Generating Random Dynamic Resource Scheduling Problems

Wolfgang Haas and **William S. Havens**
Intelligent Systems Laboratory
Simon Fraser University
Burnaby, British Columbia
Canada V5A 1S6
{haasw, havens}@cs.sfu.ca

Abstract

Dynamic scheduling refers to a class of scheduling problems in which dynamic events, such as delaying of a task, occur throughout execution. We develop a framework for dynamic resource scheduling implemented in Java with a random problem generator, a dynamic simulator and a scheduler. The problem generator is used to generate benchmark datasets that are read by the simulator, whose purpose is to notify the scheduler of the dynamic events when they occur. We perform a case-study on an oversubscribed dynamic resource scheduling problem in which we assign unit resources to tasks subject to temporal and precedence constraints.

Introduction

Scheduling is the science of allocating limited resources to competing tasks over time (Jackson & Rouskas 2002). Most of the work in this area has concentrated on static scheduling problems in which one is given all information ahead of time and the problem does not change during execution of the schedule. But in the real-world there exists no such guarantee. A machine may fail or new tasks may arrive unexpectedly. The research described in this document deals with so-called dynamic scheduling problems in which many different kinds of unexpected events occur throughout execution of the tasks. These problems are much more difficult to solve but likewise they are also difficult to generate.

In this paper, we describe a framework for dynamic resource scheduling problems comprising a random problem generator and a dynamic event simulator. The parameterized problem generator provides a portfolio of random resource scheduling problems. Given a scheduling algorithm under test, the event simulator executes the schedule while making dynamic changes to the problem during execution. Each such problem modification requires the scheduling algorithm to reconsider its proposed solution dynamically.

In general, resource scheduling problems can be described as follows: Given a set of resources and a set of tasks, a schedule is a mapping of tasks to time intervals on each resource such that all specified constraints remain satisfied and the capacity of each resource is respected while optimizing some objective function (Hoos & Stuetzle 2005).

Resource scheduling is very important in real-world application for which there is a significant literature. However, there is much less known about dynamic scheduling problems and algorithms. Our own work was inspired by the Canadian CoastWatch project.

$CoastWatch$ is an oversubscribed dynamic multi-mode scheduling problem with unit resources and lies in the Search & Rescue domain. CoastWatch datasets simulate a typical day for the Canadian Coast Guard, where officers assign resources (planes, helicopters, ships, ...) to execute several different kinds of missions (rescue, patrol, transport, ...). The problem is inherently dynamic. Missions occur spontaneously as planes and ships are reported missing. Resources become unavailable due to equipment malfunction. Search tasks take unknown times to execute and may spawn additional rescue tasks. As well, there are a backlog of routine tasks which must be performed in the interim.

We found that obtaining sufficient real-life Search & Rescue data to be difficult. So we had to generate our own datasets while maintaining as much realism as possible. However, there is relatively little known about generating dynamic resource scheduling problems. In particular, there must be an intimate interaction between the problem generator and the scheduler. Executing a schedule can alter the problem being scheduled during execution. For this purpose a dynamic event simulator is required which is driven by the current schedule.

The result of our work is a dynamic resource scheduling framework which can be applied to many different kinds of dynamic resource scheduling problems. It has three components as illustrated in Figure 1: a random problem generator, a dynamic simulator and a scheduler under test. A special feature of the simulator is a visualization tool that creates an animation of the scheduling problem on $GoogleEarth^{TM}$. One can watch vehicle resources as they are moving around to execute missions and see the decisions made by the scheduler.

The problem generator is parameterized for generating a wide variety of random datasets. A dataset is parsed by the dynamic simulator which creates scheduling events at the appropriate time. Scheduling events include adding/deleting tasks; adjusting the duration of a task; removing/adding a re-

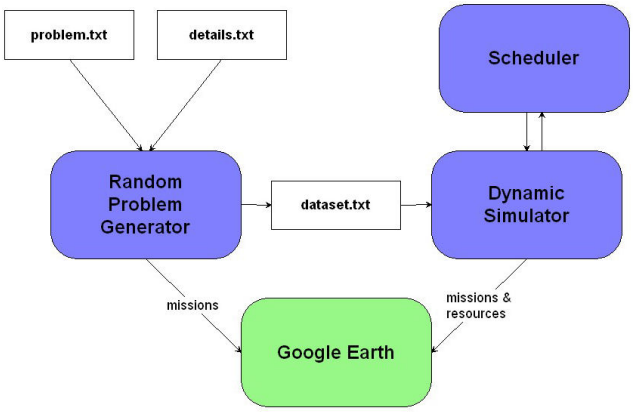

Figure 1: Overview of the Dynamic Resource Scheduling Framework

source; *et cetera*. Every time an event occurs, the problem is modified and the scheduler is invoked in order to reoptimize the schedule.

Previous Work

Usually scheduling problem generators are written as part of a research project and are tailored specifically towards the studied problem. Unfortunately, it is very unlikely that such a problem generator can be applied to a different class of problems. There do exist good general problem generators for static resource scheduling problems (*e.g.* project scheduling problems (Drexl *et al.* 2000) (Kolisch, Sprecher, & Drexl 1995)). Recently however, Policella & Rasconi (Policella & Rasconi 2005) devised a generator for a limited class of dynamic project scheduling problems (Brucker *et al.* January 1999).

Policella & Rasconi

Policella & Rasconi define a dynamic resource problem as containing two sub-problems:

- **Static sub-problem**: Given the problem definition, find a schedule that optimizes the objective function. This is equivalent to the usual static problem.
- **Dynamic sub-problem**: Given the solution to the static sub-problem, monitor the execution of the schedule. Should a dynamic event invalidate the current schedule, then repair it while trying to maintain the quality of the current solution and continue execution.

Policella & Rasconi developed a testset generator for the dynamic sub-problem. A model was defined to allow for a variety of dynamic events. An absolute event time is associated with each dynamic event in order to determine when an event occurs. Using a relaxed version of the scheduling problem allowed them to compute the feasible range for the starting times of all activities. However, to ensure that the event times remain valid throughout execution, it was necessary to make certain restrictions to their dynamic events. For instance, an activity could not be completed earlier than expected. We overcome these limitations by using dynamic event times while also ensuring that no invalid events are created.

Dynamic Resource Scheduling Framework

We describe a dynamic resource scheduling framework which can be applied to many different kinds of dynamic resource scheduling problems. We assume the existence of renewable unit resources and schedule tasks subject to temporal and resource constraints. Since this framework has been designed for tasks with multiple modes of execution, it can also be used for single-mode problems by simply specifying only one mode. Furthermore, we assume semi preemption (Hua & Qu 2003) meaning the execution of a task may be interrupted but must be restarted completely. However, the framework can be easily extended to cover non-preemptive scheduling[1] by implementing only scheduling algorithms that will not consider retracting a task that is currently executing. Additionally, this scheduling framework can be applied to oversubscribed as well as undersubscribed dynamic resource scheduling problems.

The dynamic resource scheduling framework is implemented in Java and has three components: a random problem generator, a dynamic simulator and a scheduler. The random problem generator is a stand-alone component and can be used to create instances of the problem. It provides great flexibility in generating datasets for dynamic resource scheduling problems. The specification of the problem and the parameters for all missions and events are passed into the problem generator. This makes it as general as possible in order to allow for generating benchmark datasets with very different kind of characteristics. Changing a single parameter value for an event might cause the dynamic event to have a very different influence on the whole scheduling problem. By inputting the problem specification we ensure that the problem generator can be applied easily to different dynamic scheduling problems. This is achieved by making the appropriate changes in the specification file.
Dynamic events that are generated by the problem generator include:

- resources can be either added or deleted from the problem
- new missions and tasks can be added to the problem
- tasks can be completed earlier or later than anticipated
- a task can be delayed which alters the tasks time window

The dynamic simulator parses the resulting dataset and creates all tasks and events at the appropriate time. The simulator is necessary to hide all future events from the scheduler. Every time an event occurs, the scheduler is invoked in order to make adjustments to the schedule to accommodate the new event. Figure 1 gives an overview of the dynamic resource scheduling framework.

Random Problem Generator

While there exist more general problem generators that can be used to generate datasets for many different static

[1] In non-preemptive scheduling, tasks must be executed to completion and may never be interrupted.

scheduling problems, the same cannot be said for their dynamic counterparts. We address the need for such random problem generators by creating one for dynamic resource scheduling problems as part of a larger framework.

Input Files

The random problem generator requires two input files, the Scheduling Problem file and the Mission and Event file.

Scheduling Problem file This file lists all capabilities, bases and resources that exist within the scheduling problem, respectively. These items have to be specified exactly as stated in the description of our model in the previous section.

Mission and Event file This file contains all parameter values for any mission, task or dynamic event type that is defined in the dynamic resource scheduling problem. Changing just a single parameter might have a strong effect on the characteristics of the generated datasets. Dynamic events can also specify different parameter values for different task types. A simple example might look as follows:

```
horizon         0 1440
numBases        2
numResources    3
events
delay           probability=0.1
                time=normal(10,4)
tasks
transport       numStatic=poisson(0.9)
                numDynamic=5
                priority=random(5,10)
                relativeTime=normal(60,10)
```

This scheduling problem contains 2 bases and 3 resources and runs for 24 hours. The number of transport tasks known at the beginning follows poisson distribution while 5 transport tasks will be added throughtout the simulation.

This input file contains all the information that is necessary for the random problem generator to create benchmark datasets. In fact, by specifying no dynamic events and setting the *numDynamic* parameter of all task types to zero, we could use this framework for static resource scheduling problems. By modifying the input files, we can introduce additional flexibility. We associate a priority with each mission and assume that the objective function is to maximize the sum of priorities of completed missions. This objective can easily be modified to maximize the number of completed missions by setting all priorities to 1. Similarly, we can remove time window constraints from the problem by setting the time windows for each task to $(-\infty, \infty)$.

Dynamic Events

In this section, we provide a detailed description of the dynamic events that have already been implemented into our framework. These events are very common and apply to most dynamic scheduling problems.

New Mission This dynamic event introduces a new mission during execution of the problem. The mission must consist of at least one task that needs to be executed in order to achieve some goal.

New Task The new task event adds a task to the dynamic scheduling problem. It must be created in the body of a mission or a task.

Add Resource This event dynamically adds a resource to the scheduling problem. To be consistent with our model, we assume that the new resource is also a renewable unit resource. Additionally, the type of the new resource has to be from the set of possible resource types specified initially in the problem instance.

Remove Resource This event removes a resource from the scheduling problem. If, at the moment of removal, the resource was currently executing a task, it will be unscheduled. Similarly, all tasks that were assigned to the resource to be executed in the future will also be unassigned. It is the task of the scheduler to reschedule them on one of the remaining resources.

Disable Resource This dynamic event disables a resource for a period of time. The purpose of this event is to simulate that a resource encounters a mechanical problem which needs to be fixed. The selected resource is removed from the scheduling problem and added again after the specified delay. We assume the resource will remain at the current location. As a consequence of the disable resource event, all the tasks that were assigned to it, will be unscheduled. This includes the currently executing task as well as all its future tasks.

Delay Task This dynamic event shifts time window of a task by a given delay. This delay can be positive or negative.

Change Duration The change duration event modifies the required execution time of a task for the assigned resource. The purpose of this event is to simulate unexpected events that might occur during execution which have an effect on the duration. For instance, a plane might arrive early because of strong tailwind. Additionally, this event may be used instead of the disable resource event, for example when a vehicle runs out of gas. In such an event, we know that the problem can be resolved very quickly, and we can simulate the resulting delay without having to unschedule all assigned tasks.

Other Issues

Mission Event Times Suppose we were to generate event times such that the latest finish time of all tasks lies within the scheduling horizon. This way every task can be completed before the end of the scheduling horizon. Suppose further, we schedule tasks which typically require 500 minutes to execute and assume that the end of the scheduling horizon is set to 800. Then, all missions have earliest starting times ≤ 300 and as a result there will be 500 more minutes during which no additional mission is created.

Instead, the random problem generator tries to spread out the generated tasks. We uniformly distribute the absolute event times for the creation of missions. As a result, there

might be several tasks which cannot be executed completely before the simulator halts. We deal with that problem by considering these tasks as completed as long as the scheduling algorithm was able to assign them to a resource such that they can be executed within their respective time windows.

Time Windows An important consideration for dynamic resource scheduling datasets with task time windows is the size of the generated time windows. On the one hand they should be large enough so that they can be completed successfully. But on the other hand the generated problem instances become too easy if task time windows are too large.

We require that any dataset with a single task and a single resource should be solved optimally. Therefore, in computing the size of the time window we need to include the time it takes the assigned resource to get to the starting location of the task. Since we are dealing with multi modes for execution, we set it to the sum of the best positioning time and the average duration.

Precedence constraints Precedence constraints express the starting time of an activity in terms of another activity's starting time. Precedence constraints can be specified between two tasks belonging to the same mission and between a task and the mission itself. It turns out that implementing these kinds of constraints into our dynamic scheduling framework is not that simple.

Assume there exists a task A which adds two subtasks to the problem instance sometime during its execution. We denote the subtasks of A as B and C. Suppose we want to add a precedence constraint between B and C with delay d. Assume that B is created earlier than task C. After running the scheduling algorithm, task B might start execution right away. Eventually task C will be created, but what if the other task has already been executing for d or more minutes? Then the constraint has been violated without any fault of the scheduler. It is not possible to guarantee that this precedence constraint is obeyed at all times.

Consider the following solution to the problem: Instead of adding the given precedence constraint to the scheduling problem, modify the dataset by moving the task C into the body of task B. Setting the release date of task C to a value $\geq d$ enforces the constraint. Hence, we can completely ignore the use of precedence constraints if we generate datasets such that they are implied automatically by the definition of our dynamic scheduling model.

Dynamic Simulator

Figure 2 shows an overview of the dynamic simulator, which forms a feedback loop with the underlying scheduling problem. Given a problem instance to solve, the scheduler produces a new schedule. The execution of this schedule produces a stream of events which are interpreted over time by the simulator. The results of these events are a sequence of incremental changes to the scheduling problem which are then iteratively re-solved by the scheduling algorithm. Each new schedule may produce more events in the future as scheduled tasks are being executed. This process is driven by a simulation clock which iterates through the scheduling horizon.

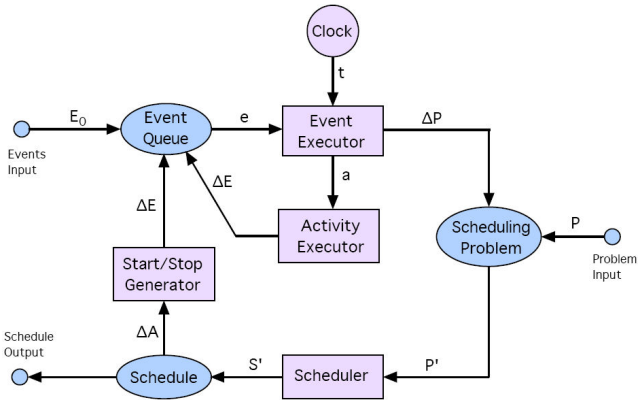

Figure 2: Dynamic Simulator Model

The Event Queue stores all future events E to be processed by the Event Executor. The Start/Stop Generator generates events representing the start and end of the execution of a task. The Activity Executor parses the body of missions and tasks and creates the corresponding events. In the event that a task has been completed or terminated abnormally, the Activity Executor is also responsible for removing the corresponding future events from the queue. The Event Executor module uses the Event Queue to organize events in temporal order and repeatedly removes the first event e from the queue. If this event is a dynamic event which introduces a modification ΔP to the scheduling problem, it is executed and the scheduler is invoked.

Visualization Tool The dynamic simulator includes a visualization tool which creates an animation of the scheduling problem on $GoogleEarth^{TM}$. Models for resources were obtained from an online database[2]. The animation steps through the scheduling horizon and visualizes the different entities. It is even possible to halt the simulation clock anytime in order to investigate some state in detail. Figure 3 shows a sample screenshots of the visualization tool for a Search and Rescue mission off the west coast of Vancouver Island, British Columbia.

Data Model

We describe a general model for dynamic multi-mode resource scheduling problems with unit resources subject to temporal and resource constraints. For a task we assume that there are multiple modes of execution and its duration depends on the assigned resource.

We extend static resource scheduling problems to include dynamic events where tasks and resources can be added, modified and deleted from the schedule during execution thereby possibly interrupting some already scheduled tasks. Unlike Policella & Rasconi, we use relative event times while guaranteeing that no dynamic event will occur at an

[2]3D Warehouse, http://sketchup.google.com/3dwarehouse/

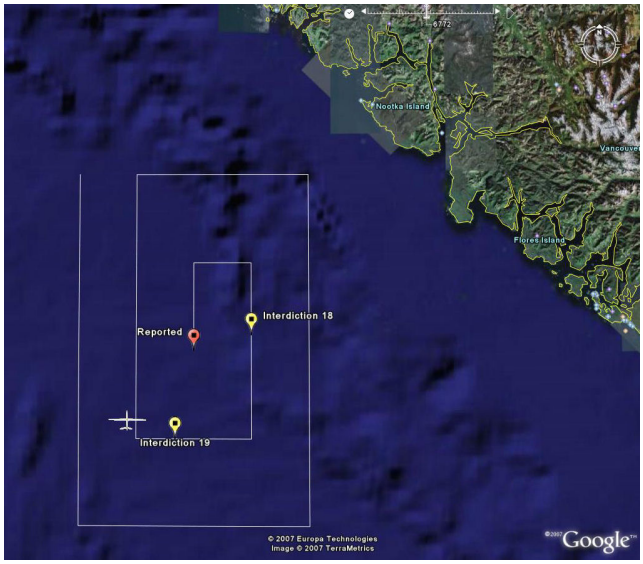

Figure 3: Visualization Tool - SAR mission

infeasible event time. In the following sections, we will describe each entity in detail along with a simple BNF syntax that we have implemented into our dynamic resource scheduling framework.

Scheduling problems

A scheduling problem for the CoastWatch project contains the following entities. Note that our generator is not specifically tied to the Search and Rescue domain.

- **Bases**: Bases are the home locations of resources. This includes air bases for aircrafts and ports for ships.
- **Resources**: Resources such as aircraft, helicopters and ships, execute tasks and have a designated home base.
- **Capabilities**: A mapping of task requirements to resource capabilities. Tasks specify the capability that is necessary to execute them and every resource has a pre-determined list of capabilities they can perform.
- **Tasks**: The activities that are scheduled and executed according to their time window and resource requirements.
- **Missions**: A partially ordered set of tasks that have to be completed to achieve some mission with specified priority.

We propose a simple regular language in BNF to specify a dynamic resource scheduling problem. We use '*' to denote zero or more of the preceding element and '+' to denote one or more repetitions.

```
<capability>*
<base>*
<resource>*
problem <horizon>
<event>*
```

First we specify all capabilities, bases and resources that exist within the problem. We then set the scheduling horizon by providing a start time and an end time for the schedule followed by a list of events, which include the creation of new missions. The syntax for the horizon is:

```
<horizon> --> (<start-time>, <end-time>)
```

where <start-time> and <end-time> are integers such that <start-time> $\leq$ <end-time>. Typically, <start-time> = 0. We assume that time is measured in minutes but any other integral time unit should also be acceptable. For instance, a scheduling problem spanning 24 hours would be specified in minutes as:

```
problem (0,1440)
```

Bases Bases give a physical start location for resources at the start of the scheduling horizon. They are defined by a unique base name and a location, which is specified by its latitude and longitude values. We assume that bases are at sea level. The syntax for defining a base is:

```
<base> --> base <id> <location>
```

where the location is defined as follows:

```
<location> --> (<lat>,<long>)
```

Resources Resources are defined by a unique identifier and a resource type. They move at a pre-determined speed (in km/h) and are assigned a home base which is the starting location at the beginning of the scheduling horizon. Their syntax is:

```
<resource> --> resource <resource-type>
               <id> <base> <speed>
```

Depending on the types of resources in the scheduling problem, <resource-type> may be very general (plane, ship, ...) or very specific (F/A-18 Hornet, Boeing 747, ...).

Capabilities The <task-type> of a task field specifies the capability a resource requires in order to be able to execute it. A capability is a mapping of a task type to a set of resource types which are able to perform it. The syntax for this relation is the following:

```
<capability> --> capability <task-type>
                 (<resource-type>*)
```

For example, a rescue task out in the ocean could be specified as follows:

```
capability rescue (helicopter ship)
```

For resources that have specialized capabilities, we can define subclasses by name which possess those capabilities.

Constraints

Time window constraints Given a task t, its time window specifies the earliest possible starting time est_t and the latest possible finish time lft_t. A task must not be executed earlier than the given est_t nor later than the given lft_t.

Resource constraints Resources are renewable, meaning that they can serve another task as soon as their current task is completed. Additionally, resources may only execute one task at a time. Similarly, a task can use only one resource for execution which will execute it from the very beginning to the very end.

Missions & Tasks

Missions in CoastWatch may be made up of a several tasks. For example, in Search & Rescue (SAR), a search task has to be completed before the rescue task is initiated. We define a mission to be a collection of tasks that need to be executed. The mission is only considered accomplished if all of its tasks have been completed successfully. As a consequence, priorities are specified with missions rather than the tasks themselves. On the other hand, execution time windows are associated with tasks. This is because not all tasks of a mission are known at beginning of execution and some tasks are created dynamically.

It is important to note that these definitions for missions and tasks do not exclude in any way activities that are only composed of one task.

Missions The syntax for defining a mission is:

```
<mission> --> mission <id> <priority>
   {<new-task> <body>} {<precedence>*}
```

where <id> is a unique identifier and <priority> is a positive integer specifying the priority of the mission with value 1 representing the lowest possible priority. A mission must have at least one "New Task" event that creates a new task. The body of a mission contains a set of dynamic events, possibly including more new tasks, and is executed once the mission is introduced into the scheduling problem. Finally, <precedence>* is the set of precedence constraints that must be obeyed. There may exist at most one precedence constraint between any ordered pair of tasks belonging to this mission.

Tasks Each mission contains a partially ordered set of tasks that need to be executed to complete the mission. Tasks are defined by a unique identifier, a task type and a time window for execution. Their syntax is:

```
<task> --> task <time-window> <task-type>
           <id> {<body>} {<precendence>*}
```

where the body of a task is a set of dynamic events that is parsed once the execution of this task has started. A task is deemed to execute when the time of the simulator reaches its start time. Similarly to missions, there may exist at most one precedence constraint between any ordered pair of subtasks. The field <time-window> specifies the earliest start time (EST) and latest finish time for a task (LFT). The syntax for time windows is:

```
<time-window> --> (<EST>,<LFT>)
```

where both fields are positive integers such that <EST> $\leq$ <LFT>.

Body In dynamic scheduling problems, unexpected events can occur during the execution of tasks. We model this behaviour by associating a set of statements, called the *body*, to tasks and missions.

The <body> field defines the set of changes to the scheduling problem which can occur as a result of scheduling and executing a task. The syntax is as follows:

```
<body> --> <event>*
```

where <event>* is the set of unexpected events. For missions, these statements are evaluated when the mission is created, which may contain the creation of new tasks as well as mission events. For tasks, evaluation happens when the execution of the task commences and possible event types include new subtasks as well as task events.

Dynamic Scheduling Events

Static scheduling problem models are not concerned with simulating the execution of tasks in their schedules. In dynamic scheduling we assume that executing a task at a particular time affects the world and introduces changes to the scheduling problem itself. In our model we use relative event times while guaranteeing that no invalid events are create. To achieve this, we differentiate between regular events, task events and mission events.

Regular Events Regular events are events that do not directly influence tasks or missions. Examples are the addition or removal of a resource. For these kinds of events, obeying causality is very straight-forward, because we are not concerned with the schedule produced by running the scheduling algorithm. Removing a resource can happen anytime and under any circumstances. One still has to be careful, though, because, for instance, if the same resource breaks down twice during the scheduling horizon, the second event should happen after the resource has been fixed. The syntax for regular events is as follows:

```
<event> --> <event-time> <event-type>
            <additional-parameters>*
```

where <event-time> is an absolute time within the scheduling horizon and <event-type> is a name that uniquely identifies the type of the event. This is followed by an optional set of additional parameters.

Task Events Task events are events that directly influence a task such as the change of its duration or the addition of a new sub task. These events differ significantly from regular events, because their event time t_{rel} is relative. We impose the restriction that $0 \leq t_{rel} \leq 1$ and treat the event time as a percentage of the task duration. For a task a, the absolute event time t_{abs} can be computed by the following formula:

$$t_{abs} = \textit{start-time}_a + t_{rel} * duration_a$$

It is guaranteed that all events will occur during execution of the task since the relative event time is a fraction of the total task duration. When a SAR mission is being executed we can create the rescue task anywhere during the execution of the search task. For instance, an event time of 0.9 would signal that the missing person is found after completing 90% of the search path. Assigning various resources with various speeds to the same rescue task, will result in different absolute event times. However, the rescue location will always be the same.

To guarantee that no invalid events will ever occur, we need to make sure that all dynamic events also obey causality. In particular, we need to ensure that neither start nor end time of a task can shift into the past. Luckily, the use of

relative event times simplifies this issue significantly. Suppose there is a dynamic event which lowers the duration of a task. If the event happens at relative event time t_{rel}, then the delay $delay_e$ must obey the constraint $delay_e \geq (t_{rel}$ - 1). This ensures that the event obeys causality; the updated end time of the task cannot be in the past after execution of the event. We can use the same argument for any other task event: we are aware exactly how far into the task execution the event happens and consequently, we know the maximum shifts that are possible.

Task events can be defined as follows:

```
<event> --> <event-time> <event-type>
   <task-id> <additional-parameters>*
```

where <event-type> must be the name of a task event and <task-id> the unique identifier of a previously defined task.

Mission Events Mission events are events that influence a mission or one of its tasks. Examples include adjusting the mission priority or adding a new task. Additionally, delaying a task (i.e. shifting it's time window) should also be considered a mission task. This is because the delay has to happen before the start of a task while task events only get executed once execution has commenced.

For mission events, the event time t_{rel} is also relative. But here it is relative to the creation of the mission and t_{rel} doesn't represent a fraction because it is independent from resources. For a mission *m*, the absolute event time t_{abs} can be computed by the following formula:

$$t_{abs} = \textit{creation-time}_m + t_{rel}.$$

Obeying causality is very straight-forward for mission events. We need to ensure that the task doesn't start executing before the delay task event is executed. This is achieved, by setting *t* to be smaller than the task's earliest starting time *est*. In addition, if the delay *d* is negative, that is a task can be started earlier than first anticipated, we need to ensure that the event doesn't move *est* into the past. This can be achieved by choosing a value for delay during event generation such that it obeys the constraint $d \geq (t$ - *est*). We can give a similar argument for any possible mission event.

Case-Study: CoastWatch

CoastWatch is an oversubscribed dynamic multi-mode scheduling problem with unit resources. The task is to schedule both routine and emergency missions within a Search & Rescue (SAR) operational command. There are more routine patrol missions than can be flown by the available resources. Unexpected SAR missions are of highest priority and must be accommodated in the schedule if possible.

Problem Definition

CoastWatch can be defined as follows:

- **Missions**. *M* = $\{m_1, m_2, ..., m_n\}$ is a set of missions which have to be completed. A mission has a priority p_i and is composed of a set of tasks T_i, all of which have to be completed in order for the mission to be considered accomplished. Every mission has an associated set of dynamic events called its *body*. These events occur at specified times after the creation of the mission.
- **Tasks**. *T* = $\{T_1, T_2, ..., T_n\}$ is a set of set of tasks which have to be scheduled. T_i contains all tasks that belong to mission m_i. Similar to missions, every task has a *body* which contains a set of dynamic events that occur at specified times after the start of execution of this task. A task is characterized by the following parameters:
  - *rd*: the release date of the task. It must be scheduled at this time or later.
  - *dd*: the due date of the task. Execution must terminate on or before this time in any schedule.
  - *CR*: the set of resources which can service the task. We also refer to them as capable resources.
  - *type*: the type of the task. This parameter determines the set of capable resources.
  - *D*: the set of durations containing the execution times of the task depending on the assigned resources. The duration may be altered during execution by dynamic events.

  The different types of tasks in CoastWatch are: Search, Interdiction (identifying an object), Rescue, Transport and Patrol.
- **Body**. A set of dynamic events associated with a task or mission. These events, which may affect the underlying scheduling problem, occur at specified times after the creation of the mission or the execution of the task.
- **Resources**. *R* = $\{r_1, r_2, ..., r_k\}$ is a set of renewable unit resources which are scheduled to perform tasks. *C* is the set of capabilities, or task types, a resource is able to perform.

The scheduling problem is semi-preemptive, meaning the execution of a task may be interrupted and restarted from the beginning at a later time.

Benchmark Datasets Generation

This section contains a description of all parameters values that we have selected for generating benchmark datasets for the CoastWatch dynamic resource scheduling problem.

- **Scheduling horizon**: 0 to 1440. We measure time in minutes and set the size of the scheduling horizon to equal a whole day.
- **numBases**: 4. We define a set of 4 real-world bases and include them in every dataset.
- **numResources**: 10. We specify 18 different resources in the scheduling problem file for CoastWatch: 2 Aurora aircrafts, 4 Cormorant and 4 Cyclone helicopters, 4 Eagle Unmanned Aerial Vehicles and 4 Frigate ships. Aurora aircrafts are kept very scarce, since they are much faster and would otherwise dominate the other resources by performing most of the tasks. Every problem instance contains 1 resource for every type and randomly selects the remaining 5.

Dynamic Events

Delay Task We assume that 10% of all tasks will be delayed. However, we allow no such event for interdiction and rescue tasks. Additionally, the execution of search tasks should never be delayed. But this is already guaranteed because we set its earliest possible starting time to be the time at which the task was created. Additionally, we set the maximum possible delay to be one hour and allow a task to be started at most 10 minutes earlier than first anticipated.

Change Duration 20% of all tasks, excluding interdiction tasks, experience a change in its duration. The relative time is set to a value between 1 and 99 meaning this event can occur anywhere during the execution of a task. A task can be executed up to 10% faster than first anticipated, which equals a delay of -10%, but its duration may be increased by up to 25%.

Disable Resource In our experiments, we simulate a typical day for the Canadian Coast Guard by assuming that two resources will experience technical difficulties and be temporarily disabled. We assume that repair will take anywhere from 30 minutes to 2 hours.

Conclusion

As more and more researchers are working on dynamic scheduling problems, the need for good problem generators will only increase over time. We have taken one step towards this direction: developing a random problem generator that is flexible enough to be used for many different kinds of dynamic resource scheduling problems. In this paper we described a framework for dynamic resource scheduling problems with unit resources subject to temporal and resource constraints. It is composed of three components: a random problem generator, a dynamic simulator and a scheduler. We proposed a model for dynamic resource scheduling problems and incorporated it into our framework. We hope that the development of this dynamic resource scheduling framework will simplify the work required to attack such problems. This paper is an attempt to spark more interest in studying dynamic scheduling problems.

Unlike Policella & Rasconi, we use relative event times for dynamic events while guaranteeing that all event times will be valid. We achieve this by differentiating between three different types of dynamic events: regular events, task events and mission events. Regular events have an absolute event time anywhere within the scheduling horizon. The event time for mission events is relative to the creation of the mission. For task events, the event time is also relative, but unlike mission events it represents a percentage. The dynamic event is created after the assigned resource has completed the specified percentage of the parent task.

We performed a case-study on the CoastWatch problem whose goal is to schedule both routine and emergency missions within a Search & Rescue operational command. Possible future work could include generating more datasets for the CoastWatch problem. It would be interesting to see if changing some of these parameters changes the characteristics of the generated datasets significantly. Additionally, it would be beneficial to identify a subset of parameters that significantly influences the difficulty of the generated datasets.

The size of the execution time windows for tasks can have a significant impact on the difficulty of the resulting problem instances. Consequently, we need to test several strategies for determining their size and analyze the resulting datasets. Currently, we generate the execution time window for a task by considering the positioning times and durations of the capable resources. However, there is a disadvantage to this approach: resources that are either much faster or much slower than other ones, influence the resulting size significantly. In the future, we could look for alternative ways such that time window sizes are not dependent on the available resources.

Acknowledgements

This research has been supported in part by Precarn Associates and the Natural Sciences and Engineering Research Council of Canada.

References

Brucker, P.; Drexl, A.; Mohring, R.; Neumann, K.; and Pesch, E. January 1999. Resource-constrained project scheduling: Notation, classification, models, and methods. *European Journal of Operational Research* 112:3–41 30 31 32 33 34.

Drexl, A.; Nissen, R.; Patterson, J. H.; and Salewski, F. 2000. Progen/pix - an instance generator for resource-constrained project scheduling problems with partially renewable resources and further extensions. *European Journal of Operational Research* 125:59–72(14).

Hoos, H., and Stuetzle, T. 2005. *Stochastic Local Search: Foundations & Applications*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc.

Hua, S., and Qu, G. 2003. A new quality of service metric for hard/soft real-time applications. In *ITCC '03: Proceedings of the International Conference on Information Technology: Computers and Communications*, 347. Washington, DC, USA: IEEE Computer Society.

Jackson, L. E., and Rouskas, G. N. 2002. Deterministic preemptive scheduling of real-time tasks. *Computer* 35(5):72–79.

Kolisch, R.; Sprecher, A.; and Drexl, A. 1995. Characterization and generation of a general class of resource-constrained project scheduling problems. *Manage. Sci.* 41(10):1693–1703.

Policella, N., and Rasconi, R. 2005. Designing a testset generator for reactive scheduling. *Intelligenza Artificiale* 3:29–36.