Experiments with a distributed architecture for predictive scheduling and execution monitoring[*]

Claude Le Pape

ILOG S.A., 2 Avenue Galliéni, B.P. 85, 94253 Gentilly Cedex, France

An important distinction is made in the planning domain between predictive planning problems and reactive execution problems [1]. The following paper discusses the distinction in the case of manufacturing scheduling. Given a set of jobs, an ordered set of production steps for each job, and a set of machines to perform these production steps, the predictive scheduling problem consists in determining an appropriate allocation of production steps to machines over time. Given partial or complete information about the current state of the manufacturing system, the reactive execution (or dispatching) problem consists in deciding "in real-time" which production steps to execute next on each machine. The paper presents an architecture allowing to associate a predictive scheduler and a reactive dispatcher. The predictive scheduler constructs a predictive schedule and updates it occasionally in response to arising problems and opportunities (in this sense, the predictive scheduler is a bit reactive!). The reactive dispatcher decides in real-time which operations to start on the shop floor given the actual course of events. It relies on the schedule to make its decisions, but overrides it in response to arising problems and opportunities. Our implementation allows the user to tune the behavior of both the predictive scheduler and the reactive dispatcher, and therefore to compare various couplings of predictive and reactive scheduling procedures. Experimental results provide evidence that independent settings of predictive scheduling and reactive execution algorithms can result in poor performance on the shop-floor.

1. PREDICTIVE AND REACTIVE SCHEDULING

Basically, scheduling is the allocation of resources over time to perform a collection of tasks. Scheduling is a decision-making process: the process of determining a schedule. A variety of *constraints* affect this process. Scheduling decisions have to satisfy *pure constraints*: release dates, operation durations and precedences, transfer and set-up times, resource availability constraints (shifts, down time), and sharing of resources. These *restrictions* define the space of admissible solutions. Furthermore, *relaxable constraints* characterize the quality of scheduling decisions. These *preferences* are related to due dates, productivity, frequency of tool changes, inventory levels, shop stability. They lead to priority relations between manufacturing orders, alternative production routings, manufacturing operations, and alternative resources. Since preference constraints may conflict with one another, the scheduling problem also consists in deciding which preferences should be satisfied and to what extent others should be relaxed.

In the context of manufacturing, two problems can be distinguished. *Predictive scheduling* consists in building a schedule to be executed in the future. For example, given the state of the shop in the evening, it may be worth selecting and scheduling operations to be performed the next day. *Real-time reactive scheduling* consists in making scheduling decisions in real-time with respect to the actual state of the shop. This does not mean every decision is made in real-

[*] In: Roger Kerr and Elizabeth Szelke (editors), “Artificial Intelligence in Reactive Scheduling”, Chapman & Hall, 1995.

time, but previous decisions are confronted with unexpected events (e.g., machine breakdown, operation tardiness, reworking). Obviously, a reactive scheduling system is all the more useful as the shop floor is a dynamic environment where unforeseeable events occur. A predictive scheduling system is not absolutely necessary, but its usefulness may be significant: because it is not subjected to real-time constraints, computational time can be spent satisfying preference constraints and ensuring the global quality of the solution.

The aim of this paper is to present an architecture which allows a predictive scheduler and a reactive dispatcher to run in parallel and deal with environmental uncertainty in a consistent fashion. Section 2 presents the overall system architecture and the communication protocols that guarantee the correctness of the overall system, assuming correctness of its parts. Section 3 presents an implementation of the architecture, based on previous work on the SONIA predictive and reactive scheduling system [2-4]. Section 4 presents experimental results and the most important conclusion of these experiments: the fact that independent settings of predictive scheduling and reactive execution algorithms can result in poor performance on the shop-floor.

2. OVERALL SYSTEM ARCHITECTURE

2.1. A distributable representation of a schedule

The overall system is an implementation of ideas developed by Smith, Keng and Kempf at Carnegie-Mellon University and Intel Corporation [5]. It consists of three "agents" operating simultaneously and exchanging messages in an asynchronous fashion:

- The *scheduling* agent generates a predictive schedule and updates it *occasionally* in response to arising problems and opportunities.
- The *execution monitoring* or *dispatching* agent decides in real-time which operations to start on the shop floor given the actual course of events. It relies on the schedule to make its decisions, but overrides it in response to arising problems and opportunities.
- The *interface* agent serves as a mediator between the scheduling agent and the dispatching agent. It determines which agent is allowed to make which changes in which part of the schedule.

Central to this approach is a "distributable" representation of the schedule [5]. The schedule is represented as a set of *schedule objects*. The most common type of schedule object is a production step. Preventive maintenance and repair operations constitute other types of schedule objects (see [5] for details). A production step has the basic form *(step job machine st et)*, where, for any given production step *s*, *step(s)* is the process step to be performed, *job(s)* is the job being operated on, *machine(s)* is the machine allocated for the purpose of performing this production step, *st(s)* is the scheduled start time of the production step, and *et(s)* is the scheduled end time of the production step. The first two attributes of a production step uniquely identify a particular production activity to be performed. The final three represent the decisions of the scheduling agent. These decisions are made with respect to a number of constraints (such as precedence and duration constraints) imported or derived from generic product/process models and made accessible to both the scheduling agent and the dispatching agent.

An admissible schedule is a schedule which satisfies all the constraints associated with its schedule objects. It is represented as a graph $(X\ U)$, where X is the set of schedule objects and $U = (U_J \cup U_M)$ is defined as follows:

- $(s_1\ s_2) \in U_J \Leftrightarrow s_1$ and s_2 concern the same job and s_2 is the immediate successor of s_1 for this job.

- $(s_1\ s_2) \in U_M \Leftrightarrow s_1$ and s_2 concern the same machine and s_2 is the immediate successor of s_1 for this machine.

Figure 1 presents an admissible schedule and the corresponding graph. We use the J and M marks to distinguish arcs in U_J and U_M. The JM mark is used when the same arc $(s_1\ s_2)$ belongs to both U_J and U_M.

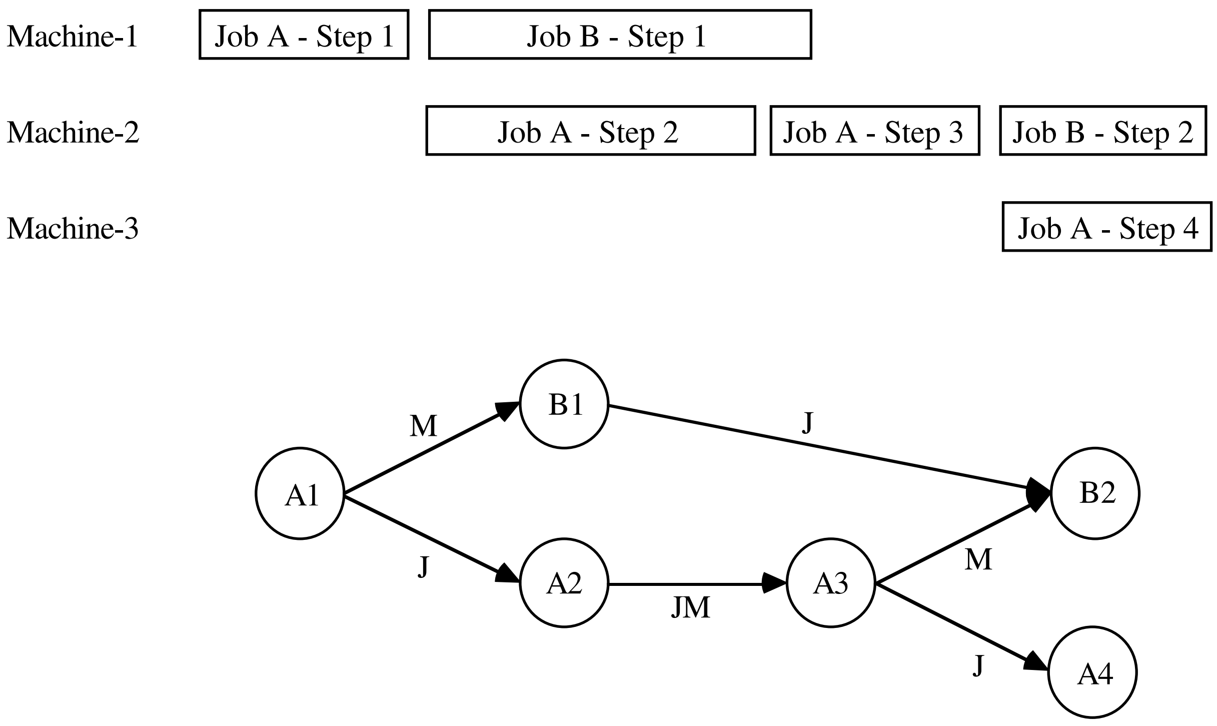

Figure 1. An admissible schedule and the corresponding graph.

To facilitate interaction between the scheduler and the dispatcher, the set of all schedule objects is divided into six subsets (as shown in Figure 2). Schedule objects are created in response to production and maintenance requests and loaded initially into subset Y. The main task of the scheduler is to move schedule objects from subset Y to subset X where schedule objects have all scheduling decision variables assigned. Within the jurisdiction of the dispatcher, there are three subsets of interest. Subset A contains all the schedule objects that are currently executing. Subsets B and C contain the "close to execute" (a notion to be discussed below) according to the schedule: the principal task of the dispatcher is to move schedule objects from subsets B and C to subset A. The schedule objects in C constitute the *frontier* between the jurisdiction of the scheduler and the jurisdiction of the dispatcher. The scheduler cannot touch them and the dispatcher must start them on time. The schedule objects in B are the focus of attention of the dispatcher. The dispatcher is allowed to override the scheduling decision variables in all possible ways provided that it respects the scheduling constraints and the decisions made for the schedule objects in C. When the dispatcher is not able to respect these decisions (or when a schedule object fails to successfully execute), it determines which schedule objects are causing problems. These schedule objects are then returned to Y indicating the necessity of a global schedule repair.

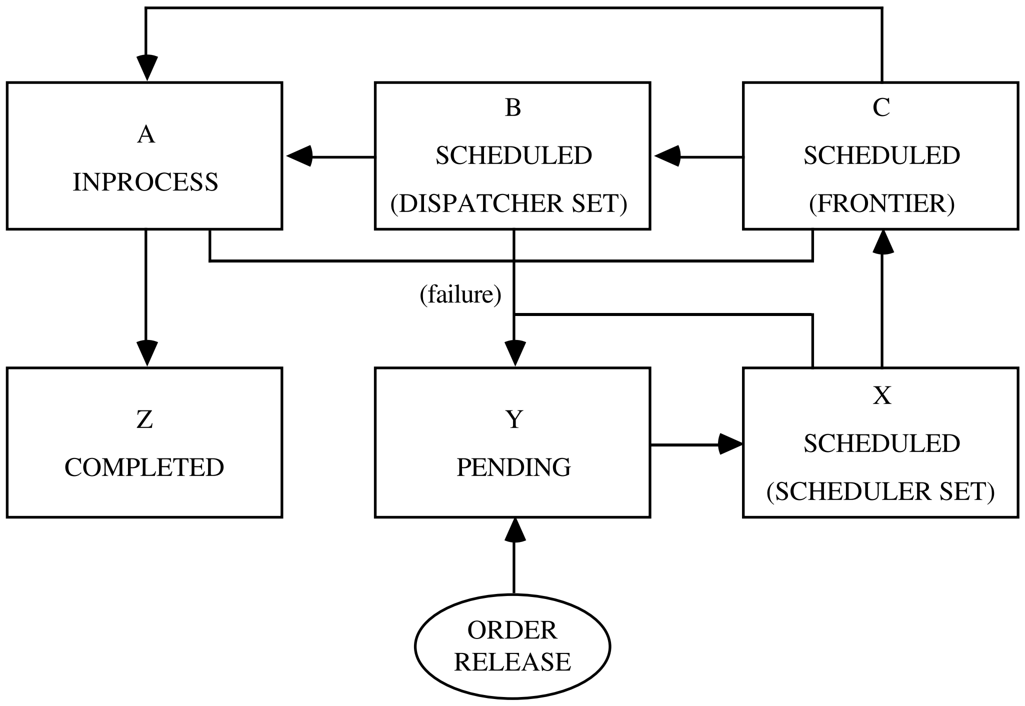

Figure 2. Sets of schedule objects.

When a schedule object finishes executing, it moves to the last subset, Z, providing a record of the actual behavior of the manufacturing system (as a schedule object *s* moves from *scheduled* to *inprocess* to *completed*, *st(s)* and *et(s)* are updated to reflect the actual times). The dispatcher moves schedule objects from subsets B and C to subset A and requests new schedule objects in replacement. Figures 3 to 5 illustrate this process in a simple case. In this example, we assume that the dispatcher is set to control (a) the executing schedule objects, (b) their immediate successors in the graph, and (c) the immediate successors of the immediate successors of the executing schedule objects. Figure 3 shows the initial situation. *C1* (the first production step of job *C*), *B2* (the second production step of job *B*), and *A3* (the third production step of job *A*) are *inprocess*. When *C1* completes, *D1* starts, as shown on figure 4. *D2* and *E1* become the immediate successors of an executing schedule object, so the dispatcher requires successors for them and obtains *D3*, *E2* and *F1*.

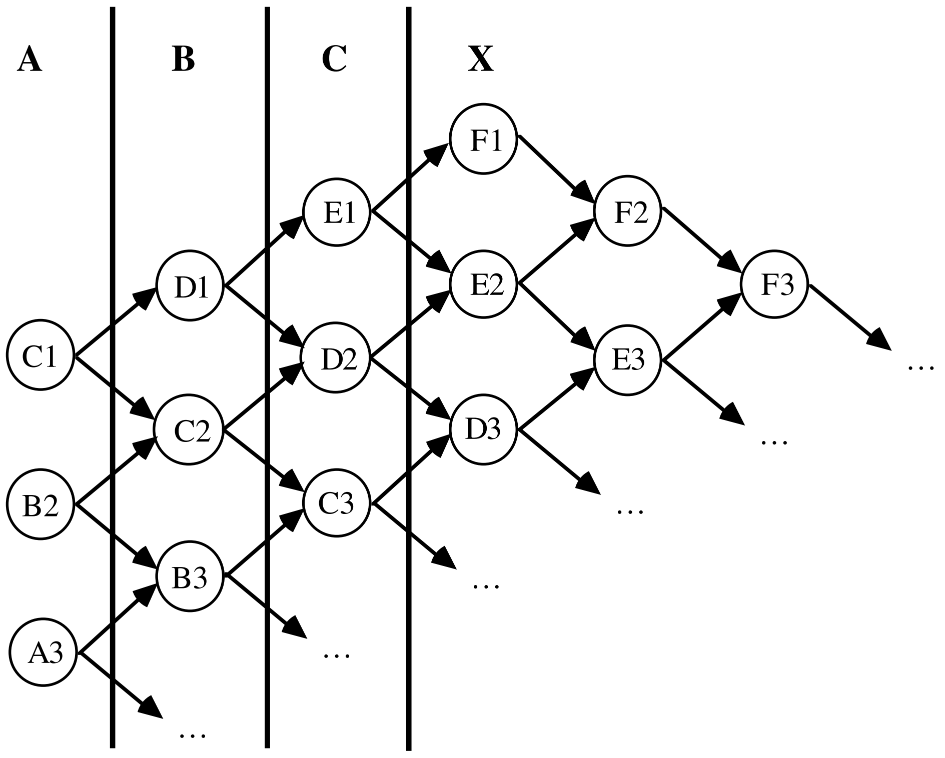

Figure 3. An initial situation.

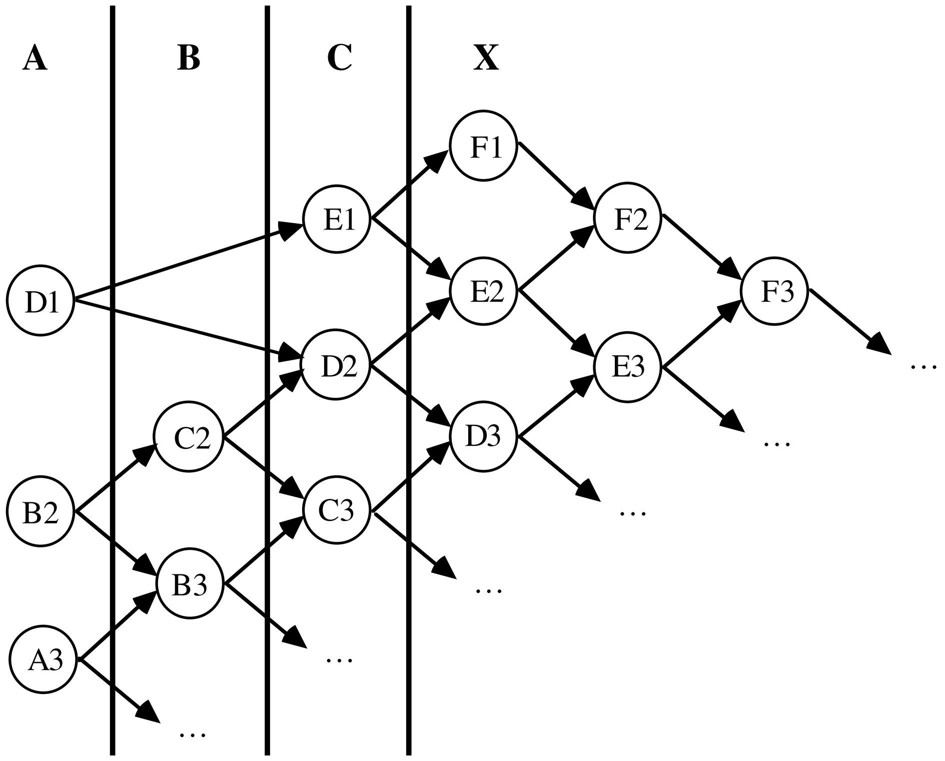

Figure 4. An execution decision.

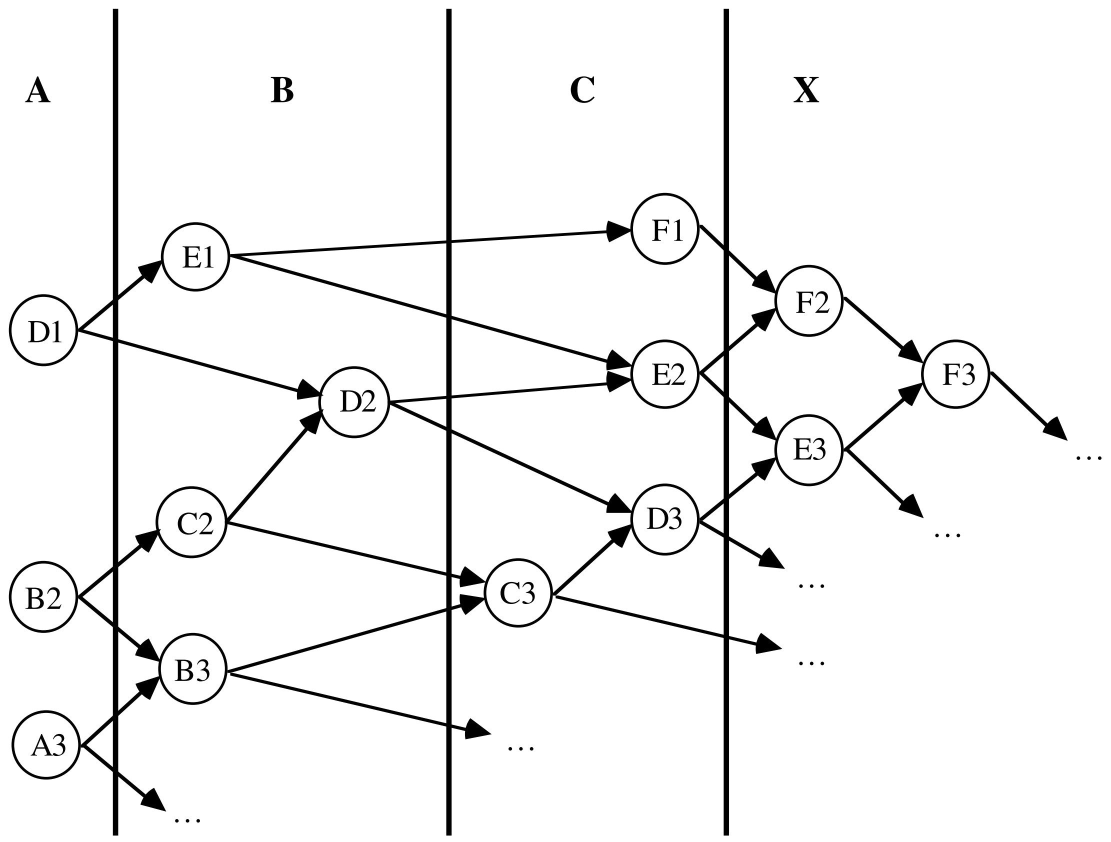

Figure 5. The transmission of new schedule objects.

Figure 3 to 5 correspond to the simplest possible case that can occur. In practice, three problems are encountered:

- The structure of the graph is not that simple. The dispatcher must therefore determine the jobs and the machines for which to require schedule objects. When the dispatcher is set to control n levels of schedule objects (including the *inprocess* level), the number of requests following an execution decision can vary between *0* and 2^{n-1}. In addition, the interface agent must manage to communicate the corresponding schedule objects in an order compatible with the schedule. Figure 6 illustrates this problem. Even though *C5* is now *inprocess*, transmitting *D5* before its predecessors *D2* to *D4* does not make sense. The interface agent must choose to transmit either *D5 with its predecessors* or nothing.

- When the scheduling agent is making changes to the schedule, the interface agent must make sure that it will not transmit a production step under modification. This means the interface agent must know what the scheduler is doing.

- When the dispatching agent is returning schedule objects after a failure, the interface agent must make sure that it will not provide successors of these schedule objects to the dispatching agent. This means the interface agent must know when the dispatcher is about to return schedule objects.

Three precise protocols, described in the following sections, are used to solve these problems.

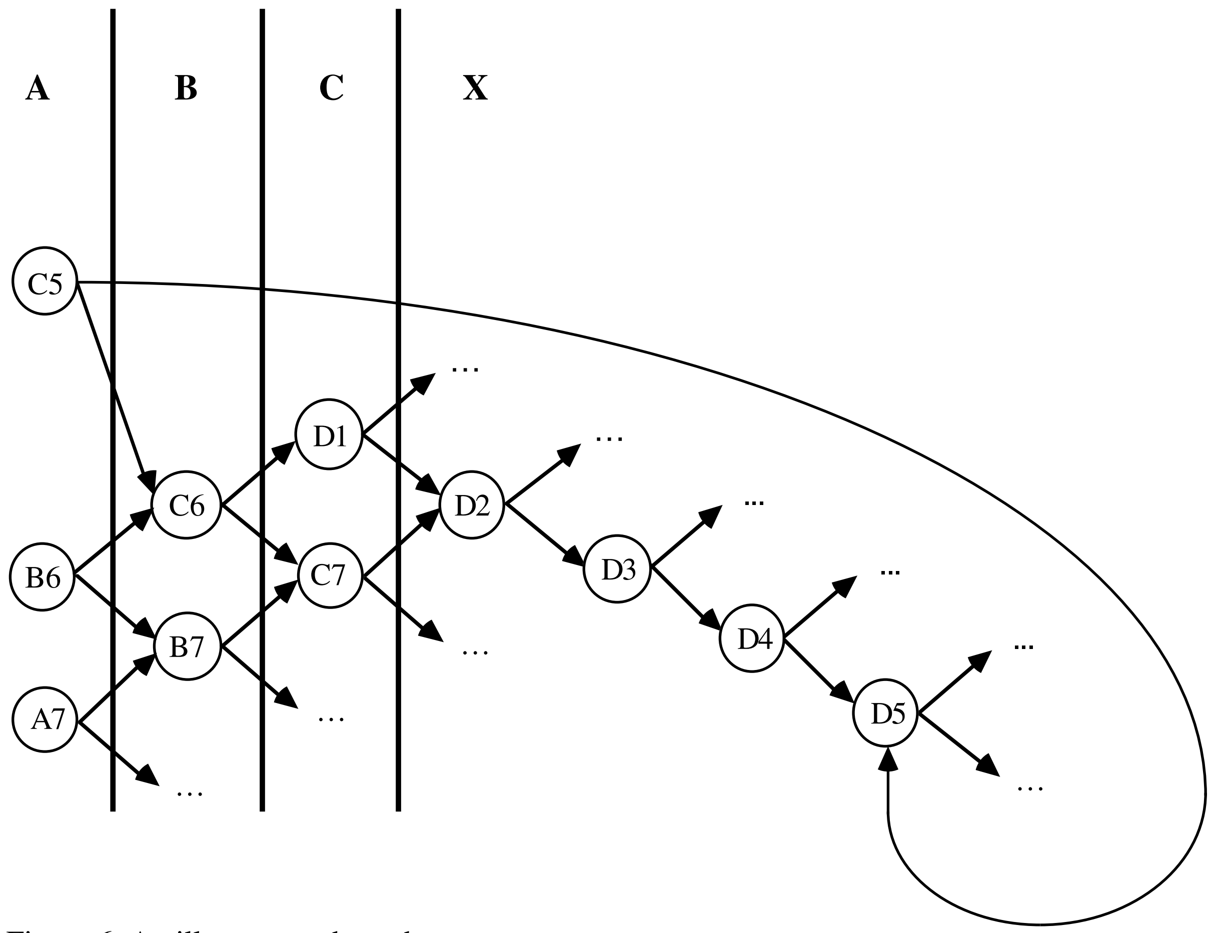

Figure 6. An ill-structured graph.

2.2. The scheduler-interface protocol

The *scheduler-interface* protocol applies when the scheduler wants to update the schedule. It consists of four phases:

- The scheduler sends a request to protect a portion P of the schedule (a subset of X) that it wants to update.
- The interface processes the request. It decides to protect a subset S of P. A message describing S is sent to the scheduler. Then the scheduler knows that the interface will not provide members of S to the dispatcher before the schedule revision is done.
- The scheduler revises the schedule. It sends messages to the interface when it can release schedule objects in S.
- When it receives release messages, the interface agent notes that the corresponding schedule objects are no longer under protection and determines whether pending requests from the dispatcher are now processible.

In its principles, the protocol allows all kinds of subset descriptions for P and S. But in practice there is a tradeoff between the generality of the subset description language and the speed of the subset protection mechanisms. A simple implementation assumes that the scheduler is able to update the schedule in a reasonable amount of time. When this is the case, the interface agent can afford to protect the complete schedule while the scheduler updates it. There is in fact no subset description to manipulate: the scheduler requires the complete schedule X and gets it.

(Note that X can change between the time of the request and the response of the interface, because the interface can provide schedule objects to the dispatcher in the meantime. However, "X" always denotes X.)

2.3. The dispatcher-interface protocol

The *dispatcher-interface* protocol applies when the dispatcher needs a schedule object for a machine or a job. It consists of four phases:

- The dispatcher searches its part of the graph to determine how its decisions to start schedule objects on the shop-floor call for new schedule objects. A simple breadth-first search algorithm is sufficient for this purpose. Then the dispatcher sends its requests to the interface. Each request specifies either a job or a machine for which a schedule object is needed.

- In the second phase, the interface agent processes requests and sends schedule objects to the dispatcher. This is the most complex phase as several cases can occur. Let us consider the different cases for a machine request:

    - (a) If the interface agent has sent a schedule object for this machine in the past (the schedule object can have been sent in response to a job request) and did not receive acknowledgement of receipt (see the next phase), it knows that the dispatcher is going to receive (or has received in the meantime) a schedule object for this machine. The interface can then ignore the new request.

    - (b) If the machine is down or (c) the next schedule object on the machine according to X concerns a broken job or (d) the next schedule object on the machine according to X is protected or (e) there is no next schedule object on the machine according to X, then the request is not processible. It will become processible when the scheduler updates the schedule.

    - (f) If none of the previous conditions applies, the interface must determine whether the next schedule object on the machine has predecessors in X and decide (using a heuristic) either to send the schedule object with all its predecessors or to keep the request pending. If the decision is to keep the request pending, the interface must record the reason for the failure in order to reconsider the request when some predecessors are sent to the dispatcher. If the decision is to send the schedule object with its predecessors, the interface must record the decision (to allow the consideration of case (a) for future requests) and determine whether it enables the reconsideration of pending requests.

  The different cases for a job request are symmetric to the above, except for case (e): if there is no next schedule object for a job according to X, but there is one according to Y, then the job request can become processible when the scheduler updates the schedule; if there is no next schedule object for a job according to X and Y, then the job is done and the interface can ignore the request.

- When the dispatcher receives new schedule objects, it incorporates them into its schedule and sends an acknowledgement to the interface. In some cases, the incorporation can call for new requests, sent right after the acknowledgement.

- When the interface receives the acknowledgement, it takes note that the dispatcher has received the schedule objects. The fact that the dispatcher has integrated the schedule objects in its schedule modifies the conditions in which case (a) applies.

An important remark is that this protocol supposes that the interface receives and processes the messages of the dispatcher in the order in which the dispatcher sends them.

2.4. The cleaning protocol

The *cleaning* protocol applies when the dispatcher is not able to respect the decisions made for the schedule objects in C or when a schedule object fails to successfully execute. It consists of six phases:

- The dispatcher sends a message to the interface agent mentioning the need for cleaning its schedule. From this point, the dispatcher stops requesting new schedule objects. It will resume requesting schedule objects at the end of the third phase.

- The interface agent receives the message and sends an acknowledgement of receipt to the dispatcher. From this point, the interface agent stops processing requests from the dispatcher. Assuming the interface processes the messages of the dispatcher in the order in which these messages are sent, all the requests have been considered at least once. The interface agent will resume processing these requests at the end of the sixth phase.

- The dispatcher decides which schedule objects to remove from its schedule (possibly using a heuristic) and sends them to the interface agent, together with new requests reflecting the changes made in the B and C sets. From this point, the dispatcher resumes its normal activities: it will request new schedule objects when needed.

- The interface agent forwards the schedule objects to the scheduler.

- The scheduler removes these schedule objects — and their successors with respect to the job links — from its schedule (these schedule objects go back to subset Y) and sends an acknowledgement to the interface agent.

- When the interface agent receives the acknowledgement, it knows that the schedule is correct again and resumes processing the requests emanating from the dispatcher. It also has to process requests emanating from the scheduler as the scheduler wants to re-schedule the schedule objects in Y. Being back to a normal situation, the interface agent is responsible for the arbitration between the two other agents.

3. A TESTBED FOR THE EVALUATION OF SCHEDULE ROBUSTNESS

It is important to notice that the three protocols above do not refer or depend upon the scheduling techniques used in the scheduler and the dispatcher. This means one can replace the scheduling agent with another scheduling agent and the dispatching agent with another dispatching agent *without revising the protocols*. The current version integrates a reduced version of the SONIA scheduling system (Section 3.1) with the simplest possible dispatcher (Section 3.2). A generic simulation system, described in [6], is used to simulate the communication protocols. Simple performance and robustness measures are then defined to obtain a complete testbed for the evaluation of various configurations of the overall scheduling and execution monitoring system.

3.1. The predictive scheduler

SONIA [2-4] is a knowledge-based job-shop scheduling system designed to detect and react to inconsistencies between a schedule and the actual events on a shop floor. The system is built upon a BB1-like blackboard architecture [7]. It is provided with both predictive and reactive scheduling knowledge sources which are used to build and modify schedules. Analyzing knowledge sources can be employed to evaluate both predictive and reactive problem-solving contexts. Control knowledge sources may use analysis results to choose the most appropriate knowledge sources to execute and determine which "behavior" the scheduling knowledge sources should adopt (e.g., which heuristics they should use). A schedule management system (built on a flexible constraint propagation system which allows tuning of the trade-off between the anticipation of inconsistencies and the cost of propagation) is used to update schedule descriptions and detect inconsistencies as scheduling knowledge sources make decisions and unexpected events happen on the shop floor.

Within SONIA, a shop schedule is represented as a set of resources, manufacturing orders and operations to which various kinds of constraints are attached. As in the ISIS [8] and OPIS [9-11] scheduling systems, resources are described at various levels of abstraction. At each level, time-tables composed of reservation constraints are associated with resources. To each manufacturing order are associated a release date, a due date, and a production plan represented as a hierarchy of operations. An *actual-status* and a *schedule-status* are defined for each operation: they indicate whether the operation is *completed*, *in-process* or *ignored* (i.e., not started) on the shop floor and whether it is *scheduled*, *selected* or *ignored* by SONIA. Temporal constraints are generated in accordance with the status information.

The schedule management system is responsible for evaluating the consequences of both (1) decisions made by the various scheduling knowledge sources (or by the user of the system) and (2) unexpected events happening on the shop floor. Decisions are related to status of operations, release dates, due dates, and resource capacity (e.g., work shifts, resource sharing). Unexpected events are machine breakdowns and delays. The schedule management system is used to create reservation constraints or temporal constraints (e.g., temporal inequalities or disjunctions of temporal inequalities) [4] according to the type of scheduling decisions or shop floor events. Constraint propagation (a deductive technique which consists in deriving new constraints from existing ones) is used in SONIA for two reasons: (1) some derived constraints, like time bound constraints, are very useful data; (2) constraint propagation enables the detection of inconsistencies between decisions and events happening on the shop floor. When an inconsistency is detected, an appropriate description of the conflict is built. According to the conflicting constraints, a conflict belongs to one or more of the following categories: the *delays* category gathers all the conflicts that result from unexpected delays (for example, a *global tardiness conflict* states that since some operations have begun or ended later than expected on the shop floor, it is impossible to perform all of the selected operations during the open work shifts); the *capacity conflicts* category gathers all the conflicts involving reservation constraints (for example, an *out-of-shift conflict* means that a resource is allocated to an operation outside of its work shifts); the *breakdowns* category is composed of *delays* and *capacity conflicts* caused by machine breakdowns.

The following knowledge sources integrate predictive and reactive scheduling in the SONIA system:

- <u>Selection</u>
  The *Selection* knowledge source is used to choose a set of operations to be performed during the open work shifts and to assign resources to them. More precisely, this knowledge source is employed when resources are under-loaded. Operations which require the use of such resources are then selected. Operations are selected according to the capacity of these resources. Heuristics are used to determine which operations should be performed. These heuristics can vary with both the shop and the problem-solving context. For example, it may be interesting to give priority to manufacturing orders for which the remaining processing time required for achievement is important compared to the remaining time made acceptable by the due date of the order. In some cases, it may be more profitable to take into account other scheduling criteria such as minimizing tool changes. When an operation is selected, its *schedule-status* is modified and the relevant constraints are created and propagated by the schedule management system.

- <u>Order Selection</u>
  The *Order Selection* knowledge source is employed to select operations which belong to the same production plan. (In the first version of SONIA [2], this knowledge source was also responsible for scheduling these operations; it was called "Order Scheduler.") It selects operations one after the other and stops as soon as an operation cannot be selected within the open work shifts. This component is often used when operations have been discarded by the *Rejection* component. This allows improvement of the schedule.

- <u>Ordering</u>
  The *Ordering* knowledge source is used to make ordering decisions when disjunctive constraints (e.g., resource sharing) must be satisfied. It consists of an iterative constraint satisfaction process: disjunctive constraints which characterize the various orders in which operations can be performed are satisfied by choosing underlying temporal inequalities. At each iteration, heuristics are used to make ordering decisions which are propagated through the schedule management system. Various backtracking procedures can be invoked when some ordering *free decisions* (as opposed to *imperative decisions*) are conflicting with other constraints. These procedures select one of the conflicting *free decisions* to be rejected; the effects of this decision are undone by the means of the schedule management system and the reverse ordering decision is made as an *imperative decision* (forced by the other conflicting decisions); then the process re-starts. The constraint satisfaction process fails when a conflict is derived from *imperative decisions*. When the *Ordering* knowledge source fails (for instance, when too many operations have been selected by the *Selection* knowledge source), some of the less important operations are rejected by the *Rejection* knowledge source.

- <u>Rejection</u>
  The *Rejection* knowledge source is responsible for rejecting selected operations. It is employed when the *Ordering* knowledge source fails or when unexpected events prevent from executing the whole predictive schedule. It uses heuristics to choose the operations to be rejected among the operations involved with conflictual situations. When an operation is rejected, its *schedule-status* is set back to *ignored*; constraints and conflicts

concerning this operation are consequently removed by the schedule management system.

- Global Re-Scheduling
  The *Global Re-Scheduling* knowledge source is used in order to process and modify the whole plan forward from the current date. Its process rests upon the fact that any deviation can eventually be expressed by precise delays associated with resources. The forward propagation of these delays results in conflicts relating to due dates or to ends of work shifts, which can be solved by relaxing due dates, extending work-shifts or rejecting operations. However, if processing time is available (i.e., if it is not necessary to correct the plan in a few seconds), it is worth reducing resources idle times by permuting operations. Consequently, two strategies are available. The first one consists of a simple right shifting strategy: for each resource, the operations to be performed remain in the same order and the schedule is moved forward from the current date. The other strategy allows permutations in order to optimize the plan and to reduce the effects of the original delay. In any case, when an operation cannot be performed within a shift, heuristics are used to determine whether the shift is extended or whether the operation is discarded or postponed until the next shift.

- Capacity Analysis
  The *Capacity Analyzer* is used to detect both bottleneck and under-loaded resources. As the shop-level capacity analyzer of the OPIS system [10], it divides the time-line into time periods and determines both the available capacity and the demand over each period for each considered resource. Capacity and demand are computed from the existing reservations and by building a rough predictive schedule (which is discarded once the analysis is achieved) for selected operations.

- Analysis of Conflicts
  The *Analyzer of Conflicts* is used to examine in details a set of conflicts in order to determine appropriate reactions. The analysis results in various proposals in order to solve all or some of the contemplated conflicts. Very simple rules are applied in order to determine which, and how, available reactive components should be used. For example, when several *global tardiness conflicts* are to be solved, it is recommended to use the *Global Re-Scheduler* knowledge source. Conversely, it is more appropriate to use the *Rejection* knowledge source in order to quickly solve an isolated global tardiness conflict. Of course the default strategy which consists in using the *Global Re-Scheduler* is always considered. In addition, heuristics can be used to focus the analysis on a particular category of conflicts or on conflicts related to a given work-area or manufacturing order.

- User Interface
  The *User Interface* allows the user of the system to build and to update schedules. More precisely, the system allows the user: to select and reject operations; to order and permute operations; to cut operations (pre-emption); to allocate a resource to an operation over an interval of time; to add and delete work shifts; to update release dates and due dates; to add and delete any constraint which can be expressed in the underlying constraint language [4]. After each decision (or modification of original data), the user can activate the constraint propagation system and visualize the results in the form of a Gantt chart.

- Schedule Application
  The *Schedule Application* knowledge source provides the user of the system (e.g., the shop floor manager) with a schedule generated or corrected by the predictive or reactive scheduling knowledge sources of SONIA. It is called each time a new (updated) schedule

is considered acceptable. The behavior of this knowledge source must be defined with respect to the execution environment under consideration. In the context of the distributed architecture described in this paper, it consists in sending messages to the interface agent, as specified in phase 3 of the scheduler-interface protocol.

- <u>Execution Controller</u>
  The *Execution Controller* informs the schedule management system about the actual course of events in the shop floor (i.e., the schedule execution with regard to unexpected events). The behavior of this knowledge source must be defined with respect to the execution environment under consideration. In the context of the distributed architecture described in this paper, it consists in processing the messages sent by the interface agent.

- <u>Control Knowledge Sources</u>
  Control knowledge sources are used to implement the basic control loop that runs the SONIA system (as in [7]) and control the behavior of the scheduling and analyzing knowledge sources [4].

In the context of the distributed architecture under consideration, SONIA is used as a predictive scheduler. Nevertheless, the reactive knowledge sources are used to avoid reconstructing schedules from scratch each time a new manufacturing order or a dispatching failure occurs. As a matter of fact, one of the main idea underlying the design of SONIA was that reactive knowledge sources could intervene in the resolution of predictive scheduling problems, and conversely [2].

3.2. The reactive dispatcher

Using the schedule management system of SONIA, the dispatcher updates the earliest and the latest start time of each schedule object under its jurisdiction each time (a) it receives new information from the shop floor or (b) it receives new schedule objects from the interface. It detects a conflict when a schedule object has its earliest start time greater than its latest start time.

The schedule management system provides the dispatcher with a description of the conflict. This allows the dispatcher to identify the schedule objects that are causing problems — and return these schedule objects, in accordance with the cleaning protocol described in Section 2.4.

3.3. Performance and robustness measures

The generic simulation system described in [6] is used to simulate the reasoning operations of the three agents. It is also used to simulate expected and unexpected events occurring on the shop floor, as well as the actions of a client posting new orders with respect to given statistical laws. From an experimental point of view, the overall system allows its user to evaluate the robustness of the schedules the scheduler can produce: the use of the cleaning mechanism indicates that the dispatcher is not able to use the schedule to make execution decisions consistent enough with the predictions of the scheduler.

Providing a formal definition for plan or schedule "robustness" is a difficult task. Informally, a plan is "robust" when the violation of the assumptions upon which it is built is of no or little consequence. One can however measure consequences of assumption violations along two dimensions: (a) in terms of the negative effect on performance metrics or (b) in terms of the effort required on the part of the prediction/execution system to recover. Robustness is "the ability to satisfy performance requirements predictably in an uncertain environment" and/or "the degree to which a plan or a schedule provides valid guidance over the range of situations that may be encountered at execution time." Despite their difference in perspective, these two definitions provide similar properties to the robustness concept:

- Robustness is an issue when (1) there are response time constraints at execution time and (2) there is unpredictability in the environment.

- A "universal" plan mapping each possible situation onto the best possible course of action in this situation is the most robust since it always provides optimal guidance. But in most domains, such a plan is not constructible: there are too many non-similar possible situations to determine in advance how to react to all of them.

- Even though robustness relates to the coverage of different execution states and/or combinations of events, coverage alone does not constitute a practical measure of robustness. There are two reasons for this: (1) different events have different probabilities of occurrence and (2) there are cases in which the negative effect on performance metrics and/or the revision effort required to recover is small even though it is not zero. From a decision-theoretic point of view, this remark suggests that the consideration of robustness is a compensation to the approximate nature of the performance metrics chosen to evaluate plans and schedules. In most cases, no one knows how to compute the expected value of the actual utility function u, so the planning system is set to optimize an explicit or an implicit function f which approximates the value of u for some probable or non-extreme scenario. Then the notion of robustness appears to denote the degree to which we can expect the execution of a similar scenario and/or the obtainment of a similar outcome.

In the following, we consider the average number of times a schedule object returns from the jurisdiction of the dispatcher to the jurisdiction of the scheduler as a measure of schedule robustness with respect to guidance. To evaluate the robustness of a schedule with respect to performance, we use two classical metrics, average tardiness, and average work-in-process time (WIP time). Let us note that, given a schedule, execution results can vary, not only with the environmental uncertainty, but also with the possibilities made available to the dispatcher. In particular, the criterion determining which schedule objects fall under the jurisdiction of the dispatcher (what is "close to execution") has considerable influence on its ability to move schedule objects in time without "breaking" the schedule.

4. EXPERIMENTAL RESULTS

Up to now, only a small series of experiments has been carried out. In this series, each experimental condition is determined by two parameters N and R. N is the number of levels of schedule objects that the dispatcher is set to control. R is a rule used to distribute job slack and machine idle time in the schedule. The *slack time* between two schedule objects s_1 and s_2 is defined whenever $(s_1\ s_2) \in U_J$ as the difference between the start time of s_2 and the end time of s_1. The *idle time* between two schedule objects s_1 and s_2 is defined whenever $(s_1\ s_2) \in U_M$ as the difference between the start time of s_2 and the end time of s_1. Two values (3 and 5) are considered for N. Four values (EB LB EA LA) are considered for R.

- The EARLIEST-BLIND (EB) rule is the one that SONIA uses as a default. SONIA schedules several time periods one after the other. The number and the duration (a few hours to a few days) of these periods are parameters set prior to run the system. The scheduling mechanism assigns a number of production steps to each resource over each period and orders the production steps which cannot execute in parallel. The earliest possible start and end times compatible with this order (and with the assignment of time periods to production steps) are provided to the dispatcher. This implies that the dispatcher has the smallest conceivable set of possibilities to update the schedule when a problem occurs on the shop-floor.

- The LATEST-BLIND (LB) rule gives more responsabilities to the dispatcher. SONIA provides the dispatcher with the latest possible start and end times compatible with the chosen order and the assignment of production steps to time periods.

- The EARLIEST-AWARE (EA) rule is similar to EB except that the scheduler maintains a "realistic" amount of idle time for each resource over each period. This amount is chosen with respect to statistics gathered on the shop-floor. The scheduler provides the earliest start and end times to the dispatcher: the dispatcher cannot use the additional idle time made available for the ongoing period but it can use the idle time made available at the end of the preceding periods (when this idle time is still available).

- The LATEST-AWARE (LA) rule is similar to LB and EA. The scheduler (a) maintains a "realistic" amount of idle time for each resource over each period and (b) provides the dispatcher with the latest possible start and end times. As a result, the dispatcher can use the additional idle time made available for a period as soon as the first production steps of the period reach the C subset.

Both LB and LA result in a pure propagation of time bound constraints from the scheduler to the dispatcher. The dispatcher can execute each schedule object at any time consistent with the current scheduling decisions. When the dispatcher fails, the scheduler also fails (and needs to revise its allocation and ordering decisions). In comparison, EB and EA result in the definition of stronger constraints in the jurisdiction of the dispatcher than in the jurisdiction of the scheduler.

Environmental uncertainty concerns the duration of schedule objects. The duration of each schedule object follows a "truncated" normal distribution. Given this distribution, a few simulations are made to determine what is a realistic proportion of resource idle time in the schedule. When R belongs to {EA LA}, the scheduler is set to maintain this proportion of idle time in each period. The resulting job slack and resource idle time is then distributed according to the chosen rule R. Figure 7 summarizes the results. For each experimental condition, the table provides five figures:

- The average tardiness in the original predictive schedule (interpreted with respect to the chosen rule R).

- The average WIP time in the original predictive schedule (interpreted with respect to the chosen rule R).

- The average tardiness resulting from the schedule execution.

- The average WIP time resulting from the schedule execution.

- The average number of times a schedule object returns from the jurisdiction of the dispatcher to the jurisdiction of the scheduler.

(N R)	Tardiness (in X)	WIP (in X)	Tardiness (in Z)	WIP (in Z)	Return Rate
(3 EB)	0	415	40	484	1.99
(3 LB)	5	456	2	413	0.54
(3 EA)	6	446	32	471	3.79
(3 LA)	12	484	0	405	0.14
(5 EB)	0	415	17	441	2.95
(5 LB)	5	456	13	430	0.70
(5 EA)	6	446	20	450	3.16
(5 LA)	12	484	14	441	1.71

Figure 7. Experiments with four slack/idle distribution rules.

The return rate is (as we could expect) much smaller for rules LB and LA than for rules EB and EA. More interesting is the fact that when N equals 3 this difference in guidance stability results in an important difference with respect to the performance metrics. The overall system is such that the pressure that EB and EA put on the dispatcher results in lots of inefficiencies at execution time.

Altogether, these results suggest that the tuning of predictive and reactive scheduling systems *so that they can run efficiently together* is a task far more complex than generally expected. Indeed, it appears clearly that the predictive schedule that looks best, under the assumption that the schedule executes as planned, can lead to the worst results at execution time, because (1) the so-called "optimal" schedule is fragile, and (2) as the dispatcher tries to keep close to the schedule, it does not consider the best possible dispatching decisions. In this respect, it it quite interesting to notice what happens when N is increased from 3 to 5: as the dispatcher gains more freedom to patch the schedule, it manages to make good decisions in the EB and EA scenarios, which result in better performance; but a reverse effect occurs in the LB and LA scenarios, i.e., the dispatcher starts making bad decisions, compared to what the predictive scheduler would have done, and this results in a degradation of performance as well as in an increased return rate (indeed, after several bad decisions, the predictive scheduler must eventually get the dispatcher out of the mess!). It is our belief that much more research work is needed to decide under which circumstances a schedule should be patched, carefully repaired, or fully reconstructed, so as to provide the best overall manufacturing performance.

ACKNOWLEDGEMENTS

This research has been funded by the Center for Integrated Facility Engineering (CIFE), the Center for Integrated Systems (CIS) and Digital Equipment Corporation (DEC), while the author was a post-doctoral fellow at Stanford University. The first version of the execution monitoring system was designed by Karl Kempf, Naiping Keng and Stephen Smith at Intel Corporation. The paper reflects intensive discussions with them and with Malcolm Acock, Mark Drummond and Srinivas Naranayan.

REFERENCES

[1] J. Hendler, A. Tate, and M. Drummond, AI Planning: Systems and Techniques, AI Magazine 11 (1990) 61-77.

[2] A. Collinot, C. Le Pape, and G. Pinoteau, SONIA: A Knowledge-Based Scheduling System, International Journal for Artificial Intelligence in Engineering 3 (1988) 86-94.

[3] A. Collinot and C. Le Pape, Adapting the Behavior of a Job-Shop Scheduling System, International Journal for Decision Support Systems 7 (1991) 341-353.

[4] C. Le Pape, Scheduling as Intelligent Control of Decision-Making and Constraint Propagation, to appear in: M. Zweben and M. Fox (editors), Intelligent Scheduling, Morgan Kaufmann, 1994.

[5] S. F. Smith, N. Keng, and K. G. Kempf, Exploiting Local Flexibility During Execution of Pre-Computed Schedules, Technical Report, Carnegie-Mellon University, 1990.

[6] C. Le Pape, Simulating Actions of Autonomous Agents, CIFE Technical Report, Stanford University, 1990.

[7] B. Hayes-Roth, A Blackboard Architecture for Control, Artificial Intelligence 26 (1985) 251-321.

[8] M. S. Fox and S. F. Smith, ISIS: a Knowledge-Based System for Factory Scheduling, Expert Systems 1 (1984) 25-49.

[9] C. Le Pape and S. F. Smith, Management of Temporal Constraints for Factory Scheduling, in: Proceedings of the Working Conference on Temporal Aspects in Information Systems (1987).

[10] S. F. Smith, P. S. Ow, C. Le Pape, B. McLaren, and N. Muscettola, Integrating Multiple Scheduling Perspectives to Generate Detailed Production Plans, in: Proceedings of the SME Conference on Artificial Intelligence in Manufacturing (1986).

[11] S. F. Smith, M. S. Fox, and P. S. Ow, Constructing and Maintaining Detailed Production Plans: Investigations into the Development of Knowledge-Based Factory Scheduling Systems, AI Magazine 7 (1986) 45-61.