On-line Planning and Scheduling for High-speed Manufacturing

Wheeler Ruml and **Minh B. Do** and **Markus P. J. Fromherz**
Palo Alto Research Center
3333 Coyote Hill Road
Palo Alto, CA 94304
ruml, minhdo, fromherz at parc.com

Abstract

We describe a real manufacturing problem that lies between job shop scheduling and temporal planning. The setting is on-line in the sense that new jobs arrive asynchronously, perhaps several per second, while plans for previous jobs are being executed. We formalize the problem as a variant of STRIPS extended with action durations and resources. We present a hybrid algorithm for this problem that combines techniques from partial-order scheduling and state-space planning. No domain-specific search control is used. Our current implementation successfully controls two prototype plants and our technology is anticipated to enable a new line of products. By integrating planning and scheduling, we enable high productivity even for complex plants.

Introduction

There is currently great interest in extending planning and scheduling techniques to handle more of the complexities found in real industrial applications. For example, PDDL (Fox and Long, 2003) has been extended to handle continuous quantities and durative actions. There are additional dimensions to planning complexity besides expressivity, however. In this paper, we describe a manufacturing problem domain that emphasizes on-line continual problem solving. The domain semantics are more complex that in job shop scheduling but simpler in many ways than PDDL2.1. A problem instance specifies a manufacturing plant consisting of several machines and a series of jobs to be completed, each of which might require several actions on different machines. As in classical scheduling, resource constraints are essential because the machines in the plant often cannot perform multiple actions at once. But action selection and sequencing are also required because a given job can usually be achieved using several different sequences of actions. Moreover, in our setting the set of jobs is only revealed incrementally over time, unlike in classical temporal planning where the entire problem instance is available at once. And in contrast to much work on continual planning (desJardins, Durfee, Ortiz, and Wolverton, 1999), we must produce a complete plan for each job before its execution can begin.

After discussing this problem domain in more detail, we will present our current solution: an on-line temporal planner that combines constraint-based scheduling with heuristic state-space planning. Jobs are optimally planned on an individual basis, in order of arrival, without reconsidering the plans selected for previous jobs. To mitigate the restrictiveness of this greedy scheme, we represent action times using temporal constraints instead of absolute times. By maintaining temporal flexibility as long as possible, we can shift plans for older jobs later in time to make room for starting a new job earlier if that improves overall plant throughput. Although this basic architecture is specifically adapted to our on-line setting, the planner uses no domain-dependent search control knowledge. We present some empirical measurements demonstrating that significant plants can be controlled by the planner while meeting our real-time requirements. Our integrated on-line approach allows us to achieve high throughput even for complex plants.

A Simple Manufacturing Domain

The domain is based on a real manufacturing process control problem encountered by one of our industrial clients. The application is reminiscent of 'mass customization,' in which mass-produced products are closely tailored and personalized to individual customers' needs. It also shares characteristics with document printing, although on a much larger scale. It involves planning and scheduling a series of job requests which arrive asynchronously over time. The plant runs at high speed, with several job requests arriving per second, possibly for many hours. Each job request completely describes the attributes of a desired final product. There may be several different sequences of actions that can be used to produce a given job. The planning system must decide how to manufacture all of the desired products as quickly as possible. In other words, it must determine a plan and schedule for each job such that the end time of the plan that finishes last is minimized. This is an on-line task because the set of jobs grows as time passes. In fact, because it is the real-world wall clock end time that we want to minimize and because the job cannot start until it is planned, the speed of the planner itself affects the value of a plan! However, the plant is often at full capacity, and thus the planner usually need only plan at the rate at which jobs are completed, which again may be several per second. While challeng-

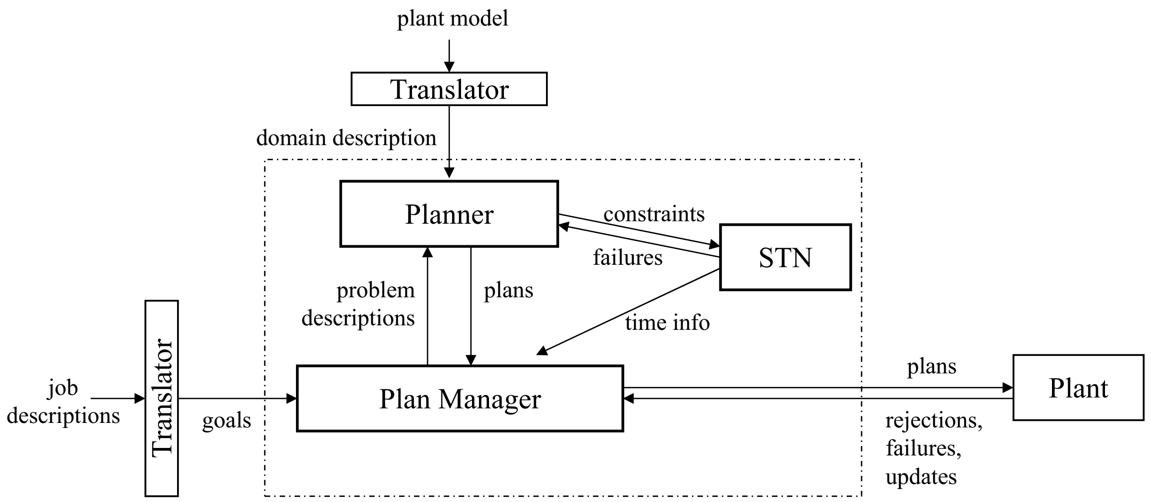

Figure 1: The system architecture, with the planning system indicated by the dashed box.

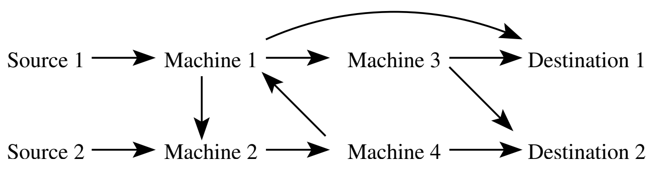

Figure 2: A schematic view of a manufacturing plant.

ing, the domain is also forgiving: feasible schedules can be found quickly, sub-optimal plans are acceptable, and plan execution is relatively reliable. Figure 1 indicates the relationship between the planning system and the plant. We will discuss the planning system itself in detail below.

The typical plants in our application can be schematically represented as a network of transports linking multiple machines (Figure 2). A typical plant might have anywhere from a few to a few hundred machines and transports. Unfinished blocks of raw material can enter the plant from multiple sources and completed jobs can exit at multiple destinations. Transports take a known time to convey jobs between machines. Each machine has a limited number of discrete actions it can perform, each of which has a known duration and transforms its input in a known deterministic way. These durations may vary over three orders of magnitude. For simplicity, we currently only consider actions that manipulate single blocks of material at a time. Actions may not split a block into two pieces for use in different jobs or join multiple blocks from different paths in the plant together. This means that a single job must be produced from a single unit of material, thereby conflating jobs with material and allowing plans to be a linear sequence of actions. In our domain, adjacent actions must meet in time; material cannot be left lingering inside a machine after an action has completed but must immediately begin being transported to its next location.

Most manufacturing actions require the use of physical

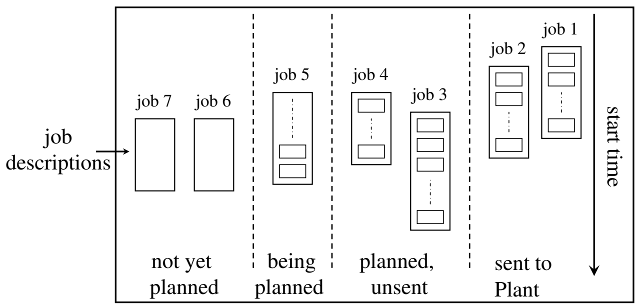

Figure 3: Stages in the life of a job in the planning system.

plant resources, so planning for later jobs must take into account the resource commitments in plans already released for production. Typically there are many feasible plans for any given job request; the problem is to quickly find one that finishes soon. The optimal plan for a job depends not only on the job request, but also on the resource commitments present in previously-planned jobs. Any legal series of actions can always be easily scheduled by pushing it far into the future, when the entire plant has become completely idle, but of course this is not desirable.

Additional Complications

In our application, the planner communicates on-line with the physical plant, controlling production and responding to execution failures (right side of Figure 1).[1] After a completed plan is transferred to the lower-level plant controller software, the planner cannot modify it. There is thus some benefit in releasing plans to the plant only when their start times approach. Figure 3 shows the different stages a job passes through in our system. In the figure, time advances

[1]In fact, the plant may also reject plans outright without executing them. We ignore this complication in our presentation.

	Job-23
initial:	Location(Job-23, Some-Source)
	Uncut(Job-23)
	Color(Job-23, Raw)
	¬Aligned(Job-23)
goal:	Location(Job-23, Some-Destination)
	HasShape(Job-23, Cylinder-Type-2)
	Polished(Job-23)
	Clean(Job-23)
	Color(Job-23, Blue)
background:	CanCutShape(Machine-2, Cylinder-Type-2)
batch:	5

Figure 4: A sample job specification, including background literals.

	CutOn2(?block)
preconditions:	Location(?block, Machine-2-Input)
	Uncut(?block)
	Aligned(?block)
	CanCutShape(Machine-2, ?shape)
effects:	Location(?block, Machine-2-Output)
	¬Location(?block, Machine-2-Input)
	HasShape(?block,?shape)
	¬Uncut(?block)
	¬Aligned(?block)
duration:	13.2 secs
allocations:	M-2-Cutter at ?start + 5.9 for 3.7 secs

Figure 5: A simple action specification.

downward so plans starting earlier are higher in the figure. Note that jobs 3 and 4 have been planned but not yet released from the plan manager to the plant.

Jobs are grouped into batches. A batch is an ordered set of jobs, all of which must eventually arrive at the same destination in the same order in which they were submitted, so that they may be immediately packed and delivered to the end customer. Multiple batches may be in production simultaneously, although because jobs from different batches are not allowed to interleave at a single destination, the number of concurrent batches is limited by the number of destinations.

Occasionally a machine or transport will break down, in effect changing the planning domain by removing the related actions. The plant is also intentionally reconfigured periodically. This means that precomputing a limited set of canonical plans and limiting on-line computation to scheduling only is not desirable. For a large plant of 200 machines, there are infeasibly many possible broken configurations to consider. Depending on the capabilities of the machines, the number of possible job requests may also make plan precomputation infeasible. Furthermore, even the best precomputed plan for a given job may be suboptimal given the current resource commitments in the plant.

To summarize, our domain is finite-state, fully-observable, and specifies classical goals of achievement. However, planning is on-line with additional goals arriving asynchronously. Actions have real-valued durations and use resources. Plans for new goals must respect resource allocations of previous plans. Execution failures can occur, but are rare enough that, at least in our current solution, we don't explicitly plan ahead for them.

Modeling the Domain

This manufacturing domain can be modeled by a straightforward temporal extension of STRIPS. As in PDDL, we distinguish between two types of input to the planner. Before planning begins, a domain descriptions containing literal and action templates is provided (top of Figure 1). Then the problem descriptions arrive on-line, containing initial and goal states, which are sets of literals describing the starting and desired configurations (left of Figure 1). A simple example job specification is given in Figure 4. In this example, `Some-Source` and `Some-Destination` are virtual locations where all sources or destinations are placed. In addition to job-dependent literals, sometimes it is convenient to specify actions using preconditions that refer to literals that are independent of the particular goals being sought. This 'background knowledge' about the domain is supplied separately in the job specification, although it could also be compiled into the action specifications. In our example, the possible shapes that a machine can cut are specified in this way.

The movement of material by transports and the transformation of material by machine actions can be directly translated from the plant model into traditional logical preconditions and effects that test and modify attributes of the material. A simple example is given in Figure 5. Our action representation is similar to the durative actions in PDDL2.1 with actions having specified real-valued duration bounds. Although the example shows a constant duration, one may also specify upper and lower bounds and let the planner choose the desired duration of the action. (This is helpful for modeling controllable-speed transports.) The intended semantics is that the logical effects become true exactly when the action's duration has elapsed. The notable extension of our representation over PDDL2.1 is the explicit representations of resources. Actions can specify the exclusive use of unit-capacity resources for time intervals specified relative to the action's start or end times. For example, the CutOn2 action in Figure 5 specifies exclusive use of the M-2-Cutter from 5.9 seconds after the start of the action until 3.7 seconds later. Machines with several independent resources or with actions that have short allocation durations relative to the overall action duration can work on multiple jobs simultaneously, so locations and resource allocations are not equivalent. In PDDL, arbitrary predicates can be made to hold at the start, end, or over the duration of an action. This expressivity is not needed in our domain.

To summarize, there are two inputs to the planner:

- A domain is a set of actions, each of which is a 4-tuple $a = \langle \mathit{Pre}, \mathit{Eff}, \mathit{dur}, \mathit{Alloc} \rangle$, where *Pre* and *Eff* are sets of literals representing the action's preconditions and effects; *dur* is pair $\langle \mathit{lower}, \mathit{upper} \rangle$ of scalars representing the upper and lower bounds on action duration; and *Alloc* is a set of triplets $\langle \mathit{res}, \mathit{offset}, \mathit{dur} \rangle$ indicating that action a uses

```
OnlinePlanner
1. plan the next job
2. if an unsent plan starts soon, then
3.     foreach plan, from the oldest through the imminent one
4.         clamp its time points to the earliest possible times
5.         release the plan to the plant

PlanJob
6. search queue ← {final state}
7. loop:
8.     dequeue the most promising node
9.     if it is the initial state, then return
10.    foreach applicable operator
11.        undo its effects
12.        add temporal constraints
13.        foreach potential resource conflict
14.            generate all orderings of the conflicting actions
15         enqueue any feasible child nodes
```

Figure 6: Outline of the hybrid planner.

resource *res* during an interval $[s_a + \mathit{offset}, s_a + \mathit{offset} + \mathit{dur}]$ where s_a is the starting time of a.

- A problem description for a particular job is a 4-tuple of $\langle \mathit{batch}, \mathit{Initial}, \mathit{Goal}, \mathit{Background} \rangle$, where *batch* is a batch id and *Initial*, *Goal*, and *Background* are sets of literals.

Given a domain description (top of Figure 1) and a low-level delay constant t_{delay} capturing the latency of the plant controller software, the planner then accepts a stream of jobs arriving asynchronously over time (left side of Figure 1). This stream corresponds to the standard notion of a PDDL problem instance. For each job, the planner must eventually return a plan: a sequence of actions labeled with start times (in absolute wall clock time) that will transform the initial state into the goal state. Any allocations made on the same resource by multiple actions must not overlap in time. Happily, plans for individual jobs are independent except for these interactions through resources. Additional constraints on the planner include 1) plans for jobs with the same batch id must finish at the same destination, 2) plans for jobs with the same batch id must finish in the same order in which the jobs were submitted, 3) the first action in each plan must not begin sooner than t_{delay} seconds after it is issued by the planner, and 4) subsequent actions must begin at times that obey the duration constraints specified for the previous action (thus it is assumed that the previous action ends just as the next action starts).

A Hybrid Planner

We have implemented our own temporal planner using an architecture that is adapted to this on-line domain. The overall objective is to minimize the end time of the known jobs. We approximate this by optimally planning only one job at a time instead of reconsidering all unsent plans. As we will see below, the large number of potential plans for a given job and the close interaction between plans and their schedules means that it is much better to process scheduling constraints during the planning process and allow them to focus planning on actions that can be executed soon. The planner uses state-space regression to plan each job, but maintains as much temporal flexibility as possible in the plans by using a simple temporal constraint network (STN) (Dechter, Meiri, and Pearl, 1991). This network represents a feasible interval for each time point in each plan. Time points are restricted to occur at specific single times only when the posted constraints demand it. Because the planner maintains the partial orders between different actions in plans for different jobs through the STN while conducting the backward state-space search, it can be seen as a hybrid between state-space search and partial-order planning. A sketch of the planner is given in Figure 6. The outer loop corresponds to the plan manager in Figure 1.

After planning a new job, the outer loop checks the queue of planned jobs to see if any of them begin soon (step 2). It is imperative to recheck this queue on a periodic basis, so 'soon' is defined to be before some constant amount after the current time and we assume that the time to plan the next job will be smaller than this constant. The value of this constant depends on the domain and is currently selected manually. If this assumption is violated, we can interrupt planning the next job and start over later. As plans are released and execute, resource contention will only decrease, so the time to plan the new job should decrease as well. It is important that new temporal constraints are added by the outer loop only between the planning of individual jobs, as propagation can affect feasible job end times and thus could invalidate previously computed search node evaluations if planning were underway.

Due to details of the plant controller software, the planner must release jobs to the plant in the same order in which they were submitted. This means that jobs submitted before any imminent job must be released along with it (step 3). Only at this stage are the allowable intervals of the job's time points forcibly reduced to specific absolute times (step 4). Sensibly enough, we ask that each point occur exactly at the earliest possible time. Because the temporal network uses a complete algorithm (a variation on Cervoni, Cesta, and Oddi, 1994) to maintain the allowable window for each time point, we are guaranteed that the propagation caused by this temporal clamping process will not introduce any inconsistencies.

Planning Individual Jobs

When planning individual jobs, the regressed state representation contains the (possibly partially-specified) state of the job. The start and end of each action and each resource allocation are represented by timepoints in the temporal network. Temporal constraints are used to represent the order and durations of actions and to resolve resource contention. Action schemata are kept in lifted form—variables can appear in states but must be bound in the final plan. A* search is used to find the optimal plan for the job, in the context of all previous jobs. We will discuss the state representation, regression semantics, branching rule, and objective function in turn.

State Representation Because the plan must be feasible in the context of previous plans, the state contains information both about the current job and previous plans. More specifically, the state is a 4-tuple $\langle \mathit{Literals}, \mathit{Bindings}, \mathit{Tdb}, \mathit{Rsrcs} \rangle$ in which

Literals describes the regressed logical state of the current job.

Bindings are the variable bindings for all the variables occurring in literals associated with any planned job. Thus, in Figure 3, the variable bindings are used to ground all lifted actions in the plans for jobs 1–4 and actions in the partial plan for job 5.

Tdb is the temporal database represented as a simple temporal network (STN) containing all known time points and the current constraints between them. Examples of time points include the start/end times of actions or resource allocations. As soon as a plan for a given job is sent to the plant (jobs 1 and 2 in Figure 3), time points associated with that plan in the *Tdb* are no longer allowed to float but are clamped at their lower bounds.

Rsrcs is the set of current resource allocations. Each resource allocation is of the form $\langle \mathit{res}, \mathit{tp}_1, \mathit{tp}_2 \rangle$ with *res* is a particular resource and $\mathit{tp}_1, \mathit{tp}_2$ are two time points in the *Tdb* representing the duration *res* is allocated to some action. Note that there are multiple resources in the domain and each resource can have multiple (non-overlapping) resource allocations.

Regression Semantics The logical goal state of a job can consist of both positive and negative literals. Each regressed logical state in our planner is a 3-tuple $L = \langle L_t, L_f, L_u \rangle$ where L_t, L_f, and L_u are the disjoint sets of literals that are true, false, and unknown (or 'don't care'), respectively (Le, Baral, Zhang, and Tran, 2004). The distinction between false and unknown literals is important in our domain because there may be fine-grained restrictions on the acceptable values for unspecified attributes of the job. If $\mathit{Pre}^+(a)$ and $\mathit{Pre}^-(a)$ are the sets of positive and negative preconditions and $\mathit{Add}(a)$ and $\mathit{Del}(a)$ are the sets of positive and negative effects of action a, then the regression rules used to determine action applicability (step 10 for Figure 6) and to update the state literals (step 11) are

Applicability Action a is applicable to the literal set L if 1) none of its effects are inconsistent with L and 2) any preconditions not modified by the effects of a are consistent with L. More formally, 1) $(\mathit{Add}(a) \bigcap L_f = \emptyset) \wedge (\mathit{Del}(a) \bigcap L_t = \emptyset)$, and 2) $(\mathit{Pre}^+(a) \bigcap L_f \subset \mathit{Del}(a)) \wedge (\mathit{Pre}^-(a) \bigcap L_t \subset \mathit{Add}(a))$.

In many planning settings, an additional criterion for applicability can be added without destroying completeness: at least one effect of a must match L $((\mathit{Add}(a) \bigcap L_t \neq \emptyset) \vee (\mathit{Del}(a) \bigcap L_f \neq \emptyset))$. This is not necessarily valid in our setting because adding a 'no-op' action a may give more time for an existing resource allocation to run out, enabling other actions to be used which might lead to a shorter plan.

Update The regression of $L = \langle L_t, L_f, L_u \rangle$ over an applicable action a is derived by undoing the effects of a and unioning the result with a's preconditions. For a given literal l modified by an effect of a, its status will be unknown in the regressed state unleast it is also specified by a corresponding precondition of a (e.g., $\neg l$ is a precondition of a). More formally, 1) $L_t = (L_t \setminus \mathit{Add}(a)) \bigcup \mathit{Pre}^+(a)$; 2) $L_f = (L_f \setminus \mathit{Del}(a)) \bigcup \mathit{Pre}^-(A)$; and 3) $L_u = (L_u \bigcup (\mathit{Add}(a) \bigcup \mathit{Del}(a))) \setminus (\mathit{Pre}^+(a) \bigcup \mathit{Pre}^-(a))$

Given that $|L_f|$ is usually much larger than $|L_u|$ in our domain, we explicitly store L_t and L_u in our current implementation and assume that all other literals belong to L_f.

Although it is not usually the case in our domain, we should note that if the goal state were always fully specified (with no unknown literals) and every action's effects had corresponding preconditions, all states would be fully specified. One could then simplify the logical state representation to $L = \langle L_t, L_f \rangle$ and simplify the regression rules to

Applicability Action a is applicable iff all its effects match in L: $\mathit{Add}(a) \subset L_t$ and $\mathit{Del}(a) \subset L_f$.

Update Regressing $\langle L_t, L_f \rangle$ through a gives $\langle (L_t \setminus \mathit{Add}(a)) \bigcup \mathit{Del}(A), (L_f \setminus \mathit{Del}(a)) \bigcup \mathit{Add}(a) \rangle$

Branching rule While the first step in creating regressed states is to branch over the actions applicable in L, each candidate action a can in fact result in multiple child nodes due to resource contention. Some scheduling algorithms use complex reasoning over disjunctive constraints to avoid premature branching on ordering decisions that might well be resolved by propagation (Baptiste and Pape, 1995). We take a different approach, insisting that any potential overlaps in allocations for the same resource be resolved immediately. Temporal constraints are posted to order any potentially overlapping allocations (step 14 in Figure 6) and these changes propagate to the action times. Because action durations are relatively rigid in typical plants, this aggressive commitment can propagate to cause changes in the potential end times of a plan, immediately helping to guide the search process. Because multiple orderings may be possible, there may be many resulting child search nodes.

For example, in Figure 3, assume that a is the current candidate action when searching for a plan for job 5 and that a uses resource r for a duration $[s, e]$. We also assume that there are n actions in the plans for jobs 1–4 that also use r, implying n existing non-overlapping resource allocations $[s_1, e_1]....[s_n, e_n]$ and corresponding time points in the temporal database. When trying to allocate r for a, one obvious and consistent choice is putting it after all other previous allocations by adding the temporal constraint $e_n < s$. However, there can also be gaps between the existing allocations $[s_i, e_i]$, allowing us to post constraints such as $e_i < s < e < s_{i+1}$. Each such possible allocation for r generates a distinct child node in the search space. Because action a can use several different resources r, the number of branches is potentially quite large. However, immediately resolving any potential overlaps in allocations for the same resource avoids the introduction of disjunctions in the temporal network, maintaining the tractability of temporal

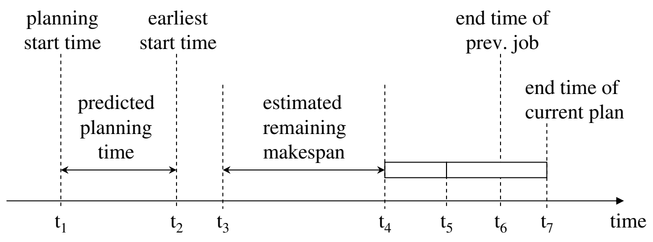

Figure 7: Important time points for constructing and evaluating a plan.

constraint propagation.

The planner has many attributes of a state-based planner: it maintains a totally specified logical state and uses regression to branch on applicable actions. However, it also branches on partial orders introduced to resolve resource contention between actions in plans for different jobs. Therefore, our planner can be seen as a combination of both state-space regression and partial-order planning. This hybrid behavior is also reflected in the plan representation. While the plan for a single job is a total-ordered sequence of actions, there are partial orders between actions that belong to plans of different jobs. Due to these partial orderings, the starting times of all actions in the plans that belong to jobs not yet sent to the plant (jobs 3–5 in Figure 3) are not fixed but merely constrained by the lower and upper bounds in the temporal database.

The actions in a plan for a single job are sequential and abut each other in time. Figure 7 illustrates some of the relevant time points. In this figure, *planning start time* refers to the real-world clock time at which the planning process started, *earliest-start-time = current wall time + predicted planning time* is the estimated time at which we will find a plan for the current job and thus is the earliest time that any action can be scheduled to begin. A plan is constrained to end after those for previous jobs in the same batch (t_6 in the figure), but is not necessarily constrained to start after or before plans for previous jobs. The start time of an action (t_4, for example) is constrained (step 12 in Figure 6) to occur before the end time (t_5), according to the action's duration, and after the time we are predicted to be done planning this job (t_2).

The remaining component *Bindings* of the regressed state is updated with the variable bindings necessary for the instantiation of a to match with the regressed logical state L.

At every branch in the planner's search space, we either modify the logical state differently or introduce different temporal constraints in order to resolve resource contention. Because each branch results in a different irrevocable choice that is reflected in the final plan, the state at each node in the planner's search tree is unique. Therefore, we do not need to consider the problem of duplicating search effort due to reaching the same state by two different search paths.

Objective function Our overall objective function is to minimize the earliest possible end time of the plan for the current job. Because it is constrained to end after the completion time of all the other jobs in the same batch, the objective function is essentially minimizing the end time of the batch of jobs involving the current job. To support this objective function, the primary criterion evaluating the promise of a partial plan (step 8 in Figure 6) is the estimate of the earliest possible end time of the partial plan's best completion.

To improve our estimates of this quantity, we compute a simple lower bound on the additional makespan required to complete the current plan. We use a scheme similar to the h_T^1 heuristic of Haslum and Geffner (2001) by building the bi-level temporal planning graph Smith and Weld (1999) without mutex and resource contention constraints that estimates the fastest way to achieve the logical state L. Starting from the initial state (and background knowledge), we apply all applicable actions, labeling each resulting literal that was not previously known with the end time of the action used to produce it. The new literals may help enable new actions, which are then applied, possibly producing yet further new literals. This process continues until either all literals in $L = \langle L_t, L_f \rangle$ have been produced or no new literals can be made. Our lower bound is then the maximum over the times taken to produce the individual literal in L (infinity if a literal cannot be produced).

This heuristic value is indicated in Figure 7 by *estimated remaining makespan*. It is inserted before the first action in the current plan (t_4) and after the earliest plan's start time (t_2). By adding the constraint $t_2 < t_3$, the insertion may thus change the end time of the plan. It may also introduce an inconsistency in the temporal database, in which case we can safely abandon the plan. Given that the current plan should end after the end time of all previous jobs in the same batch ($t_6 < t_7$), our objective function is to minimize t_7 without causing any inconsistency in the temporal database. Any remaining ties between search nodes after considering end time t_7 are broken in favor of nodes having smaller predicted makespan values ($t_7 - t_3$). In case there are still ties, they are broken in favor of the node that has the larger currently realized makespan ($t_7 - t_4$). This is analogous to the usual practice of breaking ties on $f(n)$ in A* search in favor of larger $g(n)$, and encourages further extension of plans nearer to a goal. Because our heuristic is admissible, the plan found is optimal according to our objective function.

A plan is considered complete if its literals unify with the desired initial state (step 9 in Figure 6). After the optimal plan for a job is found, the variable bindings and temporal database used for the plan are passed back to the outer loop in Figure 6 and become the basis for planning the next job. Because feasible windows are maintained around the time points in a plan until the plan is released to the plant, subsequent plans are allowed to make earlier allocations on the same resources and push actions in earlier plans later. If such an ordering leads to an earlier end time for the newer goal, it will be selected. This provides a way for a complex job that is submitted after a simple job to start its execution in the plant earlier. Out of order starts are allowed as long as all jobs in each batch finish in the correct order. This can often provide important productivity gains.

Additional Features

Our implementation elaborates on the basic algorithm presented above in certain ways. It includes full support for unbound variables, which are tolerated during planning but are unacceptable in a complete plan. In this sense, it is a lifted planner like SNLP (McAllester and Rosenblitt, 1991). This capability is used, for example, in ensuring that subsequent jobs in the same batch end at the same destination. The destination actions each have an effect like `Dest(D1)`. All jobs in the same batch include in their goal the atom `Dest(?batch23dest)` where the variable `?batch23dest` is shared among all the jobs. This variable will be bound by the first job to be planned, and will constrain the subsequent jobs. The job specification is elaborated by including non-codesignation constraints on `?batch23dest` that prevent it from codesignating with variables representing destinations of other current batches. (The planner is notified after the last job in a batch, allowing it to free the batch's destination for use by a new batch. We assume that the job source does not submit more active batches than the plant has destinations.)

Our planner also checks for messages from the plant controller during the search process. These can be of two types: domain model updates or execution failures. In either case, the current search is aborted. This allows us to assume that the planning domain remains constant during the planning of individual jobs. Domain updates are straightforward modifications of the set of possible actions. Currently, we make several assumptions to simplify the handling of execution failures. We assume that the transports remain reliable, that the job continues on its planned course, and that a diverter is present at each destination that, when commanded by the planner, can divert the faulty job for disposal. The planner's job is thus reduced to diverting the botched job and any subsequent jobs in the same batch that have already been released to the plant. The diverted jobs are then replanned from scratch.

In addition to unit-capacity resource constraints, we have found that some actions require state constraints, in which two allocations for the same resource may overlap only if they both request that the resource be in the same state.

Experience in Practice

In collaboration with our industrial client, we have deployed the planner to control two physical prototype plants. These deployments have been successful, and our client currently anticipates development of a new line of products spanning multiple markets that crucially depends on this technology. (Precompiled plans are unacceptable and high plant throughput is essential.) The planner is written in Objective Caml, a dialect of ML, and communicates with the job submitter and the plant controller using ASCII text over sockets. To give a sense of the performance of our implementation, we present simulation results on a variety of plants. Figure 8 shows the time taken to plan each job in a large batch (in seconds on a 2.4Ghz P4). The plant model used in this example yields a domain with 19 action schemata. Plans for individual jobs typically use three to five actions. Two versions are shown, differing in the algorithms they use to manage the temporal constraints. One uses an incremental directed path consistency (IDPC) algorithm (Chleq, 1995), which may change the values on edges in the constraint graph as well as introduce new edges but requires only linear time to find the minimum and maximum interval between any two time points in the database. The other uses arc consistency (Cervoni et al., 1994) and maintains for each point its minimum and maximum time from t_0, the reference time point. One cannot easily obtain the exact relations between arbitrary time points, but this is rarely needed during planning. New arcs are never added to the network during propagation and existing ones are not modified, which means that copying the network for a new search node does not entail copying all the arcs. As the figure attests, this results in dramatic time savings. Planning time in the faster implementation was never longer than 50 milliseconds. In this domain, the planner quickly outpaced the plant—the plan for job 4 was released to the plant just before planning began on job 100. Note that this means the planner could consider interleaving actions in the new plan with the actions in 96 previous plans. All of their time points and actions must be maintained in the STN and the plan manager and then consulted during the planning of a new job. However, these jobs all belong to the same batch, so the constraint on end time ordering, combined with goal regression, quickly eliminates most interleavings.

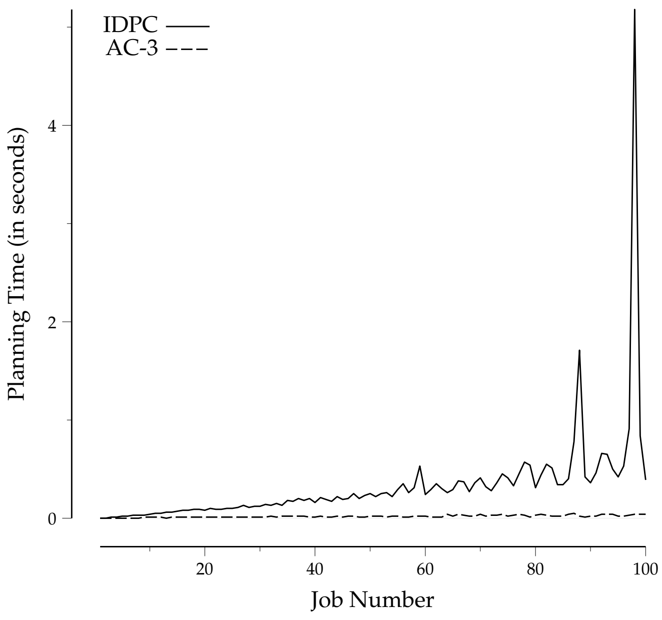

Figure 8: Simple arc consistency is faster than incremental directed path consistency.

We also evaluated the contribution of the lower bound computations in guiding the search. Figure 9 shows planning time in a slightly more complex plant, with 16 machines and transports, yielding a domain with 73 action schemata. Plans here typically involve five to ten actions. The lower bound clearly improves planning time. Although

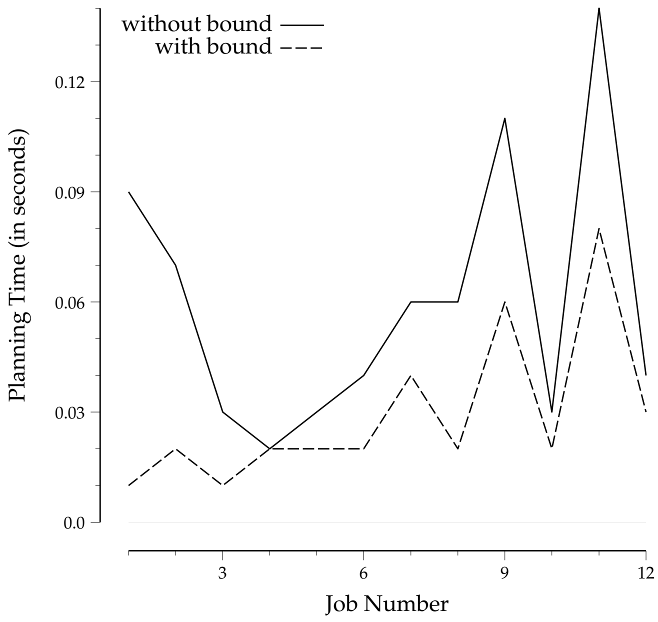

Figure 9: Heuristic guidance helps even for a small plant.

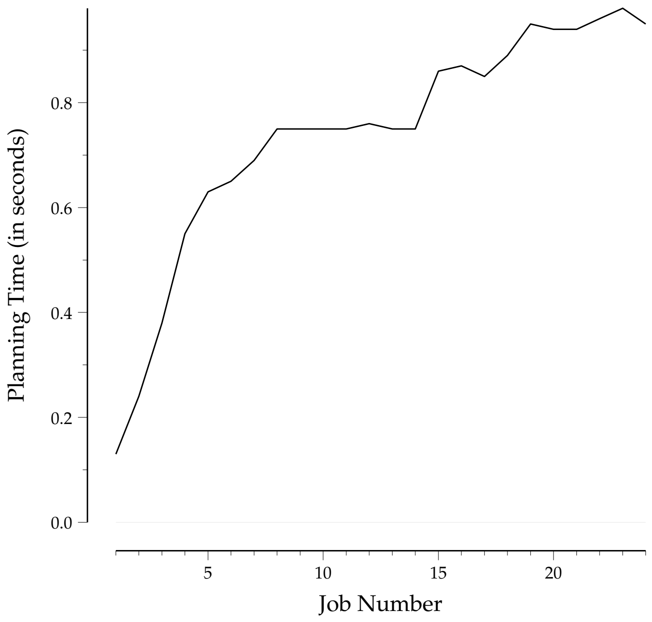

Figure 10: Running times increase, but do not necessarily explode for large plants.

the difference between 0.06 and 0.11 seconds may seem irrelevant in most planning research, we should emphasize that in our setting it may determine the system's feasibility!

Finally, we present in Figure 10 preliminary measurements of planning time using a large simulated plant model with 104 machines and transports, totaling 728 action schemata. Plans here typically involve over 30 actions. This test used simple job requests, so the lower bound estimates were often very accurate. Planning time rises quickly above the 0.2 seconds per job which we take as our goal, but does not explode. We believe that with the extensions discussed below and further implementation tuning, we should be able to handle domains of this size reliably.

Relations to Previous Work

There has been much interest in the last 15 years in the integration of planning and scheduling techniques. HSTS (Muscettola, 1994) and IxTeT (Ghallab and Laruelle, 1994) are examples of systems that not only select and order the actions necessary to reach a goal, but also specify precise execution times for the actions. However, these systems are often demonstrated on complex domains such as spacecraft or mobile robot control which can be difficult to simulate and thus make awkward benchmarks. Most popular temporal planning benchmark domains are off-line in the sense that the planner's speed does not affect solution quality. There remains a need for a simple yet realistic benchmark domain that combines elements of planning and scheduling, especially in an on-line setting.

Fromherz, Saraswat, and Bobrow (1999) discuss on-line constraint-based scheduling methods for controlling physical machines, although they use precomputed plans and their formalization cannot model systems such as ours with possible cycles in the plant graph and hence an infinite number of potential plans (see Figure 2).

Our domain formalization lies between partial-order scheduling and temporal PDDL. Because the optimal actions needed to fulfill any given job request may vary depending on the other jobs in the plant, the sequence of actions is not predetermined and classical scheduling formulations such as job-shop scheduling or resource-constrained project scheduling are not expressive enough. This domain clearly subsumes job-shop and flow-shop scheduling: precedence constraints can be encoded by unique preconditions and effects. Open shop scheduling, in which one can choose the order of a predetermined set of actions for each job, does not capture the notion of alternative sequences of actions and is thus also too limited. The positive planning theories of Palacios and Geffner (2002) allow actions to have real-valued durations and to allocate resources, but they cannot delete atoms. This means that they cannot capture even simple transformations like movement that are fundamental in our domain. In fact, optimal plans in our domain may even involve executing the same action multiple times, something that is always unnecessary in a purely positive domain. However, the numeric effects and full durative action generality of PDDL2.1 are not necessary. Because of the on-line nature of the task and the unambiguous objective function, there is an additional trade-off in this domain between planning time and execution time that is absent from much prior work in planning and scheduling.

Although we present our system as a temporal planner, it fits easily into the tradition of constraint-based scheduling (Smith and Cheng, 1993). The main difference is that actions' time points and resource allocations are added incrementally rather than all being present at the start of the search process. The central process of identifying temporal conflicts, posting constraints to resolve them, and computing bounds to guide the search remains the same. In our approach, we attempt to maintain a conflict-free schedule rather than allowing contention to accumulate and then carefully choosing which conflicts to resolve first.

Our current planner uses a hybrid state-space and partial-order search framework. While maintaining partial orderings between actions seems necessary to mitigate our job-at-a-time greedy strategy, the planning for individual jobs need not necessarily take the form of state-space regression. We have considered a forward search strategy, such as employed by many state-of-the-art planners such as FF (Hoffmann and Nebel, 2001). Initial investigation and preliminary empirical comparisons showed that while a progression planner is easier to implement and is also easier to extend to handle additional domain complexities, the performance of the regression planner (using the same heuristic) is significantly better in many problems. This seems to be due mainly to the temporal constraint enforcing that a given job should end after the end time of all the previous jobs in the same batch. This constraint interacts well with searching backward from the goal, immediately constraining the end time of the plan. Together with the constraint that actions must abut in time, many possible orderings for resolving resource contention are immediately ruled out. For example, the current job cannot be transported to its destination before the previous job in the same batch. In addition, some orderings may immediately push the end time of the plan even later, further informing the node evaluation function.

When planning in the forward direction, the planner benefits slightly from avoiding logical states which are unreachable from the initial state. However, without a similar temporal constraint for the first action in the plan, few resource allocation orderings can be pruned and the branching due to resource contention increases in direct proportion to the number of plans for previous jobs maintained in the plan manager. Furthermore, the end time of the plan rarely changes until far into the planning processes, making the heuristic less useful. In short, for the first job, the performance of forward or backward planners are similar, while as the number of plans managed by the plan manager increases, the backward planner seems to perform better. We are already investigating scenarios in our application domain where there is no constraint on the end time of a job and where the objective function is not to minimize the end time. In these cases, we believe forward search may prove more useful than regression.

In our current on-line setting, even though we plan for multiple jobs belonging to different batches, we build plans for a single job at the time. Even if there are several submitted jobs waiting to be planned, this strategy is reasonable given that jobs arrived in sequence and, until the arrival of the last job, we do not know how many jobs are in each batch and when will the planner receive the individual job specifications. Our basic approach of coordinating separate state-space searches via temporal constraints may well be suitable for other on-line planning domains. By planning for individual jobs and managing multiple plans at the same time, our strategy is similar in spirit to planners that partition goals into subgoals and later merge plans for individual goals (Wah and Chen, 2003; Koehler and Hoffmann, 2000). In our framework, even though each job is planned locally, the plan manager along with the global temporal database ensures that there are no temporal or resource inconsistencies at any step of the search. It would be interesting to see if the same strategy could be used to solve partitionable STRIPS planning problems effectively.

Possible Extensions

While the results we have presented indicate that our 'optimal-per-job' strategy seems efficient enough, further work is needed to assess the drop in quality that would be experienced by a more greedy strategy, such as always placing the current job's resource allocations after those of any previous job. Similarly, during a lull in job submissions, it might be beneficial to plan multiple jobs together, backtracking through the possible plans of the first in order to find an overall faster plan for the pair together. Jobs that have been planned but whose plans have not been released to the plant represent opportunities for reconsideration in light of the newer jobs submitted more recently.

The most pressing extension concerns the heuristic: we would like to take some mutex relations into account in our heuristic using something similar to the H_T^2 function of Haslum and Geffner (2001) or the temporal planning graph of Smith and Weld (1999). Our initial experimental results in this direction are quite encouraging.

Initial investigations with a grounded version of our planner have also been promising, taking about half the time per search node as the current lifted scheme. If our planner is still too slow for large configurations, we are planning to investigate non-optimal planning for individual jobs.

Our handling of execution failures currently makes a number of strong assumptions, and we would like to investigate on-line replanning of jobs that have already begun execution. Our implementation also currently deletes information on completed jobs from the temporal database only when the machine is idle—this must be fixed before true production deployment.

Another direction is to investigate a different objective entirely: wear and tear. Under this objective, one would like the different machines in the plant to be used the same amount over the long term. However, because machines are often cycled down when idle for a long period and cycling them up introduces wear, one would like recently-used machines to be selected again soon in the short term.

Conclusions

We described a real-world manufacturing domain that requires a novel on-line integration of planning and scheduling and we formalized it using a temporal extension of STRIPS that falls between partial-order scheduling and temporal PDDL. We presented a hybrid planner that uses state-space regression on a per-job basis, while using a temporal constraint network to maintain flexibility and resolve resource constraints across jobs. No domain-dependent search control heuristics are necessary to control a plant of 16 machines in real time, although further work will be necessary to scale to our ultimate goal of hundreds of machines with ten or more jobs per second. Our work provides an example of how AI planning and scheduling can find real-world application not just in exotic domains such as spacecraft or mobile robot control, but also for common down-to-earth problems such as manufacturing process control.

Acknowledgments

The members of the Embedded Reasoning Area at PARC provided helpful comments and suggestions. Our industrial collaborators not only provided domain expertise but were invaluable in helping us to simplify and frame the application in a useful way.

References

Baptiste, Philippe, and Claude Le Pape. 1995. A theoretical and experimental comparison of constraint propagation techniques for disjunctive scheduling. In *Proceedings of IJCAI-95*, 600–606.

Cervoni, Roberto, Amedeo Cesta, and Angelo Oddi. 1994. Managing dynamic temporal constraint networks. In *Proceedings of AIPS-94*, 13–18.

Chleq, Nicolas. 1995. Efficient algorithms for networks of quantitative temporal constraints. In *Proceedings of Constraints-95*, 40–45.

Dechter, Rina, Itay Meiri, and Judea Pearl. 1991. Temporal constraint networks. *Artificial Intelligence* 49:61–95.

desJardins, Marie E., Edmund H. Durfee, Charles L. Ortiz, Jr., and Michael J. Wolverton. 1999. A survey of research in distributed, continual planning. *AI Magazine* 20(4):13–22.

Fox, Maria, and Derek Long. 2003. PDDL2.1: An extension to PDDL for expressing temporal planning domains. *Journal of Artificial Intelligence Research* 20:61–124.

Fromherz, Markus P.J., Vijay A. Saraswat, and Daniel G. Bobrow. 1999. Model-based computing: Developing flexible machine control software. *Artificial Intelligence* 114(1–2):157–202.

Ghallab, Malik, and Hervé Laruelle. 1994. Representation and control in IxTeT, a temporal planner. In *Proceedings of AIPS-94*, 61–67.

Haslum, Patrik, and Héctor Geffner. 2001. Heuristic planning with time and resources. In *Proceedings of ECP-01*.

Hoffmann, Jörg, and Bernhard Nebel. 2001. The FF planning system: Fast plan generation through heuristic search. *Journal of Artificial Intelligence Research* 14: 253–302.

Koehler, Jana, and Jörg Hoffmann. 2000. On reasonable and forced goal orderings and their use in an agenda-driven planning algorithm. *Journal of Artificial Intelligence Research* 12:338–386.

Le, Tuan C., Chitta Baral, Xin Zhang, and Son Tran. 2004. Regression with respect to sensing actions and partial states. In *Proceedings of AAAI-04*.

McAllester, David, and David Rosenblitt. 1991. Systematic nonlinear planning. In *Proceedings of AAAI-91*, 634–639.

Muscettola, Nicola. 1994. HSTS: Integrating planning and scheduling. In *Intelligent scheduling*, ed. Monte Zweben and Mark S. Fox, chap. 6, 169–212. Morgan Kaufmann.

Palacios, Héctor, and Héctor Geffner. 2002. Planning as branch and bound: A constraint programming implementation. In *Proceedings of CLEI-02*.

Smith, David E., and Daniel S. Weld. 1999. Temporal planning with mutual exclusion reasoning. In *Proceedings of IJCAI-99*, 326–333.

Smith, Stephen F., and Cheng-Chung Cheng. 1993. Slack-based heuristics for constraint satisfaction scheduling. In *Proceedings of AAAI-93*, 139–144.

Wah, Benjamin W., and Yixin Chen. 2003. Partitioning of temporal planning problems in mixed space using the theory of extended saddle points. In *IEEE international conference on tools with artificial intelligence*.