because the RPIs are relatively long, the SP can reschedule without becoming overworked.

Because the robotic processes on the average require a WCET of about 15 seconds, interesting low-level scheduling problems arose that had not previously been considered. Previously, because tasks were assumed to take on the order of milliseconds or tenths of seconds, the entire Spring schedule was generally not more than 10 seconds into the future. In the FMT, a single three-ball order submitted to an idle system causes the generation of a schedule that is projected to take 166 seconds in its worst case![1] A schedule of this duration showed that the following implicit assumption is flawed:

- Real-time tasks that need to interact with the environment at a precise time can be scheduled "properly" no matter how long into the future they need to be executed.

The way this assumption caused errors was the manner in which we had hoped to schedule the *Flap* process. When this process executes, it waits until the bin number that it is supposed to flap rotates in front of the flapper, at which point it flaps the ball onto the linear table. At worst, this process could take the length of time of an entire revolution of the rotating table, which is around 15 seconds, but at best it could take about 2 seconds, if the bin that the flapper needs is right in front of it at the time that it is invoked. We intended to schedule the *Flap* processes (one per ball) in such a way so as to minimize the total amount of time waiting for the bins to rotate in front of the flapper. This would produce the most efficient manufacturing system. Predicated on doing this is the ability to accurately predict which bin would be in front of the flapper at the point the *Flap* process was scheduled to execute. Upon scheduling a *Flap* process, we would select that bin of the set of bins that we needed to flap that would cause the minimum amount of waiting time.

There were two reasons why this policy could not be implemented. First, the speed at which the table rotates is a function of the weighting (i.e. the number of balls) on the table. Therefore, assuming that balls arrive dynamically, the prediction of which bin of the rotating table would be in front of the flapper at a particular time in the future can not be made because there is no way to correctly predict the speed of the rotating table from the current time until the ball is flapped. Second, even if an accurate prediction can be made in relation to the speed of the rotating table, the time at which the *Flap* process is scheduled to execute is only correct if every task before it in the schedule takes precisely its WCET. In the FMT, a robotics task rarely takes its WCET. A way to approach this is to recompute upon every rescheduling which bin will be in front of the flapper, but the added complexity to do this was judged to be not worth the effort. Of course, we could have not rescheduled after each task completes (which implies that the *Flap* process will always execute at the time that it is scheduled), but that would severely cripple performance. The WCET of the *Flap* process is currently assumed to be the entire rotation of a fully-loaded table.

Adding a high level scheduler that required feedback from the Spring scheduler made us reevaluate and redesign certain operations of the Spring scheduler. Previously, the Spring scheduler did not provide feedback concerning schedulability, because no process that spawns real-time tasks could use such information. The Spring scheduler was changed in four ways to provide such information. These changes were touched upon in the previous section about the AI Planner, and are repeated here from a different perspective:

1. Spring provides the scheduled start time (*sst*) and the scheduled finish time (*sft*) of an order, when the order can be scheduled.

2. Spring provides the *sst* and the *sft* of an order, as response to a direct query from the AI Planner.

3. Spring provides the status of the system when queried. While the simple version that returns the *sst* and *sft* of each order in the system is still operational, the Spring scheduler can also respond with the projected resource utilizations for a particular time period. Spring returns the percentage

[1]There is inevitably more parallelism available within the system which we, for debugging purposes, have chose to ignore for the moment. The focus of the research thus far has been not to create the most efficient manufacturing system but rather to create a realistic model of the manufacturing system in which to investigate research issues.

of this amount of time into the future that each resource is used. Spring includes every resources that might be of some information to the AI Planner, but ultimately will let the AI Planner decide the relevance of each resource to its concerns. An example of this is shown in Figure 6, which shows the projected amount (of the next two minutes after a certain time) that each resource has been reserved in exclusive mode.

```
At t= 14246:
Resource utilization (w=120 sec)
   R0: 0   R1: 0   R2: 1
   R3: 0   R4: 92  R5: 24
   R6: 12  R7: 49  R8: 25
   R9: 24  R10: 25 R11: 0
```

Figure 6: Example Spring Status-Of-System Message

4. When an order cannot be scheduled, Spring provides some information explaining why it cannot be scheduled. As is the case concerning the status of the system, there are two versions that are operational. The first version replies with the order or orders that are preventing the requested order from being scheduled. The second provides the resource or resources that are preventing the requested order from being scheduled. For each response type, this meant augmenting the vector that contained the next available times for each resource, which the scheduler used to build schedules, to include the id of the process group that last required the resource.

It should be noted that determining what exactly prevented an order from being schedulable is a complicated problem because of the manner in which *any* multiprocessor scheduler without ample restrictions on the environment is forced to operate. Because the multiprocessor scheduling problem is in general NP-hard, a multiprocessor scheduler must be heuristic. This means that a multiprocessor scheduler's inability to find a schedule for a particular set of tasks *may not be a product of any intrinsic properties of the tasks, but rather of the manner in which the schedule was attempted to be built.* This manifests itself in the FMT when the Spring scheduler attempts to determine why an order is not currently schedulable, given the existence of other orders in the current schedule. The Spring scheduler, or any other multiprocessor scheduler for that matter, cannot definitively state the reason why a schedule cannot be built—at most, a scheduler can provide its best estimate. An example of the information Spring must use in order to determine why an order cannot be scheduled is shown in Figure 7. In that example, a new order with an ID of 545 was given to the Spring schedule. It consisted of 27 tasks, and there were 99 tasks in the schedule when the new order arrived. When Spring attempted to build a schedule in which all 126 tasks were guaranteed their resources, after 105 of the tasks had been added to a tentative schedule, Spring failed to add a new task to the schedule such that all remaining tasks looked like they would be schedulable if it continued building the tentative schedule. Of the 105 tasks, 95 were from the already-existing schedule, and 10 were part of the new order. While "resource 9" or AP2 appears to be culpable, the intrinsic lack of available time on resource 9 or AP2 cannot be proven. This complicates the AI Planner, because certain responses from a low level scheduler that uses multiple processors must include some uncertainty. Reasonable approaches for the AI Planner to deal with this uncertainty are currently being investigated.

```
Failure at time 17455
---------------------
The new event id is: 545
95 of 99 old tasks were scheduled.
10 of 27 new tasks were scheduled.

Failure task:
task spring_D T3 (3913)
        event id: 544
        pg id: 3913
        sst: 35183
        sft: 35186
        event deadline: 36217
        deadline: 36217
        processor resource number: 2
        resource table
          resource 9:  access code: 1
```

Figure 7: Example Spring Scheduler Trace

5 Communication Issues between the Schedulers in the FMT

In the FMT, the effectiveness of the communication between the schedulers is primarily a function of what the AI Planner does with the information returned from the Spring scheduler. Because the design and implementation of the AI Planner is evolving, efforts to establish the effectiveness of the communication are only preliminary. The general research issues concerning communication are: what information should be passed, how often should the information be passed, and what is the system-wide performance as a function of the communication policy between the two schedulers. These questions have been reformulated to be specifically addressed in the FMT:

1. How much of the communication should be at the resource level, and how much should be at the order level? Are there any situations in which order level status information is more effective than resource-level status information?

It has been established that information concerning the inability of the low level scheduler to schedule an order from the AI Planner should generate communication on the resource level, as discussed in Section 3.

2. How often should status messages be generated by the low level scheduler? In what circumstances?

3. How much into the future should the status messages project?

4. Should the status information of the system as a whole be described in terms other than percent utilization of its resources? Would some description of time dependency of utilization be more informative? Is it worth the cost to the low level scheduler to produce this information?

If the Spring scheduler communicates with the AI Planner only as a response to a direct query, the AI Planner might base decisions on outdated information. This is because the environment is not static, and the Spring schedule is constantly changing, because the Spring scheduler reschedules when tasks take less than their WCET. Because of this, the Spring scheduler could send periodic update messages to the AI Planner, indicating the state of the low level processing. The difficulty of this approach is determining how often these periodic messages should be sent, and how to represent the status of the system. One approach is to describe, with certainty, a history of the most recent states of the low level system. Another approach is to describe the projected near-future state of the system. This, of course, is only the anticipated state and is subject to change. In both approaches, the size of the time window must be defined. If the window is small, the frequency of messages must increase in order to maintain within the AI Planner a view of the state of the manufacturing system. The frequency does not have to be as high if the projected future states of the system is longer into the future. However, in this case, as the time into the future increases, the certainty of the projection decreases. In other words, the likelihood that the anticipated future state of the manufacturing system equals the actual future state decreases as the amount of time into the future increases. Deciding which of these policies is most appropriate for the FMT is a complicated process, and at which we have only begun to evaluate.

5. To what extent does the AI Planner need to understand the functionality of the Spring scheduler (and Spring node architecture, as well as the physical manufacturing resources) in order to understand its messages?

This is perhaps the most complicated issue involving the communication between the two schedulers in the FMT. It is hoped that the two schedulers can be built independently, and interact only through a well-defined interface. However, the AI Planner is already beginning to require knowledge of the manufacturing processes in order to function effectively. For example, the AI Planner must understand the availability of slots in the linear table in order to schedule opportunistically. To what extent the AI Planner needs to understand the functionality of the Spring scheduler remains an open issue.

6 Related Work

The AI Planner is based in part on the design-to-time real-time scheduling policies [3, 4], which are applicable in environments in which complex task interrelationships exist and can be exploited by the scheduling process. An example of a relationship between two tasks is the *facilitates* relationship, which states that, if task a facilitates task b, then executing task a *before* task b causes task b to take less time than if task b were executed before, or concurrently, with task a. Exploiting this type of information in a heuristic approach allows the scheduler to generate more effective schedules than if this information were not utilized. Ultimately, the goal is to modify the environmental characteristics of the FMT so that these principles can be studied further.

The two-level scheduling hierarchy is also used in the CIRCA architecture [8, 9]. In that architecture, the low level real-time system (RTS) provides the guarantees necessary for real-time computing, while the high level AI system (AIS) reasons in a more opportunistic, and less predictable, manner. the FMT is similar to CIRCA in that each use two scheduling subsystems that attempt to provide different functionality to the overall system. System execution is achieved through the cooperation and negotiation of the two scheduling subsystems. Both the FMT and CIRCA are interested in using the interaction of the two systems to enhance system responsiveness and performance guarantees. However, there are many key differences between CIRCA and the FMT. While in the FMT the knowledge representation scheme and the objects reasoned about in the two subsystems are *different*, the objects and representation in the two levels of CIRCA are essentially the same. In CIRCA, a goal–an activity to be achieved–can be placed in either of the two subsystems if the application designer so chooses. Another key difference is the actions that the low level system executes. In Spring, these actions (tasks) are slightly-restricted C functions that are fed through the Spring-C compiler in order to determine a WCET. In addition, tasks have precedence constraints, resource requirements, and worst-case execution times. In CIRCA, the RTS executes simple test-action pairs (TAPs) that have known worst-case execution times. The test-action pairs are independent from each other. A third difference is the interaction between the higher level and the lower level. In the FMT, the exchange is through a defined discourse language, because neither system has the ability to reason about objects in the other's schema directly. In CIRCA, the basic exchange is started when the AIS suggests to the RTS a set of test-action pairs to execute. If the RTS cannot create a schedule of these test-action pairs, the RTS tells the AIS, and the AIS suggests a new set of test-action pairs. Finally, the way each system reacts to a task taking less than its worst case duration is different. In Spring, rescheduling or resource reclaiming algorithms adjust the start time of tasks that already exist in the current schedule. That is, no new tasks are introduced as a result of a task taking less than its worst-case execution time. In CIRCA, the AIS provides the RTS with a list of "unguaranteed" test-action pairs to execute in the event that test-action pairs take less than their worst case execution times. Because test-action pairs are independent, this type of scheduling policy will enhance system performance in general without effecting the guarantees provided to test-action pairs in the existing schedule.

The MARUTI real-time operating system has been used as a basis on which to build AI systems that integrate planning and reaction [6]. The goals of that work are similar to the goals of the two-level scheduling work of the FMT–that is, to schedule at multiple levels of abstraction in order to provide responsiveness while maintaining guarantees. However, in that work, low level monitoring and high-level planning do not exist on separate systems. In addition, both the high level and low level reason about the same basic objects, so they don't have to consider lack of language by which to communicate. The question of direct feedback when some request by the AI can not be fulfilled by the real-time system is not addressed. Also, MARUTI assumes a single processor architecture, which greatly reduces the complexity of the issues.

The general issues of deliberating versus acting are discussed in [5]. They note that neither thinking ahead nor acting at the last moment should be pursued to the exclusion of the other. The high level scheduler must be aware of this issue when it submits process groups to the low level scheduler. If the process groups are submitted too far in advance, the state of the world could change significantly by the time the first task of the process group begins executing. At worst, executing a process group might not be important anymore. On the other hand, the high level scheduler cannot wait until the last moment to submit a process group, because resource utilization by the low level scheduler will inevitably suffer.

An approach that allows the high level scheduler to reason about the low level capabilities is proposed in [7]. That work proposes a representational framework for describing that low level primitives, specifically what they accomplish when they execute. This type of framework might be used to formalize the analysis of the high level scheduler of the dynamic capability of the low level scheduler.

7 Conclusions

While the evaluation of the two-level scheduling paradigm in the FMT is preliminary, there are many important contributions of this work. Primarily, a real manufacturing system, albeit a simplified version, has been implemented and directly tests the assumptions upon which much of our previous work has been formulated. New scheduling problems within the Spring scheduler have arisen. The FMT represents an operational attempt at recognizing that scheduling in certain applications must exist at two levels, and presents a testbed in which to test theories of how to effectively schedule the system as a whole. We have presented a partitioning of the overall scheduling in the flexible manufacturing domain to two levels. We have discussed our initial attempts at passing information between the two schedulers, and have begun to quantify the performance of each scheduler as a result of the information it receives from the other.

Far-term research in multi-level scheduling in the FMT includes designing, implementing, and evaluating some of the following:

- The high level scheduler can exercise more meta-level control on the low level scheduler [11].

For example, in some situations, the high level scheduler can instruct the low level scheduler that it is allowed more time in which to search for a schedule, for whatever reason the high level scheduler decides. Another paradigm for meta-level control is to make the interactions more negotiation-based [2].

- The high level scheduler can ask hypothetical "what if" questions to the low level scheduler in order to better anticipate scheduling failures. The low level scheduler would only answer these questions if it would not jeopardize the low level scheduler's more important concern, which is to actually schedule real tasks.

- Machine learning techniques can be used to have the high level scheduler learn from its scheduling failures and successes. Presumably, the high level scheduler can learn to avoid certain context-specific failures. The high level scheduler has the opportunity to do this because in general it does not have the strict timing requirements of the low level scheduler.

- The high level scheduler can plan for the *average* case performance of activities. This is in contrast to the low level scheduler, that has to guarantee tasks based on the worst case execution time. Average case performance allows the high level scheduler to more predictably anticipate the general requirements of a process group, which allows less rescheduling when tasks take less than their WCET. Of course the high level scheduler must be aware that sometimes tasks *do* take their WCET, and must plan accordingly.

Multi-level scheduling provides a real-time system with the ability to be more adaptive and responsive, while providing the necessary guarantees that many applications require. Continued research with the FMT will allow us to design and test more sophisticated scheduling paradigms.

Acknowledgments

The development of ideas concerning the high level scheduler greatly benefited from discussions with Rob St. Amant. Gary Wallace and Mike Pasieka provided additional system support needed to implement the two-level scheduler system of the FMT. Elizeth Araujo and Manfred Huber provided the graphical interface and robotics and vision routines for the FMT. We also thank Krithi Ramamritham for his helpful comments.

References

[1] Robert St. Amant. Planner interaction with a real-time scheduler in a flexible manufacturing environment. To appear, 1994.

[2] Alan Garvey, Keith Decker, and Victor Lesser. A negotiation-based interface between a real-time scheduler and a decision-maker. Computer Science Technical Report 94–08, University of Massachusetts, 1994.

[3] Alan Garvey, Marty Humphrey, and Victor Lesser. Task interdependencies in design-to-time real-time scheduling. In *Proceedings of the Eleventh National Conference on Artificial Intelligence*, Washington, D.C., July 1993.

[4] Alan Garvey and Victor Lesser. Design-to-time real-time scheduling. *IEEE Transactions on Systems, Man and Cybernetics*, 23(6):1491–1502, 1993.

[5] Steve Hanks and R. James Firby. Issues and architectures for planning and execution. In *Proceedings of the Workshop on Innovative Approaches to Planning, Scheduling and Control*, pages 59–70, November 1990.

[6] James Hendler and Ashok Agrawala. Mission critical planning: AI on the Maruti real-time operating system. In *Proceedings of the Workshop on Innovative Approaches to Planning, Scheduling and Control*, pages 77–84, November 1990.

[7] Robert C. Kohout. Representing reactive competences for use in hard real-time systems. Ph.D. Dissertation Proposal, University of Maryland, College Park, 1994.

[8] David J. Musliner, Edmund H. Durfee, and Kang G. Shin. CIRCA: A cooperative intelligent real-time control architecture. *IEEE Transactions on Systems, Man and Cybernetics*, 23(6), 1993.

[9] David J. Musliner, Edmund H. Durfee, and Kang G. Shin. Integrating intelligence and real-time control into manufacturing systems. In *Working Notes of the SIGMAN Workshop on Intelligent Manufacturing Technology*, July 1993.

[10] Krithi Ramamritham, John A. Stankovic, and Perng-Fei Shiah. Efficient scheduling algorithms for real-time multiprocessor systems. *IEEE Transactions on Parallel and Distributed Systems*, 1(2):184–195, April 1990.

[11] Krithi Ramamritham, John A. Stankovic, and Wei Zhao. Meta-level control in distributed real-time systems. In *International Conference on Distributed Computer Systems*, West Berlin, September 1987.

[12] Chia Shen, Krithi Ramamritham, and John A. Stankovic. Resource reclaiming in multiprocessor real-time systems. *IEEE Transactions on Parallel and Distributed Systems*, 4(4):382–398, April 1993.

[13] John A. Stankovic and Krithi Ramamritham. The Spring kernel: A new paradigm for real-time systems. *IEEE Software*, 8(3):62–72, May 1991.

[14] Wei Zhao and Krithi Ramamritham. Simple and integrated heuristic algorithms for scheduling tasks with time and resource constraints. *Journal of Systems and Software*, 7(3):195–205, September 1987.