Proceedings of the 1991 IEEE International Conference on Robotics and Automation, Sacramento, CA - April 1991, pp. 810-17

Real-Time Motion Scheduling for a SMALL Workcell

Gerardo Pardo-Castellote* Henrique A. S. Martins

Hewlett-Packard Laboratories
Palo Alto, CA 94304

Abstract

A high-level controller for a SMALL workcell (Sawyer-Motor Assembly workceLL) is presented. It allows multiple independent processes to share the robot modules and workspace of a single workcell, by planning and scheduling collision-free paths for module motions in real time. Planning may involve moving around other modules, stationary or in motion, waiting for motions to complete, or displacing obstructing *idle* modules.

The controller handles any number of robots in simultaneous motion along paths consisting of multiple rectilinear segments. Collision avoidance and path planning are addressed in two dimensions, with extensions to three dimensions briefly discussed. Design considerations for the algorithms involve throughput, protection, fairness, and starvation considerations. Capabilities are provided to allow independent processes to cooperate without interference.

1 Introduction

Multiple robots sharing the same workspace raise problems of collision avoidance and path planning. Substantial work has been done in these areas, involving both single and multiple robots. Proposed solutions have used different methods, such as configuration space [1] or potential field [2] methods. Complete planners for the most general case [3, 4] have also been addressed.

Commonly discussed solutions are usually not practical to implement, since they frequently tackle the most general case, involving multiple degrees-of-freedom robots, with a corresponding sacrifice in efficiency. The most general planners have running times exponential to the number of degrees of freedom. The majority of the work done also assumes complete control of the position, velocity, and acceleration for each planned motion. A good reference for the different approaches to robot motion planning can be found in [5].

Another area of concern in a multi-robot environment is task planning and scheduling [6, 7]. Most controllers assume that scheduling and planning are done off-line, or by a *know-all-does-all* controller, rather than in real time with limited task level knowledge.

SMALL-type workcells, i.e., multi-robot workcells based on Sawyer-motor technology [8], have been introduced in the past few years by Automatix Inc. [9], AT&T Bell Laboratories [10], Megamation [11], among others. These workcells consist of multiple air-bearing-supported magnetically adhered miniature robot modules that ride on the bottom of a horizontal surface or *platen*. The modules are equipped with umbilical cords that carry power, electric signals, and air.

The SMALL domain is sufficiently unique to warrant dedicated solutions to the above problems that are both simple and fast. Previous work has addressed other aspects of this architecture [12, 13]. In section 2 we describe our SMALL workcell controller. In section 3, we present the motion scheduler. In sections 4 and 5 we describe the collision-avoidance and path-planning computation algorithms that allow these tasks to be performed in real time. In section 6 we perform a brief complexity analysis of some of the algorithms. In section 7 we describe the kinematic simulator used by the planner. In section 8 we present simulation results, and conclude with planned extensions to our work.

2 The SMALL Controller

In a typical SMALL workcell several tasks can be performed simultaneously, competing for the use of the same modules and/or work areas. These tasks can be multiple instances of the same task, cooperating tasks, or totally independent tasks, each unaware of the existence of the others. It is desirable to provide an environment in which such tasks can co-exist without human programmer intervention to prevent task conflicts.

At Hewlett-Packard Laboratories we built such an environment on top of an Automatix Inc. low-level motion controller. The robot tasks are developed and executed on an HP 9000 general-purpose UNIX workstation, which interfaces with the motion controller. A dedicated user interface was built on top of MIT's X Window System and OSF's MOTIF widget set.

In our controller only one process, the *daemon*, is allowed to talk to the low-level controller; all other processes communicate with the hardware via this dae-

*Gerardo Pardo-Castellote is a Ph.D. candidate in the Department of Electrical Engineering, Stanford University. His work was performed as a SEED student at Hewlett-Packard Laboratories.

mon. The daemon is in charge of scheduling and servicing all motion requests from other tasks, and of performing collision avoidance and path planning.

In this architecture only the daemon knows about the actual hardware; all other tasks talk to the daemon in a hardware-independent language. Application tasks are thereby buffered from changes in the low-level controller that will affect only the daemon.

All motions are initiated by the daemon. To move a particular module to a destination, the daemon searches for a suitable path in the presence of possible simultaneous motions of the other modules. If such a path is not feasible, the daemon should attempt to displace *idle* modules, which may cause cascading displacements of other modules.

3 Scheduler

3.1 Scheduling Philosophy

The daemon implements a real-time on-line path planner, and has no knowledge of the multiple individual tasks that are running. (This is very similar to an operating system which has no knowledge of the actions of individual user processes running in a multi-tasking environment but merely provides orderly use of shared resources, avoiding conflicts and increasing the total throughput of the system.)

To solve the scheduling problem for SMALL we need to create a suitable abstraction that will allow user processes to request motions for robot modules and restrict the actions of the daemon. The goal is to permit several cooperating and/or independent processes to share the same physical space and modules without interference. We also need to construct a suitable algorithm to prioritize and grant individual requests, with main emphasis on throughput and fairness.

The daemon must provide means by which the individual processes can monitor the progress of their individual requests, and mechanisms to prevent other processes from interfering with specific modules or regions of the workspace. This is accomplished by qualifying each move request with one or more *attributes*. The following attributes are currently available:

Simple - A move is scheduled for execution as soon as possible.

Immediate - A move is executed immediately, if possible, or else discarded.

Notify - The requesting process will be notified upon completion of the move. A sequence of moves with notification requests may be pending, and may not be executed in order. Notification currently occurs through polling.

Non-disturbing - A move will be executed when it can proceed without requiring the daemon to displace other modules in its path.

Ownership request - Upon completion of the move the requesting process will *own* the module. This enables the module to be locked in place, immune from other processes and the daemon.

Ownership release - Upon completion of the move the process releases its lock on the module.

Straight-line - A move will be executed when it can proceed as a straight-line motion.

Since most of the above attributes are mutually independent, it is possible to request any combination of them. An obvious exception is a simultaneous ownership request and release.

3.2 Servicing Requests

The most simple and flexible solution is to let the daemon schedule moves in any order. In this scheme the daemon's role would be to queue all outstanding requests for each module and to schedule them accordingly. The primary disadvantage to this solution is that the client processes require more complicated control structures and spend time waiting for notifications.

The daemon needs to plan motions that take into account other potential moves, scheduled but not in progress, and avoid generating mutually incompatible paths. To tackle this problem, we introduce the concept of a *committed* move, as a move for which a path has already been planned and will not change, and that is guaranteed to occur in a collision-free environment at some time in the near future.

Committing a move allows it to be taken into account when planning moves for other modules, preventing collisions and allowing simultaneous motions. However, it might result in a non-optimal path, since at the time the move finally starts, shorter paths might have become available.

Once a process has obtained ownership of a module, its requests should be satisfied as soon as possible. This results in each module actually having two request queues: the *owner queue*, which contains the requests of the owner process, and the *others queue*, which contains the requests of the remaining processes. These queues are serviced as follows:

- Requests in the *others queue* are ignored while the module is owned by some process. As long as a process owns a module its requests are placed in the *owners queue* and serviced. Access to the module and the area it occupies is denied for the other processes.
- Requests for any given queue are satisfied on an availability basis, rather than in order, for the purpose of maximizing throughput.
- When an owner process releases a module, any requests in the *owner queue* are added to the *others queue*, to give equal access to the locked-out processes. This equal access is not guaranteed as the next possible move might again belong to the releasing process and be serviced.

Within this framework a particular module can be in one of the following states:

MOVING - The module is moving.

COMMITTED - The module is not MOVING, but a move has been committed.

WAITING - The module is not MOVING or COMMITTED, but there are outstanding requests.

IDLE - The module is not MOVING or COMMITTED and there are no outstanding motion requests to satisfy, i.e., either both queues are empty or the module is owned and the *owners queue* is empty.

We can look at an obstacle module under one of the following three perspectives: STATIC, IN-PROGRESS, and COMMITTED. The location of the obstacle module and the area it occupies, in each of those perspectives, depends on the state of the obstacle module. WAITING and IDLE modules are at their current positions for all perspectives. MOVING and COMMITTED modules are treated as follows:

- STATIC perspective - MOVING and COMMITTED modules are at their final destinations.
- IN-PROGRESS perspective - MOVING modules occupy the area swept from their current positions to their final destinations. COMMITTED modules with a higher precedence than the move being considered are treated as MOVING modules. (The definition of move precedence is given later.) COMMITTED modules with a lower precedence than the move being considered are treated as IDLE modules.
- COMMITTED perspective - MOVING and COMMITTED modules occupy the area swept out from their current positions to their final destinations.

Note that a *move* consists of a sequence of one or more straight-line segments that connect the origin and destination of a move request. Note also that the *current position* of the module is that computed by the planner using a kinematic simulator, as described in section 7.

Illegal motion requests, i.e., those with destinations outside the boundaries of the workspace or requiring other modules to move out of the workspace, are discarded and never queued. These requests would otherwise accumulate in the queues, as they can never be scheduled, and degrade the performance of the scheduling process, which could eventually run out of space.

3.3 Control Cycle

During the main control cycle, the daemon looks at the current state and outstanding queued requests of each module and proceeds accordingly, changing the state of each module if possible. Module state-transitions occur synchronously via periodic timer interrupts, asynchronously with the arrival of new requests and after the completion of motions. Requests are queued asynchronously whenever they arrive from the user processes. Before describing state transitions, we define the following additional terms:

Move - A move is defined by an origin, a destination, and a set of *via* points describing the path and composed of segments linking the origin and the destination. A move carries information about initial and goal positions and the path.

Move request - A destination for a module and a set of motion attributes. (A move request does not specify the initial position nor the path to be taken.)

Precedence of a move - Each move has an associated precedence. This precedence is significant only for moves not in progress, and imposes a partial ordering among these moves, since several moves can have the same precedence.

Compatible set of moves - A set of moves is said to be compatible if an only if it satisfies the following requirements:

- All moves in progress are along paths that are collision-free with respect to each other and the stationary modules. That is, all modules will arrive at their destinations without collisions.
- At least one move among the ones not in progress will become *viable*, i.e., collision-free, once all moves in progress terminate.

Planning state of the world - A set of modules, their current state, and moves associated with moving or committed modules.

Planned moves - The set of moves associated with the modules that are either moving or committed.

Consistent planning state - A planning state in which the set of planned moves is compatible.

Committed moves - Any planned move, in a consistent planning state, that is not currently in progress.

The goal of the daemon is to flush all the queues, i.e., satisfy all requests, and move all modules into the IDLE state, preserving at all times a consistent planning state. Thus, from any given consistent planning state of the world, the daemon has two tasks to perform:

- Scan the queues for a request to be committed, and commit whichever requests it can. This may involve simultaneous commitment of moves to displace other modules in its path.
- Start all committed moves that can proceed, while maintaining a consistent planning state.

3.3.1 Committing moves

The daemon must find, among all pending requests, a move that can take a module to the requested destination, leaving the world in a consistent planning state. To accomplish this, it sequentially attempts the following for each move request:

1. Find a path in a COMMITTED perspective of the world: If any is found, commit and start it immediately. Under this perspective all planned moves occupy the area swept out from their current positions to their final destinations. If a path is found under these circumstances, it will not interfere with any of the planned paths, thus maintaining a compatible set of moves. Obviously, the motion can also be started.
2. Find a path in the STATIC perspective of the world: If any exists, commit it. Under this perspective, all modules occupy the area corresponding to their final destinations, and thus the move can eventually be initiated. This is guaranteed by our strategy.

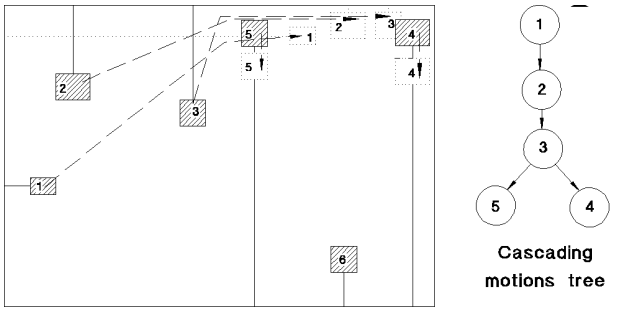

Figure 1: **Cascading motions generated by a request to move module 1: First move modules 4 and 5; then module 3; then module 2; and finally module 1.**

3. Start a recursive search. At each stage, obstructing modules will be identified in order to attempt to move them out of the way. (A module can be displaced only if it does not have a committed move and is not owned by some other process.) The move requests to displace modules in its path are initiated by the daemon, but are otherwise identical to other motion requests. The search builds a tree in which each node represents a destination to which a particular module must move. Its descendants represent destinations to which modules must be displaced to allow the parent displacement to occur. When traversing the tree the world is looked upon in the STATIC perspective. The traversal of the tree is depth-first without backtracking, i.e., as soon as one node fails the tree traversal fails. At any given node in the tree, modules corresponding to moves in already-visited nodes are represented at their destination. If the complete tree traversal succeeds, a post-order traversal of the tree gives the sequence of moves that provide a compatible set of motions, taking the module from the root of the tree to its destination. All these moves can be committed. It can easily be argued that a consistent planning state is also preserved. Fig. 1 shows a plan view of the workspace where a motion request causes the daemon to displace other modules in its path. The search tree is also shown.

3.3.2 Starting Committed Moves

We always start from a consistent planning state. There is at least one sequential ordering that will allow committed moves to proceed without collisions (in fact there might be many). The problem is to decide when a committed move can be started.

For each committed move, it is sufficient to look at the world from an IN-PROGRESS perspective. If a path can be found in that perspective, the move can be started, maintaining the consistent planning state. To understand why, recall that we introduced a precedence for each move, but never defined it. A global precedence index is maintained, and decremented, each time a request is transformed into a a set of committed moves. All moves cascading from a single request will be assigned its precedence. Moves generated by future requests will have lower precedence indices.

The significance of this precedence is to ensure that, in case of conflict, we preserve the partial ordering among the committed moves. In the IN-PROGRESS perspective, MOVING modules occupy the area swept out from their current position to their destination. Modules with COMMITTED moves and higher precedence also occupy the area swept out from their current position to their destination since, should there be a conflict, they should be executed before the current move can proceed. It is under these assumptions that the moves were committed in the first place. Modules with COMMITTED moves and the same or lower precedence are considered to be at their current position. WAITING and IDLE modules are also considered to be at their current positions.

4 Collision Avoidance

SMALL robot modules are boxes containing the Sawyer motors that provide horizontal motion. The boxes are cuboidal, about six inches across. Module motion is purely translational. Modules generally carry either end-effectors or video cameras.

The umbilical cable for each module rides approximately perpendicular to one edge of the workcell. The cables are treated as an integral rigid part of the module, forming a barrier to the motion of other modules. Collision checking and path planning is performed in the two-dimensional domain. Plans for extension to three dimensions and to more general obstacles is discussed in section 8.

4.1 Problem Formulation

Given a module, an origin, and a destination, the collision avoidance problem is to determine whether there is a collision-free *path* for the module linking the initial to the goal positions.

A path is a sequence of rectilinear motion segments defined by the sequence of via points that the module will visit in its travel from origin to destination. The

collision-checking program needs to know the *planning state of the world*, which consists of the number of modules, their geometrical models, the path each module is following, and their current locations (as determined by the kinematic simulator).

The collision checking is done in a virtual coordinate system. This is obtained by a coordinate transformation that makes the cable of the *planning module*, i.e., the module for which the path is being planned, extend along the negative y-axis of that system. This will be designated as the SOUTH orientation. The other modules, referred to as *obstacle modules*, will be designated as WEST, EAST, NORTH and SOUTH modules, depending on their relative cable orientation from the point of view of the planning module.

What makes our problem unique and simple is the fact that all the obstacle modules are connected to the workspace boundary and thus can be represented as a modified contour of the workspace. This will be referred to as the *obstacle contour* seen by the planning module. A key property of this contour is that it can be represented as the graph of a single-valued real function, consisting of straight-line segments. The actual representation for the function is a linked list of segments defined by a point and a slope.

The obstacle contour takes into account both static and moving modules. Static modules produce a single zero-slope segment of the contour. Moving modules produce several segments, some with non-zero slopes.

The description of the obstacle created by a single module or by a combination of modules has the same representation, and an incremental algorithm can be used to obtain the obstacle contour generated by all obstacle modules. Another advantage of this representation is that it contains the connectivity information about the free space in an explicit way. Thus collision checking and path planning are trivially done, once the obstacle contour has been built.

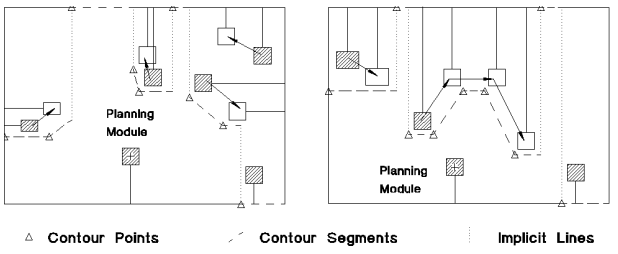

Figure 2: **Typical obstacle contours**

Fig. 2 shows typical obstacle contours with several modules in motion. As usual, obstacle contours take into account the physical dimensions of the planning module so it can be treated as a single point.

4.2 Constructing the Obstacle Contour

The construction of the obstacle contour can be broken into a sequence of four steps: obtain the *relevant points* of each obstacle module, find the *critical point* of each obstacle module, create the *obstacle contour* due to each obstacle module, and merge all these into one *obstacle contour*.

4.2.1 Obtaining the relevant points

The *point of interest* of a module is some meaningful point on the module, e.g., the location of the end-effector or the axis of the camera. A module is said to be at a position when the point of interest is at that position. To compensate for cable curvature, the cables are assumed to be delimited by the two lines extending the sides of the module toward the relevant workspace edge. The *relevant points* of an obstacle module are the vertices of the resulting rectangular area.

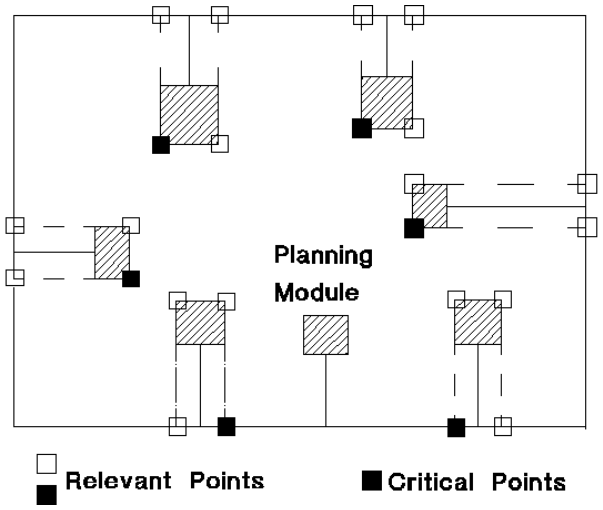

Figure 3: **Relevant Points and Critical Points**

Fig. 3 shows the relevant points for all obstacle modules.

The relevant points are computed from a geometric model during initialization, assuming that the module is at the origin of its virtual coordinate system. When planning a path, these points are corrected to account for the present position of the module, and brought into the virtual coordinate system of the planning module.

4.2.2 Finding the critical point

For a given planning module, each obstacle module has a unique *critical point*. Any of the four relevant points of the obstacle module can be critical depending on the relative orientation of the two modules. It is critical in the sense that is the only one we need to know to

find the equivalent obstacle presented by the obstacle module to the planning module.

A very simple algorithm determines which of the relevant points is critical, for each pair of planning and obstacle modules. This is done during initialization and stored in a table for run-time use. Fig. 3 also shows the critical points for all obstacle modules.

4.2.3 Creating the obstacle contour for each obstacle

For a given planning module we need to construct the obstacle contour associated with each obstacle module. Notice that an obstacle module might be in motion along a multi-segmented path. The kinematic simulator keeps the path of the obstacle module updated in order to delete the part of the path that the module has already covered. Different obstacle contours built by the collision-checking program were shown in Fig. 2.

For the algorithm that creates the obstacle contour to work, the path of the obstacle module must be *reasonable*. A multi-segmented path is *reasonable* if, and only if, the virtual x coordinates of the initial configuration, via points, and goal configuration, in the order in which they are reached, are monotonic. In other words, a reasonable path does not move back and forth along the axis orthogonal to the module's cable. In particular, paths that close in a loop are not reasonable.

There are several reasons for restricting the paths to reasonableness. First, it is not really a restriction since a general path can always be divided into a sequence of reasonable ones. Second, it results in a more efficient algorithm; dealing with a completely generic path can be quite involved due to several pathological situations. Last, but most important, our planner only generates reasonable paths.

The first two steps in building the obstacle contour are (1) to update the obstacle module's path, bringing the module to its present position, and (2) to compute the critical points corresponding to the obstacle module, in its current position and on each of the successive via points in its path, through to the goal. The remaining steps depend on the relative orientation of the planning and obstacle modules. The area swept by the obstacle module when moving along its path, from its current position to its final destination, is taken into account. These steps are straightforward and not included for the sake of brevity.

4.2.4 Merging two obstacle contours

Recall that the construction of the obstacle contour proceeds incrementally. Starting with a *merged contour* that represents the boundaries of the workspace, we proceed to merge the *obstacle contour* created by each obstacle module, one at a time. On any iteration we have a *merged contour* (the result of merging a series of previously considered *obstacle contours*) and the current obstacle contour. However, as far as the algorithm is concerned these are just two contours that need to be merged into a single one.

While merging the two contours we shall refer to an *inner* and an *outer* contour (illustrated in Fig. 4). The *outer* contour is the one that restricts the planning module's motions more severely, and will become the merged contour when the merging process completes. The *inner* contour is the other one, and will be discarded after merging.

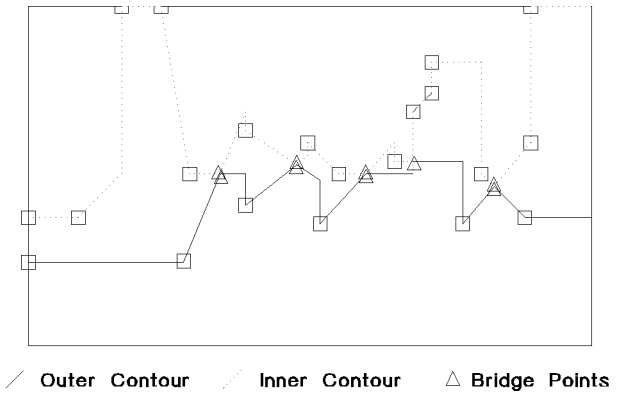

Figure 4: **Merging a new obstacle contour**

The merging process keeps two pointers. One points to the inner contour under construction and the other to the outer contour. Initially the outer pointer points to the *merged contour* and the inner pointer to the obstacle contour to be merged. The outer contour will be modified by absorbing the sections of the module contour that protrude from the merged contour. This is accomplished by simple pointer manipulation and insertion of *bridge* points, with a single pass through the contours.

5 Path Planning

Given the obstacle contour for a giving planning module, we can readily see whether the goal configuration lies in free space. Free space is always connected, except in the case when a module splits the workspace. A preliminary test takes care of this condition.

Assuming a collision-free path exists, there is an infinite number of them. When comparing two paths we assume that the one with fewer segments is better, except if it is much longer. All possible destinations can be reached by a path consisting of at most three segments, e.g., by retracting along the cable until we can avoid all obstacles, then displacing orthogonally until we can reach the destination by extending along the cable. This path would not be optimal in most cases, but an optimal single- or double-segment path may exist.

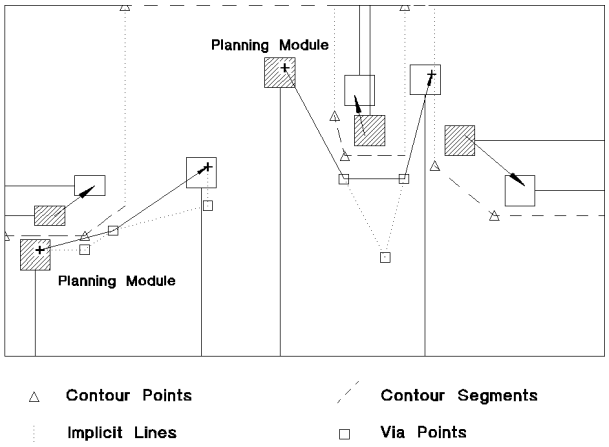

Figure 5: **Examples of paths planned by the controller**

The path-generating algorithm (not presented here for sake of brevity) always attempts to move the planning module in a straight-line path. This is the preferred solution if it is collision-free. If this is not possible, two- or three-segment paths are generated, based on geometrical considerations and the *shape* of the obstacle contour. Fig. 5 illustrates some of the different cases.

6 Complexity Analysis

The overall complexity of the scheduler cannot be determined unless complicated assumptions are made about the requests sent by the user processes. These requests determine the composition and size of the queues the scheduler has to service. Simulations could be performed but the results obtained would be of limited interest, as they would be extremely dependent on the specific details of the requests, their attributes, and frequency of arrival. Instead, a complexity analysis of the relevant parts of the scheduler, those that are independent of user requests, is done.

The usual measure of problem size in path planning is the sum of the degrees of freedom of each of the robots involved. Since in our case each robot has two degrees of freedom, an equivalent measure is the number of robots, n. Table 1 summarizes the results of the complexity analysis. A brief justification follows:

1. *Find the obstacle contour seen by a planning module.* For a given planning module, the obstacle presented by each of the other modules is independent of the number of modules, and thus computed in $\mathcal{O}(1)$ time, i.e., constant time. Merging two obstacle contours takes time proportional to the sum of points in each contour: the *merged* contour has $\mathcal{O}(n)$ points, the *outer* contour $\mathcal{O}(1)$, as above. A merging operation is done for each obstacle module, or $\mathcal{O}(n)$ times. The obstacle contour can thus be calculated in $\mathcal{O}(n^2)$.
2. *Check for the existence of a collision-free path.* Once the obstacle contour has been built, this takes time proportional to the number of points in the obstacle contour, i.e., $\mathcal{O}(n)$. Finding the path also takes $\mathcal{O}(n)$ time.
3. *Check whether a move is possible and commit it* (cascading moves included). This step involves a recursive search, without backtracking. Each node of the search tree has a module associated with it and each module can appear in the tree only once. Thus, there are at most $\mathcal{O}(n)$ visited nodes. Each time a node is visited an obstacle contour must be built ($\mathcal{O}(n^2)$), an obstacle module candidate for displacement must be identified ($\mathcal{O}(n)$) and a destination for that module found ($\mathcal{O}(n)$). This results in a time of $\mathcal{O}(n^3)$.
4. *Check whether a committed move can be started.* Once the obstacle contour has been built, checking a committed move for collision entails checking each subsegment of the path for collision. This check is proportional to the number of points in the obstacle contour, i.e., $\mathcal{O}(n)$.

Table 1: Worst-case complexity of different algorithms as a function of the number of modules.

Algorithm	Complexity
0. Update the position of a module using the simulator.	$\mathcal{O}(1)$
1. Find the obstacle contour seen by a planning module.	$\mathcal{O}(n^2)$
2. Check for the existence of a collision-free path.	$\mathcal{O}(n)$
3. Check whether a move is possible and commit it.	$\mathcal{O}(n^3)$
4. Check whether a committed move can be started.	$\mathcal{O}(n)$

It can be argued that this worst-case analysis is overly pessimistic. The characteristics of the obstacle contours, will usually result in a number of points that is not $\mathcal{O}(n)$ but much smaller. This significantly reduces the complexity of the above algorithms.

7 Kinematic Simulator

The low-level controller in our SMALL workcell is capable of generating straight-line moves with user-defined *acceleration profiles*. These acceleration profiles consist of a sequence of pairs $\{(a_i, v_i)\}_{i=1}^{n}$, where a_i is the acceleration to be used until the velocity v_i is reached. Modules accelerate until the maximum velocity is reached or the middle point of the the motion

is achieved, and then decelerate. The velocity vs. time profile is time-symmetrical.

The low-level controller is capable of buffering a sequence of motion requests. When the scheduler needs to send a module along a certain multi-segmented path, it can do so with a single transaction.

The kinematic simulator is straightforward to implement, and is based solely on the equations of motion. In order for the simulator to compute instantaneous module locations, we associate with each segment of the path the acceleration profile to be used, and the *time* when the motion along this path was started.

The time is obtained by querying the system clock. Since this is done after sending the command that starts the motion, and before performing the calculation of the module location, the simulated position of the module will be slightly behind the actual position. This adds a safety margin to the planned motions.

In the actual implementation, the kinematic simulator will be replaced by a query to the low-level controller. The kinematic simulator provides the capability of debugging the code off-line, avoiding potential damage to the workcell. In addition, it allows several people to develop their applications simultaneously, decoupled from the workcell.

8 Conclusion and Future Work

The scheduler, collision-checking, and path-planning algorithms have all been implemented in a simulated environment, and are extremely fast. The complexity analysis shows the general problem of planning a path for a module scales, in the worst case, as $\mathcal{O}(n^3)$ with the number of modules. This includes planning paths for other modules that are displaced. Up to eight modules were used in the simulator, with their positions shown in real time on a graphical display. Move requests were entered by the user via a mouse. All the algorithms described are in the process of integration into the current controller as of this writing.

After we have feedback from a real application environment, we will review whether the abstractions presented to the user processes by the controller suffice, or if they need to be augmented and/or modified.

The present planning algorithm throws away the partial ordering information of committed moves, available at planning time. Were this to be preserved, we could avoid wasting time during the initiation of the moves.

The third dimension needs to be incorporated. A way of doing this would be to classify the fixed obstacles in the workspace into two groups depending on whether or not a collision can be avoided by a suitable retraction of the vertical axis. Obstacles that can't be avoided might initially be treated as two-dimensional and modeled as such. (This would require a modification of the algorithms as they are rectangular *islands*). Another alternative is to treat unavoidable obstacles as east, west or north *modules*, depending on the destination of the module. For the obstacles that are avoidable, the third dimension can be ignored at planning time, and only when a move is launched does the planner need to check whether a retraction is necessary.

The problem of mutual interactions between z and θ-axes of different modules needs to be addressed. For most practical cases we can reduce the infinite set of possible combinations to a hopefully small finite set, and plan in two dimensions for all cases in that set.

References

[1] Lozano-Pérez, T., "Spatial Planning: A Configuration Space Approach," *IEEE Transactions on Computers*, C-32(2), 1983, pp. 108–120.

[2] Khatib, O., "Real Time Obstacle Avoidance for Manipulators and Mobile Robots," *International Journal of Robotics Research*, 5(1), 1986.

[3] Schwartz, J., Sharir, M. and Hopcroft, J., *Planning, Geometry, and Complexity of Robot Motion.* Ablex Publishers, Norwood, NJ, 1987.

[4] Canny, J., *The Complexity of Robot Motion Planning.* MIT Press, Cambridge, MA, 1987.

[5] Latombe, J., *Robot Motion Planning.* Kluwer Academic Publishers, Boston, MA, 1990.

[6] Lieberman, L. and Wesley, M., "AUTOPASS: An Automatic Programming System for Computer Controlled Mechanical Assembly," *IBM Journal of Research and Development*, 21(4), 1977.

[7] Lozano-Pérez, T. and Winston, P., "LAMA: A Language for Automatic Mechanical Assembly," *Proceedings of the Fifth International Joint Conference on Artificial Intelligence*, 1977.

[8] Sawyer, B., "Magnetic Positioning Device," U.S. Patent 3,376,578, April 1968.

[9] Scheinman, V., "Robot World: A Multiple Robot Vision Guided Assembly System," *Proceedings of the Fourth International Symposium on Robotics Research*, 1987.

[10] Lilienthal II, P., Fleming, J., Harkopos, E. and Van Orden, G., "A Flexible Manufacturing Workstation," *AT&T Technical Journal*, 67(2), 1988.

[11] Schneider, J. and Kaufman, A., "Printed Circuit Robotics and Transitioning Technologies," *Printed Circuit Assembly*, 1(4), 1987.

[12] Buckley, S., "Fast Motion Planning for Multiple Moving Robots," *Proceeding of the IEEE International Conference on Robots and Automation*, Scottsdale, AZ, 1989.

[13] Sinden, F., "The Tethered Robot Problem," *International Journal of Robotics Research*, 9(1), 1990.