A Schedule Optimization Tool for Destructive and Non-Destructive Vehicle Tests

Jeremy Ludwig, Annaka Kalton, and **Robert Richards**
Stottler Henke Associates, Inc.
San Mateo, California
{ludwig, kalton, richards} @ stottlerhenke.com

Brian Bautsch, Craig Markusic, and **J. Schumacher**
Honda R&D Americas, Inc.
Raymond, OH
{ CMarkusic, Bbautsch, JSchumacher } @ oh.hra.com

Abstract

Whenever an auto manufacturer refreshes an existing car or truck model or builds a new one, the model will undergo hundreds if not thousands of tests before the factory line and tooling is finished and vehicle production beings. These tests are generally carried out on expensive, custom-made vehicles because the new factory lines for the model do not exist yet. The work presented in this paper describes how an existing intelligent scheduling software framework was modified to include domain-specific heuristics used in the vehicle test planning process. The result of this work is a prototype scheduling tool that optimizes the overall given test schedule in order to complete the work in a given time window while *minimizing* the total number of vehicles required for the test schedule. Initial results are presented that show a reduction in required test vehicles compared to manual scheduling of the same tasks as well as increased capability to ask "what-if" questions to further improve the schedule.

Introduction

Vehicle testing is an essential part of building new cars and trucks. Whether an auto manufacturer refreshes an existing model or builds a new one, the model will undergo hundreds if not thousands of tests. Some tests are exciting, such as a 48 km/h dynamic rollover and measuring the impact on the crash-test dummies. Other tests are not quite as sensational but still important, like testing the heating and air conditioning system.

What these tests have in common is that they are generally carried out on hand-built vehicles because the new factory lines for the model do not exist yet. These vehicles can each cost as much as an ultra-luxury Bentley or Lamborghini, which results in pressure to reduce the number of vehicles. There are two additional complications with the test vehicles. First, the hand-built vehicles take time to build and are not all available at once but become available throughout the testing process based on the *build pitch* of the test vehicles. An example of this is one new test vehicle being made available each weekday. Second, there are many particular types of a model and each test might require a particular type or any of a set of types (e.g., any all-wheel-drive vehicle). There may be dozens of types of a particular vehicle model to choose from, varying by frame, market, drivetrain, and trim.

At the same time, market forces dictate when new or refreshed models must be released. The result is additional pressure to complete testing by certain dates so model production can begin.

Finally, testing personnel and facilities are limited resources. For example, it would be desirable to schedule all of the crash tests at the very end of the project so other tests could be carried out on those vehicles first. However there aren't enough crash labs or personnel to support this so the crashes must be staggered throughout the project.

The work presented in this paper describes how Aurora, an existing intelligent scheduling software framework, was modified to include domain-specific algorithms and heuristics used in the vehicle test planning process. The framework combines graph analysis techniques with heuristic scheduling techniques to quickly produce an effective schedule based on a defined set of activities, precedence, and resource requirements. These heuristics are tuned on a domain-specific basis to insure a high-quality schedule for a given domain. The resulting domain-specific scheduler is named Hotshot.

The result of this work is a prototype system that optimizes the overall given test schedule in order to complete the work in a given time window. The schedule optimization process includes determining which vehicle types are built and the order in which they are built to *minimize* the total number of vehicles required for the entire test schedule. Initial results are presented that show a reduction in required test vehicles compared to manual scheduling of the same tasks as well as increased capability to ask "what-if" questions to further improve the schedule.

Copyright © 2014, Association for the Advancement of Artificial Intelligence (www.aaai.org). All rights reserved.

In the remainder of this paper we first discuss related work. Following this we describe the Aurora scheduling framework and the changes made to create the domain-specific Hotshot scheduling tool. The methods and results sections contain the details of our comparison between an existing schedule created manually and one created with the Hotshot tool. Finally, we present future work in the conclusion.

Background and Related Work

Despite the invaluable role played by scheduling software in a number of industries, the cost and expertise involved in creating a system suited to each new area has restricted the adoption of such tools. Unfortunately, although there are a variety of high-quality *customized* scheduling systems available, off-the-shelf systems rarely fulfill the scheduling needs of any one domain. This is, in large part, because domain knowledge is crucial to the efficient and effective solution of scheduling problems in general.

The result of this is that the industries/domains that realize the advantages afforded by intelligent scheduling systems are either those that can afford a full custom solution, or those that fall within the narrow commercial off-the-shelf domain coverage (e.g., for project planning).

To make scheduling software attainable by a broader audience, it must be possible to create new scheduling systems quickly and easily. What is needed is a framework that takes advantage of the large degree of commonality among the scheduling processes required by different domains, while still successfully expressing their significant differences, i.e., with parts of the scheduling process broken out into discrete components that can easily be replaced and interchanged for new domains. Framinan and Ruiz (2010) present a design for general scheduling framework for manufacturing.

Aurora is one example of an implemented scheduling framework, which distills the various operations involved in most scheduling problems into reconfigurable modules that can be exchanged, substituted, adapted, and extended to accommodate new domains (Kalton & Richards, 2008). The OZONE Scheduling Framework (Smith et al., 1996) is another example of a system that provides the basis of a scheduling solution through a hierarchical model of components to be extended and evolved by end-developers. Becker (1998) describes the validation of the OZONE concept through its application to a diverse set of real-world problems, such as transportation logistics and resource-constrained project scheduling.

The artificial intelligence and operations research academic communities continue to investigate and report the benefits of heuristics as part of improving scheduling results (e.g., Kolischa & Hartmannb, 2006). Aurora, to more quickly find a good schedule, leverages both domain-independent and domain-dependent heuristics in addition to leveraging the hierarchical model of components.

Scheduling Framework

Aurora was designed to be a highly flexible and easily customizable scheduling system. It is composed of a number of components that can be plugged in and matched to gain different results. The scheduling system permits arbitrary flexibility by allowing a developer to specify what components to use for different parts of scheduling. Aurora has been successfully applied in a number of domains. The steps in the scheduling process are described in detail below. All configurable elements are shown in bold. Elements that were modified for the test vehicle domain will be discussed further in later sections.

Scheduling Process

Schedule Initialization

1. Aurora undoes any previous post-processing (to get back to the "true" schedule result state), and applies the **Preprocessor** to the schedule information.
2. Aurora uses the **Queue Initializer** to set up the queue that will be used to run the scheduling loop. A standard Queue Initializer puts some or all of the schedulable elements—activities, flows, and resources —onto the queue.
3. The queue uses the **Prioritizer** to determine the priority of each element. Depending on the execution strategy, these priorities may be used to periodically sort the queue, or to schedule the element with the highest priority at each stage. Note that some priorities may change in the course of scheduling.
4. The Schedule Coordinator triggers the scheduling of the elements on the queue by starting the Scheduling Loop. In many cases, a more complex element will recursively set up and execute its own queue, allowing greater control over the scheduling process.

Scheduling loop

1. A schedulable element (task, project, or resource) asks the **Scheduler** to schedule it.
2. The Scheduler calls constraint propagation on the schedulable so as to be sure that all of its requirements and restrictions are up to date.
3. The Scheduler looks at the element, considers any **Scheduling Method** that is associated with it (e.g., Forward, Backward). A Scheduling Method determines how the system goes about trying to schedule an element. The Scheduler also selects which **Quality Criterion** to associate with the selected scheduling method; the Quality Criterion determines what makes an assignment "good."

4. The Scheduler calls the Schedule Method on the schedulable. The process depends a great deal on the Schedule Method, but the result is that the schedulable element is assigned to a time window and has resources selected to satisfy any resource requirements. It also returns a list of the conflicts resulting from the given assignment.
5. The Scheduler calls constraint propagation on the schedulable (again) in order to update all of the neighbors so that they are appropriately restricted by the newly scheduled element. This process may result in additional conflicts; if so, these are added to the list of conflicts from scheduling.
6. The Scheduler adds the conflicts to the **Conflict Manager**, and asks the manager to attempt to resolve those conflicts.

Schedule Finalization

1. When the queue is empty, Aurora goes through a final conflict management step. The conflict management that occurs during scheduling is primarily local conflict management (it looks at ways of fixing the current conflict, but does not consider the broader context). In this step Aurora applies the same Conflict Manager, but this time it tries to solve all remaining conflicts, and the attempts may have more far-reaching consequences (e.g., instead of shuffling 2–3 elements, it may try to shuffle 6–7).
2. Aurora calls the **Postprocessor** on the schedule, so that any additional analysis may be done before Aurora returns the schedule results.
3. Aurora sends the schedule results to the GUI for display.

Domain-Specific Customization

Two different types of modifications were made to the Aurora framework to create the Hotshot tool. First, the user interface front end was modified to import the testing model, display and edit domain-specific properties, and to perform the optimization to minimize the number of required vehicles. Second, components in the scheduling back end were updated specifically for this domain.

User Interface Customization

There are five features added to the general scheduling user interface that are specific to the vehicle test domain: import an Excel model of the testing problem, view and edit build pitch, view and edit vehicles and build order, minimize the number of vehicles required, and export the schedule to a client-specific format. The first four of the features will be described in greater detail.

The starting point of the Aurora customization for the vehicle testing domain is importing the testing tasks, resources (vehicles, personnel, facilities), resource sets (groups of vehicles), resource requirements, constraints (temporal, sequence, and resource), build pitch information, and calendars from a set of Excel spreadsheets. These Excel spreadsheets represent a model of the overall testing problem. Once imported, the general user interface supports graphically viewing and editing most of the model elements such as tasks, resource requirements, resources, resource sets, constraints, and calendars. Changes were made to support task properties specific to this domain. For example, tasks that render the vehicle useless for future testing are marked as *destructive* and tasks that must be performed on a vehicle before any other tests are marked as *exclusive*.

A Build Pitch dialog was added for viewing and editing the general build pitch per week (number of vehicles that can be built) as well as a maximum build pitch for each vehicle type. For example, 10 test vehicles per week can be built, but only 5 all-wheel-drive can be built in a week.

A Manage Vehicles dialog specifically for managing the vehicles was also added. This dialog is used to manually change the vehicle build order as well as to manually create and remove vehicles. Vehicles with a flexible start date will be assigned a build date based on the assigned build order. Build dates are assigned based on the build order, moving from 1 to n and selecting the first available date that meets two criteria: number of vehicles/week is not exceeded and vehicle type per week is not exceeded. The build order will be assigned automatically during the optimization process.

The Optimization Dashboard is used to minimize the number of vehicles required to schedule the testing tasks (Figure 1). The upper left summarizes the current state of the schedule, showing the number of vehicles required, the number of destructive and exclusive tasks, and the utilization of vehicles in the testing schedule. The upper right shows the current status of optimization, which will change once the Start button is pressed. This portion of the dialog also provides an estimate of how long the remaining optimization will take. The central portion of the dialog contains the four parts of the optimization process. Check boxes allow advanced users to selectively turn off part of the optimization process, but these should generally all be turned on in normal use. Buttons for starting and controlling the optimization are found along the bottom of the dialog.

There are four steps in the optimization process:

1. **Set Backward Schedule.** Mark all tasks to be backward scheduled. This means that the schedule will be created from the end of the project to the beginning, with all tasks scheduling as close to their late end dates as possible.

2. **Date Optimizer.** Once the tasks are backward scheduled, assign build order based on the earliest dates tasks are assigned to vehicles. That is, if the first task assigned to Vehicle A starts on Jan 15 and the first task assigned to Vehicle B starts on Jan 18, then A will come before B in the build order. The heuristic is that vehicles that are needed earlier should be built earlier.
3. **Meta Disabler Optimizer.** For each non-exclusive vehicle, temporarily disable the vehicle and try to create a schedule without the vehicle. If this succeeds, permanently delete the vehicle. If this fails, restore the vehicle and continue. Only non-exclusive vehicles are tested because every exclusive vehicle is required by one task. Vehicles are checked for disabling by starting with the vehicles with the greatest available time and working towards those with the least available time.
4. **Set Forward Schedule.** Return each task to forward schedule mode, where all tasks try to schedule as close to the project start date as possible. However, it will still be the case that (i) exclusive tasks will remain the first tasks assigned to an exclusive vehicle and (ii) destructive tasks will be the last tasks assigned to an exclusive vehicle.

At the end of each scheduling run, the vehicle utilization is calculated. This is the sum of the duration of all the tasks assigned to a vehicle divided by the total days available for each vehicle from creation to project end.

Scheduling Component Customization

The scheduling component customization focused on three central areas: scheduling direction maintenance, special handling for vehicle testing's unusual requirements, and more standard heuristic tailoring for the domain. Each area is discussed in a section below along with the impact on the scheduling plugins. Impacted plugins are noted in italics.

Scheduling Direction Management

The dominant scheduling direction is backward scheduling for most of the optimization cycle, and forward scheduling at the end of the optimization cycle. This presents a challenge because if a conflict-free schedule can be found for one direction, then a conflict-free schedule should also be found for the other direction. This is challenging given the NP-complete nature of the scheduling problem. Just because there is a solution, there is no guarantee that the system can find the solution coming at the problem from a different direction.

This would be less problematic for a less constrained domain. However, between the domain-specific complications (discussed below), and the fact that the optimization process is iteratively reducing the solution space, without supplemental logic the system would frequently encounter conflicts when forward scheduling even though there were none when backward scheduling.

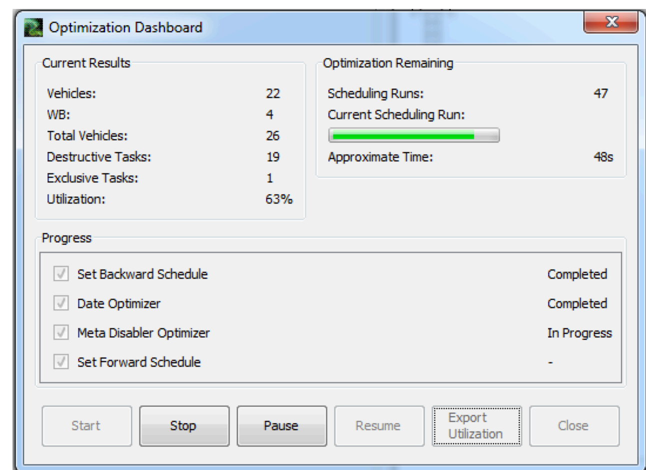

Figure 1. Optimization Dashboard.

The solution in this case is to always backward schedule first, given that that is the dominant scheduling direction for optimization. Once the system has backward scheduled successfully, it then iteratively forward schedules in date-assigned order such that it can derive a conflict-free forward schedule from the backward schedule. This may not produce as tight a forward schedule as is theoretically possible, but provides a consistency guarantee that would not otherwise be possible. This logic is handled by alterations to two plugins: *Preprocessor* — All tasks (except exclusive tasks, discussed below) are marked to backward schedule, regardless of the current dominant schedule direction. The schedule direction requested by the front end is cached so that the postprocessor can restore it. *Postprocessor* — This attempts to move tasks earlier within the limits of their temporal constraints and vehicle availability dates, starting with earlier tasks and iterating through the schedule in date order. Unrelated tasks remain scheduled while the target tasks and their tightly constrained neighbors are moved; this is done to insure that a conflict-free schedule is maintained.

Vehicle Testing Special Support

The vehicle testing scheduling had a few unusual aspects that required special handling in the scheduling framework: exclusive tasks, destructive tasks, and series of inter-constrained tasks.

Exclusive tasks represent tests that must be the first kind of test on a given vehicle. This means that they must be scheduled before anything else on a given vehicle, and nothing can be allowed to subsequently slip in before the exclusive task. Each exclusive task had a vehicle generated for it in the import phase, so the simplest way of handling this case is to schedule the exclusive tasks first in the scheduling process, and initially schedule them as early as possible so that nothing can schedule earlier in time. Three plugins work together to accomplish this. *Preprocessor* — Regardless of dominant scheduling direction, exclusive tasks must be set to always schedule forwards in the initial

scheduling sweep. *Prioritizer* — Exclusive tasks must always schedule first in the process, so that no other task will have the opportunity to schedule at the beginning of the target vehicle's window of availability. *Postprocessor* — The postprocessor is responsible for finalizing the schedule direction declared by the front end (discussed above). When the dominant schedule direction is backward schedule, this step reschedules exclusive tasks backward, snapping them in just before the subsequent tests on the vehicle in question.

Destructive tasks represent tests that destroy the vehicle, so no subsequent tests may be scheduled later than the destructive task. Unlike exclusive tasks, destructive tasks do not have devoted vehicles. This has the advantage of allowing the system to dynamically determine which vehicle is most appropriate for a destructive task, but it also complicates scheduling support. The basic approach is similar to that for exclusive tasks: schedule the destructive tasks backward early in the process in order to avoid conflicts. However, in order to prevent other tasks from sneaking onto the vehicle after the destructive task, the destructive task needs an additional placeholder. This placeholder locks the vehicle down from the end of the destructive task to the end of the test phase, so that no other task can schedule to the vehicle. Two plugins work together to accomplish this. *Prioritizer* — Destructive tasks must always schedule just after exclusive tasks in the process, so that extensive conflict resolution is not necessary to clear space for the destructive tasks. *Scheduler* — The scheduler has two pieces of domain-specific functionality relating to destructive tasks. When the destructive task is performing an analysis of which vehicle to select, the default logic would simply check the destructive task's desired window to make sure that the vehicle was available. In this case, the scheduler performs a secondary check to make sure that nothing is already scheduled to the vehicle later than the desired allocation window. Once the destructive task is scheduled, the scheduler schedules the placeholder to fill the time after the destructive task.

Vehicle-constrained test series are series of tests that need to occur on the same vehicle. That is, once the first test selects from among the set of viable vehicles, all following tests in the series must select the same vehicle. This is a concept that occurs in other domains, especially manufacturing domains, but the vehicle-testing domain included this structural feature to an unusual degree. The reason this complication made the scheduling more difficult is that usually each task is handled individually, checking resource availability individually. In a case like this, where a large number of tasks that are temporally constrained need the same resource, it is easy for the first task scheduled to select a resource that is unavailable for subsequent tasks. To prevent this from happening—or to prevent a poor resource choice from causing significant conflict resolution problems—two strategies are used, supported by two plugins: *Preprocessor* — When the tasks in question are exact-constrained (one must start as soon as the other finishes), a "follow-on" property may be used for easy look ahead. The preprocessor calculates what the "follow-on" duration should be for a given resource requirement, based on which requirement(s) have a constraint to use the same resource as another task. The preprocessor also marks the resource-constrained series (exact-constrained or otherwise) for special handling. *Scheduler* — A special schedule method isolates the backtracking and search logic necessary to handle a series of tasks that are resource constrained where some are not exact constrained, since in that case the "follow-on"-based look ahead is insufficient. This method schedules everything in the series in order, and maintains reasoning about which resources have proven problematic, in order to reduce the conflict resolution and search time.

Heuristic Tuning

Any new domain requires tuning of the heuristics that dictate scheduling order and resource selection. The most notable of these for vehicle testing was the heuristic relating to vehicle selection. The various tests are highly variable in terms of their degree of vehicle flexibility. Some tests can be run on dozens of vehicles; others on one or two. In order to insure that vehicles are selected appropriately based on vehicle availability and remaining tests, a vehicle load heuristic was added. This heuristic influenced both scheduling order (preferring to schedule tests with very limited vehicle options earlier in the process), and resource selection (preferring to schedule tests away from vehicles that had a large block of highly constrained tests that had not yet been scheduled). Keeping the vehicle load information up to date and applying it effectively required adjustments to several plugins: *Preprocessor* — Initialized vehicle load information based on all resource requirements and their options for all test tasks. *Prioritizer* — Apply load information to detect cases where the choices for a test had narrowed dangerously, forcing a test to force-schedule. *Scheduler Quality Criteria* — Apply load information to try to schedule away from vehicle bottlenecks when selecting vehicles. *Scheduler Post-Processing* — Update load information based on scheduled selections.

Methods

The starting point for the comparison between manual and automated planning was a small test schedule that had recently been created and carried out by the client. Planners at the client test facility manually converted the original schedule to the custom Aurora Excel model

format. To provide a rough description of the scope of the scheduling model, it included 60 tasks. Of these tasks, 18 were destructive tasks and 1 was a destructive and exclusive task. The sum of the duration of the tasks is 680 days to be carried out over the 55 days allocated to the project.

The manually created schedule only contained vehicle resource constraints; it did not contain facility or personnel resource constraints. Similarly, the Aurora model contained vehicle resource constraints but not facility or personnel constraints. The test schedule called for 25 vehicles to be built, where the initial Aurora model contains 26 vehicles to be built. This overestimate on the number of vehicles required was done purposely to test the Aurora optimization process. The build pitch supplied to Aurora was the build pitch used to create the schedule. Given the build pitch and the number of vehicles, there would be 1105 possible workdays in the schedule and the initial vehicle utilization is 62%.

The resulting model of the testing process was then imported into Aurora and the optimization process run to create a schedule that minimizes the number of vehicles required to complete the test in the given timeframe. The model was used repeatedly to test and refine the domain-specific heuristics, aiming towards a known lower bound that was calculated for the number of vehicles required. The absolute lower bound of vehicles based on the task definitions was 22 vehicles, based on 19 destructive tasks and 3 tasks that require vehicle types not included in the destructive vehicles.

Results

When the actual test schedule was carried out, 25 vehicles were used to complete the tests and some of the tests went beyond the project deadline. Initially, Aurora created a schedule that required only 22 vehicles to complete the tests in the same timeframe. The 12% reduction in vehicles resulted in intense scrutiny of the scheduling model being imported into Aurora and planners discovered several errors such as missing constraints, too aggressive build pitch, and destructive tests marked as non-destructive. With the improved model of the actual schedule, Aurora was able to complete the schedule using 23 vehicles — an 8% reduction. The model withstood additional scrutiny and the resulting schedule equates to a measurable savings for this small test schedule.

Being able to generate a schedule in under two minutes as opposed to days of labor resulted in the side effect of being able to generate numerous “what-if” scenarios. Planners were able to quantify the effect of compressing or extending the schedule in terms of how many cars would be required. Planners also demonstrated the effects that steeper and shallower build pitches have on the number of cars required for a given set of tasks and project end date. The ability to quickly examine these types of effects will support planners in future test schedules in the task of presenting options with various tradeoffs among manufacturing resources, time, and number of vehicles required.

Conclusion

This paper described a complex, real-world, scheduling problem in automotive vehicle testing prototype management. To address this problem, we added domain-specific heuristics to a general intelligent scheduling software framework to create the custom Hotshot scheduling software. The schedule generated by Hotshot is measured against a schedule completed using the current, manual method. Hotshot was able to generate a schedule with a significant reduction in the number of vehicles required that still completed in the given timeframe.

Ongoing work on this project is aimed at ensuring Hotshot scales to work on larger test schedule problems and takes into account all of the task constraints. Scaling includes testing on more complex models that require over 100 vehicles, creating test schedules with Hotshot while the human planners are still working on creating the schedule manually (to verify the heuristics work on the large model and did not over fit the initial test model), and continuing to improve the heuristics for minimizing the number of required test vehicles. Additionally, Hotshot will begin to utilize facility and personnel constraints when creating a schedule to provide more realistic results.

References

Becker, M.A., 1998. Reconfigurable Architectures for Mixed-Initiative Planning and Scheduling. Ph.D. diss., Robotics Institute and Graduate School of Industrial Administration, Carnegie Mellon university, Pittsburgh, PA.

Framiñan, J.M., and Ruiz, R. 2010. Architecture of manufacturing scheduling systems: Literature review and an integrated proposal. *European Journal of Operational Research* 205(2): 237-246.

Kalton, A., and Richards, R. 2008. Advanced Scheduling Technology for Shorter Resource Constrained Project Durations. *AACE International’s 52nd Annual Meeting & ICEC’s 6th World Congress on Cost Engineering, Project Management and Quantity Surveying*. Toronto, Canada.

Kolischa, R. and Hartmann, S. 2006. Experimental investigation of heuristics for resource-constrained project scheduling: An update. *European Journal of Operational Research*, Volume 174(1): 23–37.

Smith, S.F., Lassila, O. and Becker, M. 1996. Configurable, Mixed-Initiative Systems for Planning and Scheduling. In: Tate, A. (Ed.). *Advanced Planning Technology*. Menlo Park, CA: AAAI Press.