1996 NASA/ASEE SUMMER FACULTY FELLOWSHIP PROGRAM
JOHN F. KENNEDY SPACE CENTER
UNIVERSITY OF CENTRAL FLORIDA

SCHEDULING SYSTEM ASSESSMENT, AND DEVELOPMENT AND ENHANCEMENT OF RE-ENGINEERED VERSION OF GPSS

Dr. Rasiah Loganantharaj, Associate Professor
Mr. Bushrod Thomas, Graduate Student
Center for Advanced Computer Studies
University of Southwestern Louisiana
Lafayette, Louisiana

KSC Colleague - Nicole Passonno
Ground Systems

Contract Number NASA-NGT10-52605

August 8, 1996

ABSTRACT

Scheduling of activities that maintain, repair and configure a space shuttle orbiter must satisfy temporal, resource and configuration constraints. The GPSS, which stands for ground processing scheduling system, has been successfully used to schedule tasks that prepare an orbiter for its next launch. The features of the GPSS that support specification of a task with configuration requirements and effects is what makes the GPSS unique when compared to other scheduling software.

The objective of this project is two-fold. First to provide an evaluation of a commercially developed version of GPSS for its applicability to the KSC ground processing problem. Second, to work with KSC GPSS development team and provide enhancement to the existing software.

The Red Pepper Software (RPS) company has developed a commercial scheduling product based on the original algorithm of GPSS. The RPS team consists of many of the original members of the NASA Ames, KSC and Lockheed development. The RPS system was installed and an initial evaluation regarding its suitability for solving the ground processing scheduling problem was performed.

The GPSS in use at KSC is written in Lisp and C and has very little or no documentation available regarding the development process, or software design and algorithms. System re-engineering is required to provide a sustainable system for the users and software maintenance group. As such, the USA development team is developing a re-engineered version of GPSS in C++. The GPSS uses an iterative repair method for scheduling. The algorithm first satisfy temporal constraints among the tasks, and then it iteratively repair the conflicts associated with resource and configuration constraints. Resolving the conflicts is a difficult and time consuming process for the algorithm. The deconfliction process of GPSS is a hill climbing method coupled with a weak simulated annealing implementation. The temperature for the annealing process is fixed at 75, and it is brought down to 25 after achieving some conditions. We propose a formula that determines the initial temperature based on: (1) the initial cost, and (2) the tolerance limit with which one is willing to accept a higher cost in order to avoid local minima. We have also laid down procedures to gradually cool down the temperature as the algorithm improves the solution.

We have implemented the resource deconfliction portion of the GPSS in common LISP using its object oriented features. We have used the LISP profile prototype code, which was developed by the GPSS reverse engineering group, as a building block to our implementation. Our prototype will help the GPSS developers re-engineer the deconfliction portion in C++. It corrects and extends some of the deficiencies of the current production version, plus it uses and builds on the classes from the development team's profile prototype, which they are now porting to C++.

SCHEDULING SYSTEM ASSESSMENT, AND DEVELOPMENT AND ENHANCEMENT OF RE-ENGINEERED VERSION OF GPSS

Rasiah Loganantharaj and Thomas Bushrod

1. Introduction

The objective of this project is two-fold: first, to provide an evaluation of Red Pepper Software's Production Response Agent (PRA), a commercially developed version of the ground processing scheduling system (GPSS), for its applicability to the KSC ground processing problem; second, to work with the KSC GPSS development team and provide enhancement to the existing software. We start with an introduction of scheduling, GPSS and its limitations

Scheduling is a process of assigning time slots for activities while satisfying their resource and configuration requirements, and temporal ordering among themselves. A typical temporal constraint specifies a successor, a predecessor, and possibly a minimum delay between them. For example, a temporal constraint may have a form such as: task T_2 is a successor of a task T_1, and T_2 can start only after 10 units of time from the completion of T_1. Satisfying all the temporal constraints among the tasks provides a schedule with early and late start time of each task, critical path, etc. There are plenty of sophisticated software algorithms to schedule tasks that are constrained only by temporal relations. Satisfying temporal constraints of this sort will take $O(N^2)$ time where N is the number of tasks. When we introduce resource requirements for tasks, the problem can no longer be solved optimally in polynomial time. The requirement that a task have two overhead cranes is an example of a resource constraint. A task can be scheduled only after it is assigned all of its requested resources. If we have an unlimited quantity of resources to the extent of satisfying all the resource requirements, the problem reduces to solving only the temporal constraints. However, in many real world problems, resources are limited. Thus, decisions must be made about the order in which requests are satisfied. An optimal schedule maximizes the resource utilization and minimizes the schedule length.

The complexity of a scheduling problem is further compounded when the problem is extended to include configuration requirements and configuration effects. By *configuration requirement*, we mean a requirement of a task requesting an attribute of an object to be in a specified state. For instance, the precondition that payload-bay doors of a space shuttle orbiter be opened 75 degrees to allow loading of a satellite is an example of a configuration requirement. Here, the attribute is the state of the payload-bay doors (which may have a range of possible values from closed to fully open), and the activity is "load satellite." Similarly, an activity may affect the state of some attribute of an object. According to the law of causality, the effect cannot precede the causing activity, and it must persist until some other subsequent activity changes the attribute. Suppose the bay area is loaded with hazardous material. The area remains hazardous until the material is removed from the bay. This is an example of an effect that persists to infinity. In the presence of tasks that cause changes to attribute values, finding a proper placement for a task with configuration requirements increases the search efforts of the scheduler.

An objective function of a scheduler is to obtain a placement for each activity such that the overall schedule maximizes resource utilization and minimizes schedule length. Except for some trivial cases, obtaining an

optimal schedule that satisfies temporal, configuration, and resource constraints is NP-hard. That is, no polynomial-time algorithm is available to solve these optimization problems. The algorithm that solves the problem optimally will take time on the order $O(2^N)$, where N is the number of tasks to be scheduled. When N is in the order of hundreds, it will take thousands of years to solve the problem, even if we run it on the fastest computer available today. Typical flows at KSC involve a thousand or more tasks. Therefore, a sub-optimal solution is being sought to solve scheduling problems of this sort.

There are two major approaches to obtaining the sub-optimal solution: the constructive method, and the iterative repair method. In the constructive method, one tries to build a feasible solution satisfying temporal, resource, and configuration constraints incrementally. When there is a failure, the algorithm backtracks and a new choice is made. This approach can further be categorized based on how the backtracking is done. The backtracking can vary from simple methods, such as chronological backtracking, to very sophisticated methods, such as intelligent backtracking. Heuristic methods are used to decide how to allocate resources among the competing tasks.

On the other hand, the iterative repair method starts with a rough schedule that satisfies only the temporal constraints. Then it repairs the resource and configuration constraint violations one by one until no more violations exist or no further improvements can be made. The iterative repair method is very suitable to tackling over-constrained problem instances because it will iteratively reduce the number of violations even though there may not be any feasible solutions (solutions satisfying all constraints, though not necessarily optimally) for the problem. If the constructive method is used to solve an over-constrained problem, it will consume a substantial amount of time exploring the entire search space before it eventually fails. Many of the ground processing scheduling problems at KSC are over-constrained, and hence, GPSS uses the iterative repair method to solve the scheduling problem.

1.1 GPSS

GPSS has been used successfully at KSC for scheduling orbiter processing facility (OPF) operations. The OPF processing has three phases: (1) making the orbiter safe for processing and gaining access to the orbiter through the installation of access platforms, (2) testing, maintenance and repair operations on the orbiter, (3) close-out and checkout of orbiter. Approximately 40% of OPF processing is routine and very predictable in advance. The remaining 60% of OPF activities are very dynamic and are driven by factors such as payload of the previous and the following missions, test requirements specific to the age of the orbiter, diagnostic in-flight anomalies, and unexpected damage caused to the insulators.

GPSS provides a graphical interface for specifying tasks. Each task has a duration, work calendar, resource requirements, state requirements, and state effects. Temporal constraints between predecessor-successor pairs are specified. The number of resources available for each resource type is specified. A task may require many resource types, and for each resource type the task can specify the number of resources it needs. GPSS assumes that resources are reusable, that is, once the task is completed, the number of resources used by the task is released and can be used by other tasks. Each attribute's initial state is specified. A task may request an attribute to be in a specified state. When a task is scheduled in a time period during which the attribute has a different value than the one it requested, we say *configuration violation* has occurred.

Given an initial specification of tasks that may include resource and configuration requirements along with temporal constraints, GPSS satisfies their temporal constraints first. Resources are allocated, and a *profile*

(a.k.a. *history*) is maintained for each resource. A resource profile consists of a sequence of intervals along the time line indicating the *users* and *changers* of each interval. In an interval where the total requests of the users exceed the number of available resources, the resource type is said to be *over-allocated* and it is an indication of a *resource constraint violation*. In a system modeling consumable resources, a changer could allow a task to produce or consume some quantity of the resource, but GPSS is not used in that way at present.

When there is a resource violation, a user can either resolve the conflict manually or use the system to deconflict the violation. A user can select a task from among the tasks in violation for the same resource, and move it to another time interval where the resource request of the task can be satisfied. After each move, the system runs the temporal constraint satisfying algorithm and accepts the move only if there exists a way to satisfy all temporal constraints. Otherwise the move is rejected and the previous system status is restored.

Similar to the resource profile, a profile is maintained for each configuration attribute. Each interval of an attribute profile maintains its users and changers. Unlike the reusable resource constraints, users and changers need not be the same. The changers affect the attributes value. When there are many chanagers in an interval, the value of the attribute is determined by the effect of the latest changer. A task is said to be in configuration conflict if the requested attribute does not match with the attribute value in the interval. As with resolving resource conflicts, a user can either resolve the configuration conflict manually or use the system to deconflict the violation.

1.1.1 Limitations of GPSS

While GPSS is successfully used for scheduling activities in each flow, it has some weaknesses in regard to deconfliction. The auto-deconfliction in GPSS is very slow and it lacks some user controllable features such as *fencing* (providing time boundaries on the working schedule within which to select violated constraints to resolve) and *user-fixing* activities (holding selected tasks in their scheduled intervals) during deconfliction.

1.2 Organization of the report

Following the introduction, we briefly summarize the evaluation of Red Pepper Software. In the next section we describe about repairing conflicts and discuss about the enhancement we propose to the GPSS. This is followed by summary and discussion.

2. Evaluation of Red Pepper Software (RPS)

2.1 Installation

We received Production Response Agent software, PRA (v1.5.2), from RPS on a medium density tape. We installed the tape on a Sparc 5 workstation under the operating system Solaris 2.5. The initial installation was not successful because a motif library file (libXm.so.2) was missing in the Solaris 2.5 distribution. We called the RPS hotline (1-800-578-3345) and explained the installation problem. The next day, RPS had set up an ftp access for us to download the missing file. With the missing file in the correct directory we were able to complete the installation of the software and to run the Production Response Agent (PRA) successfully.

2.2 Evaluation of PRA for scheduling purposes

For the purpose of our initial PRA evaluation, we constructed a simple scheduling problem with temporal and resource constraints, ignoring status constraints. We studied the PRA user manual to build a scheduling model to solve the example problem. We were unable to find any materials in the manual that talk about building a model to solve scheduling problems involving temporal and resource constraints. We then examined the PRA startup menu to see if there is a way either to build a model or to set the problem instance. We failed to find a way to enter our simple example into the system.

PRA has to be initiated with a command file which comes with the software distribution. The command file loads different built-in models. It is our understanding that PRA will only work to solve a problem that fits the defined models. The models that came with the distribution do not support the scheduling model that is used by GPSS at NASA KSC.

To confirm our understanding of PRA, we contacted the technical support at RPS. The technical personnel at RPS acknowledged the following concerning the capability of PRA (V1.5.2) towards building a new model:

1) The user manual of PRA does not provide any information on building a new model to solve problems.
2) To learn about building models using PRA, one has to attend a class at RPS on SPL, their proprietary Scheduling Programming Language.
3) During the tutorial classes at RPS, students copy the existing models and modify the parameters and instructions to solve their problem.

PRA provides a graphical user interface to access and to modify the instances defined by the model. The product seems to be very capable and useful for solving problems in high level production planning and scheduling. However, for PRA to be utilized for ground processing scheduling purposes it would have to support temporal, resource, and configuration constraints at the task and sub-task levels.

Since PRA does not support scheduling problems similar to the ones solved by GPSS we did not continue with a through evaluation. Instead, we worked with the GPSS re-engineering group, focusing on enhancing GPSS and building a prototype for constraint violation deconfliction.

3. Development and Enhancement of Re-engineered version of GPSS

3.1 Repairing Conflicts

GPSS is an iterative repair scheduler which first satisfies all the temporal constraints among the tasks and then repairs all the resource and the configuration violations iteratively. The Waltz algorithm, which was originally developed for the purpose of scene analysis, was modified to propagate temporal constraints and to achieve temporal constraint satisfaction. In the process of achieving temporal constraint satisfaction, the Waltz algorithm may move tasks that cause resource or configuration constraint violations. The Waltz algorithm cannot move the tasks that are user-fixed. In such cases when temporal satisfaction cannot be achieved, the algorithm reports failure.

When there is either a resource or a configuration violation, GPSS provides two options to the user: the user can manually resolve the conflicts by moving tasks, or use the auto deconfliction option to resolve the conflicts. When a user resolves conflicts by moving tasks, GPSS allows the user to move one task at a time. Temporal consistency is maintained by running the Waltz algorithm after each movement. When temporal constraints cannot be satisfied, the move made by the user is rejected. In this section we look into how the system deconflicts the violations.

When GPSS deconflicts the violations a user has limited options: (1) select the relevant resources to focus, and (2) control the time taken for deconfliction by setting the number of iterations and the window size. The window size determines how many violated resource constraints and configuration constraints are selected for deconfliction in each iteration. For example, if a user selects window size 5 and iteration 40, 5 resource constraints and 5 configuration constraints are resolved in each iteration, or until all the constraint violations are resolved. Let us describe the algorithm believed to be used in GPSS (the algorithm is written based on the notes provided by Mr. Jim Tulley).

```
for  i = 1 to iterations do
{       Construct the list of violated constraints and categorize them into
        (1) resource constraint violations
        (2) configuration constraint violations
        Construct a list of tasks from the resource constraint violation list.
        From this list randomly select k tasks (k is the window size) and call them tasks-to-focus
        for j = 1 to k do
        {       pick up a task from tasks-to-focus and update the list
                call Repair on the earliest violated constraint of this task
        }
        for j = 1 to k do
        {       repair configuration constraint violation
        }
}
```

A conflict, either resource or configuration, can be resolved by moving a task involved in the conflict to a new time period where the conflict does not arise. Let us consider how GPSS repairs the constraint violations. During each iteration, a list of tasks called tasks-to-focus is created randomly by selecting tasks of window size from the resource conflicting tasks, or the total of the resource conflicting tasks whichever

is the smaller. For each task in the tasks-to-focus list, find the earliest constraint violation and call a routine called Repair to repair the violation. The Repair routine, knowing the resource and the time period of the violation, determines all the tasks that request the resource during the time period. A heuristics table is constructed to determine which task to move to reduce the conflicts. The heuristics table consists of ten columns to represent (1) fitness, (2) nearest-fix, (3) task-active, (4) temporal dependents, (5) proximity to now, (6) cost, (7) task, (8) start time, (9) weighted sum of scores, and (10) next time to try. Each task is given two rows: one for forward direction and the other for backward direction.

The first column calculate the measure of fitness and it is given by the formula:

$$\text{fitness} = \frac{1}{(\text{request} - \text{over-allocation})^2 + 1}$$

The fitness of a task has a value of one if its request matches the over-allocation that indicates the conflict on the resource during the interval can be resolved by moving the task.

The nearest fix value of a task is the time span of the current start and the next-time-to-try column of the table. The next column is a flag indicating whether the task is active or not. Forward and backward temporal dependencies for each task are computed and stored in the temporal dependency column.

The cost in column 6 is computed by moving the task in both the forward and the backward directions within a temporary context. This cost is the approximate cost since temporal consistency is not checked after the move.

When all the computations are done for each row, the table is normalized. The task with the best weighted summation is selected for the actual move. The selected task is moved and the Waltz algorithm is run to satisfy the temporal constraints. If the move reduces the previous cost, the move is accepted and the iteration continues. When the move increases, the cost is accepted with some probability computed using simulated annealing techniques. We discuss the techniques and how they are being used to solve the problem.

3.1.1 Simulated Annealing

Simulated annealing is a technique used to avoid local minima when using a hill climbing technique to solve problems. This technique simulates some aspects of the natural annealing process, and hence got the name simulated annealing. In an annealing process, a metal is heated to a higher temperature and then cooled down gradually to obtain fine grain. When the temperature cools down rapidly, one obtains course grain in the annealing process.

In a typical hill climbing method, heuristics are applied to find the next, most promising move. The algorithm commits itself to that move and continues forward from the new state. Move with higher cost are rejected as the search progresses through moves that always decrease the cost. Very often the algorithm attains a local minima or plateau. To avoid local minima, moves with higher cost must also be accepted with some controlled probability. This is exactly what is done in simulated annealing. Suppose delta, Δ, is the increase in cost. A move which normally would be rejected for having higher cost will now be accepted with probability $e^{-\Delta/T}$, where T is the temperature. Initially T is set to a higher temperature and it is

decreased gradually. At the higher temperature, or in the initial phase of the solution, the probability of accepting moves with higher cost is high. This probability decreases as the temperature decreases.

3.1.2 How Simulated Annealing is used in GPSS

The deconfliction algorithm used in GPSS is a hill climbing algorithm coupled with a weaker version of the simulated annealing algorithm. The temperature is set at 75 and it remains at that temperature until the cost is less than 10 and the number of iterations completed is greater than 5. When these conditions are met, the temperature is reduced to 25.

Setting of initial temperature should be dependent on the problem size, the domain of the problem, and the heuristic strategy being used to solve the problem. In the GPSS implementation, temperature remains fixed for a wide range of solutions. Hence, there is a danger of accepting moves that increase the cost by more than 50% of the best found solution. Suppose the best cost is 25 and the new cost is 50. The increment of the cost is 25, 100% higher than the best known so far. In GPSS, the probability of accepting such a move is 0.71, which is very high. The annealing portion is so weak that it allows all sorts of moves with much higher cost. This makes the algorithm wander around the search space without any focus in reducing the cost.

3.2 Enhancement of the GPSS Deconfliction Process

We can improve the resolution of conflicts in GPSS by improving the quality of the solution while decreasing the computational cost of obtaining such solutions. In a typical flow, the total number of constraint violations will be in the order of several hundreds, and the span of the schedule will be around 85 days. Very often, a user is interested in getting a conflict free schedule for a selected period of time, say for 11 days. The current GPSS does not allow such option, instead the deconfliction process selects tasks randomly and resolves the conflicts in each iteration. The algorithm may be spending time to resolve conflicts at the tail end of the schedule which is currently of much less interest to the user. A *fencing capability* will provide the user with the option to select the time period on which to focus, and enable the removal of all conflicts occurring during that period.

With the fencing capability, a user can interactively select the time period over which deconfliction takes place. The deconfliction algorithm is changed to select only the violated constraints occurring during the fencing period. During deconfliction, the violated tasks are moved forward or backward to remove the conflicts. The algorithm should make sure that a task cannot be moved before NOW. Further, the algorithm tries to keep the tasks inside the fencing windows by storing the tasks moved out of the fencing window during deconfliction into a list called try-later. After all the violations are resolved, the algorithm tries to schedule the tasks from try-later into the fencing windows without violating the existing schedule in the fencing window.

In addition to the fencing capability, a user would like to have the option to select a certain set of tasks and make them static during the deconfliction. This option can be easily implemented. First, the algorithm ensures there is no constraint violation among the selected tasks. If constraint violation exists, the algorithm reports appropriate error message and rejects the set of tasks. Once the request is accepted, the algorithm excludes these fixed set of tasks from the candidates to be moved to get a conflict free schedule. It is also possible that there may not be a conflict free schedule once a set of tasks are fixed.

The current implementation of GPSS does not enforce space capacity constraint. Without such enforcement it schedules tasks in the same time slot even though space capacity constraint is violated. Consider three tasks T_1, T_2, and T_3 each requiring access to the crew module. Assume that T_1, T_2, and T_3 respectively require 3,2, and 2 technicians and can operate concurrently. If the capacity of the crew module permits at most 4 people at a time, T_1, T_2, and T_3 can not be scheduled during the same time period without violating the space capacity constraint. GPSS does not enforce such constraint, therefore, it will not report this as a violation. This problem can be corrected by adding space capacity constraint to GPSS. The supporting implementation will be very much similar to that of resource constraint.

In addition to providing these capabilities to the user, we can improve the deconfliction process by improving the heuristics and decreasing the computational cost. A close examination into the heuristics table reveals that GPSS captured many interesting aspects of schedules such as fitness, temporal dependency, nearest fix, and cost. The computation of all the parameters except cost will take a constant amount of time. Computing the cost of a move is computationally very expensive since it involves de-allocation of all the resources for the candidate task, re-allocation of the resource to the task at the new time period, and computation of all the violations with the new placement. This is clearly the most expensive item in the table. We would like to remove the cost column of the heuristics table and add two other heuristic parameters: (1) number of constraints, and (2) ratio of violated constraints to total constraints. When a task with only a few constraints is moved, the chance of perturbation to the schedule is smaller than the chance incurred when moving a task with many constraints. This column can be normalized as:

$$1 - \frac{\text{number of constraints}}{\text{maximum number of constraints among the conflicting tasks}}$$

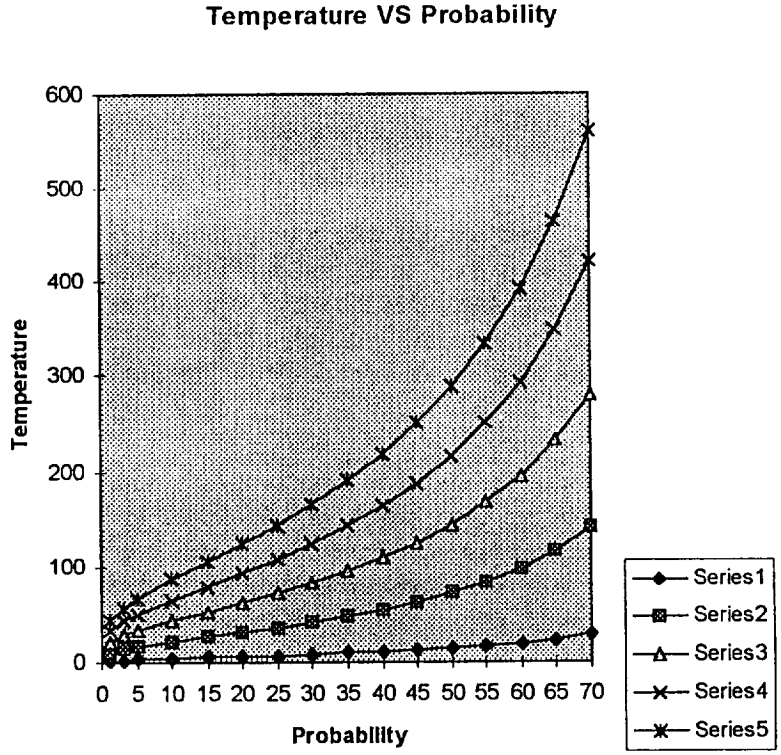
When comparing candidates for movement, we must take the percentage of violated requests into consideration. A task with a higher percentage of its requests violated should be favored as a potentially better candidate to move than the others. This is captured by the other, similarly normalized parameter.

The iterative repair method used in GPSS is basically a hill climbing algorithm coupled with simulated annealing to avoid local minima. Unfortunately, the simulated annealing as implemented in GPSS is very weak. It fixes the temperature at 75 until the cost is less than 10 and there have been at least 5 iterations. Suppose the current cost of a schedule is 15 and the new cost as a result of a new move is 30, then this move of 100% increase in cost will be accepted with 0.818 probability. Worst yet, it will accept 200% increase in cost with 0.67 probability.

The success of solving a problem using simulated annealing depends on setting the temperature at the correct value and gradually cooling it down. When the temperature is set at a higher value, as has been done in GPSS, the algorithm accepts moves with very high cost. Therefore, it may jump around the search space without focusing on the most fruitful direction of cost reduction. On the other hand, when the temperature is set too low it will not allow reasonable moves that may very well be leading to a better solution.

We reject fixing the temperature statically. Instead, we favor setting the temperature for each set of iterations. The initial temperature is based on the initial cost, say init-cost, percentage of allowable increases in cost, say per-incr-cost, with the probability, say p. Then the initial temperature is equal to:

$$\frac{\text{per-incr-cost x init-cost}}{100 \text{ x } \ln\left(\frac{1}{p}\right)}$$



This graph shows the variation of the initial temperature with the accepting probability for a series of increased cost. The series 1, 2 3, 4 and 5, respectively, represent the percentage of increased cost of 1, 5, 10, 15 and 20. The initial cost is assumed to be 1000.

4. Summary and Discussion

We have successfully completed the following overall objectives of the summer research sponsored by NASA/ASEE: (1) to provide an evaluation of a commercially developed version of GPSS for its applicability to the KSC ground processing problem; (2) to work with the KSC GPSS development team and provide enhancements to the existing software.

We have successfully installed and made an initial evaluation of PRA from Red Pepper Software for its suitability for scheduling ground processing activities. The models supplied with PRA do not support ground processing scheduling. It is a non-trivial task to build a model that supports ground processing scheduling activities using the PRA shell and SPL, their proprietary Scheduling Programming Language.

We then focused on improving and enhancing the ground processing scheduling system (GPSS). GPSS consists of a user interface and an engine. Major functional units of the engine are temporal propagation and deconfliction of violations. A modified version of the Waltz algorithm is used for propagating temporal constraints. The conflict resolution of the engine forms the bulk of the cognitive and computational complexity of the engine. The performance of deconfliction can be improved by improving the quality of the solution without sacrificing the computational cost. We have investigated into the fencing capability, and space capacity constraint and have provide some insight of how this capability can be implemented.

During deconfliction, GPSS selects a task and moves it to an interval where it can be scheduled without any violation. GPSS uses a set of heuristics to determine which task to move among the tasks in conflicts. The current GPSS system captures important parameters. We have added two more parameters that help to make a better decision.

The deconfliction process of GPSS is a hill climbing method coupled with a weak simulated annealing implementation. The temperature for the annealing process is fixed at 75, and it is brought down to 25 after achieving some conditions. We propose a formula that determines the initial temperature based on: (1) the initial cost, and (2) the tolerance limit with which one is willing to accept a higher cost in order to avoid local minima. We have also laid down procedures to gradually cool down the temperature as the algorithm improves the solution.

We have implemented the resource deconfliction portion of the GPSS in common LISP using its object oriented features. We have used the LISP profile prototype code, which was developed by the GPSS reverse engineering group, as a building block to our implementation. Our prototype will help the GPSS developers re-engineer the deconfliction portion in C++. It corrects and extends some of the deficiencies of the current production version, plus it uses and builds on the classes from the development team's profile prototype, which they are now porting to C++.

We feel that the resource deconfliction can be further improved by moving only a portion of a conflicted task if the resource is not dedicated to it, instead of moving the complete task as has been presently implemented in GPSS. Further investigation is required to substantiate the claim and fine tune the implementation.

The heuristics table is very static, and the normalization process is somewhat arbitrary. To fine-tune the weight of the table, an extensive experiment must be conducted. Use of an artificial neural network seems to be a promising approach to set the weights of the heuristics table so that the system can make better decisions about which tasks to move.

We have looked into some of the modeling capabilities provided by GPSS. We are intrigued, for instance, by the configuration requirements and effects. Suppose there are three tasks T_1 , T_2 , and T_3 which all require the bay area to be open. Assume that T_1 and T_3 are mutually exclusive, that is, they cannot be performed concurrently. A user models T_1 and T_3 with the requirement 'bay area open', and the effect 'cleared' that persists from beginning to end. The attribute here is 'bay area', which can be in an 'open' or a 'cleared' state. The effect here is not at all causal as one would suspect. T_1 and T_3 are modeled as having an effect of 'cleared' to ensure the mutual exclusion. This modeling will not permit T_2 to be carried

out concurrently with either T_1 or T_3. An in-depth study is required to provide a flexible way to model tasks and their behaviors naturally.

REFERENCES

[1] M. Deale, M. Yvanovich, D. Schnitzius, D. Kautz, M. Carpenter, M. Zweben, G. Davis, B. Daun. 1994, "The Space Shuttle Ground Processing Scheduling System," in *Intelligent Scheduling*, edited by M. Zweben, and M. Fox, *Morgan Kaufmann Publishers*, San Francisco, Calif., pp.423-449.

[2] S. Kirkpatrick and C. D. Gelatt, and M. P. Vecchi, 1983. "Optimization by Simulated Annealing," *Science* Vol. 220 #4598.

[3] D. Waltz 1985. "Understanding Line Drawing Scenes with Shadows," *The Psychology of Computer Vision*, edited by P. Winston, McGraw-Hill.

[4] M. Zweben, E. Davis and M. Deale. 1993. "Iterative Repair for Scheduling and Rescheduling," *IEEE Systems, man, and Cybernetics*, Special Issue on Planning, Scheduling, and Control.

[5] M. Zweben, B. Daun, M. Deale. 1994. "Scheduling and Rescheduling with Iterative Repair," in *Intelligent Scheduling*, edited by M. Zweben, and M. Fox, *Morgan Kaufmann Publishers*, San Francisco, Calif., pp.241-255.