Cyclic Scheduling

Denise L. Draper*
Harlequin Incorporated
1201 Third Avenue, Suite 2380
Seattle, WA 98101
ddraperSharlequin.com

Ari K. Jonsson*
RIACS
NASA Ames Research Center, M/S 269-2
Moffett Field, CA 94035
j onssonOptolemy.arc.nasa.gov

David P. Clements
Computational Intelligence Research Laboratory
University of Oregon
Eugene, OR 97403-1269
clementsQcirl.uoregon.edu

David E. Joslin
i2 Technologies
909 E. Las Colinas Blvd.
Irving, TX 75038
david_joslin<Bi2. com

Abstract

In this paper we consider the problem of cyclic schedules such as arise in manufacturing. We introduce a new formulation of this problem that is a very simple modification of a standard job shop scheduling formulation, and which enables us to use existing constraint reasoning techniques to generate cyclic schedules. We present evidence for the effectiveness of this formulation, and describe extensions for handling multiple-capacity resources and for recovering from breaks in cyclic schedules.

1 Introduction

A cyclic scheduling problem is a scheduling problem in which some set of activities is to be repeated an indefinite number of times, and it is desired that the sequence be repeating. Cyclic scheduling problems arise in domains such as manufacturing, time-sharing of processors in embedded systems, and in compilers for scheduling loop operations for parallel or pipelined architectures. In this paper, we will take the manufacturing domain as our motivation and address the cyclic version of the job shop problem, in which a set of tasks (which describe the building of a single widget) are to be scheduled into a cyclic schedule for a widget factory.

Cyclic scheduling has not received much attention in the AI community, but there is a considerable body of work available in the OR literature; [Hanen and Munier, 1995] gives an excellent overview. Many heuristic approaches have been suggested for particular problems such as hoist scheduling, but there is also work on representations for general job shop problems, such as [Hanen, 1994] and [Roundy, 1992]. We do not have the space in this paper to fully describe their approaches, but the common element involves building a special-purpose data structure which represents the problem, and solving it using techniques such as branch-and-bound search or mixed integer programming.

In this paper, we will describe a different approach to the cyclic scheduling problem, one which is a conceptually simple extension of the constraint-based formulation of scheduling problems that has enjoyed much recent success. The main advantages to using such a constraint-based framework are the availability of existing techniques and the extendability of constraint-based representations. This allows us to transparently exploit the power available in any of the modern constraint satisfaction search engines, utilize other constraint reasoning techniques such as constraint propagation and consistency checks, and use the problem formulation within larger constraint decision problems. We also find that the formulation lends itself to the use of methods for recovering schedules after a failure, and that it can be extended to handle more complex problems such as resource constrained project scheduling.

We will start with a brief description of the now-standard constraint formulation of job shop scheduling, and then go on to present our extension to handle cyclic scheduling problems. We then present experimental results on solving such cyclic scheduling problems, and briefly discuss two extensions to our approach.

*This work was done while this author was at Rockwell Palo Alto Lab.

t Some of this author's contributions to this work were made while at Rockwell and with support through CIRL.

2 Job Shop Scheduling

The job shop scheduling problem can be found in many standard texts (e.g. [Baker, 1974]); the problem is specified by a set of tasks, T, and a set of resources, *R.* There is a function on tasks, *dur(t),* specifying a non-zero duration for each task, and a predicate *u(t,r)* specifying whether task *t* uses resource r.[1] Finally, there is a set of precedence constraints, where each constraint $t_i < t_j$ specifies that the task t_i must be completed before the task t_j can begin. A valid schedule is an assignment of start times to each task such that the precedence constraints are obeyed and no two tasks require the same resource at the same time. The completion time of the final task in the schedule is called the *makespan* of the schedule. The goal of scheduling can either be satisfying (find a schedule whose makespan is at most D), or optimizing (find a schedule with minimal makespan).

The satisficing scheduling problem can be encoded as a constraint satisfaction problem, and solved using search or a combination of constraint propagation and search, and the optimizing scheduling problem can be solved with additional search or branch-and-bound over possible makespans. Early approaches focused on searching over the space of start times for tasks. More recent approaches have found it more efficient to search over ordering decisions between tasks—once sufficient orderings have been added to guarantee that the resource constraints have been satisfied, it is simple to determine a minimal start time for each task consistent with those orderings.

3 Cyclic Job Shop Scheduling

In the job shop scheduling problem, the set of tasks is fixed as given. For the cyclic job shop problem, we assume that the set of tasks is a template which we wish to repeat indefinitely—for example, the tasks represent the steps to build a single widget, and we wish to construct a schedule for a widget factory. In order to make efficient use of our resources, we will want to overlap the manufacture of multiple widgets. A cyclic schedule for building widgets is one in which a new widget is begun every *K* time units (the cycle time), and the same schedule of tasks is completed for each widget. Assuming the time to complete each individual widget is greater than *K,* the result is a pipeline in which multiple widgets are under construction at any one time.

It is not the case that a cyclic schedule is necessarily the most efficient schedule; for any fixed number of widgets, TV, a non-cyclic schedule for building *N* widgets might very well exist that is more efficient than any cyclic schedule [Hanen, 1994]. Even if we do not know *N* in advance, we could still use an 7V-widget schedule as an approximation, repeating it as necessary. There are two significant advantages to using cyclic schedules, however.

*Note that this approach to representing resource usage allows for tasks requiring more than one resource, which is a commonly used extension to the standard job shop problem.

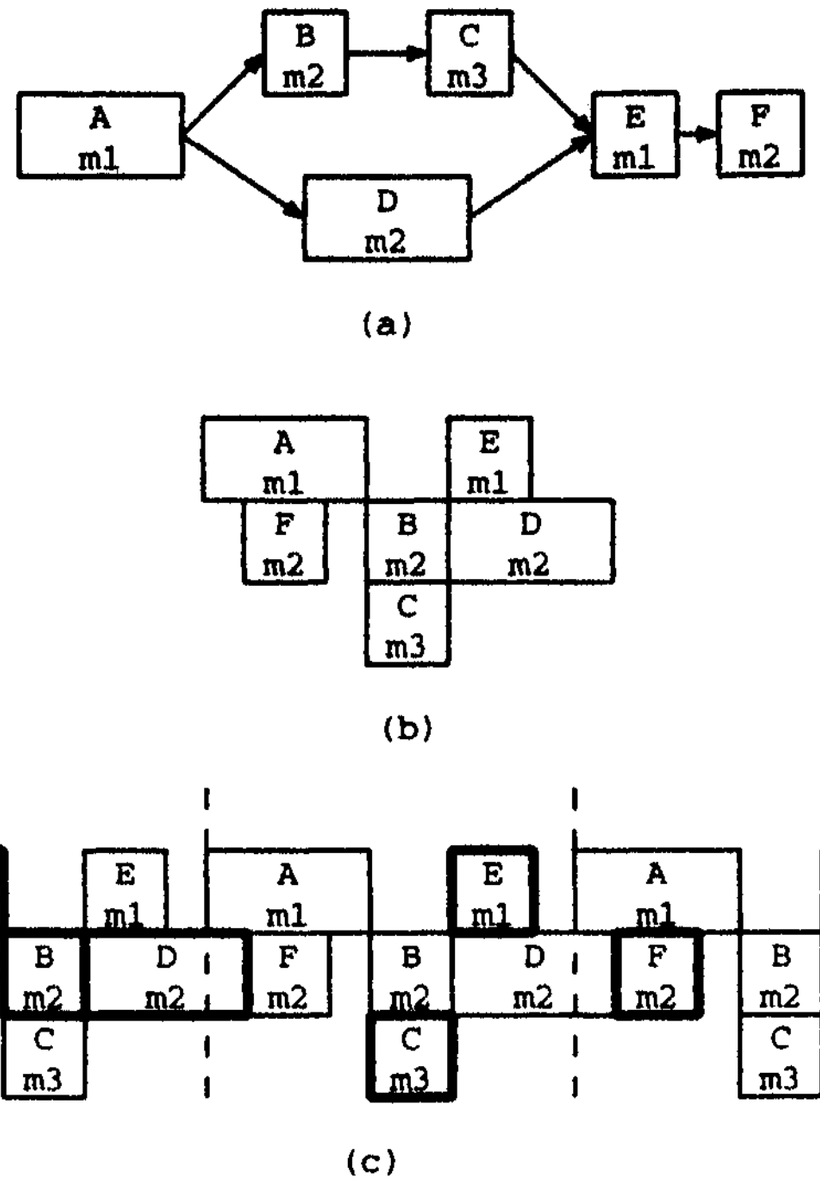

Figure 1: A scheduling problem and a cyclic solution. (a) shows a cyclic problem: each box is a task with a name and resource required, the size of the box indicates task duration and the arcs indicate required precedence. (b) shows an arbitrary assignment of starting times to tasks that respects resource constraints, (c) shows this assignment interpreted as a cyclic schedule; the boxes in bold are the instances of each task used to build a single widget.

The first is that cyclic schedules are much easier to implement: it is easier to communicate a short, repeatable sequence of actions to the agents which must carry them out, than to specify a much longer, non-repeating sequence. Secondly, the cost of computing optimal schedules for *N* widgets grows exponentially with AT, while the cost of finding a single widget cyclic schedule to make TV widgets is independent of *N.* Furthermore, as we shall see, the complexity of our cyclic algorithm has in the worst case only an extra logarithmic factor over the basic job shop algorithm.[2]

In order to explain our cyclic scheduling formulation, we will start at the end, with some observations on cyclic schedules. The main observation is that *any assignment of start times to tasks that satisfies the resource constraints can be converted into a cyclic schedule.* For example, consider the scheduling problem specified in Figure 1(a). Figure 1(b) shows an assignment of these tasks to resources. To convert this assignment into a

[2]An interesting compromise between computation cost and optimality is to build cyclic schedules for building two or more widgets at a time.

cyclic schedule, *paste* copies of the basic cycle together into a strip in such a way that they do not overlap—this induces a cyclic schedule with some particular cycle time, as shown in Figure 1(c). Then, no matter what task ordering was chosen, or what cycle time is induced, it is possible to overlay a complete schedule for one single widget such that the precedence constraints are satisfied; this is done in bold in Figure 1(c). A simple counting argument shows that this is a proper cyclic schedule for this set of tasks: every task is fulfilled, and there are no tasks left over.

This exercise demonstrates several things. First, whereas the makespan is the only measurement of interest in the traditional job shop schedule, in the cyclic case we have two: the cycle time *K,* and the *latency,* L, which is the number of cycles required to complete a single widget (which is also equal to the number of widgets in production simultaneously).

Secondly, notice how the precedence constraints were satisfied. For any precedence constraint $t_i < t_j$, it is either the case that t_i and t_j occur within the same cycle, in which case the 'normal' precedence holds between them, or it is the case that t_i is in one cycle, and t_j is in a subsequent cycle. (In Figure 1, the precedence constraints *A* < *B, A* < *D, C* < *E,* and *D* < *E* are satisfied 'normally/ while constraints *B* < *C* and *E* < *F* cross cycle boundaries.)[3] The point is: in order to satisfy a constraint $t_i < t_j$, we can either order t_i before t_j within a cycle, or *defer* the execution of t_j to the subsequent cycle. This observation is the basis of our cyclic scheduling formulation: we will restrict our attention to scheduling of tasks *within a single cycle,* modifying the definition of the precedence constraint to allow for the two different ways in which it can be satisfied.

With this formulation, we can employ any standard search procedure. There are no new decision variables: we search only over orderings of steps within a cycle. We can do satisficing search for both a given cycle time *K* and latency L, or we can optimize cycle time for fixed latency or vice versa, or we can search for a set of dominating solutions in both *K* and L.

We will now lay out our constraint encoding formally, and introduce some additional variables and consistency checks to avoid redundant or useless search.

The inputs axe:

- $t_1, \ldots, t_n$: the set of tasks,
- $r_1, \ldots, r_m$: the set of resources,
- *dur(t):* the duration of each task,
- *u(t,r):* usage predicate for each task/resource pair,
- $\{t_i < t_j, \ldots\}$: a set of precedence constraints on task pairs,
- *K:* the cycle time,
- L_{max}: maximum allowable latency (optional).

The decision variables we search over are:

- $o(t_i, t_j)$: ordering predicate on pairs of tasks within a single cycle.

We use the following non-decision variables for consistency checking:

- *8t(t):* start time of *t wrt* the beginning of a cycle,
- *cs(t):* cycle in which the first instance of *t* starts,
- *cf(t):* cycle in which the first instance of *t* finishes.

Cycles are numbered from one at the beginning of the schedule; *cs* and *cf* measure the positions of the begin and end of each task for the first widget (e.g. in Figure 1 we have: *cs{D)* = 1, *cf{D)* = 2, and *cs{F)* = 3). Note that *cs* and *cf* are only required if there actually is a desired limit on latency, which we will assume to be the case.

For conciseness, we will add notation to represent the condition that a task crosses the cycle boundary:

$$cross(t) \equiv st(t) + dur(t) \geq K.$$

The constraints we must satisfy are as follows:

1. The start time of each task is within the cycle:

$$0 \leq st(t) < K.$$

2. If task t_i is ordered before task t_j, then the termination of t_i in the cycle occurs before the start of t_j in the cycle.

$$o(t_i, t_j) \rightarrow st(t_j) > (st(t_i) + dur(t_i)) \bmod K.$$

   It is possible for both $o(t_i, t_j)$ a $o(t_j, t_i)$ o hold, if either task crosses the cycle boundary.

3. If two tasks use the same resource, one must be ordered before the other; if either task crosses the cycle boundary, they must be ordered in both directions, and both tasks cannot cross the cycle boundary.

$$\begin{aligned} &i \neq j \wedge u(t_i, r) \wedge u(t_j, r) \rightarrow \\ &\quad (o(t_i, t_j) \vee o(t_j, t_i)) \wedge \\ &\quad ((cross(t_i) \vee cross(t_j)) \rightarrow o(t_i, t_j) \wedge o(t_j, t_i)) \wedge \\ &\quad (\neg cross(t_i) \vee \neg cross(t_j)). \end{aligned}$$

4. If a precedence constraint $t_i < t_j$ exists, then either t_i is ordered before t_j, or t_i is deferred with respect to t_i, which means that t_j must start in a cycle that comes after the cycle in which t_i finishes.

$$(t_i < t_j) \rightarrow o(t_i, t_j) \vee (cs(t_j) > cf(t_i))$$

5. If a task crosses a cycle boundary, its finish cycle number is one greater than its start cycle number; otherwise they are the same.

$$\begin{aligned} cross(t) &\rightarrow cf(t) = cs(t) + 1, \\ \neg cross(t) &\rightarrow cf(t) = cs(t). \end{aligned}$$

6. The cycle numbers must be between one and L_{max}'

$$\begin{aligned} 1 \leq cs(t) \leq L_{max}, \\ 1 \leq cf(t) \leq L_{max}. \end{aligned}$$

[3] We have to be careful about tasks which overlap cycle boundaries, such as task *D;* we will address this below.

When solving this problem, we initially assign possible ranges to the variables: $st(t) \in [0, K)$, $cs(t), cf(t) \in \{1, \ldots, L_{max}\}$, and $o(t_i, t_j) \in \{true, false\}$. Then we search over assignments to $o(t_i, t_j)$, using consistency propagation techniques to prune the possible values of all the variables. When enough ordering variables have been set such that all the required constraints are guaranteed to hold, we take start times and cycle numbers to be the minimal of their remaining possible values.

This description is complete, but misses some important details concerning the efficient pruning of the supporting variables.

In the standard job shop constraint formulation it is customary to keep, in place of a set-valued st(t), two variables—earliest possible start time, *est(t)* and latest possible start time *lst(t)*—which represent the end points of *st(t).* The value of $est(t_j)$ can be efficiently updated by using the rule

$$\forall t_i : o(t_i, t_j), est(t_j) \geq est(t_i) + dur(t_i)$$

(and a symmetric rule for *lst(t)).* The advantage of this representation is that the updated values for *est(t)* and *lst{t)* can be computed quadratically in the number of tasks by traversing them in topological order (for *est(t))* and reverse topological order (for *lst(t)).*

In our formulation of the cyclic scheduling problem, we also use *est(t)* and *lst(t)* to represent the end points of *8t(t),* and we use essentially the same rules for updating these variables, except that we now must account for the cycle boundary by introducing a mod *K.* But, since the ordering decisions $o(t_i,t_j)$ may have a cycle, the propagation may not terminate at the end of the schedule, as it does for standard job shop problems. For an example, consider a task *A* that crosses the cycle boundary and as a result forces an increase in the earliest start time for another task *B.* If moving *B* then ultimately results in changing *est(A),* the standard propagation will continue propagating this cycle, until *est(A)* reaches *K*. However, we note that when the task sequence for a resource is moved more than once, the current schedule is proven to be infeasable and the propagation can therefore be halted immediately.

Our treatment of the *cs(t)* and *cf(t)* follows a similar pattern: we keep two variables to implement the lower and upper bounds on each quantity. To update the values for *ecs(t)* and *ecf(t),* we initially assign each variable to zero, then do two passes. In the first pass, *ecf(t)* is updated using constraint 5 above—that is,

$$est(t) + dur(t) > K \rightarrow ecf(t) = ecs(t) + 1$$

In the second pass, we consider the precedence constraints in topological order. For each precedence constraint, we enforce the following constraints:

$$\begin{aligned} (t_i < t_j) \wedge \neg o(t_i, t_j) &\rightarrow ecs(t_j) \geq ecf(t_i) + 1 \\ (t_i < t_j) &\rightarrow ecs(t_j) \geq ecf(t_i) \end{aligned}$$

Each time one of these constraints increments ecs(t), we reinforce constraint 5 by incrementing *ecf(t)* by the same amount.

The computation of *lcs(t)* and *lcf{t)* follow a symmetric pattern, initializing from constraint 5 and *lst(t),* and updating by considering the precedence relations in reverse topological order.

The algorithm described above produces a schedule for a fixed cycle length *K*. In order to optimize over *K,* we must iterate the scheduling process over possible values of *K.* An upper bound on *K* is given by the makespan of the problem, treated as an ordinary job shop scheduling problem (which could be bounded or approximated by any number of means). A lower bound for *K* is given by the largest total time requirement for any one resource (at which point that resource is 100% utilized).

The worst-case complexity of this algorithm, for a single value of *K,* clearly differs from the complexity of the ordinary job shop scheduler by only a small constant factor (required to process the additional constraints for computing cycle numbers). If we assume that the cycle time and the task durations are all integers, we can bound the worst case time required to search over multiple values of *K* by $T \log(K_{max} - K_{min})$, where *T* is the time to invoke the scheduler for a single instance of *K.*

4 Experimental Results

It is clear that cyclic schedules have a number of advantages over non-cyclic schedules in applications where schedules are repeated. The question is whether the cost of generating cyclic schedules is reasonable enough that these advantages can be realized. To verify that this is indeed the case for our cyclic scheduling formulation, we compare the cost of solving problems using our formulation to the cost of solving the same problems using the smaller classical formulation, with the goal of minimizing the cycle length. The comparison is done by applying an optimization search method (branch-and-bound with a time cutoff) to each formulation of a given problem and comparing the resulting cycle length. For the standard formulation, we calculate the cycle length as the longest distance between first and last use of a resource, since a new widget can be started at those intervals.[4]

The scheduling problems used in this comparison are from Norman Sadeh's scheduling test suite [Sadeh, 1992]. Disregarding the makespan limits, which are irrelevant when minimizing cycle time, the suite gives us 20 different problem instances. Each problem has 50 tasks, each task uses exactly one resource and there are 10 resources in all. It should be noted that these particular problems are not all that hard for modern search techniques and well-honed heuristics [Crawford and Baker, 1994]. But, the question here is not whether our formulation outperforms existing engines or heuristics for job shop scheduling; the question is whether this larger cyclic constraint formulation can be solved efficiently enough

[4]Note that for the standard formulation, the cycle length also replaces makespan in all pruning and heuristic calculations. In all other aspects, the search proceeds as usual; ordering decisions are made such that resource and precedence constraints are satisfied.

that the benefits still outweigh the possible increase in solving time. For answering that question, these scheduling problems are a perfectly suitable testbed.

To solve the two different formulations of the scheduling problems in the same manner, we use the gensolve system [J6nsson, 1997], which is currently a prototype of a general constraint satisfaction system that can use arbitrary combinations of procedural propagation methods to speed up the search effort. In this system, problems are represented by providing the decision variables, the constraints on value assignments, and procedures that perform propagation. For both scheduling formulations we use procedures to determine the possible start times for tasks; in the regular formulation we use the standard propagation of earliest and latest start times, while in the cyclic formulation we use the propagation algorithm described above (with a latency limit of $L_{max} = 2$). In both formulations we use the standard slack-based heuristic described in [Smith and Cheng, 1993].

The key results of this comparison are tabulated in Table 1. The results clearly demonstrate that the cyclic scheduling formulation of these problems can be solved quite effectively by standard constraint solvers. Furthermore, the results show that when it comes to scheduling for cyclic applications, using this new formulation provides significantly better results than using the standard job shop formulation.

	Cyclic	Regular
Average Cycle Length	122.2	135.5
Average Distance from Optimal	0.1%	10.8%
Number of Optimal Solutions	19/20	0/20
Average runtime (seconds)	99	1800

Table 1: A comparison of the effectiveness of minimizing the cycle time using the new cyclic scheduling formulation and doing the same using the standard scheduling formulation. Each problem instance was solved using a simple version of branch-and-bound search, on an UltraSparc-II, with a time limit of thirty minutes.

5 Extensions

One of the advantages that our approach to cyclic scheduling has over existing special-purpose algorithms is that a general constraint satisfaction formulation can be extended and adapted much more easily. As examples of this, we will briefly describe how our approach can be extended to the more general class of cyclic resource-constrained project scheduling problems, and how it can be adapted to provide an approach to recover from failures in the exectuion of cyclic schedules.

In real-world manufacturing scheduling there are typically resources that have capacity greater than one (e.g. a pool of skilled labor), tasks may require multiple instances of each resource type and each task may use multiple resource types. These characteristics have been formalized as the resource constrained project scheduling (RCPS) problem [Blazewicz *et al,* 1983], which is considered significantly more difficult than the job shop problem.

We would like to extend our cyclic formulation to create a solution for the cyclic RCPS problem. We will do this by adapting an existing approach to solving such problems. This approach is based on initially scheduling the tasks without any regard to the resource constraints, but respecting the precedence constraints, and then eliminating resource constraint violations by assigning values to corresponding ordering decision Liabks [Crawford, 1996]. First, let us point out conditions that have changed from the job shop problem:

- Tasks that use the same resource r do not necessarily need to be ordered with respect to each other, if the sum of their usage of r is less than the total availability of r.
- Tasks can have a duration that exceeds the cycle length *K,* provided that for each task *t* that uses some resource r with capacity c, we have $u(t,r) \leq c$ and
$$\lceil dur(t)/K \rceil \cdot u(t,r) \leq c$$
As a result, a task's cycle finish number can be more than one greater that its cycle start number.
- The lower bound for *K* is given by the largest total time requirement for any resource divided by its capacity.

In the cyclic formulation of the above-mentioned approach, all the tasks are initially scheduled at the beginning of the cycle, without any regard to either resource constraints or precedence constraints. The resource constraint violations are then eliminated as before, while precedence constraint violations are resolved by either making the appropriate ordering decision within the cycle or by defering the second task to a later cycle. The impact of each ordering decision is then propagated to restrict the bounds on start times and cycle numbers for later tasks, in a similar fashion as for the cyclic job shop scheduling formulation. Just as for the standard RCPS problems, systematic backtracking methods such as *limited discrepancy search* [Harvey and Ginsberg, 1995], or nonsystematic repair methods like *doubleback optimization* [Crawford, 1996], can then be used to explore the space of decisions.

Turning our attention to schedule recovery, it is well known that optimal or near optimal schedules lack tolerance for delays, and thus are easily broken when delays do occur. In the non-cyclic case, we can generate a new schedule from the point of disruption, either from scratch or by modifying the old schedule. However, since one of the goals of cyclic scheduling is regularity, generating an entirely new schedule is not a desirable option. In the cyclic case, we want to focus on returning as quickly as possible to the already-established schedule.

To do this, we consider a *rescheduling window,* which covers from when the disruption occurs to when the

cyclic schedule will be restored. Within it tasks will not follow the repeating pattern used elsewhere in the schedule. The rescheduling window may have jagged left and right edges; any tasks running at the time of the disruption, but not affected by it, jut into the window from the left side, and any tasks in progress when the cyclic schedule is restored jut into the window from the right side.

The right edge of the window can be determined by incrementing over possible times until a window that is big enough to accommodate the rescheduling is found. In some cases this may be achieved without pushing back the entire schedule. However, if the disruption is big enough, it may not be possible to recover unless the entire schedule is pushed back, thus allowing the recovery more time in which to place the schedule. Once a window has been established, any search technique can be used to schedule the tasks within it.

6 Conclusion and Related Work

In this paper we have developed a formulation for solving cyclic job shop scheduling problems as constraint satisfaction problems. We have implemented this formulation and shown that it can be used to generate good cyclic schedules, using standard constraint satisfaction methods. We have also demonstrated that for the purpose of finding cyclic schedules, the use of our formulation outperforms the use of standard job shop formulations, when the same search technique and shceduling heuristic are applied to both. In addition to this, we have described how our formulation can be generalized to cyclic resource constrained project schedules and how the formulation allows us to develop methods for recovering from failures in the execution of cyclic schedules.

As mentioned in the introduction, there are several formulations for cyclic job shop scheduling in the OR community. Furthermore, there exist specific cyclic scheduling techniques in industry, e.g, those used to schedule update cycles on recent Honeywell avionics [Boddy and Goldman, 1994]. We have not been able to do performance comparisons between our approach and other techniques, but the formulation presented here has other clear advantages in terms of understandability, and in terms of being able to exploit existing constraint reasoning techniques and heuristics. Perhaps most importantly, based on this simple constraint representation, further research can build on this formulation. For example, by adapting it to more complex scheduling problems —as we have started doing with the cyclic RCPS problem.

On the other hand, the OR techniques typically handle more complex types of precedence constraints than we do: it is allowed for a task t_j to depend on the completion of a task t_i from a different iteration—this is needed for the compiler problem where a loop may contain a statement of the form $x[i] := f(y[i-k])$. This kind of precedence does not seem to occur in manufacturing problems, our primary interest, but it would nonetheless be interesting to see if this representation can be extended to handle such more general forms of precedence.

References

[Baker, 1974) K. R. Baker. *Introduction to Sequencing and Scheduling.* Wiley, New York, 1974.

[Blazewicz *et al,* 1983] J. Blazewicz, J. K. Lenstra, and A. H. G. Rinnooy Kan. Scheduling subject to resource constraints: Classification and complexity. *Discrete Applied Mathematics,* 5:11-24, 1983.

[Boddy and Goldman, 1994) Mark S. Boddy and Robert P. Goldman. Empirical results on scheduling and dynamic backtracking. In *Proceedings of the International Symposium on Artificial Intelligence, Robotics, and Automation for Space (ISAIR AS),* 1994.

[Crawford and Baker, 1994] James M. Crawford and Andrew B. Baker. Experimental results on the application of satisfiability algorithms to scheduling problems. In *Proceedings of the Twelfth National Conference on Artificial Intelligence,* 1994.

[Crawford, 1996] James M. Crawford. An approach to resource constrained project scheduling. In G. F. Luger, editor, *Artificial Intelligence and Manufacturing Research Planning Workshop,* Albuquerque, New Mexico, 1996. The AAAI Press.

[Hanen and Munier, 1995] C. Hanen and A. Munier. Cyclic scheduling on parallel processors: An overview. In P. Chretienne, E. G. Coffman, Jr., J. K. Lenstra, and Z. Liu, editors, *Scheduling Theory and its Applications,* chapter 9. John Wiley & Sons, 1995.

[Hanen, 1994] Claire Hanen. Study of a NP-hard cyclic scheduling problem: The recurrent job-shop. *European Journal of Operational Research,* 72:82-101, 1994.

[Harvey and Ginsberg, 1995] W. D. Harvey and M. L. Ginsberg. Limited discrepancy search. In *Proceedings of the Fourteenth International Joint Conference on Artificial Intelligence (IJCAI-95),* pages 607-613, 1995.

[J6nsson, 1997] Ari K. J6nsson. *Procedural Reasoning in Constraint Satisfaction.* PhD thesis, Stanford University, Stanford, CA, 1997.

[Roundy, 1992] Robin Roundy. Cyclic schedules for job shops with identical jobs. *Mathematics of Operations Research,* 17(4):842-865, November 1992.

[Sadeh, 1992] Norman Sadeh. Look-ahead techniques for micro-opportunistic job shop scheduling. Technical Report CMU-CS-91-102, School of Computer Science, Carnegie Mellon University, 1992.

[Smith and Cheng, 1993] Stephen F. Smith and Cheng-Chung Cheng. Slack-based heuristics for constraint satisfaction scheduling. In *Proceedings of the Eleventh National Conference on Artificial Intelligence,* pages 139-44, 1993.