On-line Planning and Scheduling in a High-speed Manufacturing Domain

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

Abstract

We describe a simple manufacturing domain that requires integrated planning and scheduling and report on our experience adapting existing techniques to this domain. The setting is on-line in the sense that additional jobs arrive asynchronously, perhaps several per second, while plans for previous jobs are being executed. While challenging, the domain is also forgiving: feasible schedules can be found quickly, suboptimal plans are acceptable, and plan execution is relatively reliable. An approach based on temporal state-space planning with constraint-based resource handling suffices for small to medium-sized instances of the problem, and our current implementation successfully controls two prototype plants. By integrating planning and scheduling, we enable high productivity even for complex plants.

Introduction

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. There remains a need for a simple yet realistic benchmark domain that combines elements of planning and scheduling, especially in an on-line setting.

In this paper, we describe a relatively simple problem domain that inherently requires integrated planning and scheduling. Resource constraints are essential, yet the domain is more complex than job shop scheduling. Unlike classical temporal planning, processing is on-line, incremental, and subject to execution failures. However, the problem is simple enough that we hope others may become interested in pursuing it.

The domain is based on a real manufacturing problem encountered by one of our industrial clients. It involves planning and scheduling a series of job requests which arrive asynchronously over time. There may be several different sequences of actions that can accomplish a given job. Execution requires the use of physical plant resources, so planning for later jobs must take into account the resource commitments in plans already released for production. Jobs are grouped into batches, several of which may be in production simultaneously. The jobs within a batch must ultimately arrive at the same destination and in the same order in which they were submitted, so that they may be immediately packed and delivered to the end customer. Because the contents of each batch is only revealed to the planner incrementally, the objective is to minimize the end time of the most recently submitted job. Because the job cannot start until it is planned, the speed of the planner is linked to the objective function. 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 may be several per second.

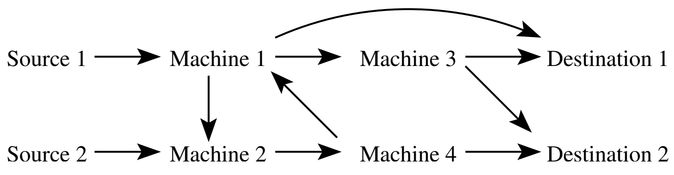

Figure 1: A schematic view of a manufacturing plant.

In our application, the planner communicates on-line with the physical plant, controlling production and responding to execution failures. 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.

After discussing the 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. Although the basic architecture is adapted to this 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 improved performance for more complex machines than previous approaches, which used on-line scheduling but precomputed plans.

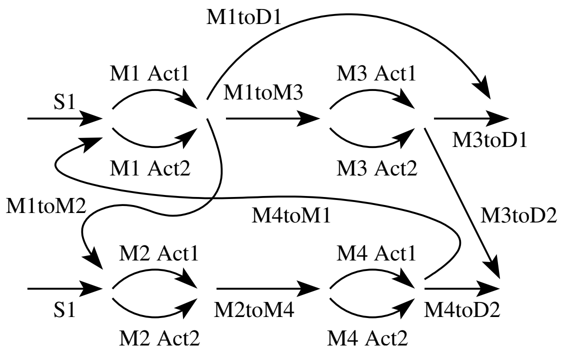

Figure 2: The domain from Figure 1, showing the effect of the actions on the `Location` predicate.

A Simple Manufacturing Domain

Schematically, a plant can be represented as a network of transports linking multiple machines (Figure 1). 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 consider only transformations that manipulate single blocks of material at a time. This means that a single job must be produced from a single unit of material, thereby conflating jobs with material. From a planning point of view, then, jobs can move through the plant as depicted in Figure 2. In our domain, adjacent actions must abut 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.

A job request specifies a desired final configuration, which may be achievable by several different sequences of actions. The plant runs at high speed, with several job requests arriving per second, possibly for many hours. A batch is an ordered set of jobs, all of which must eventually arrive in order at the same destination. 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 possible schedule for a given job's precomputed plan may be suboptimal given the current resource commitments in the plant.

	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 3: 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 4: A simple action specification.

The planner must accept a stream of jobs that arrive asynchronously over time and produce a plan for each job. These plans are given to the plant control software as a sequence of actions labeled with start times. The plant controller executes the plans it is given and reports any failures. Due to communication delays and limitations in the machine controllers, any plan that is released to the plant controller must start later than a certain time past the current instant and may not subsequently be changed by the planner.

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. The large number of potential plans 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.

Modeling the Domain

This manufacturing domain can be modeled by a straightforward temporal extension of STRIPS. A job specification corresponds to a pair of initial and goal states—sets of literals describing the starting and desired configurations.

A simple example is given in Figure 3. In the example, `Some-Source` and `Some-Destination` are virtual locations where all sources or destinations are placed. The movement of material by transports and the transformation of material by machine actions can be directly translated into traditional logical preconditions and effects that test and modify attributes of the material. A simple example is given in Figure 4. 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, keeping the representation independent of whether the planner itself searches forward or backward. In our example, the possible shapes that a machine can cut are specified in this way, rather than being compiled into the action specifications.

Our action language goes beyond classical STRIPS by incorporating time and simple resources. Instead of always taking unit time, actions have 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. To capture some of the physical constraints of common machines, we also allow actions to specify the exclusive use of unit-capacity resources for time intervals specified relative to the action's start or end times. As the example in Figure 4 implies, some machines can work on multiple jobs simultaneously, so locations and resource allocations are not equivalent. The resource allocations can be viewed as simplified versions of the maintenance conditions of PDDL-style durative actions (Fox and Long, 2003). In PDDL, arbitrary predicates can be made to hold at the start, end, or over the duration of an action.

To summarize, a domain is a set of actions, each of which is a 4-tuple $\langle Pre, Eff, dur, Alloc \rangle$, where *Pre* and *Eff* are sets of literals, *dur* is pair $\langle lower, upper \rangle$ of scalars, and *Alloc* is a set of triplets $\langle name, offset, dur \rangle$. A job is a 4-tuple of $\langle batch, Initial, Goal, Background \rangle$, where *batch* is a batch id and *Initial*, *Goal*, and *Background* are sets of literals. Given a domain and a low-level delay constant t_{delay}, the planner accepts a stream of jobs arriving asynchronously over time. 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, possibly using the background knowledge. Any allocations made on the same resource by multiple actions must not overlap in time. Plans for jobs with the same batch id must finish in the same order in which the jobs were submitted. (We will discuss below how goal literals can be used to ensure that jobs with the same batch id arrive at the same destination and that jobs with different batch ids arrive at different destinations.) The first action in each plan must begin not sooner than t_{delay} seconds after it is issued by the planner, and subsequent actions must begin at times that obey the duration constraints specified for the previous action. (It is assumed that the previous action ends just as the next action starts.) Given the set of jobs submitted up to the current time, the objective of the planner is to have the last action of the last job end as soon as possible.

```
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 both orderings of the conflicting actions
15         enqueue any feasible child nodes
```

Figure 5: Outline of the hybrid planner.

This 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. 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.

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. The planner uses state-space regression to plan each job, but maintains as much temporal flexibility as possible in the plans by using a temporal constraint network (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. In this sense, the planner is a hybrid between state-space search and partial-order planning. A sketch of the planner is given in Figure 5.

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. Resource contention will only decrease, so the time to plan the job should decrease as well. It is important that new temporal constraints are added only between the planning of individual jobs, as propagation may affect feasible job end times and thus invalidate previously computed search node evaluations.

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 any of the 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 database uses a complete algorithm (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

Individual jobs are planned using state-space regression and A* search. The regressed state contains information about the state of the job as well as information about the state of the plant. Specifically, the state is a 4-tuple $\langle$*Literals*, *Bindings*, *Tdb*, *Rsrcs*$\rangle$, where *Literals* describes the state of the current job, *Bindings* are the variable bindings for all the variables occurring in literals associated with any planned job, *Tdb* is the temporal database containing all known time points and their current constraints, and *Rsrcs* is the set of current resource allocations (which will refer to time points in *Tdb*). These data structures can be implemented in ways that allow additions to be made without copying the entire original structure.

At every branch in the planner's search space, we either modify the job state differently or introduce different temporal constraints in order to resolve resource contention. Because the options at each branch are exhaustive and mutually exclusive, each state in the planner's search tree is unique. Therefore, we do not need to consider the problem of duplicated search effort that can result from reaching the same state by two different search paths.

Because the domain is specified as a set of actions written in the standard progression style of STRIPS, as in Figure 4, our use of regression requires modification of the action specifications. When the domain is initially parsed, action specifications are rearranged into new sets of preconditions and effects for use in regression. The new preconditions are the effects of the original action as well as those preconditions that are not touched by the effects. The remaining original preconditions are the effects of the rearranged action.

When an applicable action is instantiated, it inherits the previous action's start time point as its end time point and a new time point is created to represent its start time. The start time is constrained (step 12) to occur before the end time, according to the action's duration, and after the time we are predicted to be done planning this job. If there are no actions already in the plan, then the action gets its own end time point, which is constrained to occur after the end point of the previous job in this batch, if any.

The action's resource allocations are then posted and checked, using the same temporal database as for action times. Many high-performance schedulers 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 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.

The evaluation function used to evaluate the promise of a partial plan (step 8) is our estimate of the earliest possible end time of the partial plan's best completion. In the event of ties, the makespan of the plan is minimized. To improve our estimates of these quantities, we compute a simple lower bound on the additional makespan required to complete the current plan. We use a scheme similar to the h^1_T heuristic of Haslum and Geffner (2001) that estimates the fastest way to achieve each of the preconditions P_S of the earliest action in the plan, ignoring negative interactions and resource constraints. 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. We ignore negative interactions between actions by not deleting any literals whose negation is produced. The new literals may help enable new actions, which are then applied, possibly producing yet further new literals. This process continues until either all preconditions P_S 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 preconditions (infinity if a precondition cannot be produced).

This lower bound is then inserted before the first action in the plan and after the earliest plan start time, and may thus change the end time of the plan in addition to the makespan. It may also introduce an inconsistency, in which case we can safely abandon the plan. Any remaining ties between search nodes after considering end time and makespan are broken in favor of the node that had the larger realized makespan so far before the addition of the lower bound. A plan is considered complete if its literals unify with the desired initial state (step 9).

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 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 the jobs finish in the correct order. This can often provide important productivity gains.

Additional Features

Our implementation extends 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. Also, some actions allow their duration to be specified within a given range. Although this is easily accommodated by our framework, we currently ignore this flexibility and model all actions as having a fixed duration.

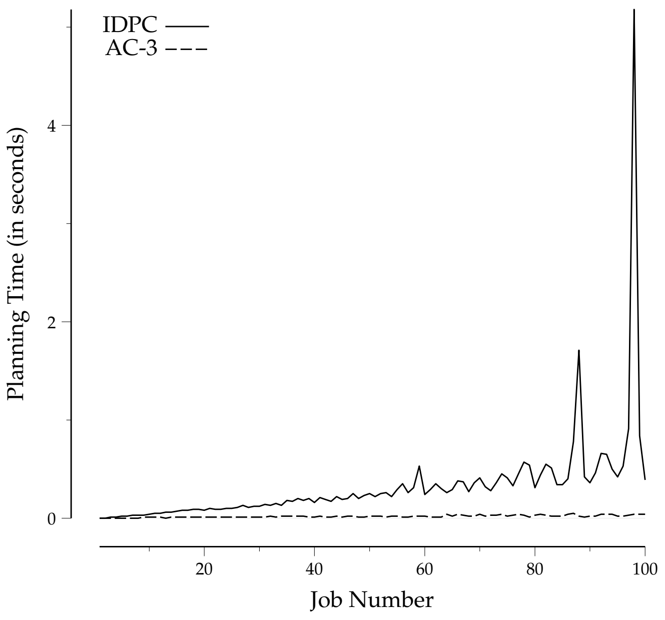

Figure 6: Simple arc consistency is faster than incremental directed path consistency.

Empirical Evaluation

In collaboration with our industrial clients, we have deployed the planner to control two physical prototype plants. These deployments have been successful. To give a sense of the performance of our implementation, we present simulation results on a variety of plants. Figure 6 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 possible actions. Plans 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 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 relations between arbitrary time points, but this is rarely needed during planning. New arcs are never added to the network during propagation, 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

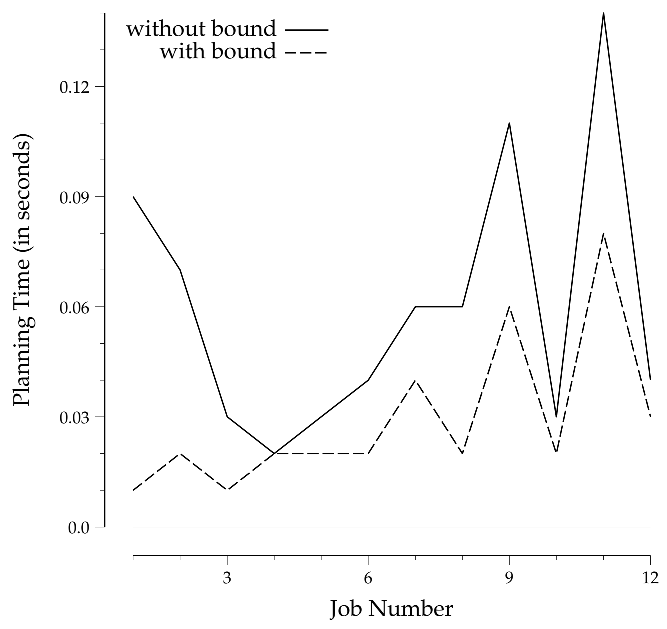

Figure 7: Heuristic guidance helps even for a small plant.

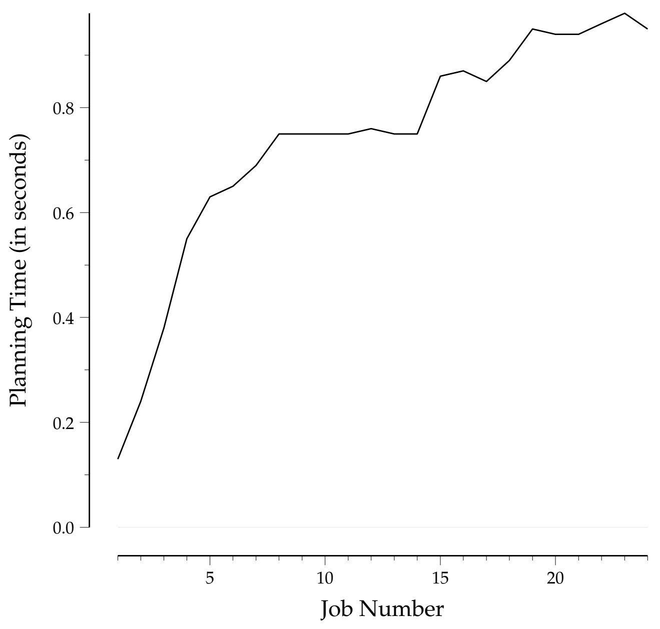

Figure 8: Running times increase, but do not necessarily explode for large plants.

longer than 50 milliseconds. The largest spikes for IDPC may be due in part to heavy demands on the garbage collector (the planner is written in Objective Caml, which features automatic memory management).

We also evaluated the contribution of the lower bound computations in guiding the search. Figure 7 shows planning time in a slightly more complex plant, with 16 machines and transports, yielding a domain with 73 possible actions. Plans here typically involve five to ten actions. The lower bound clearly improves planning time.

Finally, we present in Figure 8 preliminary measurements of planning time using a large simulated plant model with 104 machines and transports, totaling 728 possible actions. Plans here typically involve over 30 actions. This test used very 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 further implementation tuning, we should be able to handle domains of this size.

While these results 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.

Discussion

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.

In the future, we would like to take some mutex relations into account using something similar to the H_T^2 heuristic of Haslum and Geffner (2001) or the temporal planning graph of Smith and Weld (1999). 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 failure 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 have described a real-world manufacturing domain that requires integrated on-line planning and scheduling and formalized it using a temporal extension of STRIPS that falls between partial-order scheduling and temporal PDDL. We introduced 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 up to a dozen jobs per second.

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.

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.

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*.

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.