SANDIA REPORT

SAND2018-9329
Unlimited Release
Printed August 2018

Schedule Management Optimization (SMO) Domain Model: Version 1.2

Peter B. Backlund, Darryl J. Melander, Adam J. Pierson, John A. Flory, Alexander I. Dessanti, Stephen M. Henry, John H. Gauthier

Prepared by
Sandia National Laboratories
Albuquerque, New Mexico 87185 and Livermore, California 94550

Sandia National Laboratories is a multimission laboratory managed and operated by National Technology and Engineering Solutions of Sandia, LLC, a wholly owned subsidiary of Honeywell International, Inc., for the U.S. Department of Energy's National Nuclear Security Administration under contract DE-NA0003525.

Issued by Sandia National Laboratories, operated for the United States Department of Energy by National Technology and Engineering Solutions of Sandia, LLC.

Printed in the United States of America. This report has been reproduced directly from the best available copy.

Available to DOE and DOE contractors from
U.S. Department of Energy
Office of Scientific and Technical Information
P.O. Box 62
Oak Ridge, TN 37831

Telephone: (865) 576-8401
Facsimile: (865) 576-5728
E-Mail: reports@osti.gov
Online ordering: http://www.osti.gov/scitech

Available to the public from
U.S. Department of Commerce
National Technical Information Service
5285 Port Royal Rd.
Springfield, VA 22161

Telephone: (800) 553-6847
Facsimile: (703) 605-6900
E-Mail: orders@ntis.gov
Online order: http://classic.ntis.gov/help/order-methods/

SAND2018-9329
Unlimited Release
Printed August 2018

Schedule Management Optimization (SMO) Domain Model: Version 1.2

Peter B. Backlund
Systems Technology, Org. 5837

Darryl J. Melander
Math Analysis & Decision Science, Org. 8834

Adam J. Pierson
Mission Algorithms R&S, Org. 9365

Alexander I. Dessanti, John A. Flory, John H. Gauthier, and Stephen M. Henry
Advanced Decision Analytics, Org. 5493

Sandia National Laboratories
P.O. Box 5800
Albuquerque, New Mexico 87185

Abstract

Schedule Management Optimization (SMO) is a tool for automatically generating a schedule of project tasks. Project scheduling is traditionally achieved with the use of commercial project management software or case-specific optimization formulations. Commercial software packages are useful tools for managing and visualizing copious amounts of project task data. However, their ability to automatically generate optimized schedules is limited. Furthermore, there are many real-world constraints and decision variables that commercial packages ignore. Case-specific optimization formulations effectively identify schedules that optimize one or more objectives for a specific problem, but they are unable to handle a diverse selection of scheduling problems. SMO enables practitioners to generate optimal project schedules automatically while considering a broad range of real-world problem characteristics. SMO has been designed to handle some of the most difficult scheduling problems – those with resource constraints, multiple objectives, multiple inventories, and diverse ways of performing tasks. This report contains descriptions of the SMO modeling concepts and explains how they map to real-world scheduling considerations.

CONTENTS

1 Introduction 7

2 Project Structure 9
- 2.1 Projects 9
- 2.2 Tasks 10
- 2.3 Modes 10

3 Resources 13
- 3.1 Resource Description 13
- 3.2 Resource Requirement Schedules 13

4 Products 15
- 4.1 Product Description 15
- 4.2 Product Consumption and Generation 15
- 4.3 Product Inventory 16
- 4.4 Product Refills 17

5 The Schedule Builder and Evaluator 19
- 5.1 Building Resource- and Precedence-Feasible Schedules 19
- 5.2 Building Product Inventory-Feasible Schedules 21

6 Business Requirements 23
- 6.1 Business Requirements Overview 23
- 6.2 Business Requirements Objective 23

7 Multi-Objective Optimization 25
- 7.1 Multi-Objective Genetic Algorithm Overview 25
- 7.2 SMO Objective Functions 25
  - 7.2.1 Response Group 1: Minimize Total Cost 26
  - 7.2.2 Response Group 2: Maximize Requirement Coverage 26
  - 7.2.3 Response Group 3: Maximize Schedule Performance 26
  - 7.2.4 Response Group 4: Minimize Risk 27
- 7.3 SMO Constraints 28
- 7.4 Response Function Normalization 28
- 7.5 Using a Genetic Algorithm for Scheduling 29
  - 7.5.1 Initialization 29
  - 7.5.2 Crossover 30
  - 7.5.3 Mutation 31
- 7.6 Algorithm Options 32
  - 7.6.1 Random Seed 32
  - 7.6.2 Population Size 32
  - 7.6.3 Fitness Assessor 33
  - 7.6.4 Evaluation Concurrency 33
  - 7.6.5 Domination Cutoff and Shrinkage Rate 33
  - 7.6.6 Crossover Rate and Crossover Points 33
  - 7.6.7 Mutation Rate, Depth and Repeats 33
  - 7.6.8 Max Generations 34
  - 7.6.9 Max Evaluations 34

7.6.10 Max Time 34
7.6.11 Tracked Percent Change and Tracked Generations 34
7.6.12 Niching 34
7.6.13 Logging 35
References 37
Distribution 39

FIGURES

Figure 1: SMO project and task overview 9
Figure 2: SMO resource overview 13
Figure 3: SMO product store overview 15
Figure 4: Example precedence graph 20
Figure 5: Schedule built from task order [2 4 1 6 3 5 7] 20
Figure 6: A Pareto set of multi-objective optimization solutions in the performance (maximized) vs. cost (minimized) trade space 25
Figure 7: SMO response groups 26
Figure 8: Schedules with (a) highest risk, (b) medium risk, and (c) lowest risk 27
Figure 9: Example normalization function for maximization 29
Figure 10: Precedence graph for crossover examples 30
Figure 11: One-point crossover example 30
Figure 12: Two-point crossover example 30
Figure 13: Multi-point crossover example 31
Figure 14: Uniform crossover example 31
Figure 15: Example task ordering mutation operation 32

TABLES

Table 1: Example resource requirement schedule 14
Table 2: Product consumption example 15
Table 3: Product generation example 16
Table 4: Example scheduling problem 19

1 INTRODUCTION

Scheduling is the process of timing a set of tasks to complete a project. Creating schedules that meet delivery dates and minimize project duration with limited human, fiscal, and material resources is a significant challenge. Real-world scheduling problems have many characteristics that must be considered, such as resource constraints, precedence relationships among tasks, multiple ways of completing a task, concurrent projects that compete for resources, inventory constraints, and production and consumption of materials (to name a few).

Project scheduling is traditionally achieved with the use of commercial project management software or case-specific optimization formulations. Commercial software packages are useful tools for managing and visualizing large amounts of project task data. However, their ability to generate optimized schedules is limited and is typically aimed at resolving resource conflicts rather than optimizing a schedule according to the goals of the user. Furthermore, these packages do not explicitly account for many real-world constraints and decisions. Case-specific optimization formulations can be used to effectively identify schedules that optimize one or more objectives for a specific problem, but they require significant effort and an analyst with expertise in optimization formulation and programming.

Schedule Management Optimization (SMO) is a tool for automatically generating schedules that considers a broad range of real-world scheduling constraints. The resulting schedule specifies when each task should start, and how each task should be completed. Users of SMO can model a resource-constrained project scheduling problem using a simple, generic modeling language in a user-friendly environment, thus eliminating the need for a specialized optimization formulation and modeling expertise. SMO uses a genetic algorithm to identify a set of resource-feasible schedules that are optimized to multiple competing objectives. This multi-objective approach currently enables users to explore tradeoffs between cost, duration, lateness, resource utilization, and risk. Other objectives may be added in future releases.

This report provides an overview of SMO's modeling concepts and explains how the constructs used in SMO map to practical scheduling decisions, constraints, and objectives.

2 PROJECT STRUCTURE

SMO inputs are based on a language that describes the basic elements of most scheduling problems: projects/tasks, resources, and products/inventories. Figure 1 gives an overview of the project- and task-related constructs.

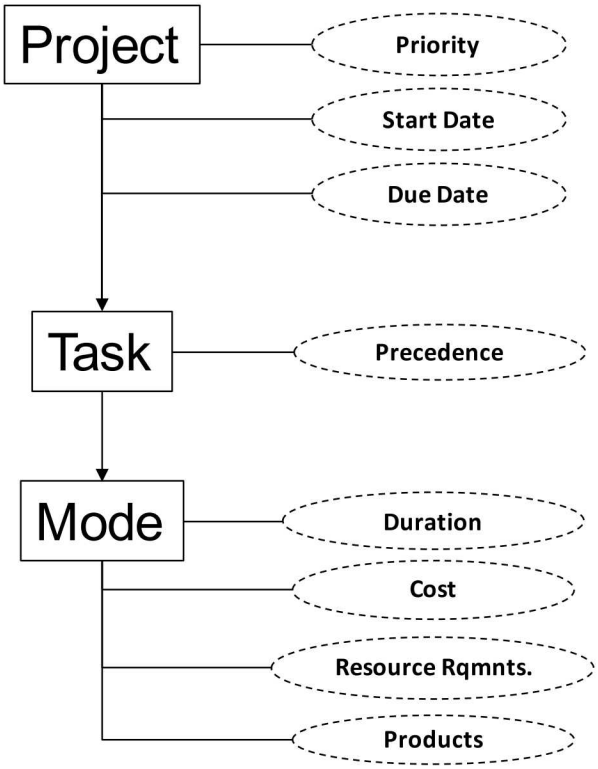

Figure 1: SMO project and task overview

In SMO, time is discretized into individual units called **time periods**. A time period can represent any length of time (minute, hour, day, 0.5 weeks, etc.). However, all time periods in an SMO model must be the same length. E.g., if Task 1 takes 5 days, and Task 2 takes 2 weeks, the model could assume that each time period represents one day, and Task 1's duration would be 5 time periods and Task 2's duration would be 14. In this case, you cannot, however, assume that each time period represents a week, because then Task 1's duration could not be represented with an integer value.

2.1 Projects

A project defines a set of tasks that once completed, achieves a high-level objective. For example, in building a housing development, each individual house could be a project. The tasks of a project would consist of all the activities necessary to construct each respective house. Other projects may be to construct the streets of the housing development or a park.

Projects can be given a priority[1], which defines the relative order in which a project's tasks are scheduled with respect to the tasks of other projects. For example, if Projects A, B, and C have

[1] Project priority is not implemented in the current release. All projects are treated with equal priority.

priorities 1.0, 3.4, and 8.0, respectively, then Project A's tasks will be scheduled first followed by those of Project B and finally Project C. If Project B's tasks use different resources and products from those of Project A, the prioritization would not affect the scheduling of Project B's tasks. However, if Project B's tasks use some or all the same resources as Project A's tasks, Project B's tasks will have to be scheduled in time periods where the commonly used resources are unclaimed by Project A's tasks.

2.2 Tasks

A task represents an activity to be completed. Every task will be included in the final schedule exactly once.

There may be more than one way to complete a task. Each alternative method of completing the task is called a mode (Section 2.3).

Tasks take time to complete, and often require resources, consume products, or generate products. A task's duration, input requirements, and output generation depend on the mode in which the task was carried out.

There are often relationships between tasks, where a task cannot be started until certain other tasks have been completed. These task order dependencies are specified via precedence relationships. Every precedence relationship involves two tasks, a *predecessor* task and a *successor* task. The predecessor must be completed before the successor can begin. Every task is a successor of its own predecessors, and a predecessor of its successors; defining one side of the relationship automatically defines the other side.

Precedence relationships can be direct or indirect. Indirect relationships arise when precedence relationships are chained together. For example, task A may be a predecessor of task B, and task B a predecessor of task C. In this case, task A is an indirect predecessor of task C because task A must be completed before task C can begin. Analysts only need to define direct precedence relationships.

Every task can have any number of predecessors and successors. However, a task is not allowed to have a precedence relationship with itself, even when considering indirect relationships. In other words, a chain of precedence relationships may not form a loop.

2.3 Modes

Modes define the ways a task can be accomplished. A task can be performed using different types and/or quantities of resources (Section 0) as well as consume or generate different types and/or quantities of products (Section 0).

A task to build a wall could be performed using different numbers of workers and material. In this case, a mode to build the wall using brick could require masons and consume bricks. On the other hand, a mode to build the wall using concrete could require a cement truck and concrete workers while consuming concrete mix.

Associated with each mode is a duration, which is defined as the time required to complete the task from start to finish using that mode. A mode that uses more resources typically has a shorter duration. For example, employing four masons instead of two should reduce the duration required to build a brick wall.

Resource requirements for a mode can vary over each time period of the mode's duration. For example, a mode may require a setup and teardown step in which personnel resources are used at the beginning and end of the task. Product consumption and generation occur at the beginning and end of the task, respectively.

The final attribute associated with modes is cost. Cost typically represents the expense of the resources and product consumed by the mode as well as any other expenses such as labor (which alternatively could be costed as a resource, see below), transport costs, insurance costs, capital costs, etc. More specifically, SMO considers three types of costs associated with each mode: resource costs, product costs, and one-time costs. The resource cost of a mode is the total cost of using each resource that the mode requires. The cost to utilize each resource is a function of the per-unit utilization cost of the resource per-unit-time as well as the quantities of the resource required during each period of the mode's duration. The product cost of a mode is the net expense incurred or revenue generated by consumption and generation of products and is a function of the product's per-unit consumption and revenue values as well as the quantities of each product generated and consumed. Lastly, one-time costs encompass other costs incurred by the mode that are not associated with products or resources (e.g., capital costs, set-up costs).

3 RESOURCES

Figure 2 gives and overview of the scheduling language constructs related to resources.

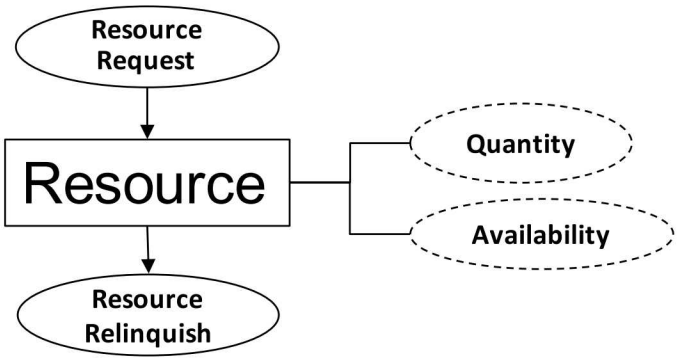

Figure 2: SMO resource overview

3.1 Resource Description

Resources are reusable items or entities that may be required to carry out a task, such as tools, facilities, or personnel. A model may include several types and quantities of resources to be used by tasks.

Each resource can be used by only one task at a time, and tasks can only be scheduled when the resources they require are available, so resource contention has a significant impact on the timing of tasks.

Resources of the same type are grouped into resource pools. When a task needs a certain kind of resource, the required resource is taken from the corresponding pool. The resource is removed from its pool while the task is using it, temporarily reducing the number of resources of that type available to other tasks. When the task is done with the resource, it is returned to the pool and the number of available resources of that type increases.

The resources in a pool are indistinguishable from each other; there is no way to identify a specific mechanic in a resource pool of mechanics, for example. If a specific mechanic is needed, this mechanic should comprise their own pool.

In some cases, a resource may not be available for one or more fixed durations. For example, an employee may have vacation scheduled, or a machine may have maintenance scheduled for a specific date. The analyst can model these situations in SMO by specifying one or more spans of resource non-availability for each resource type. During one of these periods, all resources of a type are unavailable.

3.2 Resource Requirement Schedules

The types and quantities of required resources can vary from one task to the next and from one mode of task execution to the next. The resources required to execute a task in each mode are described by a resource requirement schedule. A resource requirement schedule describes which resources are required, and when, relative to the start of the task.

A resource requirement schedule can be thought of as a list of resource types that will be needed to carry out a task in a mode. For each required resource type, there is a usage schedule that describes how many resources of that type will be required for each time period of the task's

duration. The required number of resources can be constant throughout the task's duration, or it can change from one time period to the next.

An example of a resource requirement schedule is shown in Table 1 below. In this example, a vehicle is to be repaired. The repairs need to be done in a garage bay, so a single *Garage Bay* resource is required throughout the task. The task can be started by a single mechanic without a vehicle lift, but requires an additional mechanic and a vehicle lift in the 2nd and 3rd time periods. The final time period (which might represent final cleanup) no longer requires the additional mechanic or the vehicle lift.

Table 1: Example resource requirement schedule

Resource	Cost per Unit Resource per Unit Time Period	Time Period			
		1	2	3	4
Garage Bay	$100.00	1	1	1	1
Mechanic	$50.00	1	2	2	1
Vehicle Lift	$5.00	0	1	1	0

Note that there may be other ways to carry out the same repair, represented by alternative task modes. Each alternative mode would have its own resource requirement schedule, possibly using diverse types and quantities of resources, and possibly having a different overall duration.

Resources are pulled from resource pools at the time they are needed, which may not be when the task starts. Similarly, resources are returned to their resource pool when they are no longer needed, which may be before the task ends. Also note that tasks can only use resources while the task is in progress. They cannot claim resources before the beginning of the task, nor can they hold on to resources past the end of the task's duration.

Associated with each resource is a cost, which is specified as the cost of using a single unit of the resource for a single time period. The total resource cost of a mode is computed using the cost of each required resource together with the quantities of the resource used during each time period. In the case of repairing the vehicle, the resource cost of the Mechanic is $50.00 x (1 + 2 + 2 + 1) = $300.00. The resource cost of the Garage Bay and Vehicle Lift can be similarly computed as $400.00 and $10.00, respectively, so that the total resource cost is $300.00 + $400.00 + $10.00 = $710.00.

4 PRODUCTS

Figure 3 gives an overview of the scheduling language constructs related to products and product stores/inventories.

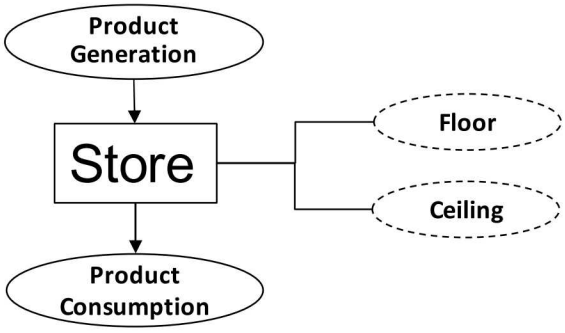

Figure 3: SMO product store overview

4.1 Product Description

Products are items that are consumed or generated by tasks, such as fuel or manufactured goods. Like resources, products may be required to carry out a task. Unlike resources, products cannot be reused – products are consumed when they are used by a task, with the amount of product used by the task permanently removed from the model. However, tasks can generate products as an output, thus replenishing the amount of product available in the product store.

4.2 Product Consumption and Generation

The types and quantities of products consumed to carry out a task can vary from one task to the next, and from one mode of task execution to the next. Similarly, the products generated by tasks can vary from mode to mode. Each mode of task execution has its own list of which products it will consume, and in what quantities, and its own list of which products it will generate, and in what quantities.

Examples of product consumption and generation lists are shown in Table 2 and Table 3 respectively below. In this example, a standard oil change consumes 2 units of engine oil and 1 unit of wiper fluid, and generates 2 units of dirty engine oil.

Table 2: Product consumption example

Product Consumption		
Product	**Quantity**	**Per Unit Cost**
Engine Oil	2	$5.00
Wiper Fluid	1	$1.50

Table 3: Product generation example

Product Generation		
Product	**Quantity**	**Per Unit Revenue**
Dirty Engine Oil	2	$0.00

Associated with product consumption and generation are costs and revenues, respectively. The cost of product consumption is specified as the monetary expenditure associated with consuming a single unit of product. Likewise, the revenue of product generation is specified as the monetary gain associated with generating a single unit of product. In the example of a standard oil change, the cost of product consumption is 2 x $5.00 + 1 x $1.50 = $11.50. There is zero revenue generated, so the total product cost of a standard oil change is $11.50. If the Dirty Engine Oil had a monetary value of $0.50 per unit, then revenue of 2 x $0.50 would be generated giving a total product cost of $10.50. In instances where revenue is generated by consuming product (e.g., removing waste) or cost is incurred by generating product (e.g. generating waste), the consumption cost or generation revenue should be input as negative numbers, respectively.

4.3 Product Inventory

Each type of product is kept in a product store. Each product store starts with some quantity of its product in inventory. As products are generated and consumed by tasks, inventory levels increase and decrease. Products consumed by tasks are removed from inventory when the task begins. Products generated by tasks are added to inventory at the beginning of the first time period after the task completes.

As an example, assume the oil change task described in the previous section has a duration of 2 time periods. If the oil change starts at time period 3, then the amount of engine oil in inventory is reduced by 2 units in time period 3, wiper fluid inventory is reduced by 1 in time period 3, and dirty engine oil inventory is increased by 2 in time period 5.

In addition to inventory changes caused by tasks, the analyst can specify additional changes that should be made to product inventory levels. The analyst specifies the type and quantity of product that should be added to or removed from inventory, and the time at which the change should occur. These analyst-specified inventory changes represent anything other than tasks that will impact the amount of product that should be available to tasks. For example, fuel delivery can be modeled as a set of inventory increases at scheduled delivery times.

Each product store has an upper and a lower inventory limit, specified by the analyst. Inventory overage occurs when inventory levels rise above the upper limit, and underage occurs when inventory levels fall below the lower limit. The overage for a single time period is defined as the amount of the inventory that is above the upper limit in that time period, or zero if the inventory level is at or below the upper limit. The total overage for a product store is the sum of overages for all time periods in the schedule. The total overage for the model is the sum of total overages for all product stores. Underage values are calculated in an analogous manner, but with respect to the inventory level falling below the lower limit.

Inventory limits do not affect the timing of tasks. Tasks are scheduled when resources are available, even if the selected time will cause inventory levels to fall outside the defined bounds.

4.4 Product Refills

The analyst may specify that some or all the product stores be refilled when, upon scheduling a task at its earliest resource- and precedence-feasible start time (Section 5.1), their inventory levels drop below their lower limits.

A refill event for a product store has a duration, a quantity, and resource requirements. The quantity represents the number of units of the product that the store will receive upon completion of the refill. The quantity gets applied to the product store one time period after the last period of the refill duration. Refills do not "top off" a product store. I.e., they do not increase the quantity up to the upper limit of the store. Rather, they increase the inventory by a fixed amount. Each project store can have only one refill type defined for it.

Refills may require resources to complete the refill. If a refill requires resources, they are required for the entire duration of the refill event. This is different than task modes, where resource requirements can vary over the duration of the task. The ability to model variable resource requirements may be implemented in future releases of SMO. The process for scheduling product refills is covered in Section 5.2.

5 THE SCHEDULE BUILDER AND EVALUATOR

SMO uses the basic elements of a scheduling problem (tasks, modes, resources, and products) to build a set of possible schedules. SMO uses a genetic algorithm (GA) to search the realm of possible schedules. To reduce the size of the search space, the variables that the genetic algorithm operates on include a priority ordering of the tasks, and the mode selections of the tasks. Therefore, the task ordering and the task mode selections on which the GA operates do not fully define a schedule. We still need to know when each task will begin. We also must evaluate the goodness of each design in terms of the optimization objectives. Furthermore, we may also be interested in other aspects of the schedule for results reporting purposes, such as resource utilization profiles over time. The schedule builder and evaluator perform these functions.

5.1 Building Resource- and Precedence-Feasible Schedules

The SMO schedule builder and evaluator takes the task ordering and mode selections from the GA and decodes them into a fully-defined schedule. The objectives and constraints are then evaluated and the results returned to the GA, which continues to iterate towards the set of Pareto optimal designs. The schedule building algorithm proceeds as follows:

1. For each task, determine the duration, resource requirements, product consumption, and product generation according to the selected mode.
2. Starting with the first task in the ordering sent from the GA, schedule it to begin at the earliest possible time without violating resource constraints or precedence relationships.
3. Repeat Step 2 for the remaining tasks in the order in which they appear in the task ordering from the GA.

A simple example of this process follows. Consider a problem with seven tasks and one resource type. Assume that there are four units of the resource available. Each task has only 1 mode with resource requirements and durations as shown in Table 4.

Table 4: Example scheduling problem

Task ID	Resource Requirement	Duration
1	2	2
2	2	3
3	3	2
4	3	2
5	2	5
6	1	4
7	2	3

The precedence relationships for the tasks are shown in Figure 4. Figure 4 is an example of a directed acyclic graph (DAG), which is a convenient way to visually represent precedence relationships. In this representation, the arrows are directed from predecessors towards successors. The graph must have no cycles. Otherwise, the schedule would be logically impossible.

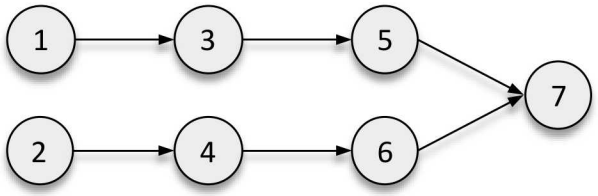

Figure 4: Example precedence graph

Given the precedence relationships, resource requirements, and durations described above, assume that the GA passes the following task order to the schedule builder: [2 4 1 6 3 5 7]. The schedule that results from this ordering is shown in Figure 5.

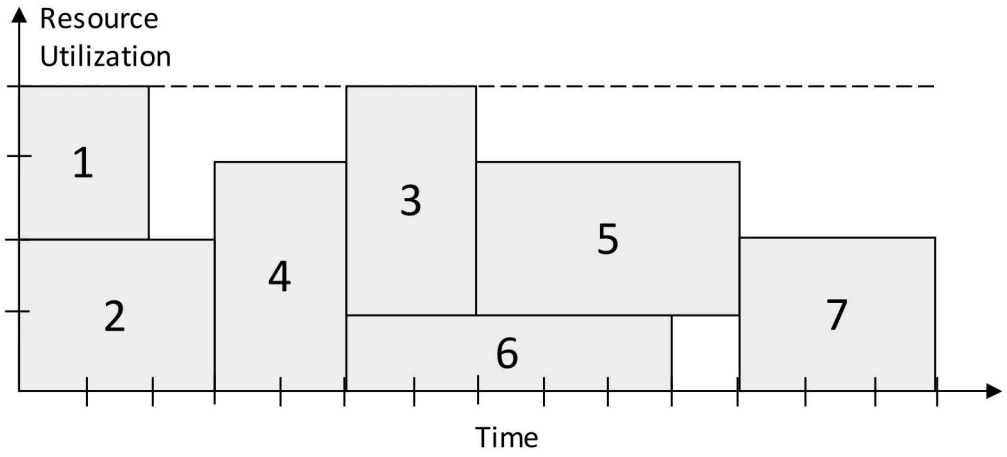

Figure 5: Schedule built from task order [2 4 1 6 3 5 7]

Task 2 is scheduled first. All resources are currently available in the first time period (because no other tasks are scheduled yet), so Task 2 can begin at time $t = 0$. Next, we schedule Task 4. Task 4 has Task 2 as a predecessor, so it cannot begin before Task 2 is complete. Therefore, we schedule Task 4 to begin at $t = 3$. Next, we schedule Task 1. Task 1 has no predecessors, and there are enough resources available in the first time period to complete it (Task 1 requires 2 resources, and Task 2 is only using 2 resources out of the total available 4), so we schedule it to begin at $t = 0$. Next, we schedule Task 6. There are enough resources available for Task 6 to begin at $t = 2$. However, Task 4 is a predecessor of Task 6, so Task 6 will begin at $t = 5$ once Task 4 has completed. This process continues until all tasks have been schedule according to available resources and precedence relationships.

The schedule building algorithm will never create a schedule that violates resource constraints, because it can always schedule tasks later when enough resources are available. It will also

never build a schedule that violates precedence constraints. It may, however, make schedules that violate upper and lower bounds on product store levels. If this occurs, the cumulative overage and the cumulative underage will be reported back to the GA as constraint violations. Design that do not violate constraints will be viewed by the GA as being more favorable than those that do.

5.2 Building Product Inventory-Feasible Schedules

Product inventory violations may be avoided by defining a type of refill event for a product store (see Section 4.4). The process for scheduling product refills proceeds as follows:

1. When scheduling a task, find the earliest precedence- and resource-feasible start time.
2. If scheduling the task at this time violates a product store lower limit, try to remedy the violation with a refill event.
    a. Start the search for the refill event start time in the last time period in which the violated store has any activity (not including the just-scheduled task if it is in the future).
    b. If possible, search forwards (in the future) incrementally for a feasible (with respect to resources and product store floor/ceiling) start time for the refill event up to the time period corresponding to the start time of the offending task minus the refill event duration.
    c. If 2B was unsuccessful (or could not even begin), search backwards (in the past) incrementally for a feasible start time for the refill event down to t = 0.
    d. If 2C was unsuccessful, find the first resource-feasible start time for the refill, using the original search start time minus the refill duration as a starting point. Then find the first resource-feasible start time for the offending task that starts after the refill finishes. If a store violation still exists, increment the refill start time and try again. Search up to last time period that any of the involved products and resources have any activity. Only do this if there is one violated product store. If there are multiple, moving the task start time has unknown effects on the other stores, including the one(s) that have already been processed.
3. If, after completing Step 2, store violations still exist (floor or ceiling), schedule the task at its earliest precedence- and resource-feasible start time without refilling and return the schedule as infeasible.

There is no single best way to schedule product refills. The approach that SMO uses has advantages and disadvantages. The most notable disadvantage is that product store lower-limit violations are not guaranteed to be fixed by a refill event. The advantages are numerous, however. First, this approach does not increase the size of the search space for the GA, because the GA does not have to make any decisions about scheduling refill tasks. Second, this approach results in a small modeling burden for the analyst. The analyst does not have to predict how many refill tasks will be needed *a priori*; they are simply scheduled as needed. The analyst also does not have to create individual refill tasks. Rather, they define one refill task type whose parameters (duration, cost, resource requirements, etc.) apply every time that type of refill is scheduled.

6 BUSINESS REQUIREMENTS

6.1 Business Requirements Overview

Business requirements are desired conditions that may be satisfied by executing tasks. Examples of business requirements include test coverage of components, risk mitigation, and safety objectives. Different modes of task execution may satisfy business requirements to differing degrees. For example, a "component testing" task might have two different modes of execution, one which tests the component thoroughly but which takes a long time, and another which can be completed quickly but provides less confidence in the results. The analyst can indicate which requirements each task mode satisfies, and to what degree.

6.2 Business Requirements Objective

The requirements objective is computed from *coverage factors* that are specified for each mode. A coverage factor is a number in the interval [0,1], where coverage factors of zero and one indicate that the mode provides no coverage or full coverage of the requirement, respectively. When multiple modes are selected that provide different coverage factors for a requirement, the coverage level of the requirement is computed as a function of the coverage factors. Let ***R*** be the set of requirements, ***M*** be the set of modes selected to be scheduled, and c_j be the coverage factor of the j^{th} mode. The coverage level of the i^{th} requirement, denoted C_i, is computed as follows:

$$C_i = 1 - \prod_{j \in M} \left(1 - c_j\right) \tag{1}$$

The equation for the coverage level of a requirement has a probabilistic interpretation. If each coverage factor is viewed as the probability that its associated mode succeeds or fails in covering the requirement, the computation of the coverage level is analogous to the probability that at least one mode succeeds in covering the requirement. It should be stressed that this probabilistic interpretation provides the motivation for the formula to compute the coverage level of a requirement, but coverage factors and the resulting coverage level are not intended to be viewed strictly as probabilities.

The requirements objective is then computed by normalizing the sum of the coverage levels of each individual requirement. See Section 7.4 for a discussion of function normalization in SMO. The requirements objective, denoted C, is computed as follows:

$$C = \phi\left(\sum_{i \in R} C_i\right) \tag{2}$$

where φ is the objective normalization function.

7 MULTI-OBJECTIVE OPTIMIZATION

7.1 Multi-Objective Genetic Algorithm Overview

Traditional optimization methods seek to optimize a single objective subject to a set of constraints. In practice, however, it is often useful to treat one or more constraints as objectives. Multiple competing objectives are optimized simultaneously for identifying and understanding the impact of decisions on performance tradeoffs. The goal of multi-objective optimization is to identify a set of *non-dominated* designs, i.e., those for which there are no known designs outside this set that are better in at least one objective and not worse in any others. The set of non-dominated designs is called the *Pareto set*, and its domain is called the *trade space* (Figure 6).

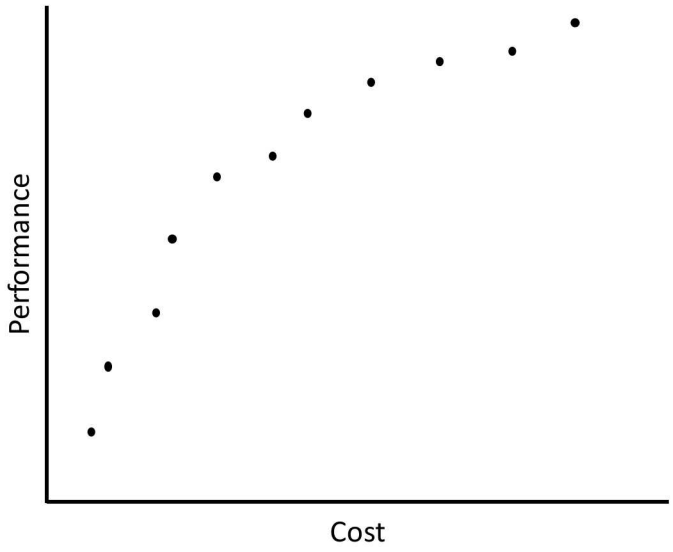

Figure 6: A Pareto set of multi-objective optimization solutions in the performance (maximized) vs. cost (minimized) trade space

SMO uses the JEGA multi-objective genetic algorithm (MOGA) to explore the trade space and identify the set of non-dominated designs. For more information about JEGA's MOGA, see the DAKOTA reference manual [1].

7.2 SMO Objective Functions

SMO seeks to optimize four overall objectives: cost, coverage, schedule, and risk. The objectives, also known as *response groups*, are comprised of one or more metrics called *responses*.

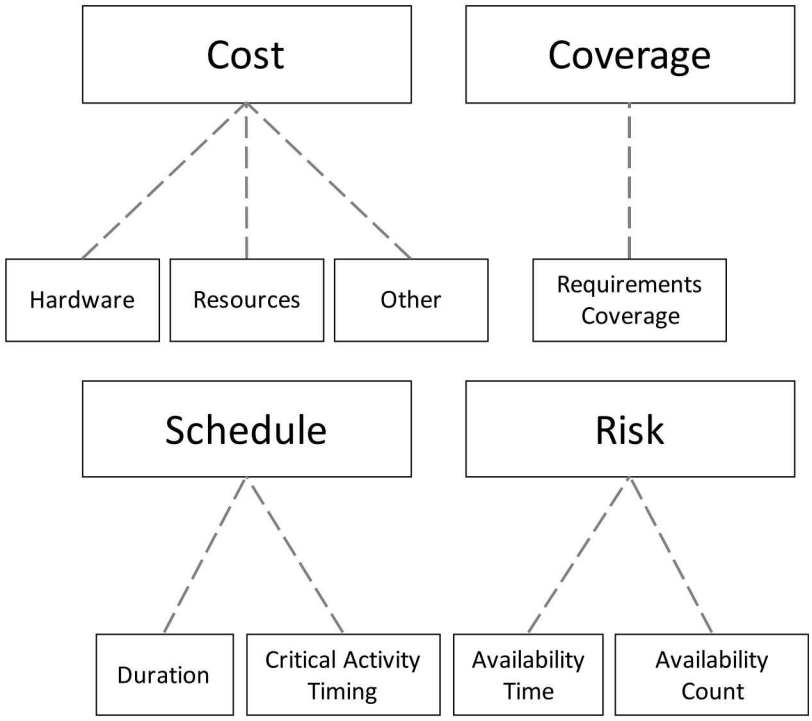

Figure 7: SMO response groups

7.2.1 Response Group 1: Minimize Total Cost

SMO seeks to minimize the total cost of completing all tasks. The total cost of completing all tasks is the sum of the costs of each selected mode in the final schedule, plus the sum of all resource costs, plus the sum of all product consumption costs, minus the sum of all product generation revenues.

7.2.2 Response Group 2: Maximize Requirement Coverage

SMO seeks to maximize the extent to which business requirements are covered. See Section 6 for a discussion of business requirements and the calculation of the coverage response group.

7.2.3 Response Group 3: Maximize Schedule Performance

The schedule response group is comprised of two responses that measure the quality of the overall schedule. First, the total duration, the time required to complete all tasks, is sought to be minimized. Second, the critical activity timing, is also sought to be minimized. For each task, the analyst may specify if that activity is considered critical. The critical activity timing response is computed by summing the start times of all critical activities in the final schedule. In practice, critical activities are those that one desires to schedule as soon as possible. Such activities may exist for a variety of reasons. For example, a critical activity may be a test whose outcome may significantly impact the future schedule if the results it yields are undesired or unexpected.

Although they both are measured in units of time, duration and critical activity timing may have significantly different scales of magnitude and have different meanings. Therefore, the schedule response group objective value cannot be generated by simply summing its two responses. Instead, they are normalized using the function normalization approach that is discussed in Section 7.4.

7.2.4 Response Group 4: Minimize Risk

In practice, predetermined schedules are rarely adhered to exactly as planned. Tasks may take longer to complete than expected. A resource may not be available when expected due to external factors, such as machine breakdown or employee illness. Schedule risk is a measure of how sensitive a schedule is to changes in task start times that result from resources not being available when expected.

Figure 8 shows three schedules for the same three-task project. The first schedule (a) is the highest risk; if tasks 1 or 2 have delayed finish times, the finish time of the final task, and therefore the overall project, will be delayed. The second schedule (b) is medium risk. If task 1 is delayed task 2 will also be delayed, but task 3 is unlikely to be delayed because the resource has downtime scheduled between tasks 2 and 3. The final schedule (c) is the lowest risk. If task 1 is delayed, tasks 2 and 3 are not likely to be delayed because the resource has down time between tasks 1 and 2 and between tasks 2 and 3.

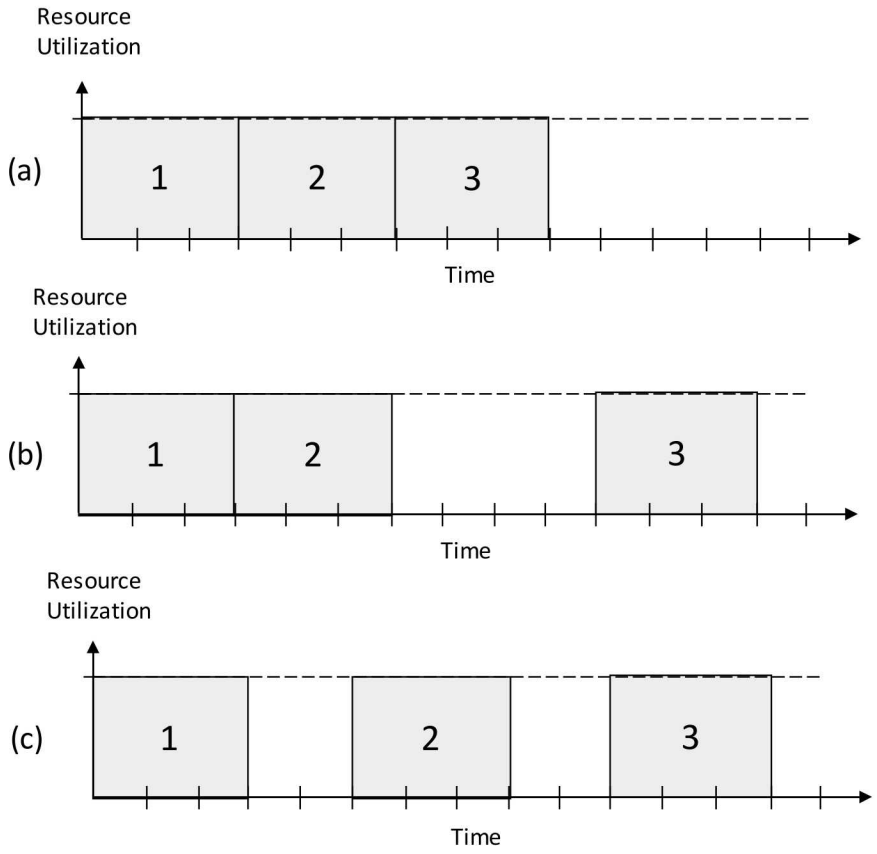

Figure 8: Schedules with (a) highest risk, (b) medium risk, and (c) lowest risk

The example in Figure 8 highlights two features of a reduced risk schedule: high total resource downtime and the high count of resource downtimes. If resources are fully utilized all the time, there is little to no wiggle room for tasks taking longer than expected when the schedule is executed in real life. Note that schedules (b) and (c) have the same total downtime, but (c) is lower-risk because there are more buffers between tasks. Both the total resource availability between times of full utilization, called "Availability Time", and the number of resource availabilities between times of full utilization, called "Availability Count", are important. This is

because it's better to have lots of little gaps between resource uses than one big one. Therefore, both the availability time and the availability count are considered when calculating the risk metric. SMO will prefer schedules that maximize these two metrics.

7.3 SMO Constraints

In other optimization formulations, particularly in the domain of linear programming (LP), many constraint equations are required to enforce precedence and resource feasibility and to ensure that each task appears exactly once in the final schedule. With SMO, these constraints are inherently satisfied because of the nature the schedule building algorithm, thus drastically simplifying the optimization formulation compared to LP approaches.

The only constraints in SMO are those that are related to product inventories. See Section 4.3 for the definition of product inventory overage and underage.

7.4 Response Function Normalization

An objective function in SMO is nonlinear and can be composed of one or more response functions. For example, an objective that measures efficiency may consist of response functions for machine utilization and quantity of working inventory. Maximizing efficiency in this example would require maximizing machine utilization while simultaneously minimizing the quantity of working inventory. The response functions could be combined into a single maximization or minimization objective by taking the negative of the working inventory quantity or utilization response functions, respectively. However, combining both these response functions into a single efficiency objective is problematic because utilization and working inventory are not commensurate with respect to their units or magnitudes. For example, utilization typically has a fractional value between 0 and 1; whereas, working inventory could have a magnitude in the thousands.

To aggregate response functions having disparate units and/or magnitudes into a single objective, each response value is first *normalized* using a function that accounts for the response goal (e.g., maximization, minimization) as well as the desired response value and acceptable limit. The normalized response values can then be summed and optimized as a single objective. Specifying a normalization function for a response requires several parameters. First, the response *objective value* must be specified which is a desired value for the response (e.g., we may desire 0.95 machine utilization) as well as the response *limit value* which is the worst response value that is acceptable (e.g., utilization below 0.60 is not acceptable).

An example of a normalization function for maximization is shown in Figure 9. For this response function, the desired value is 10.0 and the lowest acceptable limit is 7.0. As indicated by the increasing slope below the limit, the normalization function increasingly penalizes transgressions of the limit. Likewise, the response function increases modestly above the limit indicating that exceeding the limit is not especially beneficial. In addition to maximization and minimization, normalization functions can also be created for a *seek value* goal, where the goal is to keep the response as close to the objective value as possible. Only a single limit value is required to be specified and the function imposes a symmetrical limit above or below the objective. In this way, the normalization function can penalize response values that transgress either limit with more severity.

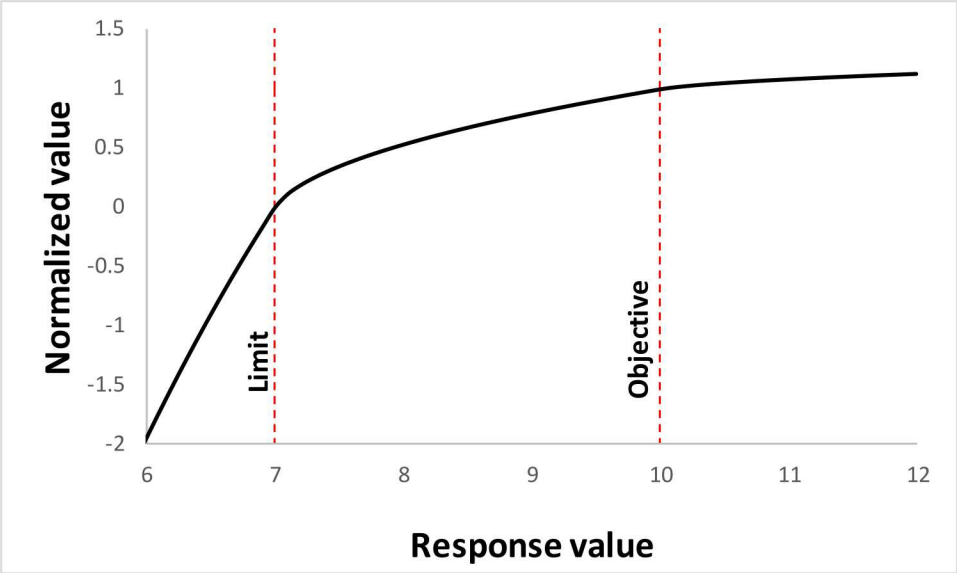

Figure 9: Example normalization function for maximization

7.5 Using a Genetic Algorithm for Scheduling

The SMO MOGA modifies both the order of the tasks and the mode selections for those tasks to fully explore the trade space and evolve the set of Pareto designs. The ordering of tasks is represented as a permutation of unique integers which each corresponds to a specific task. The schedule builder decodes this sequence into a fully defined schedule. The schedule building algorithm requires that the ordering passed from the MOGA respects precedence constraints, and that each task ID appears exactly once in the sequence. Because traditional GA operators do not meet these requirements, SMO's MOGA uses custom operators [2] to initialize, cross, and mutate the task ordering of the designs in the population. The initialize, cross[2], and mutate operators for task mode selections are more like those traditionally found in a GA.

7.5.1 Initialization

The initialization algorithm is called repeatedly to randomly generate an initial population of designs. Mode selections are generated by choosing a random number in the range [1, *M*] for each task, where *M* is the number of modes for a given task. Initializing the task order requires a more complex algorithm. In the initial task order, precedence must be respected, and each task must appear exactly once. The algorithm proceeds as follows:

1. Count the number of predecessors for each task.
2. Identify the tasks that currently have a predecessor count of zero. At least one task must have zero predecessors.
3. Randomly choose one of the tasks that have zero predecessors and place it at the end of the current task ordering.
4. Reduce the predecessor count of all successors of the selected task by one.
5. Repeat Steps 2-4 until all tasks have been placed in the ordering.

[2] Task mode crossover is not implemented in the current release of SMO. Crossover only impacts task priority order.

7.5.2 Crossover

Crossover is the process of "mating" two designs (parents) to produce two new ones (children). Crossover occurs during each generation and is intended to pass along high-performance characteristics from one generation to the next and therefore push the search towards global optima. The crossover operator for the task ordering must obey the requirements that the resulting designs feature each task exactly once and preserve task precedence.

In the following sections, the algorithms for one-point, two-point, multi-point (3+), and uniform crossover are discussed. In all the examples, there are eight tasks with the precedence relationship shown in Figure 10.

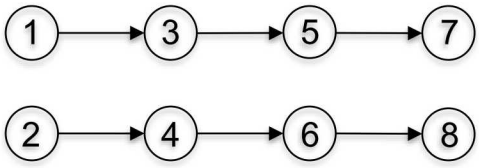

Figure 10: Precedence graph for crossover examples

7.5.2.1 One-Point Crossover

The algorithm for one-point crossover on task order proceeds as follows. Given random crossover point q, positions 1,…,q in child 1 are taken from parent 1. The remaining positions are taken from parent 2 in order, but the tasks that were taken from parent 1 may not be considered again. Child 2 is generated in the same fashion but with the inheriting parents reversed. An example of this process with eight tasks and $q = 3$ is shown in Figure 11.

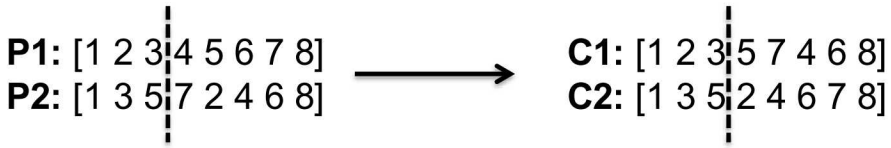

Figure 11: One-point crossover example

7.5.2.2 Two-Point Crossover

Two-point crossover in SMO is like one-point crossover, but there are two crossover points instead of one. Given two random crossover points r_1 *and* r_2 positions 1,…,r_1 and r_2 to the end are taken from parent 1 for child 1. The remaining positions are taken from parent 2 in order, but the tasks that were taken from parent 1 are ignored. An example of this process with eight tasks, $r_1 = 2$, and $r_2 = 6$ is shown in Figure 15.

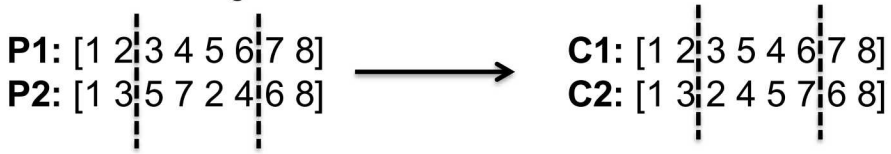

Figure 12: Two-point crossover example

7.5.2.3 Multi-Point Crossover

Following the approach for one- and two-point crossover may result in precedence violations if there are three or more crossover points. Therefore, a different algorithm is used for this case. When there are three or more crossover points, SMO will randomly divide each parent into R+1

sections, where R is the number of crossover points ($R > 2$). Starting with the first section, each child inherits its tasks and their order from each parent alternately, ignoring the tasks that are already included in the child. Note that this approach will also work if there are two crossover points, but the result will be different that the method described in the previous section. An example of this process with eight tasks and $R = 3$ is shown in Figure 13.

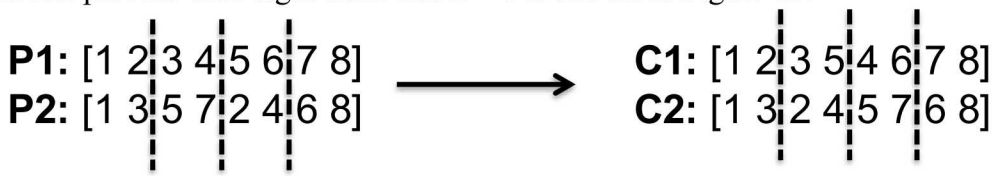

Figure 13: Multi-point crossover example

7.5.2.4 Uniform Crossover

Uniform crossover is achieved for each child by, starting at the first position, each position of the child is filled by selecting a parent at random, then selecting the first task in the parent that is not already included in the child. An example of this process is shown in Figure 14. The vectors of 1's and 2's at the top and bottom of the figure show which parent child 1 and child 2 are inheriting from, respectively.

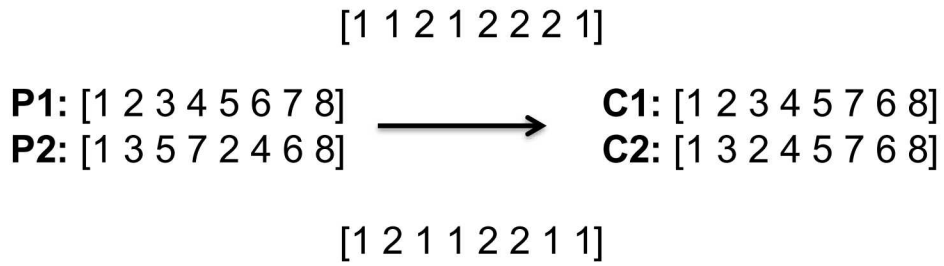

Figure 14: Uniform crossover example

7.5.3 Mutation

Mutation is an operator that is used to promote diversity from one generation of designs to the next. Mutation is performed during each generation by making random alterations to existing designs in the population. The mutated designs are similar, but not identical, to their original counterparts. Mutation prevents premature convergence to a suboptimal solution set, and encourages the GA to explore the design space more broadly.

When performing mutations on the task ordering, the mutation operator must preserve precedence and obey the rule that each task must appear exactly once in the resulting design. The mutation operator for task ordering proceeds as follows:

1. Select a random position, v, in the task order of the design to be mutated.
2. Identify the position, i, of the latest predecessor of the task in position v.
3. Identify the position, j, of the earliest successor of the task in position v.
4. Select a random position, k, between i and j.
5. Insert the task from position v immediately before position k and slide the subsequent tasks towards v.

Note that you cannot swap the elements in positions v and k, because doing so could violate precedence constraints. An example of this operation is shown in Figure 15.

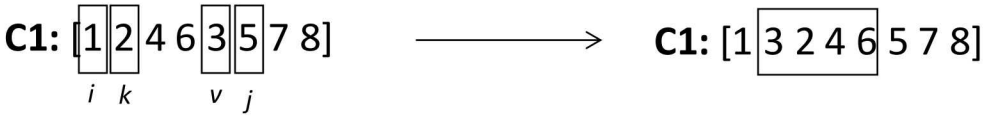

Figure 15: Example task ordering mutation operation

In the example above, Task 3 (position *v*) is randomly selected to be moved. Task 3's latest predecessor is Task 1 (position *i*) and its earliest successor is Task 5 (position *j*). Having identified positions *i* and *j*, a position between the two is selected. In this example, the eligible positions hold Tasks 2, 4, and 6 (Task 3's current position is also eligible). Say the position that holds Task 2 (position *k*) is selected. Task 3 is then moved into Task 2's position. Tasks 2, 4, and 6 move one space to the right to fill the void. You cannot swap Tasks 2 and 3, because precedence would be broken if Task 2 were a predecessor of Tasks 4 or 6.

SMO may perform this operation recursively according to the **Mutation Depth** parameter. The GA determines the depth by randomly generating a number in the range 1 up to the number specified by the mutation depth parameter. For example, if the GA generates the number 2 as the mutation depth, it will choose a random element *v* and then identify *i* and *j*. Next, it will find the latest predecessor and earliest successor of *i* and move *i* somewhere in-between. Then it will do the same thing to *j*. After moving *i* and *j*, it will move the original *v* somewhere between its current latest predecessor and earliest successor. This operation can be performed to an arbitrary recursion depth provided that the precedence chain is long enough.

The mutation operator for mode selections is like that of a traditional GA. The algorithm proceeds as follows:

1. Randomly choose a task to mutate the mode selection of.
2. Generate a random number, *m*, in the range [1, *M*], where *M* is the number of modes available for the task that was selected in Step 1.
3. Change the mode of the selected task to mode *m*.

SMO may repeatedly mutate the same design multiple times (both the task order and the mode selections) according to the user-defined **Mutation Repeats** parameter. The GA will randomly choose a number between 1 and the number of mutation repeats and repeat mutation on a single design that number of times.

7.6 Algorithm Options

7.6.1 Random Seed

The **Random Seed** is used to initialize the random number generator in the genetic algorithm. Entering a value of "0" will cause the GA to set the random seed using the current time.

7.6.2 Population Size

The **Population Size** is the size of the initial population of randomly generated designs. Note that the population size parameter sets the size of the initial population only. The actual size of the population may vary during an optimization run depending on the types of operators chosen.

7.6.3 Fitness Assessor

There are two **Fitness Assessor** options in SMO. The **Domination Count** fitness assessor determines fitness by computing the number of designs that dominate each design. Designs with low domination counts are preferred to those with high domination counts. The **Layer Rank** fitness assessor assigns all non-dominated designs a layer of 0. Then ignoring the designs assigned a layer of 0, all remaining designs that are non-dominated are assigned a layer of 1. This process continues until all designs are assigned to a layer. Designs with lower layer rankings (closer to 0) are preferred to those with higher layer rankings.

7.6.4 Evaluation Concurrency

The **Evaluation Concurrency** specifies the number of evaluations that can occur in parallel. If left blank, the number of logical processors on your machine will be used. Using a negative number will use the number of logical processors minus the number entered.

7.6.5 Domination Cutoff and Shrinkage Rate

Domination Cutoff is a selection parameter that specifies a limit on the domination count or the layer rank of designs in the population. When used with the **Domination Count** fitness assessor, all designs that are dominated by fewer than the cutoff are kept, and all others are subject to the shrinkage rate. When used with the **Layer Rank** fitness assessor, all designs whose layer is below the cutoff are kept, and all others are subject to the shrinkage rate.

The **Shrinkage Rate** is implemented to prevent sudden decreases in the population size between generations and thus a rapid loss of genetic diversity. The shrinkage rate defines the maximum amount that the population size can decrease between generations. To enforce this, all the selections that would normally be made according to the domination cutoff are made. If that is not enough, selections are made from the best of what is left (effectively increasing the cutoff as far as it must to get the minimum number of selections). This continues until enough selections are made.

7.6.6 Crossover Rate and Crossover Points

The **Crossover Rate** specifies the probability of a crossover operation being performed on any given design. Let P be the population size, and CR be the crossover rate. The number of crossovers that will occur is $(CR*P) / 2$. Note that we divide be two because two designs are crossed for each one crossover operation. For example, if the population size is 50 and the crossover rate is 0.8, there will be 20 crossover operations performed. Because each crossover operation involves two randomly selected designs, the probability that any given design will be crossed is 0.8.

The **Crossover Points** parameter specifies the number of crossover points. See Section 7.5.2 for a discussion of crossover in SMO.

7.6.7 Mutation Rate, Depth and Repeats

The **Mutation Rate** controls the number of mutations performed. The rate specifies the probability that a mutation will be performed on any given design variable. Note that the number of design variables is $2N$, where N is the number of tasks, because each design is defined by both the task ordering and the mode selections for each task. Therefore, the number of mutations that will occur is $(2*MR*P*N)$, where MR is the mutation rate and P is the population size. For example, if the population size is 50, there are 100 tasks, and the mutation rate is 0.01, there will be 100 total mutations (2*0.01*50*100).

The **Mutation Depth** controls the number of times to recursively mutate the task order. The GA will randomly choose a number in the range [1, Mutation Depth] and recursively mutate the task order that number of times.

The **Mutation Repeats** parameter controls the number of times mutation will be repeated on the same individual. The GA will randomly choose a number in the range [1, Mutation Repeats] and repeat mutation on an individual that number of times.

7.6.8 Max Generations

The **Max Generations** parameter specifies the maximum number of iterations that JEGA will perform before terminating the optimization and returning the current set of non-dominated designs at that point.

7.6.9 Max Evaluations

The **Max Evaluations** parameter specifies the maximum number of function evaluations that JEGA will perform before terminating the optimization and returning the current set of non-dominated designs at that point.

7.6.10 Max Time

The **Max Time** parameter specifies the maximum number of seconds that may pass before terminating the optimization automatically terminates and returns the current set of non-dominated designs at that point.

7.6.11 Tracked Percent Change and Tracked Generations

The **Tracked Percent Change** and the **Tracked Generations** work together to terminate the GA when the set of non-dominated designs is not changing, or is change very little, from one generation to the next. JEGA computes a Pareto metric at the completion of each generation. When the metric does not change by the specified percentage over the specified number of generations, the algorithm terminates. See the JEGA documentation for details of the Pareto metric calculation.

7.6.12 Niching

Niching is a process that is employed in multi-objective GAs to promote diversity across the set of non-dominated designs. Without niching, some Pareto sets evolve to a set of clusters that have several similar individuals while ignoring potentially interesting designs in between.

There are three **Niching Methods** to choose from in SMO. The **Max Designs** niching method chooses a limited number of designs to remain in the population. The number that remain is determined by the **Maximum Niched Designs** parameter. This method is helpful in situations where the population grows very large, thus consuming too many computer resources. The Max Designs nicher ranks designs using their fitness and a count of the number of designs that are too close to them. What is deemed "too close" is determined by the **Niching Percentage**, which sets the distance in each direction from the design of interest as a percentage of the total range of each objective function. The **Radial Distance** niching method uses the Niching Percentage to determine a minimum allowed Euclidean distance (square root of sum of squares) between any two designs. The **Orthogonal Distance** nicher works in a similar manner, except the distance is enforced along each dimension. Think of the Radial Distance nicher as disallowing two designs to reside in the same multi-dimensional sphere, whereas the Orthogonal Distance nicher disallows two designs to reside in the same multi-dimensional rectangle.

Designs that are removed by the nicher are not necessarily discarded. Checking the **Cache Niched Designs** checkbox will buffer designs up to the number specified by **Max Cached Designs**. Buffered designs are reinserted in to the population prior to the next round of selection.

All nichers will always keep the extreme points of the non-dominated set.

7.6.13 Logging

SMO offers the ability to save a log file that contains information and statistics from the optimization process. To save a log file, enter a file name in the **Log Filename** field. The **Logging Levels** determine the amount and type of logging information that is recorded:

- **Debug:** Outputs all entries.
- **Verbose:** Outputs all but the most insignificant of entries.
- **Normal:** Outputs informational entries or above.
- **Quiet:** Outputs only entries that are warning level or above.
- **Silent:** Turns logging virtually off.
- **Fatal:** Outputs only very severe errors that will presumably lead the application to abort.

Checking **Print Each Population** will create a file containing the variable values, objectives, and constraints for each member of the population after each iteration. Lastly, checking **Print Final Population** will cause SMO to save the final set of non-dominated designs with objective and constraint values. All files are saved to the user's local application data directory (e.g. **C:\Users\username\AppData\Local\SMO**). Note that the **AppData** directory may be hidden by some Windows operating systems.

REFERENCES

[1] B. M. Adams, *et al.*, "Dakota, A Multilevel Parallel Object-Oriented Framework for Design Optimization, Parameter Estimation, Uncertainty Quantification, and Sensitivity Analysis: Version 6.5 Reference Manual," Sandia National Laboratories, Albuquerque, NM, Sandia Report: SAND2014-5015, 2014.

[2] K. S. Hindi, H. Yang, K. Fleszar, “An Evolutionary Algorithm for Resource-Constrained Project Scheduling,” IEEE Transactions on Evolutionary Computation, Vol. 6, No. 5, pp. 512-518, 2002.

DISTRIBUTION

1	MS0899	Technical Library	10756 (electronic copy)

Sandia National Laboratories