12-2015

# A STUDY ON GENERAL ASSEMBLY LINE BALANCING MODELING METHODS AND TECHNIQUES

Bryan Pearce
*Clemson University*, bpearce@clemson.edu

# A STUDY ON GENERAL ASSEMBLY LINE BALANCING MODELING METHODS AND TECHNIQUES

---

A Dissertation
Presented to
the Graduate School of
Clemson University

---

In Partial Fulfillment
of the Requirements for the Degree
Doctor of Philosophy
Industrial Engineering

---

by
Bryan Wayne Pearce
December 2015

---

Accepted by:
Dr. Mary E. Kurz, Committee Chair
Dr. Laine Mears
Dr. David Neyens
Dr. Maria Mayorga

# ABSTRACT

The borders of the assembly line balancing problem, as classically drawn, are as clear as any other operations research topic in production planning, with well-defined sets of assumptions, parameters, and objective functions. In application, however, these borders are frequently transgressed. Many of these deviations are internal to the assembly line balancing problem itself, arising from any of a wide array of physical or technological features in modern assembly lines. Other issues are founded in the tight coupling of assembly line balancing with external production planning and management problems, as assembly lines are at the intersection of multiple related problems in job sequencing, part flow logistics, worker safety, and quality. The field of General Assembly Line Balancing is devoted to studying the class of adapted and extended solution techniques necessary in order to model these applied line balancing problems.

In this dissertation a complex line balancing problem is presented based on the real production environment of our industrial partner, featuring several extensions for task-to-task relationships, station characteristics limiting assignment, and parallel worker zoning interactions. A constructive heuristic is developed along with two improvement heuristics, as well as an integer programming model for the same problem. An experiment is conducted testing each of these new solution methods upon a battery of testbed problems, measuring solution quality, runtime, and achievement of feasibility. Additionally, a new method for measuring a secondary horizontal line balancing objective is established, based on the options-mix paradigm rather than the customary model-mix paradigm.

# DEDICATION

I dedicate this dissertation to my advisor, Dr. Mary Beth Kurz, without whose enduring patience, encouragement, and brilliance it would not have been possible. Thank you for seeing the better person that I could be even when I could not. I am forever grateful.

ACKNOWLEDGEMENTS

I would like to thank my committee, especially committee chair Dr. Mary Beth Kurz, for the innumerable guiding moments that pushed along this research. In addition, I'd also like to recognize Dr. Cole Smith, for assistance in tightening the IP formulation.

Martin Clark and Dr. Edward Duffy have been my personal computing wizards, helping navigate the challenges of construction and experimentation with prototype software tools. Thanks also to the kind internet citizens of stackoverflow.com, for troubleshooting support.

Thank you to the project team at BMW, for inspiring the content of this work and grounding it in the complexities inherent to industry practice. Special thanks go to research liaisons Dr. Kilian Funk and Dr. Joerg Schulte, for developing and managing the project; Julian Brockman, for crucial input on the operational constraints of the assembly line; Wolfgang Dieminger, whose mastery of product configuration was essential in developing the horizontal balancing approach; and Dr. Kavit Antani, for championing the work and pushing for implementation.

My friends and family have made all of the best parts of me. Brian Williams, Birma Gainor, Latice Fuentes, Angela Grujicic, and Russell Pearce: you teach me what loyalty is, and show me true north.

Finally, I would thank Ginger Stephens, for accepting me, feeding me, and giving thousands of hours of support. Your love and laughter make it all worthwhile.

TABLE OF CONTENTS

Table of Contents (Continued) Page

Table of Contents (Continued) Page

LIST OF FIGURES

Figure Page

List of Figures (Continued) Page

List of Figures (Continued) Page

LIST OF TABLES

Table Page

# CHAPTER ONE

# 1 BACKGROUND AND MOTIVATION

## 1.1 Terms and Concepts

This section presents the foundational concepts of assembly line balancing. Definitions are provided for italicized terms; these will be used throughout this document. The emphasis in this section will be to familiarize the reader with the simplest version of the assembly line balancing problem, forgoing problem relaxations, generalizations, and other complexity adding concepts for later.

### 1.1.1 Assembly

*Assembly*, as described by (Scholl), is a manufacturing process that develops a work-in-progress *workpiece* into finished product by sequential attachment of *parts*. Parts are the atomic physical inputs to the assembly process, each of which is typically standardized and interchangeable with other parts of the same type. A *subassembly* is a collection of parts that are attached to one another, prior to fastening to the workpiece.

### 1.1.2 Tasks and Precedence

The work performed during assembly is portioned into the smallest possible indivisible operations, or *tasks*, each of which requires an associated *task time* to complete. The sequence in which tasks are performed may be constrained such that some tasks must be done before another task begins, due to the physical architecture of the workpiece, safety reasons, or other causes. *Precedence* relationships between two individual tasks are used to codify these constraints, with the task that must come first labeled the *predecessor* and the later task called the *successor*.

The set of all binary precedence relationships between task pairs may be represented as a *precedence graph*, by first drawing each task as a node and then drawing directed arcs pointing away from each predecessor task towards its successor. An example precedence graph is shown in Figure 1. The precedence graph must be acyclic, as no task may be considered a predecessor to itself. It is not required for all nodes in the graph to communicate, as disconnected subgraphs indicate that the corresponding tasks are *precedence independent* from one another. Nor it is required to draw indirect precedence relationships on the graph. For example, in Figure 1, task 2 is a predecessor to task 7, but this relationship is implicit by considering the predecessor relationships of task 4.

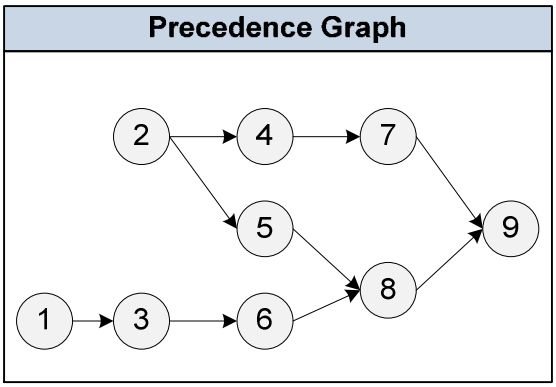


Figure 1: Precedence Graph

Alternatively, precedence relationships may be collected in the form of a *precedence matrix*. Each task may be arbitrarily assigned an indexing number, 1 to *n*, where *n* is the total number of tasks. The rows of the *n*-x-*n* matrix index to predecessor tasks and successors are indexed to columns, allowing one matrix element for each possible precedence relationship. The matrix is constructed by placing a 1 in each matrix element for which a precedence relationship exists, and a 0 if not. An example of a

precedence matrix is shown in Figure 2, containing the same precedence information as in Figure 1.

| | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|---|
| 1 | - | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 |
| 2 | 0 | - | 0 | 1 | 1 | 0 | 0 | 0 | 0 |
| 3 | 0 | 0 | - | 0 | 0 | 1 | 0 | 0 | 0 |
| 4 | 0 | 0 | 0 | - | 0 | 0 | 1 | 0 | 0 |
| 5 | 0 | 0 | 0 | 0 | - | 0 | 0 | 1 | 0 |
| 6 | 0 | 0 | 0 | 0 | 0 | - | 0 | 1 | 0 |
| 7 | 0 | 0 | 0 | 0 | 0 | 0 | - | 0 | 1 |
| 8 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | - | 1 |
| 9 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | - |

Figure 2: Precedence Matrix

Note that there are many indirect precedence relationships that are not tracked in the above example precedence graph and precedence matrix. Instead only *immediate precedence* relationships are shown, i.e. the minimal set of arcs necessary to constrain the acyclic graph. For example, task 1 is a predecessor for tasks 3, 6, 8, and 9, but only the relationship to task 3 is immediate. All indirect precedence relationships may be derived from the set of direct precedence relations, if desired.

### 1.1.3 Assembly Lines, Stations, and Workers

An *assembly line* is a type of assembly process, in which a conveyor or similar material handling equipment moves evenly spaced workpieces from the beginning of the assembly process to the end. The conveyance path is segmented according to this spacing into a series of consecutive *stations*, such that there is one workpiece in each station. Each station is given a subset of tasks to complete, and the requisite parts, tooling, and other needs in order to complete those tasks, in addition to a *worker* to provide necessary manpower. *Fixed pace* assembly lines convey workpieces at a steady

rate from one station to the next, resulting in a constant *cycle time* for each station to complete work on the current workpiece before the conveyor moves it to the next station. An example of an assembly line is shown in Figure 3. In this pictogram, each block represents a part. At each station a worker picks the parts, optionally sub-assembles some of them, and fastens them into the workpiece upon the conveyor.

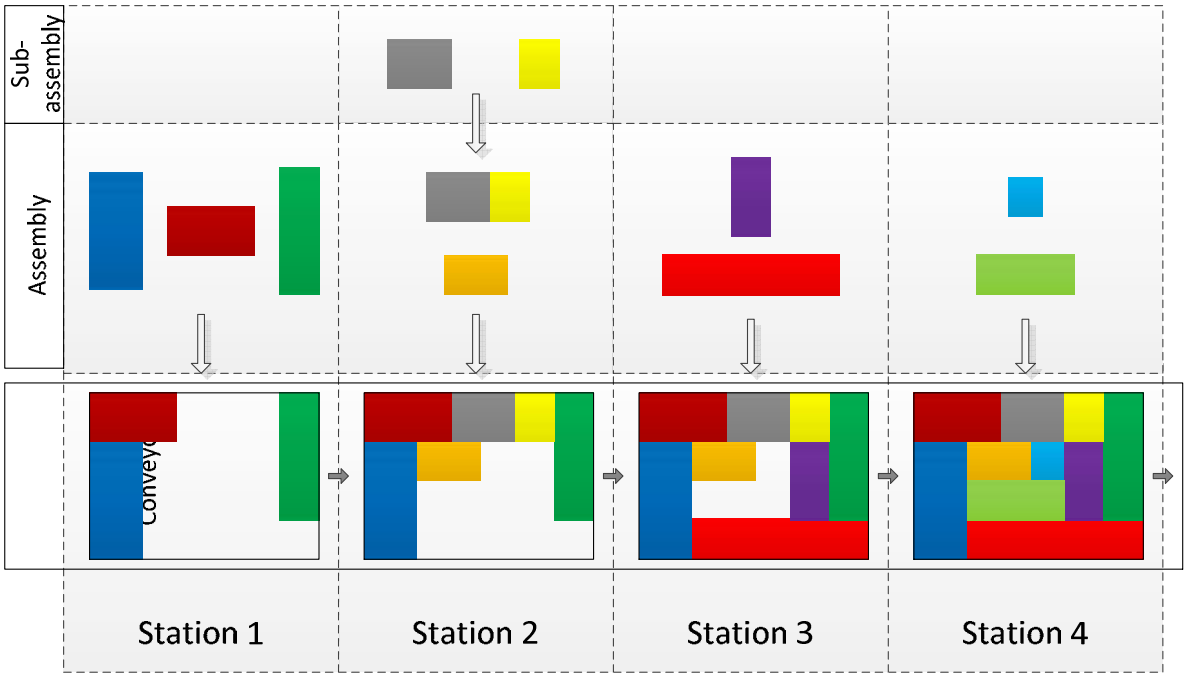


Figure 3: A Typical Assembly Line

Assembly lines were originally constructed for mass production of standardized assembly products, to increase average worker productivity and overall throughput by leveraging labor specialization along the line (Shtub and Dar-El). Modern assembly lines designed for make-to-order and mass customization production permit fast and flexible responses to customer demand (Mather) (Pine), but are associated with significant

automation and facility capital costs. Successful assembly line planning is critical to engineering a cost-effective production process.

### 1.1.4 The Assembly Line Balancing Problem

The *assembly line balancing problem* (ALB) is a production planning problem concerned with allocating tasks to the stations on the assembly line, first proposed and formulated as a mathematical programming problem in 1955 by (Salveson). A *solution* to the ALB is a set of decisions that determine which tasks are assigned to each station. (Scholl) provides a thorough modern review of assembly lines and the ALB.

## 1.2 History of Assembly Lines

Manufacturing is arguably as old as humanity itself, as records and artifacts of ancient peoples record the construction of objects from multiple components (Rekiek and Delchambre). Craftsmen such as masons and carpenters, trained specially to work various materials, have existed for at least 10,000 years. The cottage industry production system emerged some 1000 years ago, as the predominant method for fabrication of the most intricate or demanding products, and featuring skilled artisans and smiths exercising the pinnacle of their respective talents. A typical craftsman's process began with raw materials, from which components were cut, sized, or otherwise initially prepared, followed by an iterative component assembly and re-fitting procedure until the product became finished.

The Industrial Revolution brought technology to prominence in production systems, giving rise to the modern factory system. Starting in the 18$^{th}$ century, continuing through the entirety of the 19$^{th}$ and into the 20$^{th}$ century, the Industrial

Revolution encapsulates a series of discoveries and inventions related to energy, transport, and material processing technologies. The impact of these changes were felt not only within production systems, but also in society as a whole.

Assembly lines are only one of many modern production systems that sprung from the earliest factories. The assembly line depends upon key industrial innovations in material handling, the line production system, and interchangeable parts. The first industrial application of bulk material handling components is recorded in a flour mill constructed in 1785 (Roe). Multiple conveyor systems and elevators were used in the mill, allowing for completely automated movement of raw materials through the factory. A series of developments in machine tooling technology during the early 1800s allowed for hand-crafted components to be replaced by industrially fabricated interchangeable parts. This change drastically reduced the time and cost of product components, allowing for reliable access to standardized parts. The exact origin of the line production system is uncertain. By the late 1800s, at least, conveyance systems were in use in slaughterhouses, with specific butchery tasks allocated to each worker on the line, comprising a system that might be called a disassembly line. The first usage of the line production system for assembly was realized in 1901 by the Olds Motor Vehicle Company, and the concept patented as an "assembly line" by the company owner Ransom Olds (Domm). The Olds assembly line did not use a conveyor, however, as the vehicles were simply rolled on wheels from one workstation to the next. In 1913, Henry Ford's Model T assembly line first integrated conveyance with the assembly line concept, an innovation which achieved vast industrial success as well as historical acclaim.

The assembly line balancing problem (ALB), a production planning problem concerned with allocating tasks to the stations on the assembly line, was managed on an ad-hoc basis until Salveson's 1955 mathematical programming formulation of the problem (Salveson). Several additional authors followed with founding contributions between 1956-1961 (Jackson) (Bowman) (White) (Supnick and Solinger) (Hu), giving birth to assembly line balancing as a field of research. Over the last 60 years a wide variety of extensions, adaptations, and innovations have emerged, both in the technological support and complexity of physical assembly lines, as well as in the methodologies used to solve the ALB.

## 1.3 Motivation

At an undergraduate level of understanding of the ALB, one might be tempted to feel confident that any real-world ALB problem would surrender to existing methods, i.e. that the problem is trivial or solved. A glance at the continuing quantity of academic output, as neatly summarized and organized in a recent survey (Boysen, Fliedner and Scholl, A Classification of Assembly Line Balancing Problems), might suffice to scatter this confidence. This evidence would seem to indicate that practitioners do not yet possess the necessary ALB tools, insofar as industrial needs can be inferred from research activity. The need is made plain by consideration of the methods commonly used in industrial practice.

Prior to the instantiation of ALB as a research field in the 1950s, of course, all ALB problems were solved manually, as only intuitive, trial-and-error methods were available. By the early 1970s algorithmic ALB methods had proliferated, but yet a

survey at that time found that only approximately 5% of companies were using published methods to solve their internal ALB problems (Chase). Many articles attest to the continuing prominence of intuitive methods over algorithmic ones developed by the research community, covering all decades of the intervening time period (Schöniger and Spingler) (Milas) (Erel and Sarin) (Boysen, Fliedner and Scholl, Production Planning of Mixed-Model Assembly Lines: Overview and Extensions). A field book published as recently as 2012 (Townsend) makes no mention of algorithmic methods at all, instead recommending a manual approach, in consultation with a process expert to ensure the balance is feasible. Our industrial partner for this research uses a similar method, constructing balances during multi-day workshop collaborations between experts. By all accounts, there remains a significant and continuing gap between theory and practice. What is the cause of this gap, and what can be done to bridge it?

The simplest explanation is the mere fact that finding a feasible solution to an ALB can usually be accomplished by hand. The manual solution will perhaps not be optimal, but might at least be good enough to seem acceptable to management. Additionally, there are certain normal translational difficulties for any new theoretical work. Industry adoption requires potential adopters to learn that the theoretical methods exist, overcome organizational inertia resisting change, and, of course, financial investment to implement the change. Still, the gap has been persistent for the last 60 years. One would hope that to be sufficient time to overcome these issues. There are several more substantial practical reasons for the existence of the gap beyond translational difficulties, however.

### 1.3.1 Lack of Suitable Methods

The real-world ALB problem may possess features that either by themselves, or in conjunction with one another, are not modeled by any published solution procedure. Section 1.5 discusses many different generalizations of the ALB problem that may present on a real assembly line. A hypothetical facility with parallel workers, a U-shaped assembly line, and stochastic task times might find, after searching the literature, that there are no methods suitable for application.

This was the case for our industrial partner at the onset of this research, as no published contribution offered ALB modeling methods with the constraint detail necessary to capture operational dynamics at the facility. The use of ALB solution methods with insufficient constraint modeling renders any generated solution vulnerable to infeasibility, as these solutions may violate one or more of the ignored constraints.

A practitioner might develop new methods as needed, with appropriate background and skill. Given the deliverable-oriented nature of many process engineering job duties, however, it is perhaps uncommon that such a research task be undertaken.

### 1.3.2 Lack of Input Data

Algorithmic approaches to ALB are demanding in regards to input data. The simplest ALB problems require specification of task time and precedence information, i.e. a codification of which tasks must be executed before others may begin. Task times are usually established by motion time measurement (MTM) projects, in which the workplace is simulated and measurements recorded on each action undertaken by the worker. The sequencing of tasks in an MTM experiment is typically specified for the

subject, perhaps with replication under differing sequences to test for time variation. These experiments do not usually capture task-to-task precedence information, or capture it only in limited form during task sequencing variation.

In the automotive sector, a vehicle requires several hundred to several thousand assembly tasks. Lacking documented task precedence information, a typical industry practice is to partition the total line balance problem into several smaller line balancing problems, using functional domain threshold points on the physical assembly line as partition boundaries. For example, if the vehicle underbody is assembled on one continuous subsection of the assembly line, then this subset of stations and tasks can be isolated as a smaller ALB. A process expert for this subsection might then manually construct a new ALB solution, or tune an existing one, relying on their detailed knowledge of task and station characteristics in lieu of having this information documented. The overall balance for the entire assembly line may then be constructed by integrating the manually created solution from each subsection.

This approach is very time intensive and error-prone, unfortunately. The task and station details are necessary input data to any line balancing approach. If this data only exists in the minds of experts, not documented elsewhere, then no automated methods may be applied.

### 1.3.3 ALB Context

Assembly line balancing problems typically present as one of a myriad of linked production planning and operations management problems. These problems are usually tackled hierarchically, with the longest-range planning problems solved first. Facility

location, layout, capacity planning, and vendor contract problems exhibit the longest-range, the solutions of which are revisited on the order of 1 to 10 years. The ALB is a medium range planning problem, along with the linked workforce management (hiring, firing, training) problem. Medium range problems are usually revisited every few weeks or months. Short range problems include procurement, shipping, and routing, and may be revisited daily, if necessary.

Many of these assorted problems are related to one another. The decisions made by solving longer range problems may have implications for the shorter range problems solved subsequently. For example, ALB solutions are strongly influenced by the previously-determined layout of the assembly line, and the location of capital-intensive robotic support resources. Once found, the ALB solution strongly influences logistics problems related to supplying parts to the line, and may also affect quality and safety engineering efforts. These relationships between the various problems pose difficult questions for theoretical modeling approaches. How might a given problem's scope be adjusted to account for implications towards the adjacent problem? Hybrid modeling approaches combining related problem pairs are fertile ground for research activity. More conservative methods simply approximate downstream effects, incorporating them into the constraints or objective of the current problem. Although integration across production problems is a new and relatively untapped area for academic research, the importance of these efforts have long been understood in industry.

Our industrial partner for this research specifies several constraints and objectives for their manual line balancing process that impinge on problems external to line

balancing efficiency. Ergonomic considerations require that the ALB solution not pair too many tasks that require the worker to push with the elbow, lest the repetitive stress of these tasks lead to injury. Horizontal balancing concerns are important both for quality considerations and job sequencing. If an ALB solution has poor horizontal balance, then the worker might commonly run over the allotted cycle time, causing them to rush and increasing the odds of a defect. Job sequences that feature strings of consecutive vehicles that require more than cycle time will only exacerbate this problem.

In sum, the needs of industry are frequently much more sophisticated than the relatively stringent assumption set accompanying most theoretical ALB modeling approaches. For successful adoption in industry, research models will need to account for an array of interests not traditionally within the scope of the ALB problem. Many of these extensions appear to be achievable by evolving the constraint and objective functions, by addition of detailed plug-ins to the ALB problem to measure downstream effects. For example, an ergonomic modeling function might measure the cumulative repetitive stress risk of some task set upon the worker. The option-mix modeling approach in chapter 6 is another such effort, oriented toward encapsulating quality and process flow disruption risks.

## 1.4 ALB Research Patterns

After the seminal contribution of (Salveson), vast attention has been given to the ALB problem by subsequent researchers. Much of the literature can be characterized as attacking either the problem's structural characteristics or its solution methodology. Structural research may be considered to fall into two camps:

1) Relaxation of simplifying assumptions to provide application to more general environments, otherwise known as the General Assembly Line Balancing Problem (gALB). Examples include permitting task times to be stochastic, examination of U-shaped lines, and multiple workers per station.
2) Development of alternative objectives or multi-objective approaches to find solutions that satisfy other criteria. Notable alternative objectives include cost, quality, horizontal and vertical balancing metrics. These objectives are sometimes treated alone, but are frequently included as secondary objectives subsequent to the primary efficiency objective.

Methodology approaches fall into one of two categories:

1) Development of more powerful exact modeling techniques. These approaches include branch and bound, branch and cut, and integer programming methods, among others. Interests in this domain overlap with content from math programming related to true optimization.
2) Development of heuristics and/or metaheuristics to more quickly identify high quality solutions. These procedures do not guarantee optimal solutions, seeking instead to reduce the computing time required relative to exact approaches by shrinking the search space evaluated by the algorithm. Heuristic methods typically use problem insight to develop very fast algorithms that are greedy toward some problem metric. Metaheuristic methods typically employ intensification strategies that search "near" the solution space of good solutions,

along with occasional diversification strategies to break into unsearched solution space.

Many researchers publish work with both structural and methodological components. For example, (Simaria and Vilarinho) consider multi-zone assembly lines (under the gALB umbrella) while using an ant-colony metaheuristic.

As the language used by authors to address these concepts can vary greatly within the literature, the classification and terminology suggestions from the surveys of (Baybars, A Survey of Exact Algorithms for the Simple Assembly Line Balancing Problem), (Becker and Scholl), (Becker and Scholl) have been herein adopted, with any conflict favoring the more recent publication.

## 1.5 ALB: Generalizations

For any valid ALB solution, the following minimal set of constraints must be satisfied:

1. All tasks must be assigned to some station, such that the workpiece is finished upon exiting the final station.
2. All precedence relationships must be satisfied. Classically this constraint is enforced by ensuring that no task is assigned to an earlier station than one of its predecessor tasks.
3. The sum of task times at each station cannot exceed the cycle time.

Using the terminology of (Baybars, A Survey of Exact Algorithms for the Simple Assembly Line Balancing Problem) survey, Salveson's initial formulation is known as

the Simple Assembly Line Balancing Problem (sALB), as it features a number of simplifying assumptions:

1. Mass-production of one homogenous product.
2. All tasks are processed in a predetermined mode (no processing alternatives exist).
3. Paced line with a fixed common cycle time according to a desired output quantity.
4. The line is considered to be serial with no feeder lines or parallel elements
5. The processing sequence of tasks is subject to precedence restrictions.
6. Deterministic (and integral) task times.
7. No assignment restrictions of tasks besides precedence constraints.
8. A task cannot be split among two or more stations.
9. All stations are equally equipped with respect to machines and workers.

Many industrial environments do not conform to these assumptions, motivating a vast body of research addressing specific manufacturing conditions that require relaxation of one or more assumptions. Though extensive research has been--and continues to be--published relating to ALB, the field is marked by increasingly divergent extensions to the core problem. Some authors have sought to nest ALB within a larger framework of engineering decision problems such as facility design, equipment selection, production scheduling, and logistics. Others have developed focused ALB techniques that conform to specific characteristics of real-world ALB problems. Taken together, these generalizations cover a very wide, but sparse domain, as there are a huge number of

problem characteristic combinations possible, and relatively few problem extension approaches amenable to simultaneous application.

### 1.5.1 Mixed-Model

Mixed-model lines produce several different products upon the same line in an intermixed sequence (Bukchin, Dar-El and Rubinovitz). Task sets, task times, precedence, and other production requirements may vary between models, such that each station may have differing work content depending on the model. (Deutsch) assume that the cycle time restriction is enforced for each model on each station, but later authors quickly realized that cycle time restrictions can be relaxed for the mixed-model case, requiring only the average work content at each station satisfies the cycle time. High- and low-work content workpieces can be staggered in sequence to take advantage of their compensation effect, though this function motivates a production scheduling problem to determine an optimal sequence.

The *model decomposition* scheme takes the original problem, and replaces it with parallel independent ALB instances for each model. (Roberts and Villa) and (Rao) take each task that applies to multiple models, and in its place substitute a suite of tasks, such that each new task applies to a single model from the original multi-model task. By decoupling the original multi-model tasks, superior ALB solutions may become available that were not feasible with the tasks coupled—it is now permitted that multi-model tasks be performed upon a different station for each model. The solution space for the prior problem with coupled multi-model tasks is a subset of the solution space of the new problem, with decoupled single-model tasks, allowing perhaps superior solutions to be

found. Additionally, the model decomposition scheme requires no adaptation in solution methodology. The multi-model aspects of the problem have been encapsulated within the parallelization, and single-model solution methods can be applied to each parallel ALB instance.

Model decomposition's benefits to solution quality and methodological simplicity raise questions regarding task definition. Are there any associated costs to decomposing multi-model tasks into single-model ones? Several practical considerations bear mentioning:

1. Assembly tasks typically involve part installation. The necessary parts are typically stored adjacent to the assembly line, within easy reach of the worker. If a multi-model task that involves fastening a single part is decomposed into a suite of single-model tasks, and the single-model tasks are assigned to different stations, then each of these stations must allocate storage area to the part. Where there was only a single line-side part repository to support the multi-model task, there may be several repositories for the suite of single-model tasks. Logistical problems relating to delivering and storing parts for assembly line use are usually solved separately and subsequent to the ALB. The costs implicit in the logistics problem objective function may be higher with decomposed tasks. At worst, model decomposition ALB approaches may render these logistical problems infeasible, e.g. if there is not enough storage area for duplicated repositories.

2. Assembly tasks may require tooling. From simple hand tools to robotic machines, there is a wide range of costs associated with providing tools to the assembly line. Model decomposition may result in additional costs due to duplication of tooling.
3. Leveraging workforce specialization is one of the primary drivers for using assembly lines, versus other production flow alternatives. Allocating multi-model tasks to several stations requires multiple workers to perform each task, rather than a single worker for each. This diversification may result in loss of personnel efficiency to execute tasks.
4. Tasks may require setup activities before the primary installation activity can commence, such as arranging fixtures and jigs. Splitting tasks to multiple stations may result in lost efficiency in task-sequencing setup time problems, which are usually solved subsequent to the ALB problem.

(Thomopoulos) and (Macaskill) transformed the mixed-model problem into a single model version by taking the demand-averaged time for each task. This method ignores the piece-to-piece variability in work content, and may result in disruptions in line operation. (Thomopoulos) attempted to compensate for this effect by minimizing a secondary objective of the sum of absolute deviations of actual station times of each model to the average station time across models, an early form of horizontal balancing. Horizontal balancing seeks to equalize the work content at a station across all model alternatives, such that the resulting balance is more robust to changes in model demand and production sequencing. (Domschke, Klein and Scholl) proposed a refined horizontal balancing objective that seeks to minimize the sum of work overload time, i.e. the work

content in excess of the cycle time, across all models and stations. (Vilarinho and Simaria) developed a simulated annealing solution approach that incorporated both horizontal and vertical balancing objectives, within a model with parallel stations and additional assignment constraints.

The production sequencing problem that emerges from mixed-model environments can be solved in a staged fashion, subsequent to the ALB problem (Yano and Bolat); (Sumichrast and Russell, Evaluating Mixed-model Assembly Line Sequencing Heuristics for Just-in-time Production Systems); (Sumichrast, Russell and Taylor, A Comparative Analysis of Sequencing Procedures for Mixed-model Assembly Lines in a Just-in-time Production System); (Bard, Dar-El and Shtub) or the two problems can be solved simultaneously (Merengo, Nava and Pozetti).

(Kim, Kim and Kim, A Coevolutionary Algorithm for Balancing and Sequencing in Mixed Model Assembly Lines) proposed genetic algorithms to solve a simultaneous mixed-model ALB and production sequencing problem. Demand is commonly realized at a shorter time horizon than is applicable to the ALB problem, however, suggesting that ALB methods that produce solutions that are robust to demand may be more applicable than those that simultaneously sequence the workpieces.

Multi-model lines are a niche derivative of mixed-model concepts. In addition to the mixed-model features described above, multi-model lines require setup times when transitioning between products of differing type. Batches of similar product are encouraged within the job sequencing problem, as consecutive production of like products requires no setup, reducing the overall sum of setup time required in the

production schedule. Multi-model lines require additional lot sizing and job sequencing problem extensions, as discussed by (Burns and Daganzo) and (Dobson and Yano).

### 1.5.2 Stochastic Task Times

Task times may exhibit variability, especially in high-complexity, low-automation environments. Methods that ignore this variability may suffer from tightly packed solutions that fail to account for the probability that station times may exceed the cycle time. (Moodie and Young) first investigated the stochastic line balancing problem, under assumption that task times were distributed normally. The total variability of the station load was then assessed, and a heuristic procedure moved tasks to different stations in search of minimizing the most probable exceedance of cycle time. (Kao), (Sniedovich), and (Carraway) examined dynamic programming (DP) approaches with more general distributions for task times. (Nkasu and Leung) employ the COMSOAL stochastic optimization algorithm to generate suites of solution alternatives.

(Sphicas and Silverman) offer heuristics to estimate upper bounds that tasks will not exceed with a given probability, transforming the stochastic problem into a deterministic one by application of this safety factor. This deterministic transformation method disregards scenarios in which the cycle time is exceeded, though later researchers attempt to account for station overload by implementing an intervention policy. (Kottas and Lau, A Cost-oriented Approach to Stochastic Line Balancing) assume that additional workers are employed at stations that are likely to result in incomplete work.

(Kottas and Lau, A Total Operating Cost Model for Paced Lines with Stochastic Task Times) assume that incomplete work is moved to an off-line buffer, to be repaired

by a dynamically sized off-line workforce and then returned to the line. The ALB is solved via deterministic transformation, estimating repair costs afterward. A suite of balance and cost alternatives is generated by varying the transformation parameters. (Sarin and Erel, Development of Cost Model for the Single-model Stochastic Assembly Line Balancing Problem) used a hybrid heuristic with dynamic programming (DP) elements, to solve a similar problem with off-line support to resolve incomplete work, and (Sarin, Erel and Dar-El, A Methodology for Solving Single-model, Stochastic Assembly Line Balancing Problems) extend this method into a complex staged heuristic-partial DP-branch and bound optimization. (Gökcen and Baykoc) provide a simulation approach of off-line work policies to provide better estimation of incompletion probabilities and buffer costs.

(Silverman and Carter) and (Lau and Shtub) assumed that the production line would stop if all tasks were not complete at the end of the cycle time, and simply developed a suite of ALB solutions from which to select a minimal composite cost.

### 1.5.3 Supplementary Constraints

In addition to precedence constraints, many real-world problems exhibit tooling, zoning, worker skill, and other characteristics that restrict the assignment of tasks to stations and/or the relative assignment of tasks to one another. (Johnson) presents a branch and bound model that includes tooling and worker skill components, such that tasks with these "irregularities" are required to be assigned to stations with necessary tools. High complexity tasks are grouped together such that high skill operators can be assigned to those stations, and so forth, satisfying the skill requirement.

(Bautista, Suarex and Mateo) examine tasks sets that are incompatible, i.e. ineligible for assignment within the same station, and deliver a hybrid adaptive search and genetic algorithm solution. (Carnahan, Norman and Redfern) study the effects of physical fatigue on workers from tasks that vary in difficulty, minimizing a composite fatigue score for a given number of stations and cycle time with a hybrid heuristic and genetic algorithm approach.

### 1.5.4 Parallel Workers

Assembly lines with parallel workers relax the sALB assumption that one worker is allocated to each station, instead allowing multiple workers per station. Parallel worker assembly lines are appropriate for workpieces large enough to permit several workers with simultaneous access, e.g. cars or airplanes. Relative to single-worker stations, parallel worker lines offer several potential benefits. Perhaps most obvious of these is a reduction in the total line length (number of stations) required, and corresponding improvements to factory floor space utilization and facility capital construction costs. Consolidation of workers into the same station as one another may also allow sharing of fixed tooling resources between them, reducing capital costs. Material handling costs may also be reduced, as there are fewer destinations to support with part delivery, spread across a smaller footprint. Lastly, parallel worker lines may realize superior line balancing solutions due to reduced worker movement around the workpiece, as each worker can be assigned tasks that only appear in a specific zone. (Bartholdi), (Lee, Kim and Kim)

The first parallel ALB considered lines in which there are two distinct sides to the assembly line, and proposes a priority heuristic to generate a solution (Bartholdi). Tasks are classified into the sets {left, right, either} corresponding to the side of the line to which they may be assigned. A genetic algorithm methodology was later introduced for this gALB problem, to identify higher quality solutions (Kim, Kim and Kim, Two-sided Assembly Line Balancing: A Genetic Algorithm Approach). An important consideration for simultaneous work is that precedence related tasks might be assigned to separate sides of the line, resulting in one side waiting for the other to finish a task in order to begin work on the next one. (Lee, Kim and Kim) offer two supplementary objective functions designed to address this issue: 1) work relatedness, which promotes tasks that have an immediate precedence relationship to be assigned to the same station, and 2) work slackness, which promotes those tasks to have a large time gap between them if they cannot be assigned to the same station. A stochastic single-pass prioritization heuristic was created for a two-sided assembly line that manufactures appliances (Lapierre and Ruiz). (Baykasoglu and Dereli) develop one of the first applications of ant-colony metaheuristics to ALB problems, solving a two-sided assembly line that produces domestic products. This ant-colony approach was quickly extended to include secondary vertical and horizontal balancing objectives (Simaria and Vilarinho). (Chutima and Chimklai) extend previous two-sided ALB models with a multi-objective particle swarm optimization approach that incorporates work relatedness, utilization, and vertical smoothing. (Pastor and Corominas) consider a line with four work zones and additional constraints to require certain sets of tasks be assigned to the same station, using a hybrid

DP and tabu search algorithm to vertically smooth workloads over a given number of stations.

## 1.6 ALB: Optimization and Objective Functions

Classic optimization of the ALB seeks to minimize the total idle time in the line, ideally packing each station with tasks so that the cycle time constraint is tight for every station. There are three possible objective functions, depending on whether the cycle time and station count parameters are free or constrained:

1) Minimize the number of stations (workers) given a fixed cycle time. (Type 1 objective)
2) Minimize the cycle time given a fixed number of stations. (Type 2 objective)
3) Maximize utilization (or, equivalently, minimize total idle time) while varying both the number of stations and cycle time. (Type 3 objective)

All of the ALB objectives are motivated toward increasing line efficiency. For the Type 1 problem, with a fixed cycle time, costs may be minimized by reducing the total labor hour (i.e., the number of workers). For the Type 2 problem, with a fixed number of stations, throughput is an output variable that is maximized by minimizing the cycle time. The Type 3 problem draws on the economic motivations of both the Type 1 and Type 2 problems in tandem. (Wee and Magazine) prove that all three versions of the optimization problem are NP-hard, by showing that ALB is a generalization of the bin-packing problem.

### 1.6.1 Horizontal Line Balancing

Optimization of the traditional ALB problem seeks to minimize total idle time by minimizing of the number of stations (or workers) used, given a fixed cycle time. The problem is NP-hard, as shown by (Wee and Magazine). (Thomopoulos) and (Macaskill) transformed the mixed-model problem into a single model version by taking the demand-averaged time for each task. Such methods ignore the piece-to-piece variability in work content, and may result in disruptions in line operation.

Horizontal balancing seeks to equalize the work content at a station across all model alternatives, such that the resulting balance is more robust to changes in model demand and production sequencing. See Figure 45 for a visualization of two alternative solutions with the same average utilization, but drastically varying horizontal balance. In an early form of horizontal balancing, (Thomopoulos) attempted to compensate for this effect by minimizing a secondary objective of the sum of absolute deviations of actual station times of each model to the average station time across models. (Domschke, Klein and Scholl) proposed a refined horizontal balancing objective that seeks to minimize the sum of work overload time, i.e. the work content in excess of the cycle time, across all models and stations. (Vilarinho and Simaria) developed a simulated annealing solution approach that incorporated both horizontal and vertical balancing objectives, within a model with parallel stations and additional assignment constraints.

The production sequencing problem that emerges from mixed-model environments can be solved in a staged fashion, subsequent to the ALB problem (Yano and Bolat); (Sumichrast and Russell, Evaluating Mixed-model Assembly Line Sequencing Heuristics for Just-in-time Production Systems); (Sumichrast, Russell and

Taylor, A Comparative Analysis of Sequencing Procedures for Mixed-model Assembly Lines in a Just-in-time Production System); (Bard, Dar-El and Shtub), or the two problems can be solved simultaneously (Merengo, Nava and Pozetti).

Demand is commonly realized at a shorter time horizon than is applicable to the ALB problem, however, suggesting that ALB methods that produce solutions that are robust to demand may be more applicable than those that simultaneously sequence the production units. To test the effectiveness of different horizontal line balancing metrics to this end, (Emde, Boysen and Scholl) conducted extensive computational experiments to detect differences in line disruption due to product variety on assembly lines balanced with an array of different horizontal metrics. (Emde, Boysen and Scholl) also mentions in closure the reliance of all tested methods on the mixed-model paradigm, and calls for methods that are robust to product variety size.

## 1.7 Heuristic Methods

### 1.7.1 Single Pass Heuristics

Perhaps the best known of the heuristic methods is the Ranked Positional Weight (RPW) heuristic (Helgeson and Birnie). RPW is designed for fixed cycle time version of SALB, and attempts to pack stations with tasks by assigning them one at a time starting at the beginning of the assembly line. Each task is given a weight that is equal to its task time plus the task times of all of its successor tasks. The tasks are then sorted upon this weight, thereby ensuring that no task will be above one of its predecessors in the sorted list. Assignment then proceeds one task a time, placing the task at the earliest station that

meets both constraint criteria: 1) sufficient time capacity exists at the station to place the task, and 2) no predecessor of the task is at a later station.

In the same paper (Helgeson and Birnie) propose the Inverse Positional Weight heuristic, which is identical to RPW except that tasks are weighted as the sum of their task time and all predecessor task times. The assignment sequence then begins at the final station instead of the first.

(Tonge, Summary of a Heuristic Line Balancing Procedure) developed a heuristic that begins by grouping sets of tasks that are directly connected in the precedence graph, and then treating these groups as single tasks. Tasks are grouped such that each task set is approximately equal in aggregate time, and then a valid assignment is sought using the much-reduced task set size. If an assignment is found, the algorithm then seeks to improve the solution by smoothing tasks from one station into an adjacent one, while preserving precedence and cycle time constraint satisfaction. If no solution can be found with a given task grouping, then the groups are broken into smaller subsets and the process repeats. An updated version of this method was published a year later (Tonge, A Heuristic Program for Assembly Line Balancing).

(Agrawal) proposed the Related Activity heuristic, which scores tasks in the same way as the Inverse Positional Weight heuristic. The assignment method differs however, instead selecting the task with the largest score that is smaller than the cycle time. The selected task is then assigned to a new worker, along with all predecessor tasks. The algorithm then recalculates scores for all unassigned tasks, and repeats the assignment

process. Upon completion the set of workers are then sequenced according to precedence validity.

(Baybars, An Efficient Heuristic Method for the Simple Assembly Line Balancing Problem) develops a set of preprocessing steps oriented to reducing problem complexity. The input data is analyzed for complexity saving opportunities, such as decomposing the original problem into smaller sub-problems, or detection of implicit assignment constraints. After preprocessing a heuristic procedure assigns tasks one at a time beginning at the end of the assembly line. The task prioritization metric considers the subset of tasks with no currently unassigned successors, and then chooses the one with the highest number of predecessors. The chosen task is then assigned to the latest possible station.

### 1.7.2 Multi-Pass Heuristics

(Tonge, Assembly Line Balancing Using Probabilistic Combinations of Heuristics) utilizes a suite of prioritization metrics coupled with a one-at-a-time task assignment approach. After each task assignment, a randomly selected heuristic is used to select the next task. The author reports a competitive result from executing multiple runs of this heuristic, though it is likely this performance was due to the wider solution space search afforded by the element of randomness in each run. It bears noting that most heuristic procedures run incredibly fast, placing little cost on this additional search.

The COMSOAL algorithm (Arcus) expands upon the work of (Tonge, Assembly Line Balancing Using Probabilistic Combinations of Heuristics), again using probabilistic next-task selection during an iterated single-pass solution construction heuristic. On the

first pass the selection probabilities are uniform between all tasks that have no unassigned predecessors, though these probabilities may change as the algorithm iterates. COMSOAL can be characterized as a form of learning algorithm, as objective function performance is used as an input between iterations to bias the probabilistic selection steps in future iterations. A set of nine methods are given for inducing bias into the probabilistic search.

(Pinto, Dannenbring and Khumawala) show a network inspired procedure, in which tasks are grouped together via adjacencies on the precedence graph. These super-nodes are then sequenced and fitted into stations to evaluate the resulting balance. The procedure is iterated with differing heuristic rules for constructing the super-nodes.

# CHAPTER TWO

## 2 GAPS AND RESEARCH PLAN

The research area of Generalized Assembly Line Balancing (gALB) has emerged in response to the limitations of the Simple Assembly Line Balancing (sALB) to solve realistic problems in complex production environments. There are multiple facets of the sALB problem which can be relaxed or modified. In the interest of illustrating gaps in the body of literature, it may be instrumental to first show the full scope of research diversity. The survey paper (Boysen, Fliedner and Scholl, A Classification of Assembly Line Balancing Problems) presents a classification scheme for organizing the variety of efforts within the field, and tabulates the contributions of hundreds of published papers according to which generalization facets are examined. Figure 4 presents this scheme in hierarchical list form. Bolded items appear in some fashion in the gALB problem that is the focus of this research, as discussed in section 2.1.1.

No methodology yet exists that simultaneously manages all, or even most, of the problem generalizations listed in Figure 4. Research contributions in gALB typically select a combination of facets that remains unexamined, outputting a new methodology for the particular environment defined by this selection. The high water mark for methodological flexibility is perhaps provided by (Boysen and Flidener, A versatile algorithm for assembly line balancing), in which several common generalization topics are aggregated into a single approach. Still, the totality of literature only sparsely covers the domain of the field.

1) Precedence graphs and task attributes
   a) Product variety
      **i) Mixed-model. Workpieces differ in content due to customer configuration. Setup times between workpieces are negligible.**
      ii) Multi-model. Workpieces differ in content due to customer configuration, but setup times are significant. Batches of like product are launched together in attempt to minimize setup time. The associated batch sequencing and lot sizing problems may be embedded or solved separately.
   b) Special precedence structure, e.g. linear, diverging, or converging graphs.
   c) Processing time variation
      i) Stochastic task time
      ii) Dynamic task time, e.g. learning effects
      iii) Sequence-dependent task time, e.g. setup time between two consecutive tasks
   d) Assignment restrictions. Constrain assignment of tasks to stations.
      **i) Linked. Tasks that must be assigned to the same station**
      **ii) Incompatible. Tasks that cannot be assigned to the same station**
      iii) Cumulative. A cumulative task attribute (e.g. storage space) is constrained
      **iv) Fixed. Tasks that can only be assigned to certain stations, due to e.g. tooling**
      **v) Exclusion. Tasks that cannot be assigned to certain stations**
      **vi) Type. Tasks that must be assigned to stations of a certain type, e.g. lifted**
      vii) Minimum distance. Tasks that must be separated by some minimum time or space
      viii) Maximum distance. Tasks that must be within some time or space proximity
   e) Processing alternatives. The problem scope expands to select an alternative.
      i) Alternatives differ with respect to task times and cost
      ii) Alternatives affects the precedence graph
2) Stations
   a) Cycle time
      **i) Obeyed for average workpiece**
      ii) Obeyed for every workpiece
      iii) Obeyed for a given probability of workpieces
      iv) Cycle time varies between stations, e.g. via production buffers
      v) Unpaced line, movement between stations when work is finished
   b) Layout
      i) U-shaped line with one or more cross-over stations
   c) Parallelization
      i) Multiple lines work independently in parallel
      ii) Flow splits before a subset of duplicated stations, and merges after.
      iii) Tasks may be performed by different stations per workpiece
      **iv) Multiple operators work in tandem at a single station**
   d) Resource assignment. The assembly line design problem embeds selection from a set of alternative equipment with the ALB
   e) Non value-added time. E.g. walking or transportation times are incurred at the station
   f) Other
      i) Buffers. In- and/or out-buffer capacity must be determined
      ii) Feeder lines must be synchronized with the primary line
      iii) Layout of parts storage to support tasks at the station
      **iv) Positioning of workpiece within the station, e.g. tilted, lifted**
3) Objective function
   a) Primary
      **i) Min idle time. Either or both of #operators and cycle time are minimized**
      ii) Min cost. Several factors may have cost implications, e.g. equipment, differing operator wage due to proficiency constraints
      iii) Max profit. Difference of revenue from production output with costs.
   b) Secondary
      **i) Horizontal smoothing. Minimize product variety induced time variation at each station**
      ii) Vertical smoothing. Minimize station-to-station variation in time

Figure 4: Hierarchy of gALB topics. Bolded features are represented in this research.

## 2.1 Contributions

This research considers a gALB with several generalized characteristics, briefly summarized in section 2.1.1. Bolding has been applied to items in the complete gALB topic list of Figure 4 to represent characteristics in the gALB at hand. Detailed descriptions for each problem feature are given in section 3.2.

### 2.1.1 gALB Problem Characteristics

The assembly line studied produces automobiles. The workpieces are large enough to permit up to five parallel workers within each station. To prevent interference between workers, each must be assigned to a *work zone* (WZ). Tasks are labeled with a *product zone* (PZ), which indicates the location on the workpiece that the task operates upon. Workers may only be assigned a task if the worker's WZ overlaps with the PZ of the task. Figure 5 displays the WZ and PZ zoning divisions.

Some tasks require fixed tooling resources that only appear on some stations. These tooling constraints require the assignment mechanism to match task needs with tooling support. Tools only support a subset of zones within any station, depending on the location of the tool itself. A tool can only support tasks that fall within this subset of zones, called the tool coverage zone (TZ). Figure 10 displays the relationship between TZ and PZ.

Further, vehicles may assume one of eight orientations relative to the conveyor, and the orientation may change between stations. For example, the vehicle may be oriented nose-first in one station, then rotate 90 degrees such that the vehicle is sideways

for the next station. Vehicle orientation strongly affects zoning relationships, dictating which WZ, PZ, and TZ are associated with one another at each station.

Each task maintains a list of stations to which it can or cannot be assigned.

There are several assignment restrictions between tasks that either force assignment linkage or exclusion. Task-to-task assignment restrictions may occur by any of these mechanisms:

1. "Same-Station." The linked tasks must be assigned to the same station, but may be performed by different workers within the station.
2. "Same-Takt." The tasks must be assigned to the same station and worker.
3. "Adjacent." The tasks must be assigned to the same station and worker, and must be executed sequentially.
4. "Not-same-takt." Tasks related by this constraint are incompatible, and may not be assigned to the same worker.

The problem considered is mixed-model, in which each workpiece is built-to-order according to customer's specifications. A configurator encodes customer configuration choices into a set of Boolean *options*, or discretionary equipment to be installed on the vehicle. Optional equipment requires additional or specialized assembly tasks to install, and tasks are labeled according to which options require the task. The set of options available is very large, providing the customer with fine-grained control over their personal configuration. Not all configurations of options are permissible, however, as governed by a set of first-order Boolean expression configuration rules.

Two objective functions are of interest. Objective 1 is to maximize the overall efficiency of the line. Cycle time is fixed, so this objective is equivalent to minimizing the total number of workers, or, alternatively, to minimizing total idle time. Objective 2 is to horizontally smooth the balance, i.e. to reduce the aggregate product diversity induced variation in the time a worker needs to complete all tasks. Both objectives enforce cycle time constraints only in the average case, allowing total load to exceed cycle time for worst-case but not average-case workpieces.

### 2.1.2 Contribution 1: Constructive and Improvement Heuristics

A constructive heuristic is designed called the Modified Ranked Positional Weight (MRPW) heuristic, so called as it is inspired by the Ranked Positional Weight (Helgeson and Birnie) heuristic. MRPW introduces novel prioritization extensions to embed linked task relationships and fixed resource constraints. First, the concept of the *responsibility set* is defined for each task, by merging precedence and task linkage information into a composite successor set. A task is responsible for all of its successors, all tasks that are linked to itself, and all tasks that are either linked or successors to any task in the responsibility set (defined recursively). Next, an *urgency score* is defined for each fixed resource as the number of stations after the last instance of the resource on the assembly line. The highest urgency scores correspond to resources that last appear early in the line, giving importance to the tasks that use those resources. Tasks are then weighted; firstly by the highest urgency score within the tasks responsibility set, and secondly by the total time within the responsibility set.

The responsibility set and the urgency score are new contributions, used to generate a prioritized listing of tasks suitable for a one-task-at-a-time first fit decreasing (FFD) assignment heuristic. A FFD approach assigns tasks to the earliest station at which constraints are satisfied, then moves to the next task. It is necessary to prioritize tasks such that the most constrained tasks are assigned first. From the problem constraints shown in section 2.1, precedence, linkage, and fixed resources have been managed by the prioritization scheme. The primary goal of MRPW is to generate feasible solutions by navigating the myriad constraints, with a secondary goal of maximizing efficiency.

Another use for this new prioritization scheme is given by a follow-up last fit increasing (LFI) improvement heuristic, applied after the constructive heuristic is complete. Here tasks are considered in increasing order of priority, i.e. the most flexible (least constrained) tasks first. Each task is moved to the latest already-active worker on the line. If all tasks are removed from any worker then that worker can be removed from the solution. The goal of the LFI improvement heuristic is to improve the efficiency of a feasible solution.

A second improvement heuristic is developed that addresses zoning issues, called the Work Zone Blocking (WZBlock) heuristic. WZBlock considers the bifurcated ALB problem of first selecting which workers are active, and the subsequent assignment of tasks to those workers, and focuses on the first WZ selection sub-problem. Two novel metrics are introduced to measure the relative value of activating WZs, called the *flexibility* and *uniqueness* scores. The metrics are developed in consideration of each

WZ's offerings in terms of satisfying task needs with respect to zoning, tooling, and accessibility constraints. *Flexible* WZs are those that can satisfy a large proportion of the task set. *Unique* WZs are those to which difficult-to-satisfy tasks may be assigned. A composite of these scores is used to block low-value WZs. WZBlock is applied iteratively, using the MRPW heuristic to construct solutions with the blocked WZs. At each iteration an additional low-value WZ is blocked, and the best found solution is retained.

### 2.1.3 Contribution 2: Integer Programming Formulation

A new binary integer programming (BIP) formulation for the gALB problem is designed, with the goal of maximizing efficiency (minimizing the number of workers) for an assembly line with given stations. Several unique approaches are introduced to manage the zoning aspects of the problem. The problem uses a large quantity of parallel workers per station, in context of the literature. However, the distinguishing features of the problem lie in the flexibility and complexity of the interactions between the three zone types: PZ, WZ, and tool coverage zone. Vehicles may be oriented upon the conveyor in one of eight ways, each of which results in a different mapping between zones. Additionally, WZ overlap in physical space, allowing either of two workers to perform tasks at a given location, with the caveat that the two workers cannot both be assigned to the same space due to interference concerns. All of these zoning features are novel IP extensions in the gALB field.

Traditional decision variables are implemented for the assignment problem, with $x$ and $y$ binary variables controlling task assignment and worker activation, respectively.

The objective function is also traditional, minimizing the total number of workers (the sum over $y$).

Extensive preprocessing is applied to all problem input data related to zoning, in order to achieve representations that are streamlined for BIP formulation and conducive to constructing binary constraints. The key issue to resolve in preprocessing is the problem of vehicle orientation, which demands non-static mapping associations between the three zone types. The vehicle orientation input parameter determines the positioning of the vehicle relative to the conveyor, selected from eight possible settings (four 90 degree rotations, and the same four inverted top-to-bottom). As all zoning concepts are symmetrical with respect to 90 degree rotations, it is assumed, without loss of generality, that changes in vehicle orientation affect only the locations of the PZ associated with the vehicle, and that the WZ and tooling locations are unaffected. Two parameters are output from the preprocessing stage to track orientation-dependent zone mapping. The first of these is $B$, a binary parameter that manages which PZ are permissible for assignment at each WZ. The second is $Q^c$, a binary parameter that manages the subset of PZ that are covered by each tool. The definitions of these two parameters are carefully constructed to encapsulate all orientation-induced complexity in the mapping of zones, and to permit direct implementation of respective constraints.

The BIP formulation employs a large constraint set to enforce the gALB problem features. Many of these are standard fare for ALB formulation, though some are not. The parallel worker zoning scheme permits many PZ to be associated with either of two WZ. However, to prevent interference between workers, tasks of the same PZ are

required to be assigned to the same WZ within any given station. A non-overlap constraint is introduced to enforce this concept, such that the BIP formulation dynamically determines which WZ is chosen for each PZ at each station. Not-same-takt constraints require that task subsets are not assigned to the same worker. To implement this constraint, a clique approach is adopted. Finally, precedence constraints are implemented only at the station level of granularity, such that no predecessor task may be assigned to a later station than one of its successor tasks. This constraint does not fully constrain solutions to feasible space, as it is possible for a precedence violation to occur between tasks assigned to parallel workers at the same station. Overcoming this potential violation within the BIP formulation would require introducing timing-specific variables for sequencing tasks for each worker. Instead, a post-processing feasibility check is applied, and until feasibility is achieved the BIP is iteratively solved, with a new constraint forbidding the previous infeasible solution(s).

### 2.1.4 Contribution 3: Measuring worst-case cycle time

The heuristics and IP formulation enforce the cycle time constraint only in the average case, i.e. product variety is ignored in favor of minimizing the number of operators. Horizontal smoothing is a secondary objective function which seeks to minimize some composite score related to time variation between workpieces for each operator. Several alternative metrics exist for this time disparity, but all rely upon specification of the set of uniquely configured models, and their associated time demands. Consideration of each uniquely configured product alternative is the classic mixed-model ALB paradigm.

The problem environment considered in this research features a large amount of optional content, with each option relating to customizable components. If each option has Boolean (yes/no) values representing whether or not it is chosen, then the total number of unique configurations is $2^n$, where $n$ is the number of options available. Specifying the full suite of models (i.e. unique configurations) is an intractable proposition for any problem environment with sufficiently large number of options, as $2^n$ grows prohibitively large. Note that the combinatorial explosion of configurations presents problems in ALB preprocessing, when simply enumerating the suite of models. This issue will persist through any ALB algorithm that takes the model-mix paradigm in a sufficiently high-option environment.

An alternative approach is offered in chapter 6, called the options-mix paradigm. Instead of listing all unique configurations, instead two information inputs are retained to represent product diversity: 1) the set of option/task associations, that determine which options require each task, and 2) a database of rules that governs which options configurations are valid.

As yet the literature does not yield any methods for horizontal smoothing metrics in an options-mix paradigm. An algorithm is presented in chapter 6 which calculates the worst-case time for a set of tasks assigned to a worker, in consideration of the option associations of all assigned tasks. The worst-case time is the maximum cumulative time of any subset of the tasks, with the caveat that tasks in the subset must be valid for the same workpiece. To identify this worst-case time, differing task subsets are tested for

validity by consulting the rule database, and solving instances of the Boolean satisfiability (SAT) problem.

## 2.2 Limitations

The methods developed in contributions 1 and 2 are designed to manage the gALB problem features in section 2.1. Generalized problem environments with additional gALB features are not supported. Contributions 1 and 2 treat the cycle time constraint only in the average case, i.e. product variety is ignored.

The method developed in contribution 3 is functional only where configuration rules can be expressed with logical first-order Boolean propositions (If, And, Or, Not), though most higher-order ontologies can be reduced to this form. The options-mix metric from contribution 3 is used as a plug-in during contribution 3, but classically derived horizontal smoothing metrics may be substituted for suitable problem environments.

## 2.3 Implications

A manufacturer with operations encapsulated by the gALB characteristics for this problem can implement these methods. The algorithmic approach is suitable for embedding within a commercial line balancing visualization software tool for operations management. Direct use cases include support for initial (product launch) line balancing, and periodic rebalancing to respond to forecasted demand changes. Secondary use cases include exploration of input parameter variation, e.g. cycle time changes, or relocation of fixed equipment.

The SAT decomposition approach discussed in chapter 6 is strikingly similar to research in the virtual machine (VM) packing field. The VM problem emerges from

cloud computing, where a set of users (tasks) must be satisfied with a minimum of energy consumption (time). Each user (task) has a set of needs (options) from the VM (station) environment, and can share a VM with other users with similar needs. Efficiencies are gained by planning to consolidate like users and their shared needs into single VM instances. Performance is lost when duplicate instances are spawned in multiple locations on the cloud for users on separate VMs, which could perhaps have been packed together in a single instance.

CHAPTER THREE

# 3 CONSTRUCTIVE AND IMPROVEMENT HEURISTICS

## 3.1 Introduction

The traditional form of an assembly line, as described by (Scholl), is a production system consisting of a configuration of consecutive workstations, typically using a conveyor or similar material handling equipment to transport workpieces down the line at a constant rate. The total work to be performed along the assembly line is subdivided into the smallest indivisible elements of work, typically called tasks, and each task $i$ possesses an associated task time ($t_i$). Tasks are related to one another by precedence attributes, i.e. some tasks must be finished before others can begin, usually due to the physical architecture of the workpiece. These individual precedence relationships between tasks are collected and summarized by a precedence graph, an acyclic graph with each task as a node and arcs representing precedence.

Stations are spaced along the line such that there is one workpiece present at each station, and all stations will be allotted a fixed cycle time ($c$) to execute tasks before the conveyor moves the workpiece to the next station. These characteristics define the simple assembly line balancing problem (sALB), a production planning problem concerned with assignment of the set of tasks to stations, such that all work is performed upon the workpiece as it traverses the line. Optimization of the sALB seeks to minimize the total idle time in the line through one of three methods: 1) minimization of the number of stations given a fixed cycle time, 2) minimization of cycle time given a fixed number of stations, or 3) maximization of utilization while varying both the number of

stations and cycle time. All three versions of the optimization problem are NP-hard (Wee and Magazine).

Assembly lines were originally constructed for mass production of standardized assembly products, to increase average worker productivity and overall throughput by leveraging labor specialization along the line (Shtub and Dar-El). Modern assembly lines designed for make-to-order and mass customization production permit fast and flexible responses to customer demand (Mather) (Pine), but are associated with significant automation and facility capital costs. Successful assembly line planning is critical to engineering a cost-effective production process.

Though extensive research has been—and continues to be—published relating to sALB, increasingly divergent extensions have been proposed to relax the set of assumptions for more general environments. Many authors have developed specialized techniques that conform to specific characteristics of real-world problem instances, and these extensions collectively outline the general assembly line balancing problem (gALB). The gALB covers a wide but sparse domain, as there are very many combinations of problem characteristics that justify research yet few problem extension approaches amenable to application across multiple domains.

In this chapter a heuristic technique is developed based on the Ranked Positional Weight algorithm of (Helgeson and Birnie). The new heuristic is called the Modified Ranked Positional Weight (MRPW), and is designed to solve assembly line balancing problems encountered with our industrial partner. In section 3.2 the problem environment is discussed, with emphasis on the types of constraints present. Next, in

section 3.3 the classic RPW heuristic is shown in detail. This review is presented to assist in motivating the algorithmic extensions applied to the RPW heuristic, which is shown in section 3.4. These extensions permit the MRPW heuristic to respect the described constraints, allowing production of solutions that are feasible in this domain.

## 3.2 Problem Environment and Additional Constraints

This section introduces and illustrates the production environment details relevant to line balancing. Several features of this environment require relaxation of the standard assumptions of the sALB. In addition to precedence constraints, which are general to all ALB problems, the environment to be modeled contains a number of other constraint types. Table 1 provides an overview summary of the features of the problem. The classification system of (Boysen, Fliedner and Scholl, A Classification of Assembly Line Balancing Problems) is noted for each feature in the right-most column, to provide context.

| Feature | Description | Class |
|---|---|---|
| Parallel workers | Up to five workers may be assigned at each station, each with a non-overlapping work area dynamically determined by the set of tasks assigned | $\beta_3 = pwork^5$ |
| Mixed model | Intermixed sequences of different models are produced on the assembly line | $\alpha_1 = mix$ |
| Grouped tasks | Task groups define tasks that must be performed by the same worker, or in the same station | $\alpha_5 = link$ |
| Stationary resources | Tasks that require fixed resources can only be assigned to stations that possess the resource | $\alpha_5 = fix$ |
| Task exclusion | Some tasks cannot be assigned to certain stations | $\alpha_5 = excl$ |

Table 1: gALB Problem Features Considered

The problem is to be solved for an already existing assembly line. There are several physical characteristics concerning the line, as well as adjoining spaces, that must be considered. This physical data is collected into the station model, an input to the gALB that encompasses all constraints resulting from the physical architecture.

The Type 1 objective function is used here, as cycle time is fixed by consumer demand and not subject to the line balancing process. While holding cycle time constant the number of stations in the solution is a function of the optimization performance. The solution should not feature more stations than already exist, as the existing line cannot be expanded. Given that the line is already functioning, however, we may use the currently implemented line balance solution as a baseline for potential improvement.

### 3.2.1 Parallel Workers and Zoning Constraints

In the sALB tasks are assigned to stations, as it is assumed that only one worker may be present at each station. Here, however, the physical space within each station is sufficient to allow multiple workers to simultaneously process on a single workpiece in parallel. *Zoning constraints* are introduced to prevent interference between parallel workers in the same station. The physical space that the vehicle occupies on the conveyor is partitioned into a set $\Upsilon$ of five *work zones (WZ)*: {V (front), R (right), L (left), H (rear), and I (center)}. Every station will have between zero and five assigned workers, each of which is responsible for a single WZ. With parallel work tasks must be assigned to both a station and a WZ in order to ascribe them to a unique worker. Whereas with the sALB it is permissible to conflate the worker with the station, as each

worker is uniquely assigned to a single station, here the permitted conflation is between workers and their station / WZ pair.

Each task $i$ is encoded with one of nine *Product Zones (PZ)* $\Phi = \{RV, MV, LV, RM, MM, LM, RH, MH, LH\}$ corresponding to location on the vehicle with which the task interacts, divided into a 9-zone grid. Maps of the WZ and PZ are shown in Figure 5.

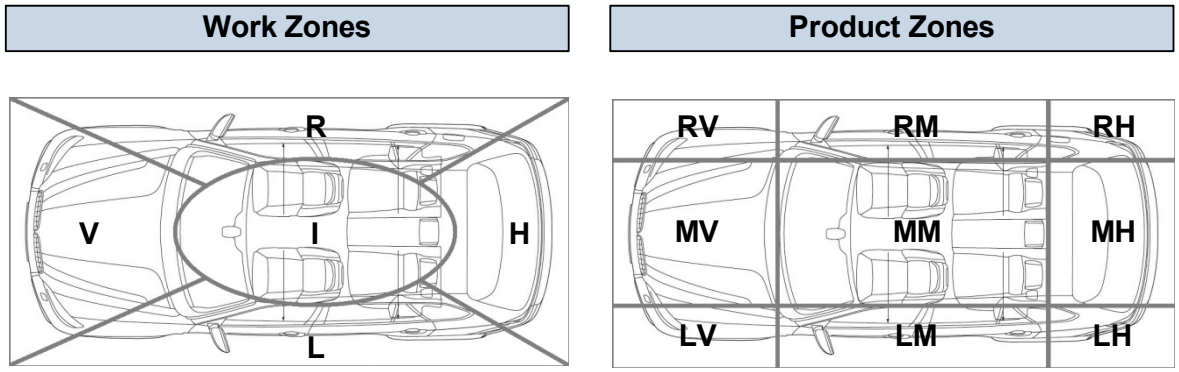


Figure 5: Work Zones (WZ) and Product Zones (PZ)

Each of the WZ is initially eligible to cover three or more PZ, as shown in Figure 6. For example, a worker in the V WZ is positioned at the front of the vehicle, putting the RV, MV, and LV product zones within reach.

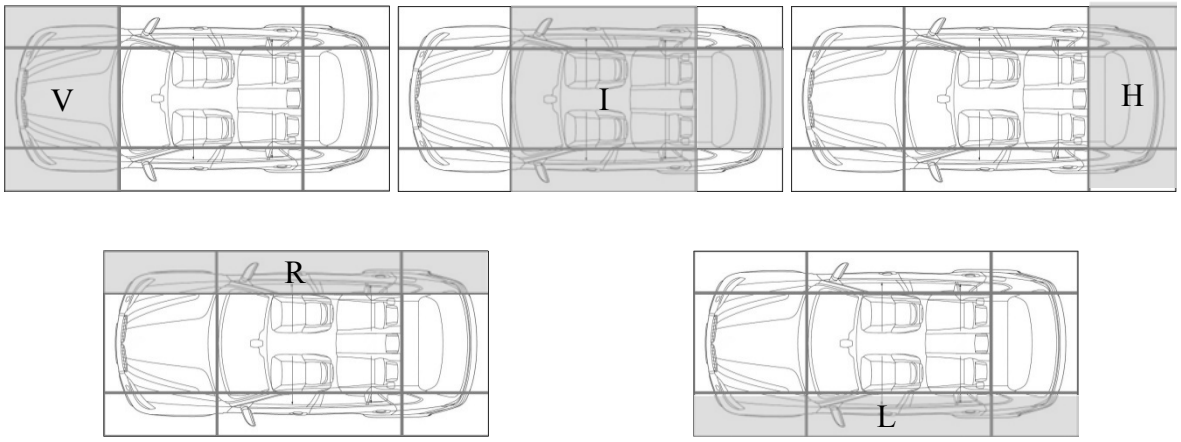

Figure 6: Product Zones Eligible in each Work Zone

While Figure 6 shows all potential matches between PZ and compatible WZ, some of these pairings cannot be activated simultaneously. To avoid interference problems between workers, each PZ may only be assigned to one WZ within each station. For example, while the LV PZ may be assigned to either the L or the V WZ, it may not be assigned to both within any given station. All tasks that are located in the LV PZ must be assigned to the same worker, either the L or the V worker. Figure 7 illustrates this zoning conflict. The lightning bolt flags 1 and 3 show workers attempting to perform tasks upon the same area of the vehicle.

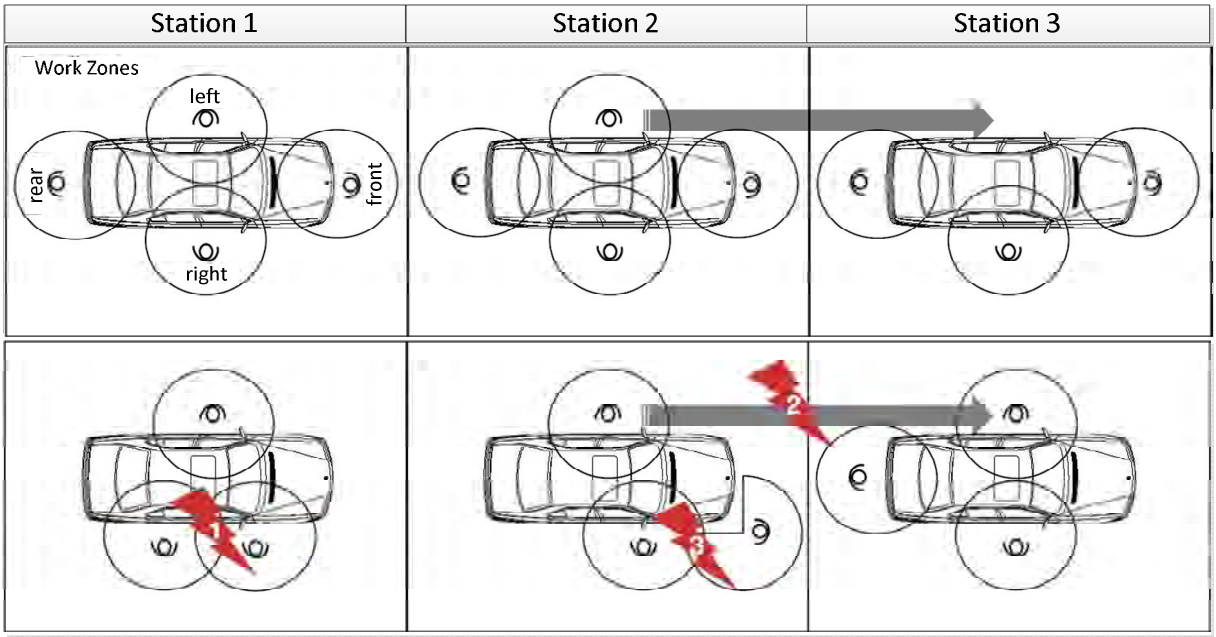


Figure 7: Zone Conflicts

### 3.2.2 Accessibility Constraints

Each PZ and WZ at a station may be deemed inaccessible, due to the structural layout of the station. An inaccessible WZ may not have a worker assigned to the zone.

Similarly, an inaccessible PZ at a station indicates that tasks of that PZ may not be assigned at that station. Physical obstructions such as pillars, robotic machinery, or the workpiece carrier itself are common causes for inaccessible zone constraints.

### 3.2.3 Mixed Model

The production environment modeled is notable for providing a large number of customer configurable options for each vehicle. Each vehicle produced is custom ordered, and none are made with stock configuration. Further, multiple platforms, also known as variants, are frequently produced upon the same assembly line. In this environment it is exceedingly likely that each individual vehicle is uniquely configured. As a result of this diversity of configuration, some tasks are applicable to only a subset of vehicles. For example, installing roof rails requires a few tasks to accomplish, yet not all vehicles have roof rails. For those vehicles without roof rails, these tasks may be skipped entirely.

In this chapter issues involving vehicle configuration diversity are ignored. The methods in this chapter enforce the cycle time constraint only in the average sense. That is, the average time per vehicle, $\bar{t}_i$, is accumulated across all tasks assigned to a worker, and this must not exceed the cycle time. Configuration diversity may result in individual vehicles exceeding the cycle time at certain stations. On average, however, that station's cycle time will not be exceeded. Chapter 6 presents an approach for calculating worst-case loading with respect to product diversity, to support implementation of a horizontal balancing objective.

In addition to task time $t_i$, each task also is delivered with an associated *volume*, $v_i$, equal to the expected number of vehicles per day that will require execution of the task. The calculation for $\bar{t}_i$, then, is given by Eq 1, where $v_{max}$ is the total daily production volume, or daily throughput..

$$average\ time\ per\ workpiece\ of\ task\ i.\ \ \bar{t}_i = t_i \frac{v_i}{v_{max}}, \forall i \in \mathrm{I}, \qquad \text{Eq 1}$$

3.2.4 Task Grouping Constraints

*Task grouping constraints* refer to sets of tasks that must be completed by the same worker, or in the same station. Scenarios that might induce these constraints include, but are not limited to:

1. Upon finishing a task, the worker has a part or tool is in hand, intended for usage on another task.
2. Some tasks require a followup self-inspection of work performed, which is a separate task.
3. Part scanning tasks exist to assure that a later installation task uses the correct part.

Four classes of task grouping constraints are outlined: adjacency, same-takt, same-station, and not-same-takt.

3.2.5 Adjacency Constraints

Adjacency requires the involved tasks to be performed consecutively by the same worker. Consider two tasks: task A requires collecting a part from the line-side storage area, and task B installs that part on the vehicle. Clearly there should be a precedence

relationship between these two tasks, to enforce that task A is performed before task B. An additional concern, however, is that after task A finishes, the worker has the part in their hands, and therefore is not free to pursue other work. Instead, the worker must immediately perform task B. For notation, let $r_{ij}^{adj} = 1$ if tasks *i* and *j* are together part of an adjacency group, and 0 otherwise. Adjacency constraints are not necessarily binary relationships between task pairs. Any number of tasks may be included in a single adjacency group.

### 3.2.6 Same-Takt Constraints

A same-takt (worker) constraint requires the involved tasks to be performed by the same worker, but not necessarily consecutively. Consider the example of self-inspection. The installation and inspection tasks will have a precedence relationship, of course. Additionally we require the two tasks to be assigned to the same worker. For notation, let $r_{ij}^{st} = 1$ if tasks *i* and *j* are together part of a same-takt group, and 0 otherwise. As with adjacency constraints, same-takt constraints are not necessarily binary. Any number of tasks may be included in a single same-takt group.

### 3.2.7 Same-Station Constraints

A same-station constraint requires the involved tasks to be performed on the same station, but not necessarily by the same worker. Consider the example of headliner installation. Due to the size of the part, installation requires several workers to hold the part during the install. Each of these workers is assigned a different task, but all tasks must be done in tandem. Collectively these tasks must be assigned to the same station. For notation, let $r_{ij}^{ss} = 1$ if tasks *i* and *j* are together part of a same-station group, and 0

otherwise. Same-station constraints are not necessarily binary. Any number of tasks may be included in a single same-station group.

### 3.2.8 Multiple Grouping Constraints

Task grouping constraints are transitive, so no task will be part of more than one group of a given type. For example, if tasks A and B have an adjacency relationship, and also tasks B and C have an adjacency relationship, then all three tasks are involved with the same adjacency group. Tasks that are within a group frequently exhibit precedence relations, as evidenced by the examples above. A general approach does not assume that precedence exists amongst grouped tasks, however, as shown in Figure 8. Tasks 2 and 3 may be part of a same-takt group (green oval) but are not related by precedence.

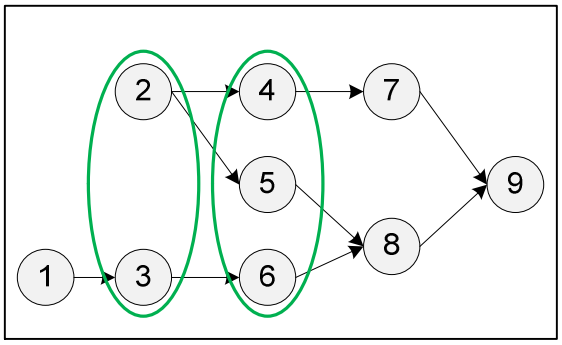


Figure 8: Precedence Graph with Groups

Note that all tasks in a same-takt group must also be assigned to the same station, as workers may only work at a single station. All tasks within a same-takt group with one another may also be considered to be within a same-station group. Adjacency groups generalize to same-takt groups using the same logic. Adjacency related tasks, then, may also be considered to be in a same-takt group together as well as being in a same-station group together. This motivates the introduction of the term $G_i$ as detailed in Eq 2,

representing the group of tasks that are related to task *i*, after fully extending the domain of each task relationship.

$$j \in G_i \; iff \; r_{ij}^{adj} = 1, r_{ij}^{st} = 1, or \; r_{ij}^{ss} = 1 \qquad \text{Eq 2}$$

See the example in Figure 9. Tasks 4, 5, and 6 are in a same-takt group (green oval), and tasks 6 and 8 are in an adjacency group together (red oval). Implicitly, then, task 8 is also included in the same-takt group. The adjacency group, however, does not expand to include tasks 4 or 5. All four tasks are mutually involved in the same *G* group.

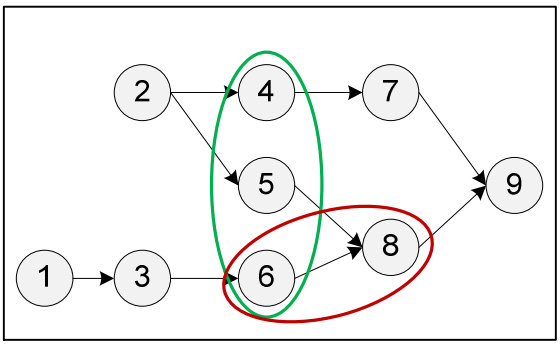


Figure 9: Overlapping Task Groups

### 3.2.9 Resource Constraints

*Resource constraints* involve stations that possess resources (a.k.a. "tooling") that are necessary to complete certain tasks, e.g. robotic lift support, pneumatic tooling, or other stationary resources. There may be one or more of each of these resources distributed along the line, and any task that interacts with a certain resource is forced to be assigned to a station that possesses the resource. Portable tools such as electric screwdrivers, wrenches, and similar are exempted, as these tools may be moved to any station along the line to support a given line balance solution.

Fixed resources generally are located on one side the line or the other, and can only be used upon the side of the vehicle that is facing it. Many resources have limited reach that only extends to the proximal side of the vehicle. Some resources might theoretically be extended for use on the other side of the vehicle, e.g. an AFCS torque driver, but would require cords or hoses to be drawn across the span of the vehicle in order to do so. Such behavior is generally forbidden due to mutilation risk. In any case, resources are typically duplicated on both sides of the line if they are needed on both sides of the vehicle, at least for all cases witnessed.

With this motivation *tool coverage zones* (TZ) are defined. As shown in Figure 10, a tool is deemed to cover the six PZ that are on the same side of the vehicle. The tool may satisfy the resource needs of any task in those PZ that is assigned at the station.

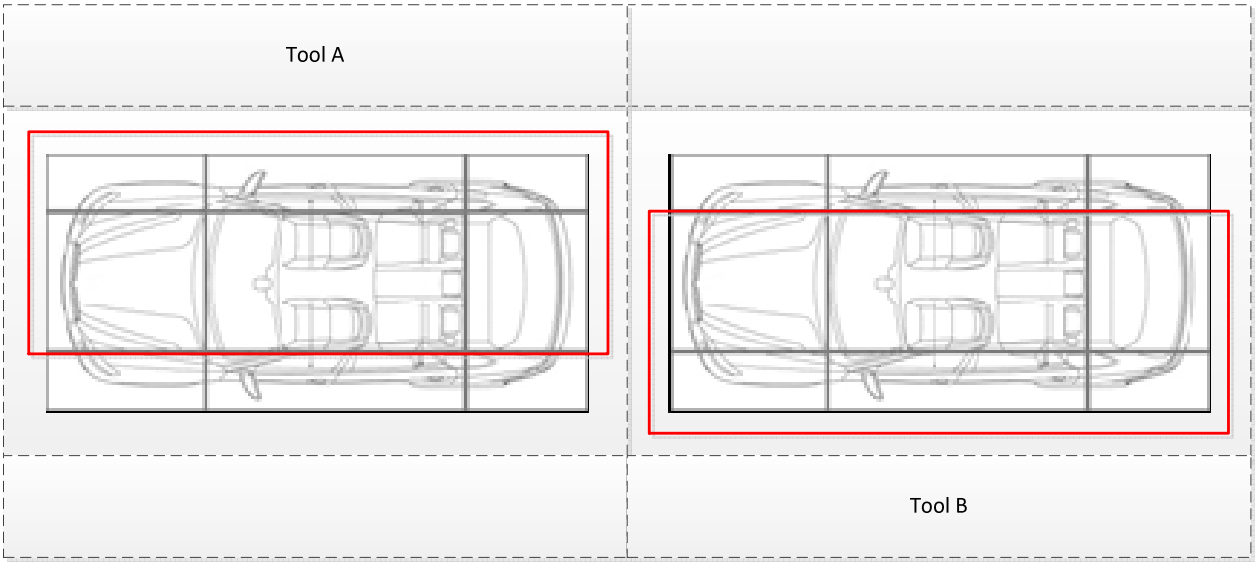


Figure 10: Tool Coverage Zones (TZ)

There is one exception to the TZ pattern. The MM (center) PZ is accessed by tools by passing through the side doors of the vehicle, or through the back hatch. The

MM PZ may not be accessed by passing over the front (V) of the vehicle, however, as such would require passing over the front hood of the vehicle, inducing mutilation risk. This is no issue if the vehicle is traversing the line in standard end-to-nose orientation, as only the vehicle sides are exposed to the line-side tooling. If the vehicle is oriented sideways on the line, however, then any tool on the line-side in front (V) of the vehicle may not be used in the MM (center) PZ.

## 3.3 Ranked Positional Weight

In this section the classic RPW (Helgeson and Birnie) heuristic is summarized, as a reference to the reader and to motivate certain characteristics of the MRPW heuristic presented in section 3.4.

Heuristic line balancing methods traditionally make task-to-station assignments one at a time within an iterative solution process. The iteration traverses both the set of unassigned tasks and the set of stations searching for a viable match, begetting two different general approaches to the structure of the iteration depending on which iterative process is nested inside the other. The *station-oriented* approach considers one station at a time, traversing the unassigned task set to identify tasks that may be assigned to the current station. When the no further tasks are eligible for assignment to the current station the next station is considered. The *task-oriented* approach selects tasks one at a time, and then searches the set of stations for a viable match. This methodology is referred to as "first fit decreasing" (FFD), as the first fit available is taken, and a decreasing priority score is used to guide the sequence of tasks chosen for assignment.

RWP (Helgeson and Birnie) is a classic sALB solution heuristic using the task-oriented approach. It solves the Type 1 ALB, with fixed cycle time, minimizing the number of stations. Let $P_i$ be the set of all tasks that are predecessors to task $i$, and $Q_i$ be the set of all tasks that are successors to task $i$. The ranked positional weight score ($r_i$) for each task is then calculated by Eq 3.

$$r_i = t_i + \sum_{j \in Q_i} t_j \qquad \text{Eq 3}$$

i.e. the RPW of a task is its own time plus the sum of all successor task times. One immediately apparent consequence of this prioritization scheme is that task $k$ will always be scored higher than task $l$ if $k$ is a predecessor of $l$, as $l \in Q_k$ and $Q_k \supset Q_l$. In addition, tasks that are unrelated by precedence are scored such that tasks with larger dependent work times are prioritized.

RPW begins by sorting tasks by $_i$ score. Taking tasks one at a time, beginning with the highest $_i$ score, the algorithm checks precedence relationships. If the chosen task has any predecessors, then they have been assigned already, as predecessors are guaranteed to have a higher $_i$. The highest station number from all predecessors is chosen as the starting point for the search (if the task has no predecessors, then station 1 is the starting point). With precedence constraints thus handled, only cycle time constraints remain to be considered. If the current station can take the task without exceeding cycle time then the task is assigned there. Else, the next station is considered, and so on. If no station can take the task then a new station is instantiated at the end of the line, and the task is assigned there. Figure 11 shows the RPW algorithm.

```
Algorithm: Ranked Positional Weight
    Inputs: Set of tasks, I
        Precedence
        Cycle time
    Output: Line balance solution

    Let I^free contain unassigned tasks.  Set I^free = I
    Let W contain stations.  Set W = {1}
    Let S_k contain the tasks assigned to station k. Set S_1 = ∅
    Calculate r_i ∀i ∈ I via Equation (1)
    While I^free ≠ ∅
        Select i = argmax_j {r_j | j ∈ I^free}, (the unassigned task with largest r_i)
        Remove task i from I^free
        If P_i = ∅, k = 1
        Else k = max{m|∀j ∈ P_i: j ∈ S_m}
        While i unassigned
            If t(S_k) + t_i ≤ c, (sufficient time remaining)
                S_k = S_k ∪ i, (assign task i to station k)
            Else, k = k + 1 (next station)
            If k > |W|
                W = W ∪ k, (induce new station k)
                Set S_k = ∅
```

Figure 11: RPW Algorithm

## 3.4 Modified Ranked Positional Weight Heuristic

In this section a constructive heuristic is presented, called the Modified Ranked Positional Weight (MRPW) heuristic, that seeks to identify a solution to the gALB problem with the additional constraints described above. The heuristic presented is an adaptation of the ranked positional weight (RPW) (Helgeson and Birnie) heuristic. A review of the traditional RPW is presented in section 3.3.

An important characteristic of RPW is that it ensures that task $k$ will always be scored higher than task $l$ if $k$ is a predecessor of $l$, as $l \in Q_k$ and $Q_k \supset Q_l$. In addition, tasks that are unrelated by precedence are scored such that tasks with larger dependent work times are prioritized. These two characteristics are preserved, with some necessary extensions, in the MRPW algorithm.

### 3.4.1 Extension: Grouping Constraints

Grouping constraints can link together precedence chains indirectly, suggesting that the classic RPW score is insufficient to ensure higher prioritization for early tasks in an indirectly linked precedence chain. The following Eq 4 and Eq 5 define a *responsibility set* of tasks that require task *i*, either directly or indirectly.

First define $\bar{Q}_i$ as the set that contains all tasks that succeed task $i$ or any task grouped with task *i*, but not any of the tasks within the group $G_i$ itself.

$$\bar{Q}_i = \left( \bigcup_{g \in G_i} Q_g \right) \backslash G_i \qquad \text{Eq 4}$$

The responsibility set $\bar{\bar{Q}}_i$ then recursively defines the set of all tasks that are dependent on task *i*, either directly by precedence relationships or indirectly by grouping with tasks that have precedence relationships, and so on.

$$\bar{\bar{Q}}_i = \left( \bigcup_{h \in \bar{Q}_i} \bar{\bar{Q}}_h \right) \cup G_i \qquad \text{Eq 5}$$

The set $G_i$ is removed from set $\bar{Q}_i$ in Eq 4 to prevent self-referencing recursion in the definition of Eq 5, as would occur in the case of precedence relationships between tasks of the same group. Note that $\bar{\bar{Q}}_i$ contains task $i$ and all tasks in the group $G_i$, enabling calculation of a ranked positional weight score $r_i^g$ that includes all tasks within the responsibility set of *i*, as shown in Eq 6.

$$r_i^g = \sum_{j \in \bar{\bar{Q}}_i} t_j \qquad \text{Eq 6}$$

This metric is analogous to combining grouped tasks into single super-tasks, and as a result all tasks $g \in G_i$ will be scored equivalently by $_i^g$. Relative scoring to break

these ties between in-group tasks can be measured with $r_i$. Figure 12 shows the growth of responsibility sets from task groupings.

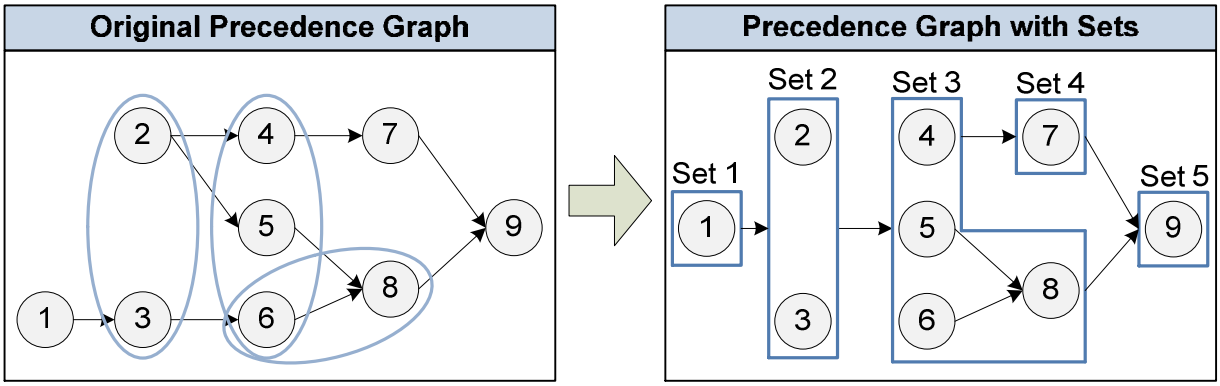


Figure 12: Group Definition of Responsibility Sets

### 3.4.2 Extension: Resource Constraints

Let $R_k^b$ denote the set of resources available at station $k$ in PZ $b$, $R_i$ denote the set of resources required by task $i$, and $m^{max}$ denote the maximum number of stations available on the line. Eq 7 is an *urgency score* that measures the relative importance of resource *res* by the last station to possess *res* on the existing assembly line. For example, if there are 17 stations and a resource last appears on station 15, then that resource has $Z_{res} = 2$.

$$Z_{res} = m^{max} - max\{k | \forall W_k^b : res \in R_k^b\} \quad \text{Eq 7}$$

Given two fixed resources $res_1$ and $res_2$, if the station number of the final appearance of $res_1$ is less than the final appearance of $res_2$ then $res_1$ will have a higher urgency score. This scoring reflects the fact that tasks that require $res_1$ have fewer opportunities to assign their predecessors during a first fit decreasing heuristic. To

impose prioritization of these predecessors each task $i$ inherits the maximum $Z_{res}$ from their responsibility set $\bar{\bar{Q}}_i$, as shown in Eq 8.

$$ {}_i^r = max\{Z_{res} | \forall\, es\colon es \in \cup_{j \in {}_i\bar{\bar{Q}}} R_j\} \qquad \text{Eq 8} $$

3.4.3 MRPW Algorithm

The following constructive heuristic algorithm is proposed to modify the ranked positional weight algorithm from section 3.3 to incorporate the environmental constraints shown in the earlier sections. The scoring metric ${}_i$ is augmented with two additional metrics; ${}_i^g$ from Equation (4) and ${}_i^r$ from Equation (6). These three metrics are combined in a hierarchy such that ${}_i^r$ dominates, followed by ${}_i^g$, and using ${}_i$ only to break ties. The dominance of ${}_i^r$ will be enforced in the algorithm by application of a BigM multiplier. In the worst case a task $i$ with no successors nor grouping must compete against task $j$, for which all tasks other than $i$ and $j$ are successors. Task $i$ requires a resource that only exists on station $k$, and the earliest resource needed in the responsibility set of $j$ is at station $k + 1$. One unit of urgency scoring must dominate the cumulative time of all tasks (except $i$), resulting in Eq 9:

$$ M = \sum_{j=1}^{n} t_j \qquad \text{Eq 9} $$

The BigM method shown here is developed to permit combining the metrics into a single composite score.

```
1   Algorithm ModifiedRankedPositionalWeight
2   Inputs:
3       Set of tasks, I
4       Precedence
5       Grouping constraints
6       Resource constraints
7       Cycle time
8       Output: Line balance solution
9
10      Let I^free contain unassigned tasks.
11      Set I^free = I
12      Let W contain stations.  Set W = {1}
13      Let S_k contain the tasks assigned to station k.
14      Set S_1 = ∅
15      Calculate r_i ∀i ∈ I via Eq 3
16      Calculate r_i^g ∀i ∈ I via Eq 6
17      Calculate r_i^r ∀i ∈ I via Eq 8
18
19      Start
20      While I^free ≠ ∅
21          Select i = argmax_i{Mr_i^r + r_i^g | i ∈ I^free}, (primary criterion)
22          In case of tie,  i = argmax_i{r_i | i ∈ Set of tied tasks}, (secondary criterion)
23          Collect group G_i
24          Remove task in G_i from I^free
25          If P_i = ∅, ∀i ∈ G_i, k = 1
26          Else k = max{m | ∀j ∈ P_i: j ∈ S_m: i ∈ G_i}
27          While G_i unassigned
28              If ∃j: z_j = z_i, j ∈ S_k^b AND t(S_k^b) + t̄_i ≤ c AND R_i ∈ R_k^b
29                  S_k^b = S_k^b ∪ G_i, (assign group to station k, WZ b)
30                  GoTo Start
31              ∀b: S_k^b ≠ ∅
32                  For each b that is compatible with z_i (PZ map eligible)
33                      If t(S_k^b) + t_i ≤ c AND R_i ∈ R_k^b
34                          S_k^b = S_k^b ∪ G_i (assign group to S_k^b)
35                          GoTo Start
36              If |∀b: S_k^b ≠ ∅| ≤ B_k^max, ∀b:
37                  For each b that is compatible with z_i (PZ map eligible)
38                      If t(S_k^b) + t_i ≤ c AND R_i ∈ R_k^b
39                          S_k^b = S_k^b ∪ i (assign group to S_k^b)
40                          GoTo Start
41              k = k + 1
42              If k > m^max
43                  m^max = m^max + 1
44                  S_k^b = ∅ ∀b ∈ B (empty station)
45                  R_k^b = ∪_{j∈V} R_j (give all known tooling)
```

Figure 13: MRPW Algorithm

### 3.4.4 MRPW Remarks

The algorithm begins by calculating $r_i$, $r_i^g$, and $r_i^r$ for each task $i$, by application of Eq 3-Eq 8. In lines 20 and 21, the set of unassigned tasks is sorted, first filtering tasks by maximum $_i^r$. If more than one task is tied in $_i^r$, then a maximum $_i^g$ filter is used to break the tie. If a tie still remains then a maximum $_i$ filter is used to break the tie. In the

event that there is still a tie, the next task *i* for assignment is chosen arbitrarily from the candidates. Next, all tasks that are linked to task *i* via adjacency, same-takt, or same-station constraints ($_{ij}^{adj} = 1$, $_{ij}^{st} = 1$, or $_{ij}^{ss}$) are

In lines 23 and 24 find the station at which to begin the search, by considering precedence. If the task has no predecessors then the station search will begin at station 1. Else the station search begins at the last station at which a predecessor task is assigned.

On line 26, three conditions are considered for assigning task *i*. First, if there is a task *j* at this station with the same PZ as task *i*, then the only WZ at this station to which task *i* can be assigned is the same WZ to which *j* is assigned. The second condition checks whether the WZ that contains *j* has sufficient capacity to add task *i* (in the average-time sense). The third condition checks whether the resource needs of task *i* are met at this location. If all of these conditions hold then task *i* is assigned to this station and WZ.

Lines 29-31 considers all WZ at this station that are not empty (possess at least one task). The motivation here is to attempt to add task *i* to an existing WZ if possible, rather than open a new WZ. If there exists an already open WZ that can hold task *i*'s PZ, and that WZ has sufficient time capacity, and the resource needs of task *i* are met at this location, then assign task *i* to that WZ.

The logic on line 34 considers relaxing the restriction that the WZ be non-empty. If the count of WZ with tasks assigned has not yet hit the $B_k^{max}$ limit at this station, then perhaps a new WZ can be opened to hold task *i*. The time capacity and resource satisfaction assignment conditions must again be met here.

Line 39 increments to the next station, as a feasible assignment at the current station was not found. Line 40 checks whether the new station index exceeds the number of stations given as input. If so, then the new station under consideration will not have any information regarding resource availability. This is considered a failure mode, as the task will be assigned to a station beyond the bounds of the given input data. All known resources are given to the new (dummy) station to ensure that task *i* can be assigned.

## 3.5 Last-Fit-Increasing Improvement Heuristic

The MRPW FFD heuristic seeks to pack tasks tightly as far to the beginning of the assembly line as possible, so that later task assignments may enjoy more freedom of assignment. The motivation for this strategy is founded in the theory of constraints, as it is more likely that a late-assigned task finds a feasible assignment location if the previously-assigned tasks have left more available locations. Efficiency is only a secondary concern for the MRPW algorithm, as it tends to produce solutions with several high-utilization workers toward the beginning of the assembly line.

In particular, the MRPW algorithm makes no attempt to consolidate takts after running to completion. If by some happenstance a worker on station 1 has been assigned no tasks at all after the first $n - 1$ assignment steps, and MRPW finds that the final task may feasibly be assigned to that worker, then the worker will be activated and assigned the task without consideration of where else the task might go. A superior alternative, in terms of efficiency, would be to assign this task to any already-activated worker. Since it is the last task to be assigned, there are no feasibility implications for consolidating, and clear efficiency implications for not doing so.

The Last-Fit-Increasing (LFI) improvement heuristic—so called because it operates on reverse logic from the First-Fit-Decreasing protocol—is designed to compensate for MRPW's disregard for consolidation. The intuition for the LFI approach is directly derived from the MRPW process. LFI begins by taking an existing feasible ALB solution, and borrows the RPW task metrics $r_i$, $r_i^g$, and $r_i^r$ and the compositing function $Mr_i^r + r_i^g$ developed in section 3.4.3. In contrast to FFD, however, LFI considers all tasks in *increasing* order of priority, first selecting task $i$ according to $i = argmin_i\{Mr_i^r + r_i^g\}$, with ties broken by $i = argmin_i\{r_i | i \in Set\ of\ tied\ tasks\}$, i.e. the lowest-priority, last-assigned task during MRPW. The small priority score for this task indicates that it is maximally free for assignment anywhere on the assembly line, relative to the other tasks, as e.g. resource and precedence constraints have not inflated its MRPW metric. This task is a good candidate for pushing as far to the end of the line as possible, so that it might be far out of the way of other tasks with more demanding constraints. However, it is entirely plausible that the worker at the end of the assembly line is currently inactive, in the ALB solution that we are trying to improve. Efficiency would not benefit from activating a new worker. Instead, the task is moved to the last already-active worker that can feasibly accept it. Iteration then continues with the next-lowest-priority task, in terms of MRPW metric. The hope is that some lightly-loaded takts, relics of the feasibility-oriented MRPW heuristic, might have all of their tasks removed to other destinations further down the line, thereby improving efficiency by consolidating two lightly-loaded takts into one. Figure 14 presents an algorithmic view of this logic.

```
1   Algorithm LFI_Improve
2       Inputs:
3       Set of tasks, I
4       Precedence
5       Grouping constraints
6       Resource constraints
7       Cycle time
8       Line balance solution
9       Output: Line balance solution
10
11      Let W contain stations.
12      Let S_k contain the tasks assigned to station k.
13      Calculate r_i ∀i ∈ I
14      Calculate r_i^g ∀i ∈ I
15      Calculate r_i^r ∀i ∈ I
16
17      Set I^free = I
18      While I^free ≠ ∅
19          Start
20          Select i = argmin_i{M r_i^r + r_i^g | i ∈ I^free}
21          In case of tie, i = argmin_i{r_i | i ∈ Set of tied tasks}
22          Collect group G_i
23          Remove tasks in G_i from current assignment and I^free
24          k = min{m | ∀i ∈ P_j : j ∈ S_m : i ∈ G_i}
25          While G_i unassigned
26              If ∃j: z_j = z_i, j ∈ S_k^b AND t(S_k^b) + t̄_i ≤ c AND R_i ∈ R_k^b AND
27                  S_k^b = S_k^b ∪ G_i, (assign group to station k, WZ b)
28                  GoTo Start
29              ∀b: S_k^b ≠ ∅
30                  For each b that is compatible with z_i (PZ map eligible)
31                      If t(S_k^b) + t_i ≤ c AND R_i ∈ R_k^b
32                          S_k^b = S_k^b ∪ G_i (assign group to S_k^b)
33                          GoTo Start
34              k = k - 1
```

Figure 14. Last Fit Increasing Improvement Heuristic

Note that in line 24 the initial station to begin the search, $k$, is chosen to be the earliest station at which is found a successor task to one of the tasks in $G_i$. This is the farthest that group $G_i$ might go toward the end of the line, lest a precedence constraint is violated. Recall that whichever successor task is found during this search has already moved previously in the course of the LFI, as it would have a lower MRPW score.

Station k is examined to determine if active takts can feasibly absorb group $G_i$. If so, the tasks are assigned and the loop proceeds to the next task group. If not, then the previous station will be considered. If no later station is found to which group $G_i$ can move, then the group will simply be reassigned at their originally assigned station.

## 3.6 Work Zone Blocking Improvement Heuristic

### 3.6.1 Motivation

The MRPW algorithm prioritizes task assignment based on precedence and task groupings through the responsibility set, and also resource and station eligibility through urgency scoring. One facet of the problem that it does not prioritize, however, is aggregate zone matching patterns between the set of all PZs on tasks and the compatible WZs on stations. Instead, MRPW considers one task group at a time, assigning the group to an already-active WZ if possible, or, failing that, merely arbitrarily opens any new WZ at the station that is compatible with the task group. Consider the highest priority task group, which will be the first to be assigned during the MRPW balancing algorithm. At the time of its assignment, no WZ are active. If all tasks within the group belong to a PZ which is compatible with more than WZ, then the algorithm picks one of the compatible WZ according to the arbitrary sequence {V (front), R (right), L (left), H (rear), and I (center)}. This choice may have repercussions in later task assignments, particularly so if there are few other tasks that can fit the selected WZ.

Consider the following degenerate example. Suppose the highest priority task group is comprised of tasks with the LV PZ. The MRPW assigns these tasks to the V WZ at station 1, and the Rubicon has been crossed. Unfortunately, all other tasks are of

the LM PZ, which is not compatible with the V WZ. As consequence, no other tasks can join the V WZ at station 1, and the associated worker will be greatly underutilized. Worse, it is possible that this poor choice of WZ results in no feasible solution being identified. One possible example of such an infeasibility is provided by supposing that station 1 may have at max one worker, and also has a tool in the L WZ that is unique. All tasks that need the tool will fail to find it. There are certainly many other ways in which infeasibilities might arise due to poor WZ choice. The MRPW heuristic attempts to pack tasks into the beginning of the line for good reason, as doing so retains latent capacity for subsequent assignment of highly constrained tasks.

The gALB problem specified in section 3.2 typically activates only a fraction of the WZs available for any solution, at least with respect to the testbed datasets acquired in conjunction with our industrial partner. Indeed, the WZs are constructed with heavy PZ overlap specifically to permit flexibility in task assignment, such that product or process system changes might induce relatively small ensuing changes to the balance. The intuition for the Work Zone Blocking (WZBlock) heuristic flows from this central concept, that active WZs only sparsely cover the set of all WZs in a solution. The ALB problem at hand may be thought of as taking two sequential stages: first, to choose which WZ are permissible for activation, and afterward to assign the tasks to them.

### 3.6.2 Work Zone Metrics

In the spirit of leveraging aggregate task PZ to WZ compatibility patterns, two metrics are introduced that provide insight into the relative quality of activating each WZ. First, let $Comp_{i,m}$ be an indicator variable on whether task $i$ is compatible with WZ $m$, as

shown in Eq 10. The indicator variable is true if and only if task *i* passes a battery of constraints. These constraints include:

1. Tooling. If task *i* requires any tools, then the associated tool coverage zone TZ at WZ's station must provide them.
2. Accessibility. This checks both WZ and PZ accessibility. The PZ of task *i* must not be blocked at this station, nor may WZ *m* itself be blocked.
3. Zone overlap. The PZ of task *i* must be associated with WZ *m* for possible assignment, as per the mapping provided in Figure 5.

Moreover, these conditions must be met for all tasks that are grouped with task *i* via adjacency, same-takt, or same-station linkages, not just for task *i* itself. Only if all of these conditions are met does $Comp_{i,m} = 1$, suggesting that task *i* might indeed be eligible for assignment to WZ *m*.

$$Comp_{i,m} = \begin{cases} 1 \; if \; Tooling, Accessibility, WZ \; to \; PZ \; overlap \\ 0 \; else \end{cases} \quad \text{Eq 10}$$

The first WZ metric is a "uniqueness" score, given in Eq 11. The internal term $\sum_{m' \in M, m' \neq m} C\,mp_{i,m'}$ counts how many other WZ are compatible with a given task *i*. This quantity is divided by the total number of other WZ on all stations combined, $|M| - 1$, and subtracted from 1, yielding the fraction of non-compatible WZs for task *i*. This value is then maximized over the set of all *i*, subject to *i* being compatible with WZ *m*. The final value delivered, $Uniqueness_m$, is a measure of the maximum degree to which WZ *m* is needed by any task, normalized on the [0,1] scale. A measure of 0 uniqueness indicates that WZ *m* is not particularly important to any task, as any task that is compatible with *m* is also compatible with every other WZ. On the other hand, a measure

of 1 uniqueness indicates that there exists some task for which WZ $m$ is the only possible assignment.

$$Uniqueness_{m\in M} = Max_{i\in I}\left\{1 - \frac{\sum_{m'\in M, m'\neq m} Comp_{i,m'}}{|M|-1}\middle| Comp_{i,m} = 1\right\} \quad \text{Eq 11}$$

The second WZ metric is the "flexibility" score, give in Eq 12. $Flexibility_m$ is simply the proportion of tasks that may be assigned at WZ $m$. A measure of 1 flexibility indicates that the WZ is compatible with every task. Zero flexibility indicates that the WZ is compatible with no tasks.

$$Flexibility_{m\in M} = \frac{\sum_{i\in I} Comp_{i,m}}{|I|} \quad \text{Eq 12}$$

Both the flexibility and uniqueness metrics provide insight into the relative usefulness and importance of each WZ. Additionally, both metrics are normalized to the scale [0,1]. To support the forthcoming WZBlock heuristic, the two metrics are simply added together to create a single composite score for each WZ, as shown in the algorithm summary of

Figure 15.

```
1  Algorithm WorkZoneScoring
2     Inputs:
3        TVGxMABR, 2-D array containing Comp_im compatibility indicators
4        numTVGs, the number of tasks
5        numMaBrTotal, the number of work zones on all stations
6     Output: scores, 1-D array scoring each WZ
7
8     For each task i
9     iCount = Sum(TVGxMABR(i,*)) - 1
10       For each WZ m
11          If TVGxMABR(i, m) = True Then
12              scores(m) = Max(scores(m), 1 – iCount/(numMaBrTotal - 1))
13
14     For each WZ m
15          kCount = Sum(TVGxMABR(*,m))
16         scores(m) = scores(m) + kCount / numTVGs)
17
18     Return scores
```

Figure 15. Work Zone Scoring Metric Computation

### 3.6.3 WZBlock Heuristic Algorithm

The WZBlock algorithm proceeds by iteratively blocking work zones from usage, by simulating accessibility constraints additional to any that may be in the original problem data. Recall, accessibility constraints prevent assignment of any task to the WZ. The approach aspires to identify and forbid the WZs that, if chosen for activation, are most likely to cause infeasibilities or sub-optimality in the objective function. The sum of the flexibility and uniqueness metrics presented in section 3.6.2 is used to discriminate between WZs. The algorithm shown in Figure 16 details the procedure.

```
1  Algorithm WZBlock
2     Inputs:
3        LBinit, the solution to be improved
4        TVGModel, the batch of all task information
5        StationModel, the batch of all station information
6        maxIter, the heuristic iteration cap
7     Output: Line balance solution
8
9     Variables:
10       NumRows, the number of tasks
11       NumCols, the number of work zones on all stations
12       TVGxMABR, a 2-D array of size NumRows x NumCols
13       mabrBlocks, a 1-D array of size NumCols
14
15    Save LBinit in LBbest
16    TVGxMABR(*, *) = False
17    For each Task i
18        Group all tasks assignment-linked with Task i
19        For each Station j
20            If Group passes Tooling, Accessibility at Station j
21               For each WZ k at Station j
22                  If Group passes Zone Compatibility at WZ k
23                     TVGxMABR(i, k) = True
24
25    mabrBlocks(*) = False
26    iter = 0
27    While iter < maxIter
28        Call WorkZoneScoring(TVGxMABR)
29        If no score is positive, end algorithm
30        Select WZ k with the smallest positive score
31            mabrBlocks(k) = True
32            TVGxMABR(*, k) = False
33            Call ModifiedRankedPositionalWeight, WZ k blocked
34            Save solution in LBtest
35            If LBtest is feasible
36                Call LFI_Improve
37                If LBbest is feasible
38                   If LBtest.Util > LBbest.Util
39                      Save LBtest in LBbest
40                Else
41                   Save LBtest in LBbest
42            ElseIf LBtest.NumDummy < LBbest.NumDummy
43                Save LBtest in LBbest
44            ElseIf LBtest.NumDummy = LBbest.NumDummy
45               If LBtest.Util > LBbest.Util
46                  Save LBtest in LBbest
47        iter = iter + 1
48    Loop
49
50    Return LBbest
```

Figure 16. Work Zone Blocking Improvement Heuristic

The variable LBbest retains the best solution found through the course of the algorithm. An initialization phase prepares the TVGxMABR matrix, which retains the compatibility indicator variables $C\,mp_{i,m}$ in Eq 10. Next, an iteration loop begins. During each iteration, the WZ with the smallest positive $Flexibility + Uniqueness$ score is chosen, and that WZ is blocked so that it may not be activated. Recall, the WZ metrics are constructed such that higher values indicate relatively higher value to a potential ALB solution, so the lowest values are targeted for exclusion. For redundancy purposes, composite scores of zero are not targeted, as a zero flexibility implies that no task can be assigned to the WZ regardless. The MRPW heuristic is then applied with the targeted WZ blocked from activation, followed by the LFI_Improve heuristic. The forthcoming solution retained if it is an improvement upon the best-yet-found solution, whereupon the WZBlock algorithm considers the WZ to block for the next iteration. Looping continues until the iteration count exceeds a user-defined maxIter hyperparameter, or no WZs are identified for potential blockage.

Note that the best-yet-found solution retained in LBbest might not be feasible. Infeasible solutions contain "dummy" stations, simulated stations with every tool from the entire line, appended to the end of the assembly line for harboring the tasks that could not otherwise be assigned to any station. Lines 35-46 perform a series of checks on the new ALB solution to determine whether it is superior to the incumbent solution. These checks prefer feasible solutions over infeasible ones, using utilization (efficiency) as a tie-breaker. Table 2 presents the four possible scenarios, and corresponding action.

| LBtest | LBbest | ACTION |
|---|---|---|
| FEASIBLE | FEASIBLE | Retain higher utilization |
| FEASIBLE | INFEASIBLE | Retain LBtest |
| INFEASIBLE | FEASIBLE | Retain LBbest |
| INFEASIBLE | INFEASIBLE | Retain fewest dummy stations. If tied, retain higher utilization |

Table 2. Criteria for Retention of Best-yet Solution

## 3.7 Conclusion

In this chapter the MRPW constructive heuristic is presented for the gALB problem described. Concepts from classic RPW have been extended to manage the features of the production environment in question. These features include zoning constraints, task groupings, and resource constraints. A task-oriented approach was selected to permit the heuristic to establish work areas for each station dynamically as needed to support a prioritized list of tasks. Due to the large number of constraints present that might prohibit assignment of tasks, several scoring metrics are introduced that prioritize tasks that have difficult satisfaction requirements. The prioritization scheme selects first tasks that require—or support successor tasks that require—stationary resources on the assembly line.

Two improvement heuristics are developed in conjunction with the MRPW constructive heuristic. The first, LFI, leverages the task prioritization metrics from MRPW to consolidate tasks and remove lightly loaded takts, thereby improving the efficiency of the ALB solution. The second improvement heuristic, considers the bifurcated problem of first selecting work zones, then assigning tasks. Two new work zone scoring metrics are developed, oriented towards superior selection of the work zones available for activation.

All heuristic methods have been coded, and a prototype delivered to our industrial partner as of December 2013. An algorithmic flow overview of this prototype is shown in Figure 17, detailed in Appendix 0. Between Spring 2014 and Summer 2015 three line balancing pilots have been launched to investigate using the prototype. Several other potential constraint classes have emerged during these discussions. An "eligible station" constraint is defined as a subset of stations to which a task may be assigned. It is similar to tooling constraints in spirit and in algorithmic interpretation, but without any physical resources or tooling coverage zones.

Another new constraint type is concerns assembly lines with more than one derivative. An additional capacity constraint is added to ensure that the average time for each derivative does not exceed the cycle time.

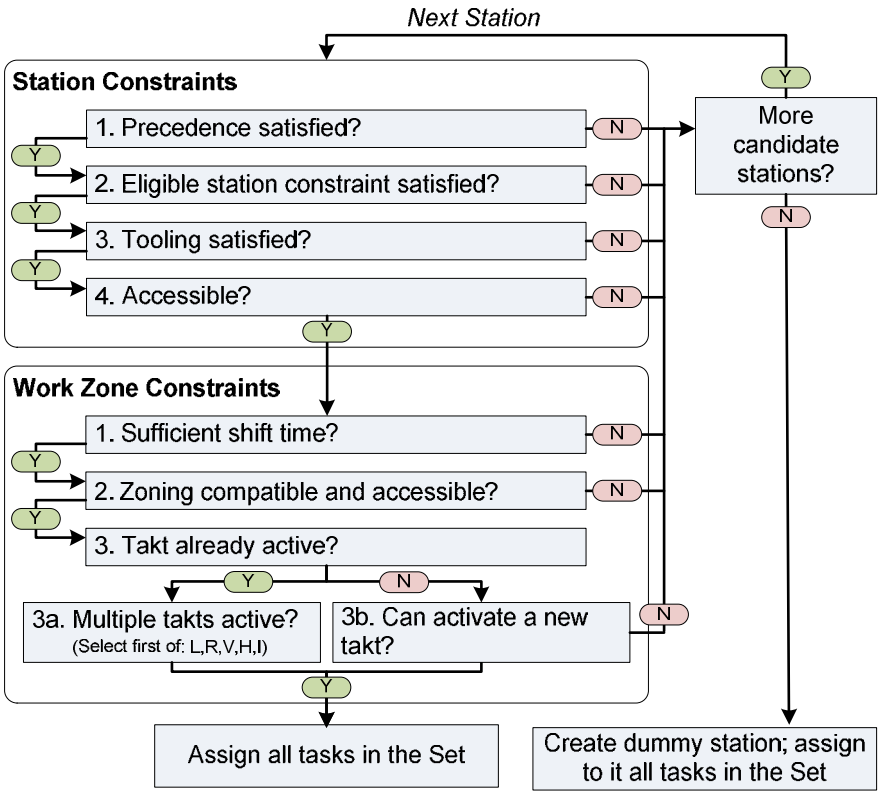


Figure 17: Heuristic Architecture

# CHAPTER FOUR

# 4 INTEGER PROGRAMMING MODEL

In this chapter an binary integer progamming (BIP) formulation is presented that models the gALB problem introduced in section 2.1.1, and solved with heuristics in chapter 3. In section 4.1 notation is introduced for the sets and input parameters for the problem. As discussed in section 4.2, many of these input parameters are preprocessed before execution of the IP, to save run time for the solver. Section 4.3 presents the IP formulation in three parts: decision variables, objective function, and constraints. Finally, section 4.4 discusses postprocessing feasibility verification, and poses a strategy for managing a type of precendence violation not handled by the IP constraints.

## 4.1 Sets and Input Parameters

Table 3 presents five sets over which the parameters and variables in the formulation are indexed, and the indexing variable typically used when quantifying over each set. Table 4 presents all input parameters for each problem instance. Preprocessing activities take some of the input parameters and compute transformed representations used in the formulation. Outputs from the preprocessing routine are shown in Table 5.

| SYMBOL | DESCRIPTION | INDEX |
|---|---|---|
| $\boldsymbol{I}$ | $set\ of\ all\ tasks\ \{1,\dots,n\}$ | $i,j$ |
| $\boldsymbol{K}$ | $set\ of\ all\ stations\{1,\dots,m\}$ | $k$ |
| $\boldsymbol{\Upsilon}$ | $set\ of\ all\ work\ zones\ \{L,R,V,H,I,P\}$ | $m$ |
| $\boldsymbol{\Phi}$ | $set\ of\ all\ product\ zones\ \{LV,MV,RV,LM,MM,RM,LH,MH,RH\}$ | $p$ |
| $\boldsymbol{\Psi}$ | $set\ of\ all\ tools\ \{1,\dots,\}$ | $t$ |

Table 3: IP Problem Sets

| SYMBOL | DESCRIPTION |
|---|---|
| $C$ | *Cycle time (sec)* |
| $T_i$ | *Time of task i (sec)* |
| $\phi_i$ | *PZ of task i* |
| $V_i$ | *Volume of task i, expected count of* $\frac{workpieces}{day}$ that require task *i* |
| $V_{max}$ | *Daily production volume, total* |
| $p_{ij}$ | *precedence relation of tasks i and j* $= \begin{cases} 1 \text{ if task } j \text{ must precede task } i \\ 0 \text{ else} \end{cases}$ |
| $r_{ij}^{adj}$ | *adjacency relation of tasks i and j* $= \begin{cases} 1 \text{ if task } i \text{ and task } j \text{ are adjacency related} \\ 0 \text{ else} \end{cases}$ |
| $r_{ij}^{st}$ | *same-takt relation of tasks i and j* $= \begin{cases} 1 \text{ if task } i \text{ and task } j \text{ are same-takt related} \\ 0 \text{ else} \end{cases}$ |
| $r_{ij}^{ss}$ | *same-station relation of tasks i and j* $= \begin{cases} 1 \text{ if task } i \text{ and task } j \text{ are same-station related} \\ 0 \text{ else} \end{cases}$ |
| $r_{ij}^{nt}$ | *not-same-takt relation of tasks i and j* $= \begin{cases} 1 \text{ if task } i \text{ and task } j \text{ are not-same-takt related} \\ 0 \text{ else} \end{cases}$ |
| $r_{it}^{tool}$ | *tool requirement* $= \begin{cases} 1 \text{ if task } i \text{ requires tool } t \\ 0 \text{ else} \end{cases}$ |
| $s_{km}^{W}$ | *WZ accessibility* $= \begin{cases} 1 \text{ if WZ } m \text{ is accessible in station } k \\ 0 \text{ else} \end{cases}$ |
| $s_{kp}^{P}$ | *PZ accessibility* $= \begin{cases} 1 \text{ if PZ } p \text{ is accessible in station } k \\ 0 \text{ else} \end{cases}$ |
| $s_k^{max}$ | *Maximum number of workers that may be assigned to station k* |

Table 4: Problem Input Parameters

| SYMBOL | DESCRIPTION |
|---|---|
| $\bar{t}_i$ | *Average time per workpiece of task i.* $\bar{t}_i = T_i \frac{V_i}{V_{max}}, \forall i \in \mathrm{I}$ |
| $Q_{kpt}^{c}$ | *Tool coverage zone* $Q_{kpt}^{c} = \begin{cases} 1 \text{ if station } k \text{ has tool } t \text{ covering PZ } p \\ 0 \text{ else} \end{cases}$ |
| $B_{kmp}$ | *WZ to PZ zoning compatibility* $B_{kmp} = \begin{cases} 1 \text{ if WZ } m \text{ can contain PZ } p \text{ at station } k \\ 0 \text{ else} \end{cases}$ |
| $w_{ij}$ | *common zone tasks.* $w_{ij} = \begin{cases} 1 \text{ if } \phi_i = \phi_j \\ 0 \text{ else} \end{cases}, \forall i, j \in \mathrm{I}$ |

Table 5. Problem Parameters Derived During Preprocessing

## 4.2 Preprocessing

Each WZ at each station lists the set of tooling resources that are provided at that location. The vehicle orientation determines which way the vehicle has been rotated upon the conveyor belt. Orientation information for each station is also given by input data. The WZ are fixed relative to the conveyor belt, and therefore do not change location if the vehicle orientation is rotated. The PZ covered by each WZ, however, are changed, as well as the tool coverage zones. There are 8 vehicle orientations possible, shown in Table 6.

| ORIENTATION | DESCRIPTION |
|---|---|
| **V-N** | Front-leading |
| **V-I** | Front-leading, inverted. Left & right are flipped, as if upside-down |
| **H-N** | Rear-leading |
| **H-I** | Rear-leading, inverted |
| **L-N** | Left-leading |
| **L-I** | Left-leading, inverted. Front & rear flipped, as if upside-down |
| **R-N** | Right-leading |
| **R-I** | Right-leading, inverted. |

Table 6. Vehicle Orientation Options

We can now derive parameters for tool coverage and zoning compatibility.

$$Q^c_{kpt} = \text{Tool Coverage} = \begin{cases} 1 \; if \; station \; k \; has \; tool \; t \; covering \; PZ \; p \\ 0 \; else \end{cases}$$

$$B_{kmp} = \text{Zoning compatibility} = \begin{cases} 1 \; if \; WZ \; m \; contains \; PZ \; p \; at \; station \; k \\ 0 \; else \end{cases}$$

These preprocessing parameters are found with the following method. Considering each station / WZ pair, select the greek code letter in Table 7, by indexing with the orientation at this station and the WZ. The greek code letter indicates the direction that the WZ is facing given the current orientation. Find the table matching the greek letter code in the set Table 8 through Table 12. There are three matrices shown for each code. The first matrix shows the PZ names, some of which are shaded. The dark gray PZ are eligible for mapping to the WZ; this information is duplicated in the second matrix. Both the light and dark gray PZ together define the tooling coverage zone for any tool that exists in the WZ; this information is duplicated in the third matrix.

| ORIENTATION | L | R | V | H | I |
|---|---|---|---|---|---|
| **V-N** | A | β | Γ | δ | ε |
| **V-I** | B | α | γ | δ | ε |
| **H-N** | B | α | δ | γ | ε |
| **H-I** | A | β | δ | γ | ε |
| **L-N** | Δ | γ | α | β | ε |
| **L-I** | Γ | δ | α | β | ε |
| **R-N** | Γ | δ | β | α | ε |
| **R-I** | Δ | γ | β | α | ε |

Table 7. WZ to Orientation Code Letters

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| LH | LM | LV | 1 | 1 | 1 | 1 | 1 | 1 |
| MH | MM | MV | 1 | 1 | 1 | 0 | 0 | 0 |
| RH | RM | RV | 0 | 0 | 0 | 0 | 0 | 0 |

Table 8. Code α: WZ / PZ Map and Tool Coverage Zone

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| LH | LM | LV | 0 | 0 | 0 | 0 | 0 | 0 |
| MH | MM | MV | 1 | 1 | 1 | 0 | 0 | 0 |
| RH | RM | RV | 1 | 1 | 1 | 1 | 1 | 1 |

Table 9. Code β: WZ / PZ Map and Tool Coverage Zone

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| LH | LM | LV | 0 | 1 | 1 | 0 | 0 | 1 |
| MH | MM | MV | 0 | 0 | 1 | 0 | 0 | 1 |
| RH | RM | RV | 0 | 1 | 1 | 0 | 0 | 1 |

Table 10: Code γ: WZ / PZ Map and Tool Coverage Zone

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| LH | LM | LV | 1 | 1 | 0 | 1 | 0 | 0 |
| MH | MM | MV | 1 | 1 | 0 | 1 | 0 | 0 |
| RH | RM | RV | 1 | 1 | 0 | 1 | 0 | 0 |

Table 11: Code δ: WZ / PZ Map and Tool Coverage Zone

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| LH | LM | LV | 0 | 1 | 0 | 0 | 1 | 0 |
| MH | MM | MV | 1 | 1 | 0 | 1 | 1 | 0 |
| RH | RM | RV | 0 | 1 | 0 | 0 | 1 | 0 |

Table 12. Code ε: WZ / PZ Map and Tool Coverage Zone

The following is an example to illustrate the orientation logic. Tasks require interaction with fixed points on the vehicle, regardless of orientation. Each task has a PZ,

as well as resource needs. Assume that at some station the orientation is R-N, i.e. the right side of vehicle leads. There is a tool located in the L WZ, i.e. on the left side of the line relative to production flow. Which PZ can be assigned to the L WZ on the vehicle? Which PZ can be serviced by the tool in the L WZ? See Figure 18 for a visual of this example. The red-shaded area is the tool coverage zone of the L WZ tool.

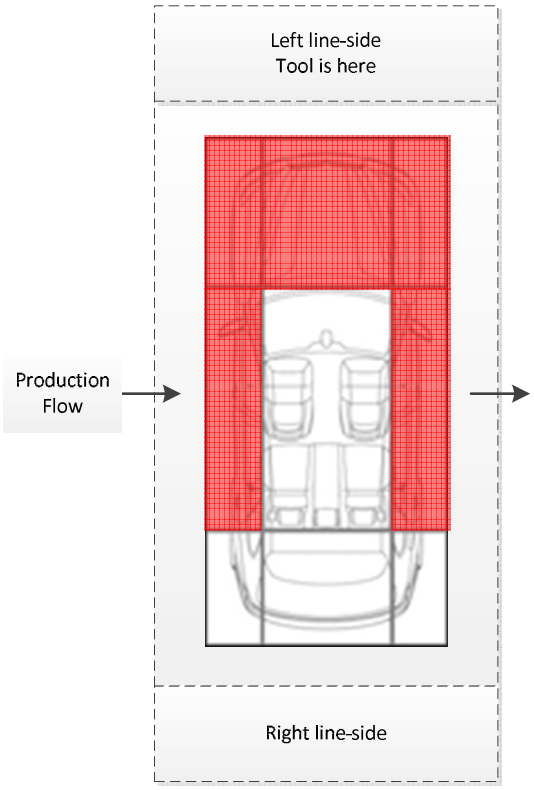


Figure 18: Orientation Example, R Leading

Looking in Table 7 above, we find the greek code letter γ. Looking in Table 10 for code γ, we see that the LV, MV, and RV PZ map to this (L) WZ. The tool covers PZ LV, MV, RV, LM, and RM. MM is not included in the tool coverage zone due to the mutilation risk of crossing over the hood of the vehicle to access the vehicle cabin.

## 4.3 IP MODEL

### 4.3.1 Decision variables

$$x_{ikm} = \begin{cases} 1 \; if \; task_i \; is \; assigned \; to \; station \; k \; and \; WZ \; m \\ 0 \; else \end{cases}$$

$$y_{km} = \begin{cases} 1 \; if \; WZ \; m \; at \; station \; k \; is \; active \\ 0 \; else \end{cases}$$

### 4.3.2 Objective

Two options are available for objective function. The first one, $z_1$, simply minimizes the number of open WZ, i.e. the number of workers. This is the classic objective for fixed cycle time ALB problems, modified only to conflate WZ (instead of stations) with workers.

$$minimize \; z_1 = \sum_{k \in \mathrm{K}} \sum_{m \in \mathrm{M}} y_{km}$$

The second objective function, $z_2$, seeks to maximize the sum of squares of each worker's task load, where the task load for a worker is simply the summed average time for all tasks assigned to the worker. Optimal solutions with this objective will favor dividing the workers into 2 camps: heavily loaded and lightly loaded. As tasks are moved off of lightly loaded workers and into heavier loaded ones, this objective improves. Consider a solution with worker A and worker B both loaded to 80% of capacity. Suppose further that a task can be moved from A to B such that the loadings are now 60% and 100%, respectively. This objective function favors the imbalanced alternative, as $80^2 + 80^2 < 60^2 + 100^2$.

$$maximize \; z_2 = \sum_{k \in \mathrm{K}} \sum_{m \in \mathrm{M}} \left( \sum_{i \in \mathrm{I}} x_{ikm} \, \bar{t}_i \right)^2$$

The motivation for the $z_2$, objective is to bias a search toward solutions with some lightly loaded workers, as these solutions may be 'close' to solutions where those same workers are empty, having all of their tasks moved elsewhere. In this parlance, 'close' solutions are similar across many decision variables, differing in only a few. Intuitively, if a search has found a solution with, say, 17 workers, and is working to find a superior solution with 16 workers, then it might be beneficial to search through solutions that are 'close' to 16, as these would more quickly lead to 16.

Such is the argument for using $z_2$, but unfortunately the objective is nonlinear, as it squares a decision variable. This IP is linear in all other respects, however. Perhaps for some heuristics or metaheuristics this objective would yield superior performance. Limited testing has shown a drastic performance penalty for using $z_2$ instead of $z_1$, presumably due to the necessity of using nonlinear solver packages instead of only linear ones.

### 4.3.3 Constraints

Table 13 presents formulas for all constraints in the IP. The left-most ID column is referenced in subsequent text to provide description for the mechanics of each constraint.

| ID | Constraint Formula | Quantification |
|---|---|---|
| C1 | $\sum_{k\in \mathrm{K}}\sum_{m\in \mathrm{M}} x_{ikm} = 1$ | $\forall i \in \mathrm{I}$ |
| C2 | $\sum_{i\in \mathrm{I}} x_{ikm}\bar{t}_i \le y_{km}C$ | $\forall k \in \mathrm{K}, m \in \mathrm{M}$ |
| C3 | $\sum_{m\in \mathrm{M}} y_{km} \le s_k^{max}$ | $\forall k \in \mathrm{K}$ |
| C4 | $\sum_{k=v+1}^{\lvert k\rvert}\sum_{m\in \mathrm{M}} x_{jkm} \le 1 - \sum_{k=1}^{v}\sum_{m\in \mathrm{M}} x_{ikm}$ | $\forall v = 1 \dots \lvert k\rvert - 1,$ if $p_{ij} = 1$ |
| C5 | $x_{ikm} = x_{jkm}$ | $\forall i, j \in \mathrm{I}, k \in \mathrm{K}, m \in \mathrm{M},$ $if\ r_{ij}^{adj} = 1\ or\ r_{ij}^{st} = 1$ |
| C7 | $\sum_{m\in \mathrm{M}} x_{ikm} = \sum_{m\in \mathrm{M}} x_{jkm}$ | $\forall i, j \in \mathrm{I}, k \in \mathrm{K},$ $if\ r_{ij}^{ss} = 1$ |
| C8 | $\sum_{\alpha\in \mathrm{clique}} x_{akm} \le 1$ | $\forall k \in \mathrm{K}, m \in \mathrm{M}, clique\ s.t.\ all\ x\ have\ r_{ij}^{nt} = 1$ |
| C9 | $Q_{it}^{u}\left(\sum_{m\in \mathrm{M}} x_{ikm}\right) \le Q_{k,b_i,t}^{c}$ | $\forall i \in \mathrm{I}, k \in \mathrm{K}, t \in \mathrm{T}$ |
| C10 | $x_{ikm} = 0$ | $\forall i \in \mathrm{I}, k \in \mathrm{K}, m \in \mathrm{M},$ $if\ A_{km}^{m} = 0$ |
| C11 | $\sum_{m\in \mathrm{M}} x_{ikm} = 0$ | $\forall i \in \mathrm{I}, k \in \mathrm{K},$ $if\ A_{k,b_i}^{p} = 0$ |
| C12 | $x_{ikm} = 0$ | $\forall i \in \mathrm{I}, k \in \mathrm{K}, m \in \mathrm{M},$ $s.t.\ B_{km,b_i} = 0$ |
| C13 | $x_{ikm} + \sum_{m'\in \mathrm{M}\setminus m} x_{jkm'} \le 1$ | $\forall i, j \in \mathrm{I}, k \in \mathrm{K},$ $m \in \mathrm{M},$ $if\ w_{i,j} = 1$ |
| | $x_{ikm}\epsilon\{0,1\}$ | $\forall i \in \mathrm{I}, k \in \mathrm{K}, m \in \mathrm{M}$ |
| | $y_{km}\epsilon\{0,1\}$ | $k \in \mathrm{K}, m \in \mathrm{M}$ |

Table 13. IP Constraints

(C1) Every task must be assigned to exactly one station and worker.

(C2) The average workload assigned to each worker cannot exceed the cycle time. No tasks may be assigned to a worker at a particular WZ and station unless the corresponding y variable is set to 1.

(C3) The maximum number of workers at station *k* is bounded by $s_k^{max}$.

(C4) Enforces precedence constraints. This constraint only applies if task *j* must precede task *i*. Consider any station *v*, except for the last station on the assembly line. If

task *i* is assigned to any station between 1 and *v*, then task *j* cannot be assigned to any station after *v*. Note that this constraint only considers precedence constraints at the station level, and does not consider task sequencing within each station. See section 4.4 for resolution of sequencing related precedence issues.

(C5) Enforces adjacency and same-takt constraints. If either of these relationship exists between tasks *i* and *j* then $x_{ikm}$ must be equal to $x_{jkm}$. If $x_{ikm}$ and $x_{jkm}$ differ then $x_{ikm} - x_{jkm}$ will be 1 for some choice of *k* and *m*.

(C7) Enforces same-station constraints. The argument presented in C5 is used here, except WZs are aggregated over rather than quantified, as individual WZs need not be examined for same-station constraints.

(C8) Enforces not-same-takt constraints. This is a clique inequality, enforced only for cliques defined by each not-same-takt group. If there is a not-same-takt relationship between any tasks *i* and *j*, then they form a clique, and $x_{ikm}$ and $x_{jkm}$ cannot be assigned to any matching station and WZ.

(C9) Enforces tooling constraints. If task *i* is assigned to station *k* and requires tool *t*, then tool *t* must exist at station *k* and cover the PZ of task *i*. See preprocessing arguments in section 4.2 for derivation of the *Q* parameters.

(C10) Enforces WZ accessibility constraints. If a WZ is not accessible at some station, then no tasks may assigned there.

(C11) Enforces PZ accessibility constraints. If a PZ is not accessible at some station, then no task with that PZ may be assigned at the station.

(C12) Enforces zoning compatibility. Tasks may only be assigned a WZ at some station if the PZ of the task is compatible with that WZ. See preprocessing arguments in section 4.2 for derivation of the *B* parameter.

(C13) Enforces zone overlap constraints. This constraint considers all task pairs *i* and *j* that share the same PZ, by consulting the preprocessing parameter $w_{ij} = 1$, established using the equation $w_{ij} = \begin{cases} 1 \; if \; \phi_i = \phi_j \\ 0 \; else \end{cases}, \forall i, j \in \mathrm{I}$. If *i* and *j* are further assigned to the same station, and task *i* is assigned to WZ *m*, then task j must also be assigned to WZ *m*. This is accomplished by restricting *j* from assignment to any other WZ that is not *m*. This constraint prevents workers from interfering with one another. Without this constraint it would be possible for two workers to simultaneously attempt tasks within the same PZ.

## 4.4 Postprocessing: Iterative Precedence Verification

The precedence constraint in the IP is $\sum_{k=v+1}^{|k|} \sum_{m \in \mathrm{M}} x_{jkm} \leq 1 - \sum_{k=1}^{v} \sum_{m \in \mathrm{M}} x_{ikm}, \forall v = 1 \ldots |k| - 1, \mathrm{s.t.} \, p_{ij} = 1$. This constraint prevents predecessor tasks from being assigned to later stations than their successors. It does not, however, prevent potential precedence violations within a single station. Consider an example where there are two workers at one station, and there is a precedence relationship between a task pair *i*, *j* that is split between the workers. If task *j* is the predecessor, then it must be completed by one worker before the other worker can begin their task *i*. Depending on the task sequences used by the two workers, it may not be possible for task *j* to be completed early enough in the cycle time. The IP does not consider task

sequencing, so an infeasibility of this kind may be present in the solution produced by the IP.

It is possible to extend the IP to manage task sequencing, and thereby prevent such infeasibilities, but at the cost of introducing a new set of decision variables related to sequencing. Instead, the IP solution is checked in post-processing to determine whether any cross-WZ precedence violations exist. If found, then the IP is re-run, introducing a new diversification constraint that prohibits the exact solution found in the prior run. The constraint is formulated as $\sum x'_{ikm} < n$, where the $x'_{ikm}$ are chosen as the assignment solutions from the previous, infeasible run. In the prior run all of these $x$ decision variables were equal to one, and their sum was $n$. This constraint restricts at least one of this set of decision variables to zero, and thereby the new solution will be different from the old one. The IP is iteratively run until a solution is produced that does not evidence cross-WZ precedence violations. In the case of multiple iterations, all diversification constraints from prior runs are retained, to prohibit any of the earlier solutions.

The test datasets available did not evidence any cross-WZ precedence violations. Likely this is due to the propensity of precedence relationships to exist between tasks of the same PZ within these datasets. These precedence features preempt the potential violation, since tasks with the same PZ cannot be assigned to different WZ within the same station. It is likely, however, that other datasets with different precedence features will run afoul of the violation. For a hypothetical dataset with many precedence relationships crossing over zoning boundaries, it may be less costly overall to adopt the necessary IP extensions to manage task sequencing within a single larger IP execution,

rather than a series of smaller IP executions. It would be very interesting to research the relative cost/benefit of such an extension. There is perhaps some means of measuring precedence zoning complexity for a dataset, in purpose of determining whether or not to explicitly model task sequencing within the IP.

CHAPTER FIVE

# 5 APPLICATION OF SOLUTION METHODOLOGIES

This chapter describes a computational experiment aimed at developing performance benchmarks for both the heuristic methods established in chapter 3 and the IP formulation established in chapter 4. For the latter, solver time requirements are the criteria of interest, and the scaling of this time as a function of problem size. The primary criteria of interest for the heuristics methods is the quality of generated solutions, as measured by optimality gap.

## 5.1 Experimental Configuration

To execute the experiment, the IP and two heuristic methods are each individually applied to a suite of testbed problem instances. Section 5.1.1 discusses the construction of the test data sets. In section 5.1.2, the run-time configuration and hyperparameters are shown for each solution method.

### 5.1.1 Test Data Sets

The ALB literature provides no testbed data sets that exhibit all constraints modeled by the methods in chapters 3 and 4. There are, however, three sets of test data collected during the development of these methods, in conjunction with our industrial partner. The three data sets are labeled “Band 1”, “Band 26”, and ”Band 30.” Table 14 summarizes some properties of each of these initial data sets.

| | | | |
|---|---|---|---|
| **BAND 1** | 13 | 396 | 12 |
| **BAND 26** | 9 | 317 | 12 |
| **BAND 30** | 10 | 300 | 3 |

Table 14. Test Data Set Properties

These three initial data sets form the testbed basis of the experiment. In addition, an array of ALB sub-problems are appended, each of which is formed as a subset of one of the initial sets. Consider an ALB instance called $ALB_0$ with $m_0$ stations, and a corresponding solution $sol_0$ assigning all $n_0$ tasks. A new ALB sub-problem $ALB_1$ may be formed by isolating any subset of $m_1 = m_0 - b$ stations from $ALB_0$, where $0 < b < m_0$ and integer $b$, and all $n_1$tasks assigned to those stations by $sol_0$. $ALB_1$ may then be re-solved as a separate instance. It cannot be assumed that the isolated portion of solution $sol_0$ is optimal for the partition $ALB_1$.

To map tasks to stations within the sub-problems, $sol_0$ is taken from the manually created line balance solution in use by our industrial partner during data collection. Any feasible ALB solution could be used for sub-problem creation. The manually created solution is chosen as it exhibits some tendencies towards vertical smoothing objectives. The manual solution is not optimally vertically balanced, of course, but effort is made to establish relatively equal average task load for each worker. In contrast, the solutions generated by the heuristics and IP in chapters 3 and 4 are generated according to efficiency objectives only, and may show poor vertical smoothing. In the worst-case, a solution generated with the algorithmic methods may assign tasks such that all workers are maximally utilized with zero idle time, except for one worker who is only lightly loaded. There are implications to using a $sol_0$ with poor vertical properties during the sub-problem partitioning process. Idle time within ALB problems corresponds to degrees of freedom for task assignment, i.e. the size of the solution space. If all workers within a

particular partition are maximally loaded with tasks, then a sub-problem generated from that partition will have few degrees of freedom.

All sub-problems created from the initial three data sets have three or more stations. This minimum problem size is chosen to prohibit inclusion of relatively trivial 1- or 2-station problem instances in the experiment. Additionally, sub-problem partitions are formed only from adjacent stations. Using only these two limiting constraints, an ALB sub-problem is created for every possible partition of the initial data sets. Figure 19 presents the concept graphically. Let a $(m_o, x)$ pair represent each partition, where $m_o$ is the number of stations in the partition, and $x$ is the sequence ID number of the partition. The first three stations and all assigned tasks are designated as sub-problem $(3,1)$. Stations 2, 3, and 4 are partitioned into sub-problem $(3,2)$, and so on. A total of $\sum_{i=3}^{m} i(m-i+1)$ sub-problems are created for each initial dataset, where $m$ is the number of stations in the dataset. Note that the full-sized (non-partitioned) initial problem is included in this accounting, totaling 130 datasets.

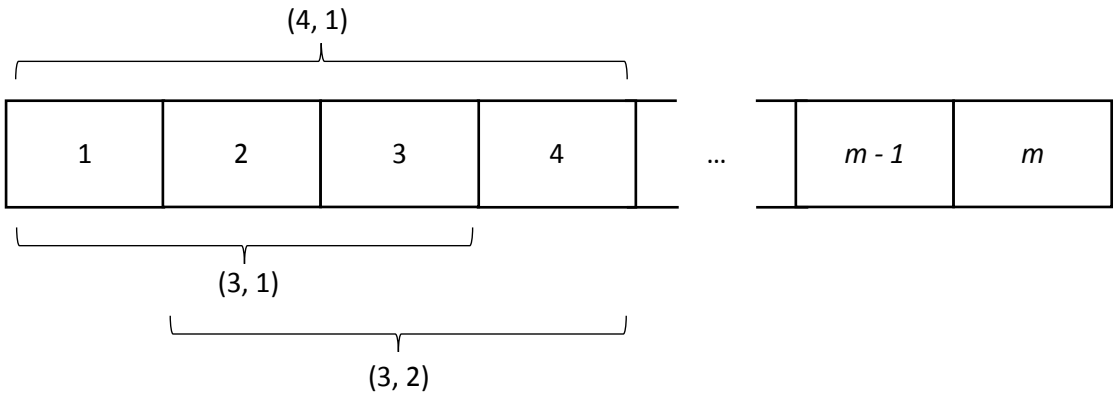


Figure 19. Sub-problem Partitioning Pattern

Three potentially significant factors influencing performance outcomes are tracked among the generated population of datasets: counts of stations, tasks, and unique tools (i.e. number of discrete types of tools on the line, regardless of multiples.) Figure 20 displays the values for these three parameters across all datasets. Note that station and task counts track very closely together, with a 0.965 correlation coefficient. The average number of tasks per station is 31.1. The count of unique tools is relatively loosely associated with the other factors. Between task and tool counts the correlation coefficient 0.647, and 0.663 between station and tool counts.

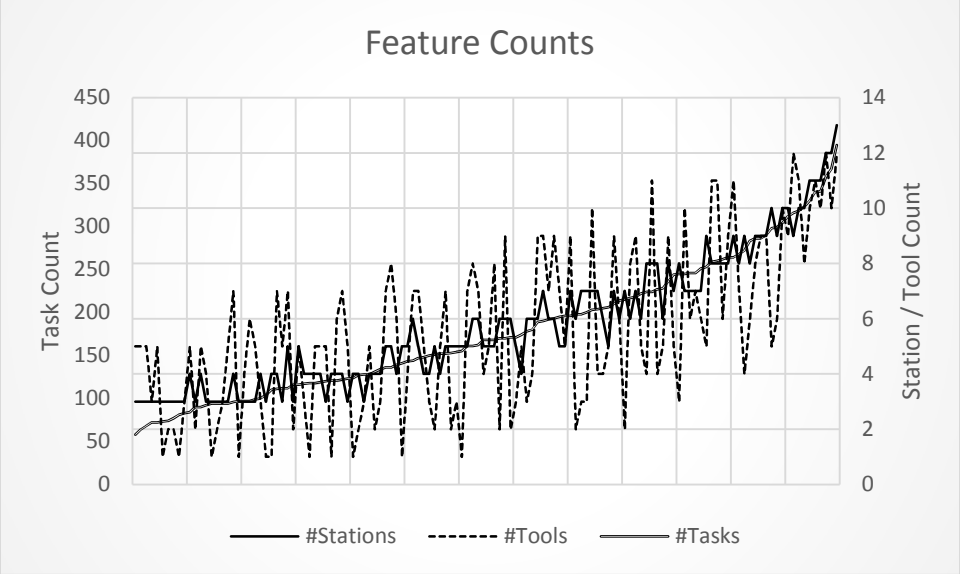


Figure 20. Relative Task, Station, and Tool Counts

### 5.1.2 Method Parameters

#### 5.1.2.1 Integer Program

The IP solutions for each sub-problem ALB instance were executed on the Linux-based Palmetto Cluster at Clemson University. The IP formulation is modeled in AMPL, and run using the Gurobi 5.0 Linux 64 solver. For each problem instance 8 processors and 120gb of RAM are allocated.

#### 5.1.2.2 Heuristics

Two heuristics are applied to each ALB sub-problem. The first is the MRPW constructive heuristic discussed in section 3.4, with the LFI improvement heuristic from section 3.5 subsequently applied if a feasible solution is found. This combination of heuristics is called H1 in the experiment. The second, called H2 in the experiment, is the work zone blocking improvement heuristic, as discussed in section 3.6. The output of the H1 heuristics is used as an input to the substation blocking improvement heuristic, so the objective function found by H2 should always be at least as good as that found by H1. The maximum iteration hyperparameter for substation blocking is set to 10.

All heuristics were implemented in VBA, and executed on a 64-bit Windows PC with 2.40GHz processor and 2GB RAM.

## 5.2 Results

The totality of experimental data is shown in Table 17, Table 18, Table 19, and Table 20. All ALB problem instances were either successfully solved or found infeasible by each of the solution methods.

### 5.2.1 Feasibility

Note that not all of the problem instances are feasible for all methods, as summarized in Table 15. IP feasibility is a particular concern, as this indicates that there is truly no solution to the problem instance. The instances were generated by consultation with actual line balance solutions, as discussed in section 5.1.1. As there existed an implemented balance upon the real assembly line for these instances, it is surprising to find that there is no solution to the IP formulation.

Closer inspection of the original, manually-created solutions used by the infeasible instances reveals the cause of this phenomenon. In each case, the original line balance evidenced violations of the zoning constraints embedded in the algorithmic methods. In several cases, the PZ of tasks associated with a single worker would span across zoning boundaries associated with any single WZ. The algorithmic methods would necessarily consider such a solution to be infeasible. In practical application upon the line, however, these violations were tolerated. This phenomenon suggests potential future work for generalizing the constraint set, perhaps by implementation of a penalty function to discourage, but not disallow, these constraint violations.

| | | IP feasibility | | Heuristic 1 feasibility | | Heuristic 2 feasibility | |
|---|---|---|---|---|---|---|---|
| Station Count | Instances | Count | Percent | Count | Percent | Count | Percent |
| 3 | 26 | 24 | 92.3% | 22 | 84.6% | 22 | 84.6% |
| 4 | 23 | 21 | 91.3% | 17 | 73.9% | 17 | 73.9% |
| 5 | 20 | 19 | 95.0% | 14 | 70.0% | 14 | 70.0% |
| 6 | 17 | 17 | 100.0% | 13 | 76.5% | 13 | 76.5% |
| 7 | 14 | 14 | 100.0% | 11 | 78.6% | 11 | 78.6% |
| 8 | 11 | 11 | 100.0% | 9 | 81.8% | 9 | 81.8% |
| 9 | 8 | 8 | 100.0% | 7 | 87.5% | 7 | 87.5% |
| 10 | 5 | 5 | 100.0% | 5 | 100.0% | 5 | 100.0% |
| 11 | 3 | 3 | 100.0% | 3 | 100.0% | 3 | 100.0% |
| 12 | 2 | 2 | 100.0% | 2 | 100.0% | 2 | 100.0% |
| 13 | 1 | 1 | 100.0% | 1 | 100.0% | 1 | 100.0% |

Table 15. Solution Feasibility

Overall only 5 of the 130 instances resulted in IP infeasibility, or 3.8%. All 5 of these instances were found in smaller problem instances with 3-5 stations, and all on data from Band 26. Interestingly, all of these infeasibilities disappear in the larger Band 26 problem instances, of which the small infeasible instances are a subset. As a general rule, larger problem instances offer more degrees of freedom for the movement of tasks. It

appears that problems encountered in the infeasible instances were able to be assuaged by addition of an extra station or two, providing more space over which tasks can mingle, despite also bringing in a fresh set of new tasks.

The heuristic methods failed to find a feasible solution for 26 out of 130, or 20% of problem instances. Further, both H1 and H2 are infeasible for the same problem instances. The zone blocking heuristic was unable to resolve the infeasibilities of any H1 solution, unfortunately. Considering only the 125 instances for which a feasible solution is possible, as evidenced by a feasible IP solution, the heuristic methods were successful in finding a feasible solution for 83.2% of them. Of the 26 heuristic-infeasible instances, 24 of them are sourced in Band 26 data, and 2 are from Band 1.

### 5.2.2 IP Runtime

The average time to execute the IP model was 3.095 seconds, aggregated across all 130 datasets. Figure 21-Figure 23 display IP runtime presented against one factor each, using task, tool, and station count, respectively. Regression lines are plotted on each graph, using a degree-2 polynomial fit and an intercept of zero. Both task and station count display relatively good fit, with $R^2$ values of 0.815 and 0.871, respectively. Tooling count did not trend as strongly with IP solution time, delivering a $R^2$ value of 0.205.

Fits with exponential regression functions yielded slightly lesser fit metrics relative to the polynomials. This result should be taken with a grain of salt, as it will almost assuredly fail to extrapolate as the problem size scales upwards. I would expect the NP nature of the problem to assert itself with larger problems. Still, it is remarkable

to have solved a 400-task problem, easily a middle-sized problem by ALB standards, in only 20 seconds.

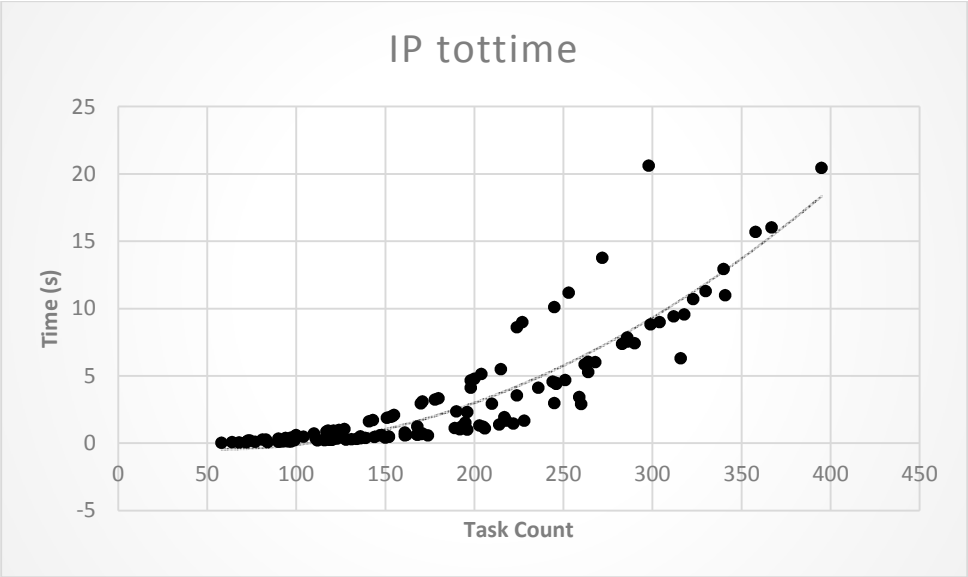


Figure 21. IP Runtime vs. Number of Tasks

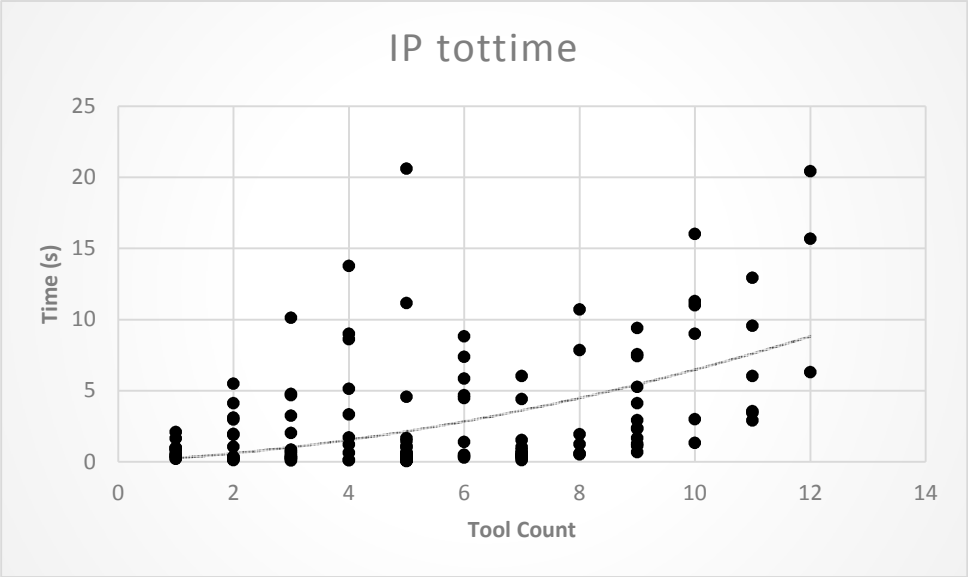


Figure 22. IP Runtime vs. Number of Tools

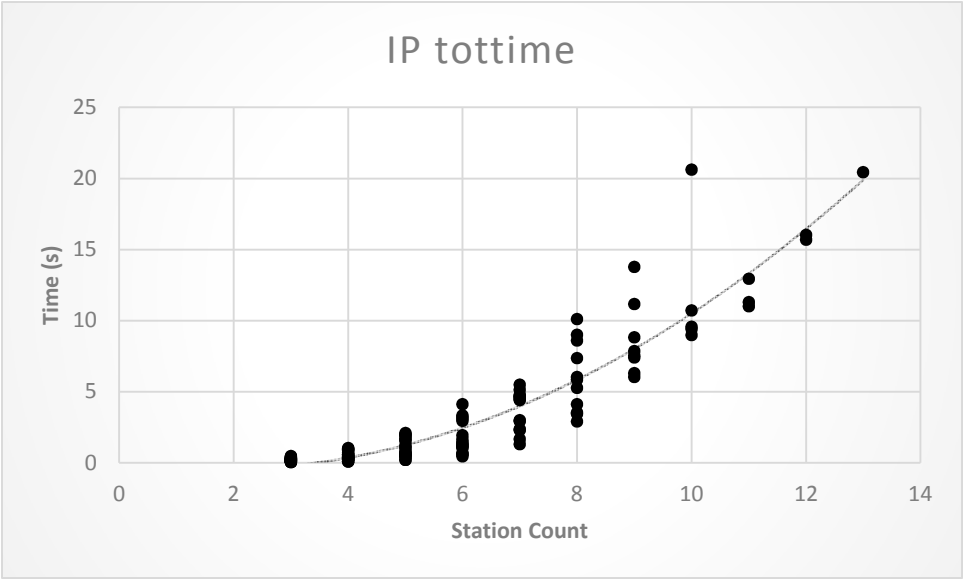


Figure 23. IP Runtime vs. Number of Stations

Figure 21 displays strong visual evidence of clustering within the data. Each finger in the scatterplot is representing a separate input data source: the three bands of original data. Figure 24-Figure 26 present IP runtime versus task, tool, and station counts, with each band's datasets collected separately. Figure 24 shows particularly strong differentiation between bands, and consistency within bands. It appears that there are some characteristics particular to each band which carry strong implications towards the IP runtime.

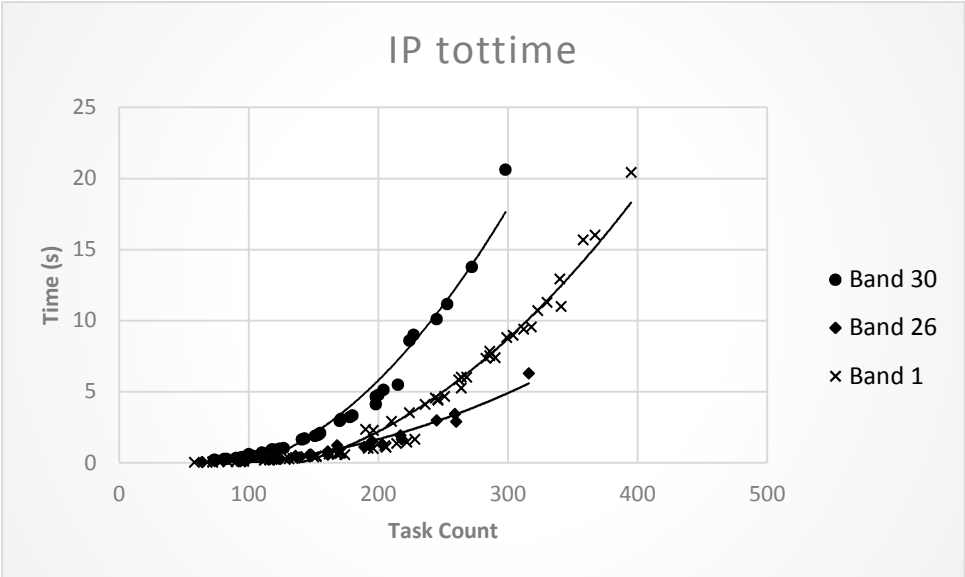


Figure 24. IP Runtime by Band and Task Count

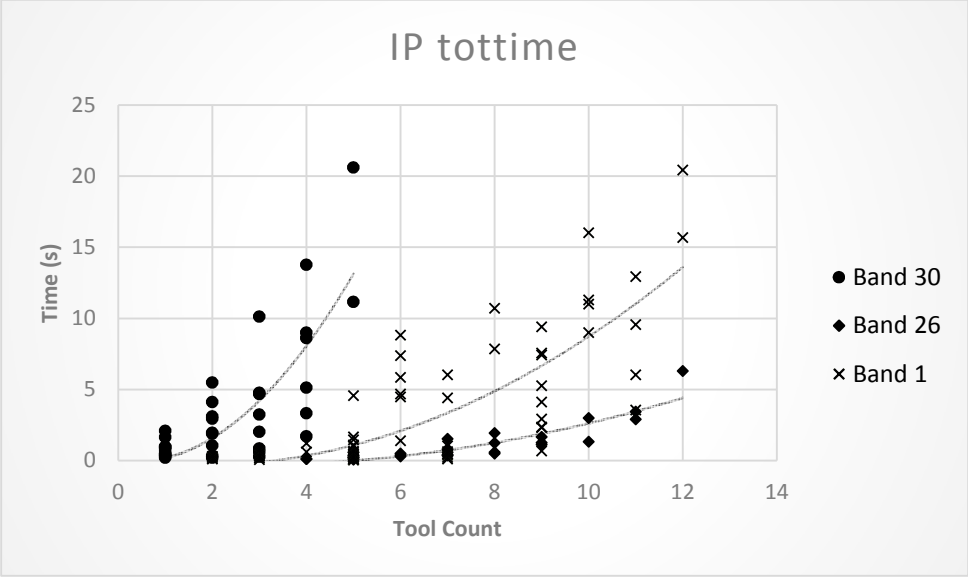


Figure 25. IP Runtime by Band and Tool Count

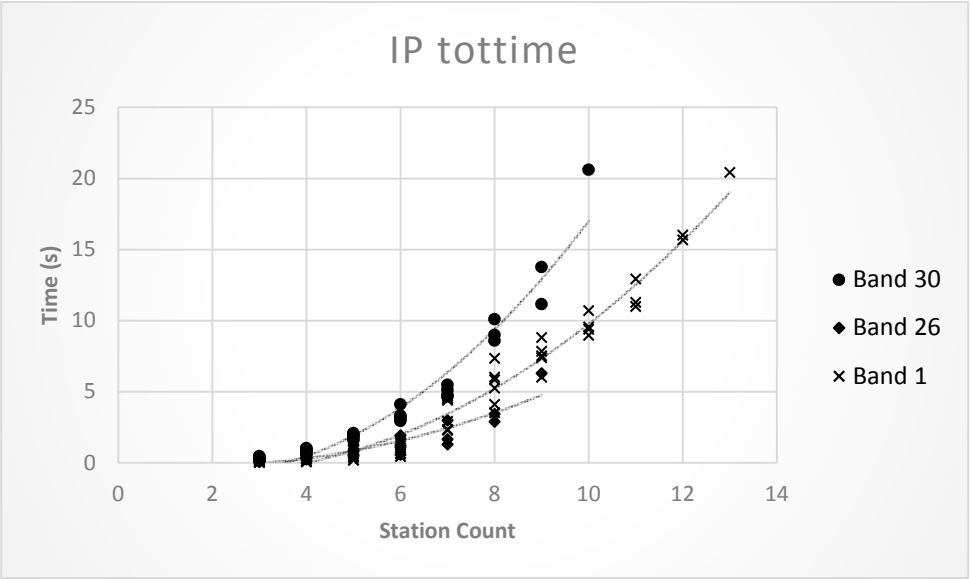


Figure 26. IP Runtime by Band and Station Count

### 5.2.3 H1 Runtime

The average time to execute the H1 heuristic was 0.341 seconds, aggregated across all 130 datasets. Figure 27-Figure 29 display H1 runtime presented against one factor each, using task, tool, and station count, respectively. Regression lines are plotted on each graph, using a degree-2 polynomial fit and an intercept of zero. Both task and station count display relatively good fit, with $R^2$ values of 0.779 and 0.724, respectively. Tooling count did not trend with H1 solution time, delivering a $R^2$ value of only 0.053. The heuristic has polynomial computational complexity, suggesting that runtime scales in a polynomial fashion with respect to problem size, as measured by task or station count. The regression fit lines might extrapolate well toward larger problems.

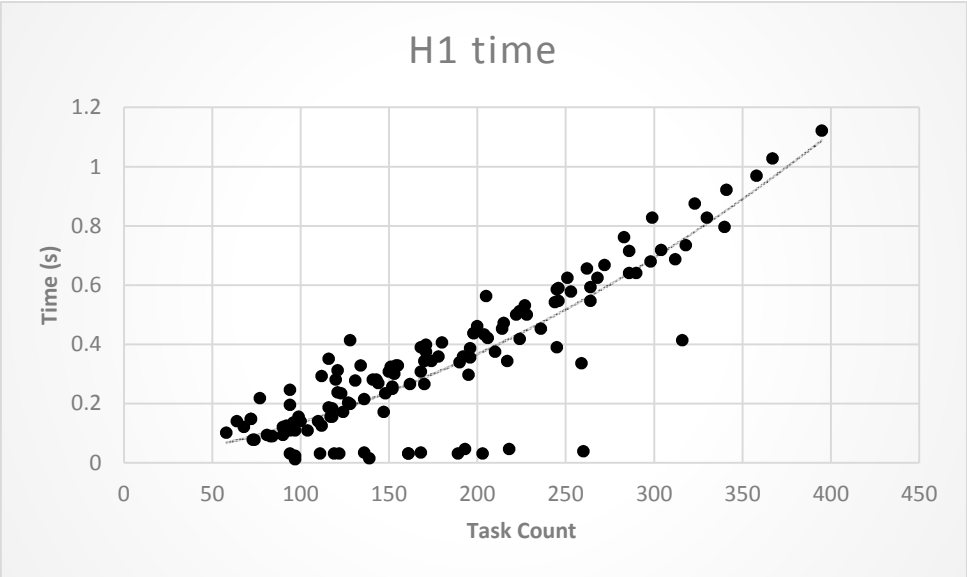


Figure 27. H1 Runtime vs. Number of Tasks

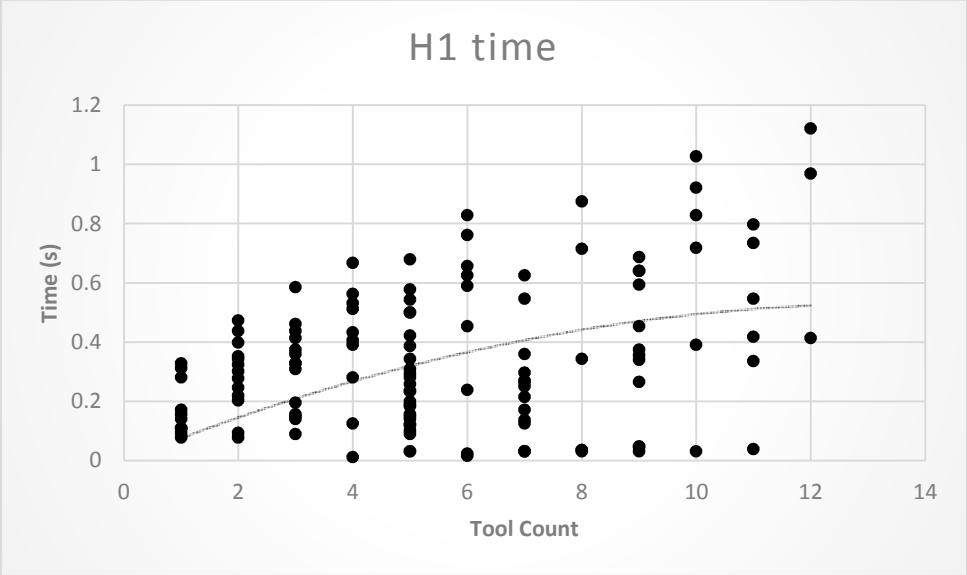


Figure 28. H1 Runtime vs. Number of Tools

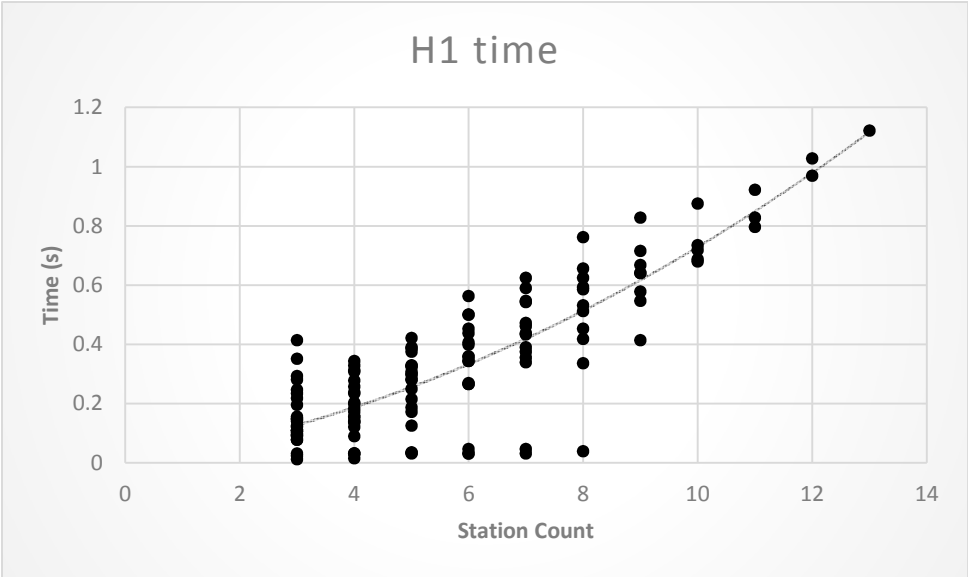


Figure 29. H1 Runtime vs. Number of Stations

Figure 27 displays visual evidence of two clusters within the data. The smaller finger of results, underneath the larger primary cluster of data, is showing the runtime of infeasible problem instances. The heuristic runs more quickly once it has become infeasible, as tasks begin to be assigned to dummy stations with all tools available, no accessibility constraints, etc, limiting the degree to which the algorithm must search for viable WZs. The low-runtime infeasible instances are also viewable in station and tool count plots. Figure 30-Figure 32 show these same plots with each band's data specified. Note that almost all of the infeasible instances belong to Band 26.

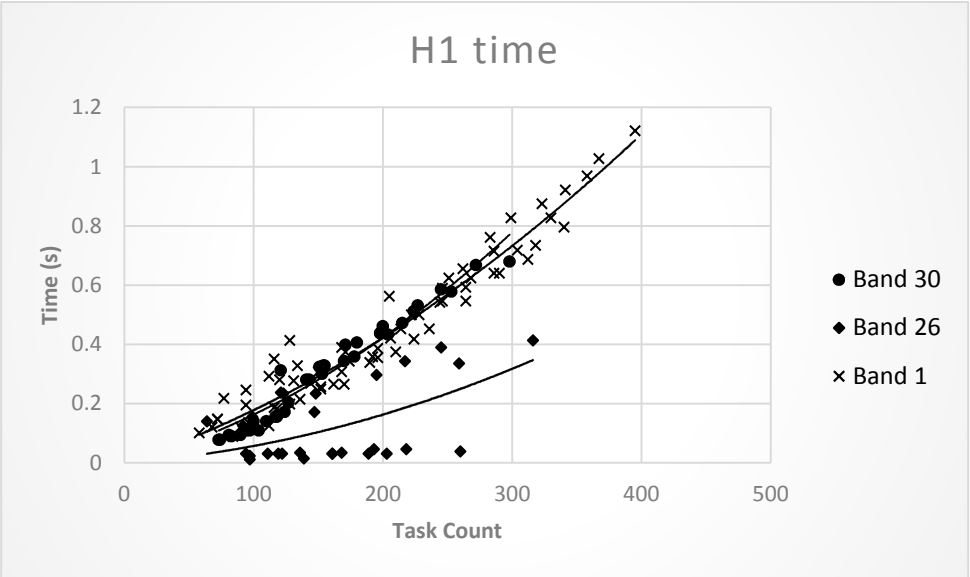


Figure 30. H1 Runtime by Band and Task Count

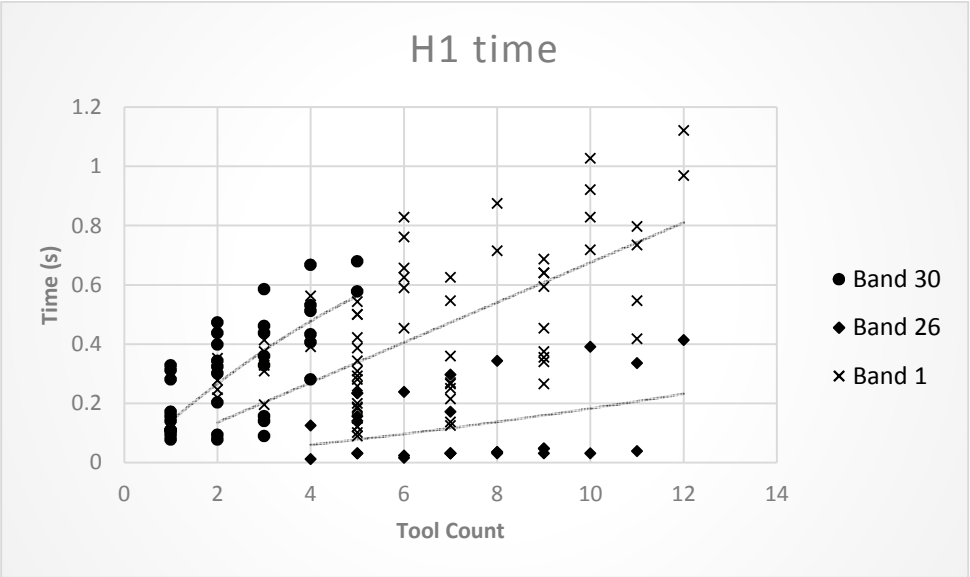


Figure 31. H1 Runtime by Band and Tool Count

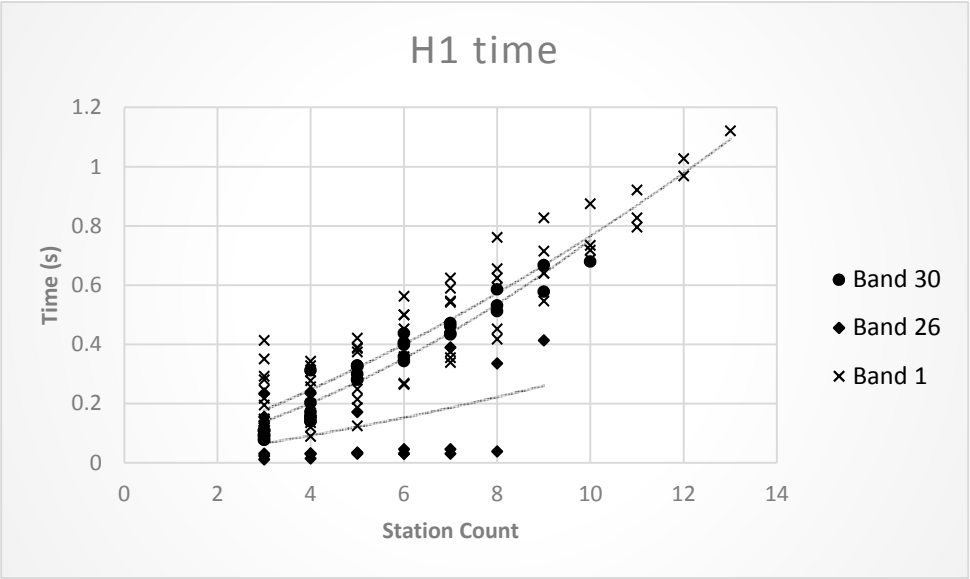


Figure 32. H1 Runtime by Band and Station Count

### 5.2.4 H2 Runtime

The average time to execute the H2 heuristic was 30.3 seconds, aggregated across all 130 datasets. This is two orders of magnitude higher than the H1 heuristic. One order of magnitude is explained by the nature of the H2 approach. 10 iterations were chosen for each H2 run, which requires the embedded H1 heuristic to be applied 10 times. The other order of magnitude difference in runtime can only be ascribed to the overhead of the H2 heuristic itself.

Figure 33-Figure 35 display H2 runtime presented against one factor each, using task, tool, and station count, respectively. Regression lines are plotted on each graph, using a degree-2 polynomial fit and an intercept of zero. None of these plots display particularly strong correlation between problem instances and runtime, with a maximum $R^2$ value of only 0.46. The fixed outer H2 iteration loop might explain the lesser correlation witnessed here. If the H2 overhead activities are relatively unresponsive to

problem size, then primarily only the size-dependent runtime within the embedded H1 runs are affecting the results seen here. The heuristic has polynomial computational complexity, suggesting that runtime scales in a polynomial fashion with respect to problem size, as measured by task or station count. The regression fit lines might extrapolate well toward larger problems, despite the relatively weaker fit. The runtime of infeasible instances is lower than feasible runs, as noted in section 5.2.3. These instances collect near the x-axis on each plot.

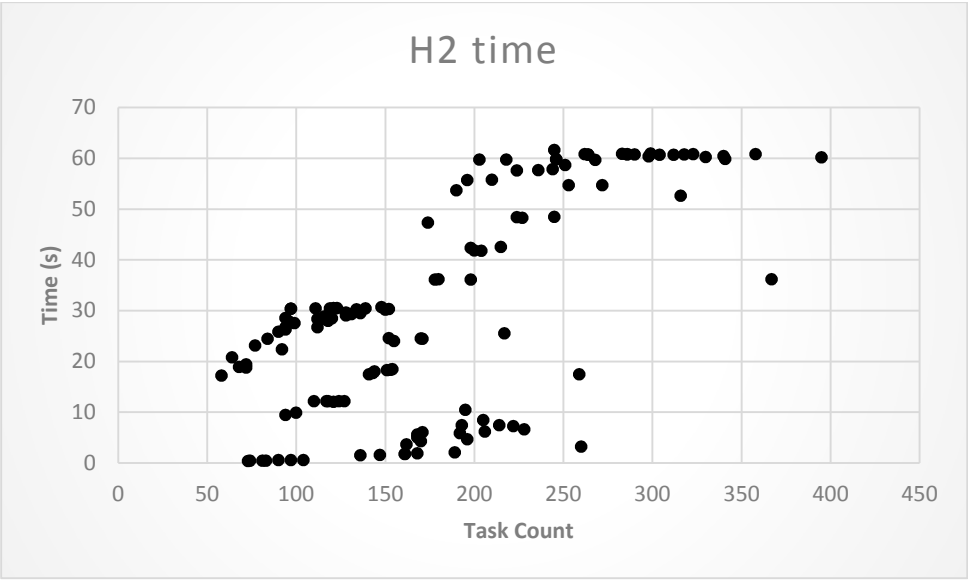


Figure 33. H2 Runtime vs. Number of Tasks

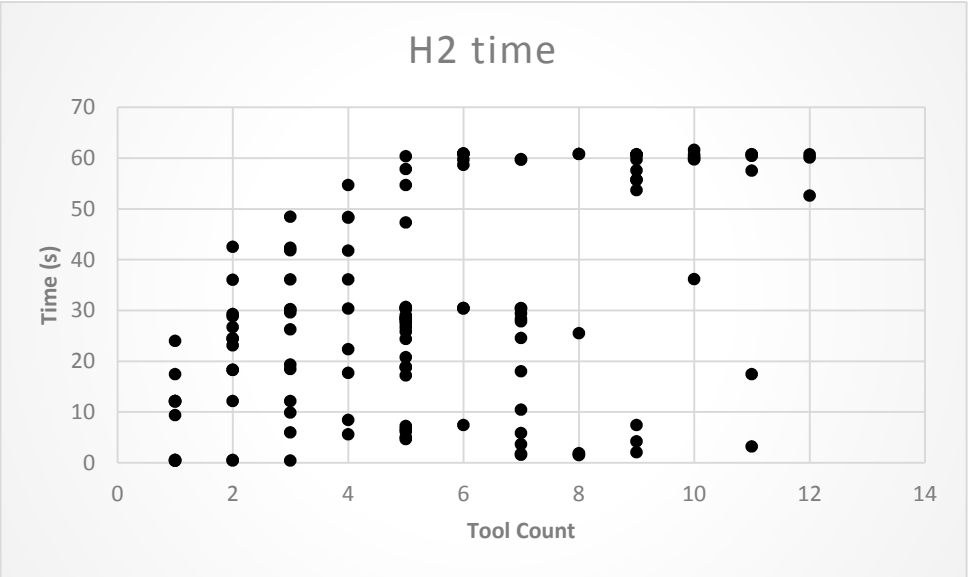


Figure 34. H2 Runtime vs. Number of Tools

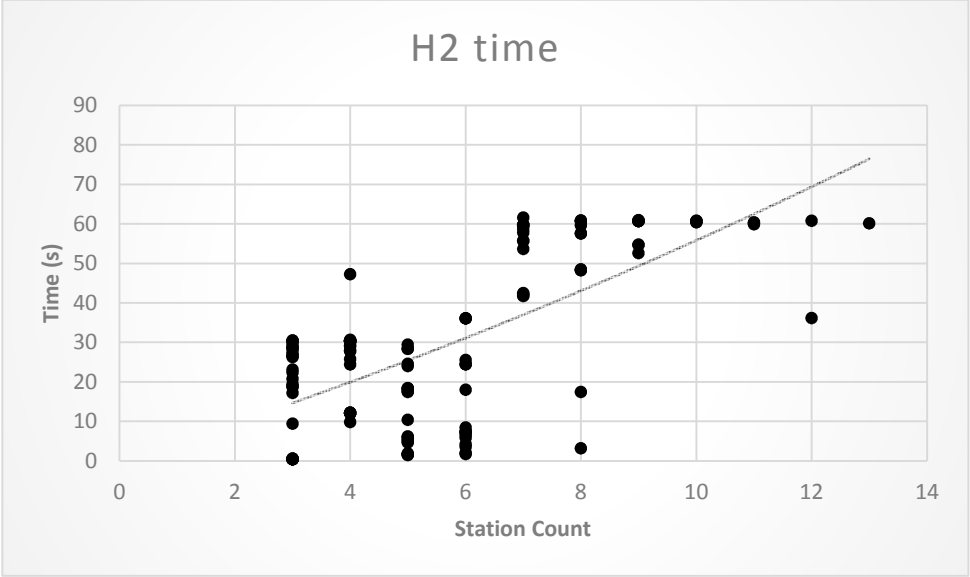


Figure 35. H2 Runtime vs Number of Stations

Figure 36Figure 38 display the same runtime plots, with each band's specific runtime results separated.

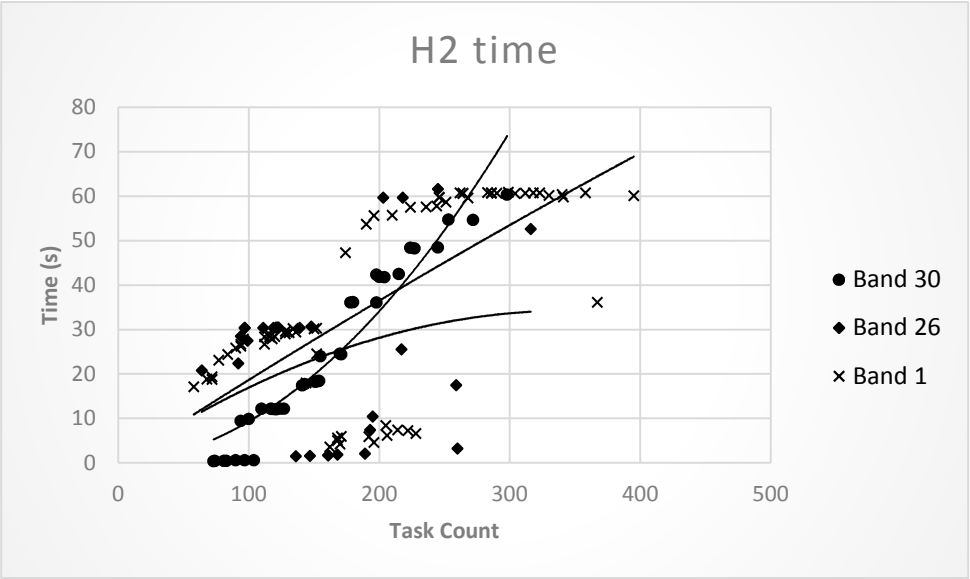


Figure 36. H2 Runtime by Band and Task Count

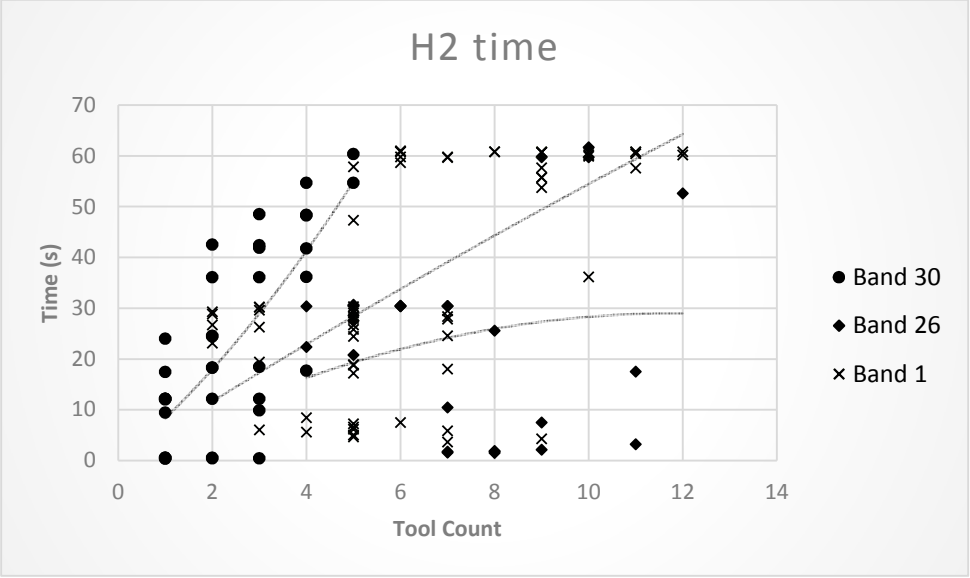


Figure 37. H2 Runtime by Band and Tool Count

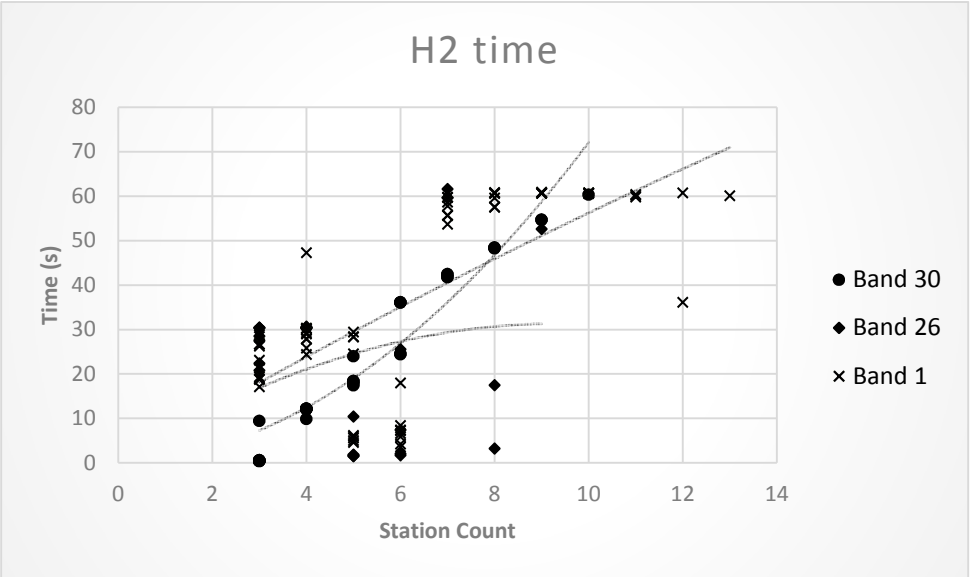


Figure 38. H2 Runtime by Band and Station Count

### 5.2.5 Heuristic Optimality Gap

The optimality gap for instance $i$ and heuristic $h$ is measured by $Gap_{h,i} = \frac{z_{h,i} - z_{opt,i}}{z_{opt,i}}$, where $z_{opt,i}$ is the optimal value of the objective function, as determined by the IP, and $z_{h,i}$ is the value of the objective function found by the heuristic. If the heuristic finds an optimal solution, then the gap is zero. Otherwise, the gap grows in inverse proportion to the quality of the objective found by the heuristic. Infeasible solutions are not included in this metric. The heuristics do find infeasible solutions for which an objective function is computed, but these solutions are discarded in this analysis. Figure 39-Figure 44 plot the gaps for each instance against the tool, station, and task counts of the instance, for both H1 and H2. All instances with a gap of zero, i.e. the heuristic found an optimal solution, are plotted on the x-axis. Many of these overlap on the plots. Overall, both heuristics found 31 optimal solutions from the 130 problem instances. Five

instances are known to be infeasible from the IP. The heuristics achieve optimality for approximately 25% of the instances for which feasible solutions exist.

The average gap for H1 is 0.22, or 22% higher objective function relative to optimal, and in the worst instance the H1 gap is 0.69. The average gap for H2 is 0.204, or 20.4% higher objective function relative to optimal, and in the worst instance is 0.62.

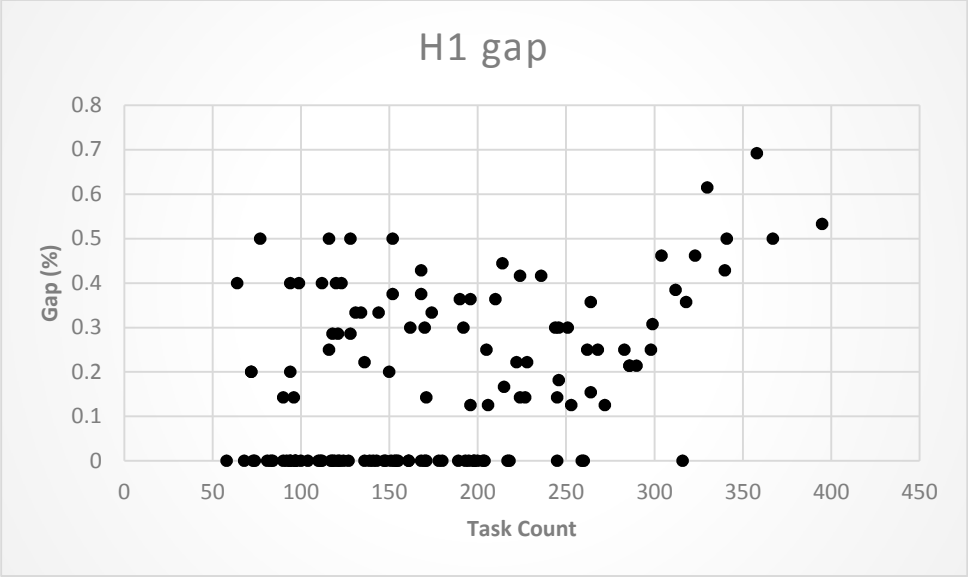


Figure 39. H1 Optimality Gap by Task Count

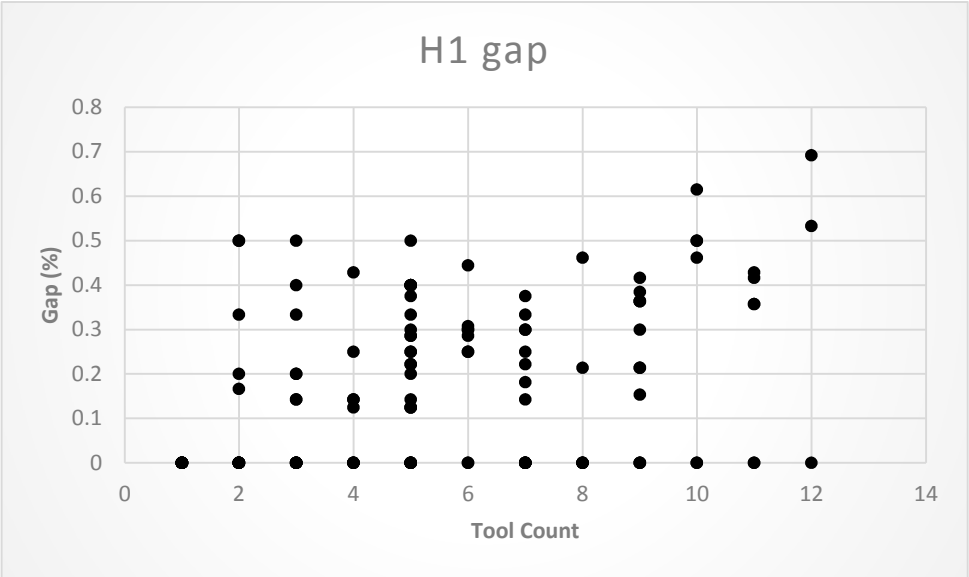


Figure 40. H1 Optimality Gap by Tool Count

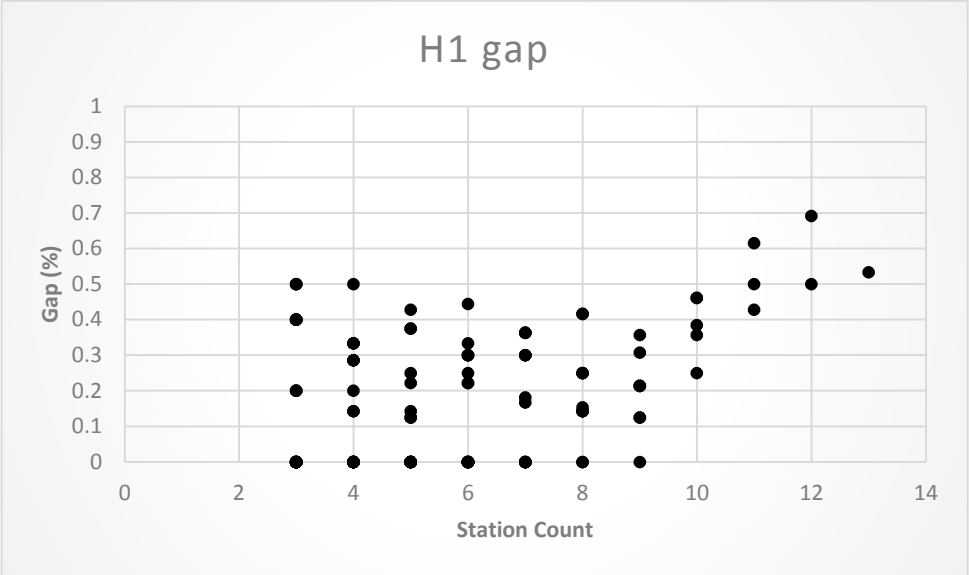


Figure 41. H1 Optimality Gap by Station Count

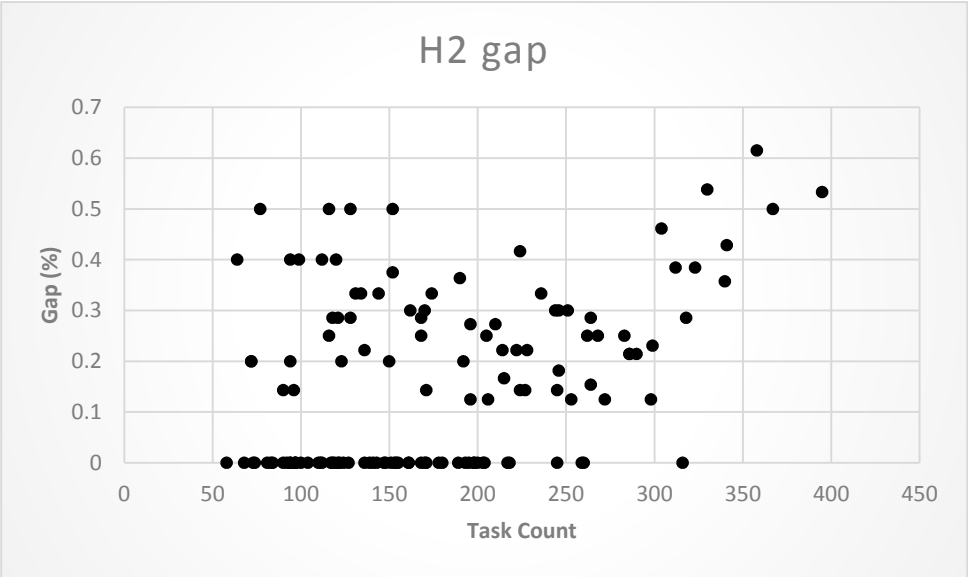


Figure 42. H2 Optimality Gap by Task Count

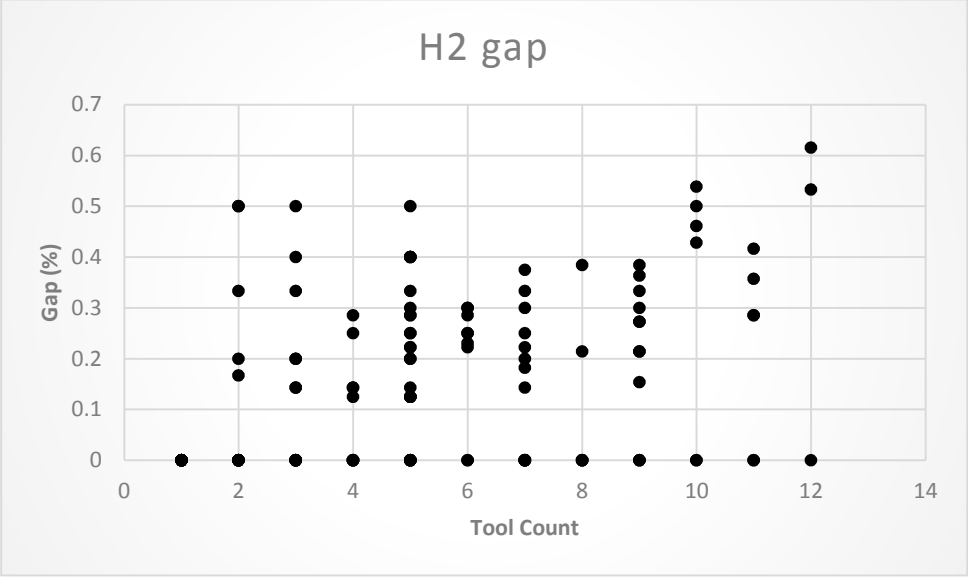


Figure 43. H2 Optimality Gap by Tool Count

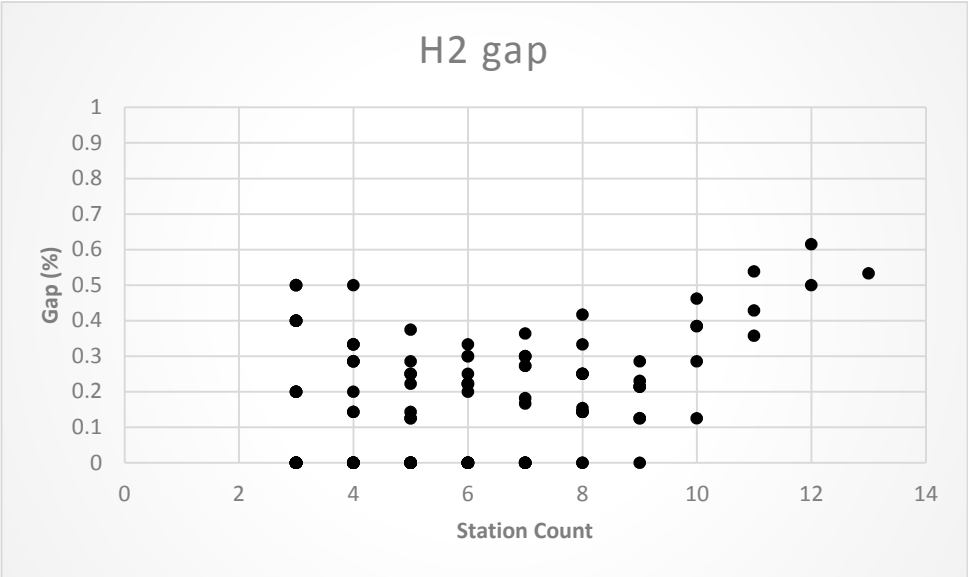


Figure 44. H2 Optimality Gap by Station Count

H2 improves the solution found by H1 in 17 instances. The average gap between between H1 and H2 solution pairs is 1.3%. Conditioning upon the 17 improved instances, the average gap grows to 8.2%, with a maximum gap of 18.2% for one instance on Band 1.

## 5.3 Discussion

### 5.3.1 Band Differentiation

The input source data draws from three complete datasets, called bands, and sub-divided into a multitude of smaller data sets to support this experiment. Each band represents an independent production process, with several key differences that might help illuminate band-specific differentiation in IP runtime.

Band 1 is a feeder line, with production buffers on either side of its product flow. The conveyor is disjoint, and representative of a pull process, allowing workers to pause the line in front of them if necessary, without disrupting the entire assembly line. All

tasks in Band 1 belong to one of the four corner PZs: {LV, RV, LH, RH}, and in most stations workers may only be assigned to the L or R work zones. There are only a few questions regarding mapping zones between tasks and workers, entirely encapsulated within the two stations that support three parallel workers and overlapping work zones. There are many fixed tools in Band 1, but for most tasks that require tools there is only one WZ which can satisfy both tooling and zoning needs, simplifying the decision problem by forcing task assignment.

Band 26 is relatively complex, with the full complement of up to 5 parallel workers permitted at many stations. Tasks are located in every PZ. Tools are common, though many are duplicated across two or more stations, permitting tool-needing tasks to be assigned in one of several WZs.

Band 30 is a single-sided assembly line, with only one worker permitted per station. Every task is located in the same PZ, and tooling is sparse on the line. Task grouping is relatively common in Band 30, but otherwise this band is easily the simplest of the three with respect to constraint complexity.

Lastly, the source data for each band was collected by a different individual. In light of the very different character of each band, and their associated ALB problem, it is perhaps not surprising that Figure 24 shows Band 26 instances requiring the most IP time to solve, Band 30 the least, and Band 1 in the middle. The additional complexity of Band 26 resulted in approximately 4x as much runtime relative to Band 30 for runs of equivalent task count, and approximately 2x the runtime relative to Band 1 instances. Switching from task count to station count produces roughly the same 4:2:1 breakdown

IP runtime between instances from each band, though this result might be expected due to the heavy correlation between task and station counts.

### 5.3.2 Performance

The heuristics failed to find feasible solutions for 16.8% of instances for which feasible solutions exist. The average heuristic optimality gaps are 0.22 and 0.204, and the maximum optimality gaps are 0.69 and 0.62, for H1 and H2 respectively. Runtime growth for H1 appears to be well-fit by a quadratic regression over problem size, as measured by either task or station count.

The IP successfully ran to completion for all problem instances, with the longest run taking just over 20 seconds to complete. There were concerns that the IP would exhibit slow runtime performance when initially preparing the IP model runs for submission to the Palmetto Cluster, due to prior runs of the IP's ancestors. The current version of the IP grew from these ancestor models by tightening the formulation across several aspects. When the final IP instances were submitted, each was budgeted for one hour of runtime. The actual runtime performance of the IP was spectacular, blowing away expectations.

The runtime plots of the IP suggest accelerating growth in runtime with respect to the size of the problem. Indeed, ALBs are NP-hard, and extremely large problems will certainly be intractable. Still, the instances solved here span up to 400 tasks and 13 stations, what might be considered mid-sized problems in ALB, with runtimes under one minute.

### 5.3.3 Extension and Adoption

Industry application of ALB methods commonly encounter difficulties in extending existing methods to account for gALB features specific to the problem environment. For the gALB environment considered by the methods here, no existing ALB methods were suitable for immediate application, largely due to the unique zoning features. During the course of this research, the heuristic methods were created first, and the IP formulation later. More than a year was spent working with our industry partner, both to collect data and to understand the various constraints that appear in the problem. Some constraint types, e.g. not-same-takt constraints, are especially rare on the assembly lines under study, and were not discovered until late in the process.

The ALB methods detailed in this work are certainly extensible for application to problem domains outside our industrial partner's, for which the methods were specifically designed. Issues related to industrial application of the IP and heuristic methods are discussed separately in sections 5.3.3.1 and 5.3.3.2, respectively.

#### 5.3.3.1 IP

The IP performed exceptionally well for all problem instances in the experiment. Assuming availability of a solver such as Gurobi, it is the recommended solution to any industry customer with an applicable ALB problem, assuming that their problem is comparably sized. It is difficult to speculate on IP runtime performance for problems larger than in the experiment, as runtime will certainly experience combinatorial growth rates at some size. Perhaps problems up to an order of magnitude of the largest instances in this experiment (similar in size to the largest ALB problems considered in any literature) would still find acceptable runtimes. Table 16 shows the worst-case IP

runtime from this experiment, scaled upward over several orders of magnitude. If, say, a one-order-of-magnitude increase in problem size resulted in a four-orders-of-magnitude increase in runtime, then this hypothetical problem might require something like 2.4 days to run. ALB problems are not typically run in time-sensitive environments, and two days might be a reasonable amount of time to wait.

| ORDERS LARGER | RUNTIME |
|---|---|
| **0** | 20.62 seconds |
| **1** | 3.44 minutes |
| **2** | 34.4 minutes |
| **3** | 5.7 hours |
| **4** | 2.4 days |

Table 16. Worst-Case IP Runtime, under Increasing Orders of Magnitude

The IP is particularly well-suited for constraint extensions that involve task-to-task or task-to-station assignment compulsion or forbiddance. Several constraints of this variety already exist within the current gALB problem, implemented with relatively clear, direct, and tight IP constraints. Presumably, extending the IP for another gALB environment by adding more constraints of this type would be relatively simple. Indeed, during development of this IP the not-same-takt constraints were added late in the timeline, but were easily modeled in the IP structure.

The IP features three distinct zone types: work zones, product zones, and tool coverage zones. The implementation details of these zones, such as their mapping relationships, are easily customizable (mapping is entirely dependent upon tunable parameters shown in section 4.2). It is possible to add, remove, or re-map any of the zoning features with reasonable effort. Such changes would require no alteration of the IP formulation itself, only redefinition of the preprocessing parameters, in which zone relationships are encapsulated.

An extension to manage a U-shaped assembly line appears to be near at hand. Such an extension would require zoning redefinition for the crossover takts, where workers access more than one station at a time, to ensure that zoning concepts are properly applied. Otherwise such an extension is fairly direct, as U-shaped lines may be considered as linear lines in which some takts share the same worker. Under this paradigm precedence constraints are unchanged, and only the cycle time constraint requires remodeling, so that crossover takt loading captures all tasks assigned to each worker, regardless of zone.

Implementing task sequencing constraints would require adding new decision variables to the IP to ensure that task start/stop times are properly managed. Adding these variables and associated sequencing constraints to the IP is relatively direct in terms of formulation, but might present significant consequences in terms of runtime. Adding decision variables might always be expected to add runtime, but in particular start/stop time variables are quantified over the real numbers. All other variables currently in the IP are binary, significantly restricting the size of the solution space. Timing variables changes the IP from a BIP to a MILP, and runtime penalties should be expected.

Problem extensions that permit the IP to touch on related production planning problems would necessitate large-scale adaptations to the IP. Examples include extensions to accommodate job sequencing, part logistics, or facility design. These issues are entirely out of the scope of the methods developed here.

#### 5.3.3.2 Heuristics

Relative to the IP, the heuristic methods are ill-suited for extensions that add constraints or other gALB features. The first ancestral version of the MRPW heuristic was developed early in the research project, before discovering many of the constraints now represented. Since that time, each constraint added has induced excessive difficulty when adapting the MRWP method. I would not recommend extension of the heuristic methods to any gALB problem with new features.

Further, the experiment has shown significant performance problems for the heuristics, both in terms of finding feasible solutions and in the quality of those solutions. There are only two scenarios in which I could recommend application of the heuristics instead of the IP. The first is the case of extraordinarily large problem size. The runtime of heuristic methods scales in a polynomial fashion with respect to problem size, and will experience a slower growth rate than the IP. At some threshold of problem size the IP will cease to be a reasonable option, due to inordinate runtime. The second scenario for application of the heuristic methods is if the resources for solving IPs are unavailable. The IP instances in this experiment were solved using the Gurobi solver, which is free for academic use but requires relatively expensive licensing costs for business use. Other solvers may of course be used instead, though to my knowledge there are no solvers that permit free business licensing at this time.

| ID | Band | #Stations | Batch# | IP obj | IP systime | IP tottime | IP status | H1 obj | H1 time | H1 status | H1 gap | H2 obj | H2 time | H2 status | H2 gap | H2 vs H1 | #Tools | #Tasks | Sum task time |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 | 30 | 3 | 1 | 3 | 0.054991 | 0.395002 | feas | 3 | 0.109375 | Feas | 0 | 3 | 9.425781 | feas | 0 | 0 | 1 | 94 | 913.62 |
| 2 | 30 | 3 | 2 | 3 | 0.008998 | 0.196195 | feas | 3 | 0.078125 | Feas | 0 | 3 | 0.375 | feas | 0 | 0 | 1 | 73 | 921.36 |
| 3 | 30 | 3 | 3 | 3 | 0.021997 | 0.266892 | feas | 3 | 0.09375 | feas | 0 | 3 | 0.4375 | feas | 0 | 0 | 1 | 81 | 896.64 |
| 4 | 30 | 3 | 4 | 3 | 0.038994 | 0.479717 | feas | 3 | 0.109375 | feas | 0 | 3 | 0.5625 | feas | 0 | 0 | 1 | 104 | 948.9 |
| 5 | 30 | 3 | 5 | 3 | 0.042994 | 0.420471 | feas | 3 | 0.109375 | feas | 0 | 3 | 0.578125 | feas | 0 | 0 | 1 | 97 | 953.58 |
| 6 | 30 | 3 | 6 | 3 | 0.032995 | 0.345457 | feas | 3 | 0.09375 | feas | 0 | 3 | 0.550781 | feas | 0 | 0 | 2 | 90 | 973.26 |
| 7 | 30 | 3 | 7 | 3 | 0.016997 | 0.209042 | feas | 3 | 0.078125 | feas | 0 | 3 | 0.453125 | feas | 0 | 0 | 2 | 74 | 995.58 |
| 8 | 30 | 3 | 8 | 3 | 0.021997 | 0.277579 | feas | 3 | 0.089843 | feas | 0 | 3 | 0.4375 | feas | 0 | 0 | 3 | 83 | 961.8 |
| 9 | 30 | 4 | 1 | 4 | 0.095985 | 0.941233 | feas | 4 | 0.15625 | feas | 0 | 4 | 12.14062 | feas | 0 | 0 | 1 | 118 | 1228.68 |
| 10 | 30 | 4 | 2 | 4 | 0.037994 | 0.724373 | feas | 4 | 0.140625 | feas | 0 | 4 | 12.15234 | feas | 0 | 0 | 1 | 110 | 1225.44 |
| 11 | 30 | 4 | 3 | 4 | 0.06799 | 0.98815 | feas | 4 | 0.171875 | feas | 0 | 4 | 12.17578 | feas | 0 | 0 | 1 | 124 | 1226.4 |
| 12 | 30 | 4 | 4 | 4 | 0.072989 | 0.924006 | feas | 4 | 0.3125 | feas | 0 | 4 | 12.04296 | feas | 0 | 0 | 1 | 121 | 1268.64 |
| 13 | 30 | 4 | 5 | 4 | 0.071989 | 1.04746 | feas | 4 | 0.203125 | feas | 0 | 4 | 12.16406 | feas | 0 | 0 | 2 | 127 | 1277.34 |
| 14 | 30 | 4 | 6 | 4 | 0.076989 | 0.86255 | feas | 4 | 0.15625 | feas | 0 | 4 | 12.17187 | feas | 0 | 0 | 3 | 117 | 1325.34 |
| 15 | 30 | 4 | 7 | 4 | 0.030996 | 0.601422 | feas | 4 | 0.140625 | feas | 0 | 4 | 9.894531 | feas | 0 | 0 | 3 | 100 | 1281.54 |
| 16 | 30 | 5 | 1 | 5 | 0.222966 | 2.09243 | feas | 5 | 0.328125 | feas | 0 | 5 | 24.02343 | feas | 0 | 0 | 1 | 155 | 1532.76 |
| 17 | 30 | 5 | 2 | 5 | 0.115983 | 1.96536 | feas | 5 | 0.300781 | feas | 0 | 5 | 18.35546 | feas | 0 | 0 | 2 | 153 | 1555.2 |
| 18 | 30 | 5 | 3 | 5 | 0.101985 | 1.63536 | feas | 5 | 0.28125 | feas | 0 | 5 | 17.4375 | feas | 0 | 0 | 1 | 141 | 1546.14 |
| 19 | 30 | 5 | 4 | 5 | 0.121982 | 1.88319 | feas | 5 | 0.324218 | feas | 0 | 5 | 18.26171 | feas | 0 | 0 | 2 | 151 | 1592.4 |
| 20 | 30 | 5 | 5 | 5 | 0.127981 | 2.01374 | feas | 5 | 0.328125 | feas | 0 | 5 | 18.46484 | feas | 0 | 0 | 3 | 154 | 1629.42 |
| 21 | 30 | 5 | 6 | 5 | 0.114982 | 1.71713 | feas | 5 | 0.28125 | feas | 0 | 5 | 17.70312 | feas | 0 | 0 | 4 | 143 | 1611.3 |
| 22 | 30 | 6 | 1 | 6 | 0.423935 | 4.12128 | feas | 6 | 0.4375 | feas | 0 | 6 | 36.08593 | feas | 0 | 0 | 2 | 198 | 1862.52 |
| 23 | 30 | 6 | 2 | 6 | 0.141979 | 2.95635 | feas | 6 | 0.34375 | feas | 0 | 6 | 24.54296 | feas | 0 | 0 | 2 | 170 | 1874.94 |
| 24 | 30 | 6 | 3 | 6 | 0.184972 | 3.09639 | feas | 6 | 0.398437 | feas | 0 | 6 | 24.46093 | feas | 0 | 0 | 2 | 171 | 1869.9 |
| 25 | 30 | 6 | 4 | 6 | 0.185971 | 3.23077 | feas | 6 | 0.359375 | feas | 0 | 6 | 36.12109 | feas | 0 | 0 | 3 | 178 | 1944.48 |
| 26 | 30 | 6 | 5 | 6 | 0.20097 | 3.32649 | feas | 6 | 0.40625 | feas | 0 | 6 | 36.14453 | feas | 0 | 0 | 4 | 180 | 1915.38 |
| 27 | 30 | 7 | 1 | 6 | 0.508922 | 5.4991 | feas | 7 | 0.472656 | feas | 0.166667 | 7 | 42.53515 | feas | 0.166667 | 0 | 2 | 215 | 2182.26 |
| 28 | 30 | 7 | 2 | 7 | 0.210967 | 4.77862 | feas | 7 | 0.460937 | feas | 0 | 7 | 41.87109 | feas | 0 | 0 | 3 | 200 | 2198.7 |
| 29 | 30 | 7 | 3 | 7 | 0.227966 | 4.67101 | feas | 7 | 0.4375 | feas | 0 | 7 | 42.37890 | feas | 0 | 0 | 3 | 198 | 2221.98 |
| 30 | 30 | 7 | 4 | 7 | 0.26396 | 5.13073 | feas | 7 | 0.433593 | feas | 0 | 7 | 41.80468 | feas | 0 | 0 | 4 | 204 | 2230.44 |
| 31 | 30 | 8 | 1 | 7 | 0.809876 | 10.12 | feas | 8 | 0.585937 | feas | 0.142857 | 8 | 48.49218 | feas | 0.142857 | 0 | 3 | 245 | 2506.02 |
| 32 | 30 | 8 | 2 | 7 | 0.369944 | 8.99572 | feas | 8 | 0.53125 | feas | 0.142857 | 8 | 48.29296 | feas | 0.142857 | 0 | 4 | 227 | 2550.78 |
| 33 | 30 | 8 | 3 | 7 | 0.448932 | 8.60774 | feas | 8 | 0.511718 | feas | 0.142857 | 8 | 48.41796 | feas | 0.142857 | 0 | 4 | 224 | 2507.94 |
| 34 | 30 | 9 | 1 | 8 | 1.14482 | 13.7687 | feas | 9 | 0.667968 | feas | 0.125 | 9 | 54.70703 | feas | 0.125 | 0 | 4 | 272 | 2858.1 |
| 35 | 30 | 9 | 2 | 8 | 0.536918 | 11.1672 | feas | 9 | 0.578125 | feas | 0.125 | 9 | 54.72656 | feas | 0.125 | 0 | 5 | 253 | 2836.74 |

Table 17. Experimental Results, Problems 1-35

| ID | Band | #Stations | Batch# | IP obj | IP systime | IP tottime | IP status | H1 obj | H1 time | H1 status | H1 gap | H2 obj | H2 time | H2 status | H2 gap | H2 vs H1 | #Tools | #Tasks | Sum task time |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 36 | 30 | 10 | 1 | 8 | 1.52377 | 20.6214 | feas | 10 | 0.679687 | feas | 0.25 | 9 | 60.39062 | feas | 0.125 | 0.111 | 5 | 298 | 3144.06 |
| 37 | 26 | 3 | 1 | 5 | 0.031995 | 0.21909 | feas | 7 | 0.15625 | feas | 0.4 | 7 | 27.53906 | feas | 0.4 | 0 | 5 | 99 | 838.32 |
| 38 | 26 | 3 | 2 | 5 | 0.005 | 0.075646 | feas | 7 | 0.140625 | feas | 0.4 | 7 | 20.79296 | feas | 0.4 | 0 | 5 | 64 | 606.24 |
| 39 | 26 | 3 | 3 | 5 | 0.030996 | 0.285508 | feas | 0 | 0.023437 | infeas |  | 0 | 30.38281 | infeas |  |  | 6 | 97 | 790.56 |
| 40 | 26 | 3 | 4 | 1 | 0.019997 | 0.171945 | feas | 0 | 0.03125 | infeas |  | 0 | 28.55078 | infeas |  |  | 5 | 94 | 817.98 |
| 41 | 26 | 3 | 5 | 0 | 0.002 | 0.119151 | infeas | 0 | 0.011718 | infeas |  | 0 | 30.39843 | infeas |  |  | 4 | 97 | 816.78 |
| 42 | 26 | 3 | 6 | 0 | 0.001 | 0.119685 | infeas | 8 | 0.125 | infeas |  | 8 | 22.38671 | infeas |  |  | 4 | 92 | 930.36 |
| 43 | 26 | 3 | 7 | 5 | 0.026996 | 0.350069 | feas | 7 | 0.234375 | feas | 0.4 | 6 | 30.48046 | feas | 0.2 | 0.167 | 5 | 123 | 1219.62 |
| 44 | 26 | 4 | 1 | 7 | 0.052991 | 0.487465 | feas | 9 | 0.238281 | feas | 0.285714 | 9 | 30.48046 | feas | 0.285714 | 0 | 6 | 121 | 1117.8 |
| 45 | 26 | 4 | 2 | 6 | 0.024996 | 0.392356 | feas | 0 | 0.03125 | infeas |  | 0 | 30.41796 | infeas |  |  | 7 | 111 | 904.74 |
| 46 | 26 | 4 | 3 | 2 | 0.027995 | 0.362166 | feas | 0 | 0.03125 | infeas |  | 0 | 30.45703 | infeas |  |  | 7 | 122 | 1030.56 |
| 47 | 26 | 4 | 4 | 0 | 0.006999 | 0.253532 | infeas | 0 | 0.03125 | infeas |  | 0 | 30.43359 | infeas |  |  | 5 | 119 | 1096.26 |
| 48 | 26 | 4 | 5 | 0 | 0.012998 | 0.387524 | infeas | 0 | 0.015625 | infeas |  | 0 | 30.43359 | infeas |  |  | 6 | 139 | 1228.86 |
| 49 | 26 | 4 | 6 | 6 | 0.041994 | 0.590973 | feas | 10 | 0.234375 | infeas |  | 10 | 30.69140 | infeas |  |  | 5 | 148 | 1459.62 |
| 50 | 26 | 5 | 1 | 8 | 0.121981 | 1.2393 | feas | 0 | 0.035156 | infeas |  | 0 | 1.902343 | infeas |  |  | 8 | 168 | 1416.3 |
| 51 | 26 | 5 | 2 | 2 | 0.010998 | 0.503037 | feas | 0 | 0.035156 | infeas |  | 0 | 1.511718 | infeas |  |  | 8 | 136 | 1144.74 |
| 52 | 26 | 5 | 3 | 1 | 0.008998 | 0.577184 | feas | 0 | 0.171875 | infeas |  | 0 | 1.5625 | infeas |  |  | 7 | 147 | 1308.84 |
| 53 | 26 | 5 | 4 | 0 | 0.020997 | 0.804048 | infeas | 0 | 0.03125 | infeas |  | 0 | 1.765625 | infeas |  |  | 7 | 161 | 1508.34 |
| 54 | 26 | 5 | 5 | 7 | 0.073989 | 1.52785 | feas | 11 | 0.296875 | infeas |  | 11 | 10.44140 | infeas |  |  | 7 | 195 | 1758.12 |
| 55 | 26 | 6 | 1 | 4 | 0.132979 | 1.24726 | feas | 0 | 0.046875 | infeas |  | 0 | 7.460937 | infeas |  |  | 9 | 193 | 1656.3 |
| 56 | 26 | 6 | 2 | 1 | 0 | 0.582737 | feas | 0 | 0.03125 | infeas |  | 0 | 1.734375 | infeas |  |  | 8 | 161 | 1423.02 |
| 57 | 26 | 6 | 3 | 1 | 0.047993 | 1.13256 | feas | 0 | 0.03125 | infeas |  | 0 | 2.105468 | infeas |  |  | 9 | 189 | 1720.92 |
| 58 | 26 | 6 | 4 | 8 | 0.094986 | 1.93804 | feas | 13 | 0.34375 | infeas |  | 13 | 25.55078 | infeas |  |  | 8 | 217 | 2037.6 |
| 59 | 26 | 7 | 1 | 3 | 0.181972 | 1.66951 | feas | 0 | 0.046875 | infeas |  | 0 | 59.72265 | infeas |  |  | 9 | 218 | 1934.58 |
| 60 | 26 | 7 | 2 | 1 | 0.035994 | 1.31454 | feas | 0 | 0.03125 | infeas |  | 0 | 59.72656 | infeas |  |  | 10 | 203 | 1835.1 |
| 61 | 26 | 7 | 3 | 10 | 0.181973 | 2.98906 | feas | 15 | 0.390625 | infeas |  | 15 | 61.66406 | infeas |  |  | 10 | 245 | 2250.18 |
| 62 | 26 | 8 | 1 | 3 | 0.287956 | 2.91045 | feas | 0 | 0.039062 | infeas |  | 0 | 3.234375 | infeas |  |  | 11 | 260 | 2346.66 |
| 63 | 26 | 8 | 2 | 11 | 0.177973 | 3.44107 | feas | 19 | 0.335937 | infeas |  | 19 | 17.48437 | infeas |  |  | 11 | 259 | 2364.36 |
| 64 | 26 | 9 | 1 | 13 | 0.576912 | 6.2955 | feas | 21 | 0.414062 | infeas |  | 21 | 52.63281 | infeas |  |  | 12 | 316 | 2875.92 |
| 65 | 1 | 3 | 1 | 5 | 0.016997 | 0.071281 | feas | 6 | 0.148437 | feas | 0.2 | 6 | 18.78906 | feas | 0.2 | 0 | 5 | 72 | 523.02 |
| 66 | 1 | 3 | 2 | 6 | 0.013997 | 0.058251 | feas | 6 | 0.121093 | feas | 0 | 6 | 18.90625 | feas | 0 | 0 | 5 | 68 | 572.28 |
| 67 | 1 | 3 | 3 | 6 | 0.006999 | 0.044115 | feas | 6 | 0.101562 | feas | 0 | 6 | 17.1875 | feas | 0 | 0 | 5 | 58 | 585.84 |
| 68 | 1 | 3 | 4 | 5 | 0.013998 | 0.075138 | feas | 6 | 0.148437 | feas | 0.2 | 6 | 19.38281 | feas | 0.2 | 0 | 3 | 72 | 550.38 |
| 69 | 1 | 3 | 5 | 5 | 0.030996 | 0.17356 | feas | 7 | 0.195312 | feas | 0.4 | 7 | 26.27734 | feas | 0.4 | 0 | 3 | 94 | 923.22 |
| 70 | 1 | 3 | 6 | 5 | 0.044994 | 0.22166 | feas | 7 | 0.292968 | feas | 0.4 | 7 | 26.71875 | feas | 0.4 | 0 | 5 | 112 | 1027.92 |

Table 18. Experimental Results, Problems 36-70

| ID | Band | #Stations | Batch# | IP obj | IP systime | IP tottime | IP status | H1 obj | H1 time | H1 status | H1 gap | H2 obj | H2 time | H2 status | H2 gap | H2 vs H1 | #Tools | #Tasks | Sum task time |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 71 | 1 | 3 | 7 | 5 | 0.031996 | 0.252868 | feas | 7 | 0.28125 | feas | 0.4 | 7 | 28.48828 | feas | 0.4 | 0 | 5 | 120 | 1023.48 |
| 72 | 1 | 3 | 8 | 4 | 0.042994 | 0.283847 | feas | 6 | 0.414062 | feas | 0.5 | 6 | 29.60156 | feas | 0.5 | 0 | 3 | 128 | 780.42 |
| 73 | 1 | 3 | 9 | 4 | 0.033995 | 0.278307 | feas | 6 | 0.351562 | feas | 0.5 | 6 | 28.86328 | feas | 0.5 | 0 | 2 | 116 | 725.1 |
| 74 | 1 | 3 | 10 | 5 | 0.020997 | 0.150993 | feas | 6 | 0.246093 | feas | 0.2 | 6 | 26.70312 | feas | 0.2 | 0 | 2 | 94 | 601.5 |
| 75 | 1 | 3 | 11 | 4 | 0.011998 | 0.099844 | feas | 6 | 0.21875 | feas | 0.5 | 6 | 23.12890 | feas | 0.5 | 0 | 2 | 77 | 661.68 |
| 76 | 1 | 4 | 1 | 7 | 0.020996 | 0.118484 | feas | 8 | 0.136718 | feas | 0.142857 | 8 | 27.86718 | feas | 0.142857 | 0 | 7 | 96 | 743.82 |
| 77 | 1 | 4 | 2 | 7 | 0.007999 | 0.090559 | feas | 9 | 0.089843 | infeas |  | 9 | 24.42968 | infeas |  |  | 5 | 84 | 743.76 |
| 78 | 1 | 4 | 3 | 7 | 0.009998 | 0.104315 | feas | 8 | 0.121093 | feas | 0.142857 | 8 | 25.85156 | feas | 0.142857 | 0 | 5 | 90 | 894.36 |
| 79 | 1 | 4 | 4 | 7 | 0.028996 | 0.261355 | feas | 9 | 0.183593 | feas | 0.285714 | 9 | 27.96093 | feas | 0.285714 | 0 | 5 | 118 | 1144.02 |
| 80 | 1 | 4 | 5 | 7 | 0.037994 | 0.280067 | feas | 9 | 0.199218 | feas | 0.285714 | 9 | 29.05859 | feas | 0.285714 | 0 | 5 | 128 | 1199.4 |
| 81 | 1 | 4 | 6 | 6 | 0.043993 | 0.479886 | feas | 9 | 0.257812 | feas | 0.5 | 9 | 30.31640 | feas | 0.5 | 0 | 5 | 152 | 1332 |
| 82 | 1 | 4 | 7 | 6 | 0.052992 | 0.584964 | feas | 8 | 0.34375 | feas | 0.333333 | 8 | 47.30859 | feas | 0.333333 | 0 | 5 | 174 | 1223.64 |
| 83 | 1 | 4 | 8 | 5 | 0.037994 | 0.42274 | feas | 6 | 0.308593 | feas | 0.2 | 6 | 30.17578 | feas | 0.2 | 0 | 3 | 150 | 1001.28 |
| 84 | 1 | 4 | 9 | 6 | 0.037994 | 0.328472 | feas | 8 | 0.328125 | feas | 0.333333 | 8 | 30.23437 | feas | 0.333333 | 0 | 3 | 134 | 905.58 |
| 85 | 1 | 4 | 10 | 6 | 0.020997 | 0.292463 | feas | 8 | 0.277343 | feas | 0.333333 | 8 | 29.3125 | feas | 0.333333 | 0 | 2 | 131 | 861.84 |
| 86 | 1 | 5 | 1 | 8 | 0.046992 | 0.213192 | feas | 11 | 0.125 | infeas |  | 11 | 28.37109 | infeas |  |  | 7 | 112 | 915.3 |
| 87 | 1 | 5 | 2 | 8 | 0.025996 | 0.214422 | feas | 10 | 0.1875 | feas | 0.25 | 10 | 28.375 | feas | 0.25 | 0 | 5 | 116 | 1052.28 |
| 88 | 1 | 5 | 3 | 9 | 0.044993 | 0.376338 | feas | 11 | 0.214843 | feas | 0.222222 | 11 | 29.46484 | feas | 0.222222 | 0 | 7 | 136 | 1337.58 |
| 89 | 1 | 5 | 4 | 8 | 0.058991 | 0.472942 | feas | 11 | 0.25 | feas | 0.375 | 11 | 24.58984 | feas | 0.375 | 0 | 7 | 152 | 1420.2 |
| 90 | 1 | 5 | 5 | 8 | 0.050992 | 0.65138 | feas | 11 | 0.308593 | feas | 0.375 | 10 | 5.054687 | feas | 0.25 | 0.1 | 5 | 168 | 1503.48 |
| 91 | 1 | 5 | 6 | 8 | 0.102984 | 1.10546 | feas | 9 | 0.421875 | feas | 0.125 | 9 | 6.199218 | feas | 0.125 | 0 | 5 | 206 | 1503.48 |
| 92 | 1 | 5 | 7 | 8 | 0.105984 | 1.01665 | feas | 9 | 0.386718 | feas | 0.125 | 9 | 4.648437 | feas | 0.125 | 0 | 5 | 196 | 1444.5 |
| 93 | 1 | 5 | 8 | 7 | 0.044993 | 0.615671 | feas | 10 | 0.390625 | feas | 0.428571 | 9 | 5.597656 | feas | 0.285714 | 0.111111 | 4 | 168 | 1181.76 |
| 94 | 1 | 5 | 9 | 7 | 0.037994 | 0.722967 | feas | 8 | 0.375 | feas | 0.142857 | 8 | 6.019531 | feas | 0.142857 | 0 | 3 | 171 | 1181.76 |
| 95 | 1 | 6 | 1 | 9 | 0.080987 | 0.462828 | feas | 12 | 0.269531 | feas | 0.333333 | 12 | 18.03515 | feas | 0.333333 | 0 | 7 | 144 | 1223.82 |
| 96 | 1 | 6 | 2 | 10 | 0.053991 | 0.633766 | feas | 13 | 0.265625 | feas | 0.3 | 13 | 3.636718 | feas | 0.3 | 0 | 7 | 162 | 1495.5 |
| 97 | 1 | 6 | 3 | 10 | 0.071989 | 0.673208 | feas | 13 | 0.265625 | feas | 0.3 | 13 | 4.257812 | feas | 0.3 | 0 | 9 | 170 | 1613.76 |
| 98 | 1 | 6 | 4 | 10 | 0.088986 | 1.03627 | feas | 13 | 0.359375 | feas | 0.3 | 12 | 5.882812 | feas | 0.2 | 0.083333 | 7 | 192 | 1724.28 |
| 99 | 1 | 6 | 5 | 9 | 0.143978 | 1.45522 | feas | 11 | 0.5 | feas | 0.222222 | 11 | 7.253906 | feas | 0.222222 | 0 | 5 | 222 | 1703.64 |
| 100 | 1 | 6 | 6 | 9 | 0.157976 | 1.66996 | feas | 11 | 0.5 | feas | 0.222222 | 11 | 6.632812 | feas | 0.222222 | 0 | 5 | 228 | 1753.02 |
| 101 | 1 | 6 | 7 | 9 | 0.12698 | 1.38232 | feas | 13 | 0.453125 | feas | 0.444444 | 11 | 7.460937 | feas | 0.222222 | 0.181818 | 6 | 214 | 1624.98 |
| 102 | 1 | 6 | 8 | 8 | 0.068989 | 1.21965 | feas | 10 | 0.5625 | feas | 0.25 | 10 | 8.457031 | feas | 0.25 | 0 | 4 | 205 | 1442.1 |
| 103 | 1 | 7 | 1 | 11 | 0.253961 | 2.36609 | feas | 15 | 0.339843 | feas | 0.363636 | 15 | 53.72265 | feas | 0.363636 | 0 | 9 | 190 | 1667.04 |
| 104 | 1 | 7 | 2 | 11 | 0.152976 | 2.30948 | feas | 15 | 0.355468 | feas | 0.363636 | 14 | 55.69531 | feas | 0.272727 | 0.071428 | 9 | 196 | 1771.68 |
| 105 | 1 | 7 | 3 | 11 | 0.245963 | 2.93077 | feas | 15 | 0.375 | feas | 0.363636 | 14 | 55.78515 | feas | 0.272727 | 0.071428 | 9 | 210 | 1917.84 |

Table 19. Experimental Results, Problems 71-105

| ID | Band | #Stations | Batch# | IP obj | IP systime | IP tottime | IP status | H1 obj | H1 time | H1 status | H1 gap | H2 obj | H2 time | H2 status | H2 gap | H2 vs H1 | #Tools | #Tasks | Sum task time |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 106 | 1 | 7 | 4 | 11 | 0.364944 | 4.41626 | feas | 13 | 0.546875 | feas | 0.181818 | 13 | 59.82031 | feas | 0.181818 | 0 | 7 | 246 | 1924.44 |
| 107 | 1 | 7 | 5 | 10 | 0.424936 | 4.56467 | feas | 13 | 0.542968 | feas | 0.3 | 13 | 57.84375 | feas | 0.3 | 0 | 5 | 244 | 1924.5 |
| 108 | 1 | 7 | 6 | 10 | 0.299955 | 4.48261 | feas | 13 | 0.589843 | feas | 0.3 | 13 | 59.77734 | feas | 0.3 | 0 | 6 | 246 | 1933.5 |
| 109 | 1 | 7 | 7 | 10 | 0.306953 | 4.68246 | feas | 13 | 0.625 | feas | 0.3 | 13 | 58.67578 | feas | 0.3 | 0 | 6 | 251 | 1885.32 |
| 110 | 1 | 8 | 1 | 12 | 0.360945 | 3.54474 | feas | 17 | 0.417968 | feas | 0.416667 | 17 | 57.57812 | feas | 0.416667 | 0 | 11 | 224 | 1943.22 |
| 111 | 1 | 8 | 2 | 12 | 0.305954 | 4.11991 | feas | 17 | 0.453125 | feas | 0.416667 | 16 | 57.64062 | feas | 0.333333 | 0.0625 | 9 | 236 | 2075.76 |
| 112 | 1 | 8 | 3 | 13 | 0.430935 | 5.27347 | feas | 15 | 0.59375 | feas | 0.153846 | 15 | 60.67968 | feas | 0.153846 | 0 | 9 | 264 | 2118 |
| 113 | 1 | 8 | 4 | 12 | 0.510923 | 6.02894 | feas | 15 | 0.625 | feas | 0.25 | 15 | 59.66796 | feas | 0.25 | 0 | 7 | 268 | 2145.3 |
| 114 | 1 | 8 | 5 | 12 | 0.346947 | 5.8574 | feas | 15 | 0.65625 | feas | 0.25 | 15 | 60.83203 | feas | 0.25 | 0 | 6 | 262 | 2104.98 |
| 115 | 1 | 8 | 6 | 12 | 0.503923 | 7.36936 | feas | 15 | 0.761718 | feas | 0.25 | 15 | 60.87890 | feas | 0.25 | 0 | 6 | 283 | 2193.84 |
| 116 | 1 | 9 | 1 | 14 | 0.620905 | 6.03563 | feas | 19 | 0.546875 | feas | 0.357143 | 18 | 60.76171 | feas | 0.285714 | 0.055555 | 11 | 264 | 2247.3 |
| 117 | 1 | 9 | 2 | 14 | 0.513922 | 7.41904 | feas | 17 | 0.640625 | feas | 0.214286 | 17 | 60.75781 | feas | 0.214286 | 0 | 9 | 290 | 2275.92 |
| 118 | 1 | 9 | 3 | 14 | 0.563914 | 7.54926 | feas | 17 | 0.640625 | feas | 0.214286 | 17 | 60.73046 | feas | 0.214286 | 0 | 9 | 286 | 2338.86 |
| 119 | 1 | 9 | 4 | 14 | 0.608907 | 7.85905 | feas | 17 | 0.714843 | feas | 0.214286 | 17 | 60.80468 | feas | 0.214286 | 0 | 8 | 286 | 2325.78 |
| 120 | 1 | 9 | 5 | 13 | 0.762884 | 8.81781 | feas | 17 | 0.828125 | feas | 0.307692 | 16 | 60.97265 | feas | 0.230769 | 0.0625 | 6 | 299 | 2365.32 |
| 121 | 1 | 10 | 1 | 14 | 0.91486 | 9.55898 | feas | 19 | 0.734375 | feas | 0.357143 | 18 | 60.77734 | feas | 0.285714 | 0.055555 | 11 | 318 | 2447.46 |
| 122 | 1 | 10 | 2 | 13 | 0.639903 | 9.41312 | feas | 18 | 0.6875 | feas | 0.384615 | 18 | 60.71484 | feas | 0.384615 | 0 | 9 | 312 | 2496.78 |
| 123 | 1 | 10 | 3 | 13 | 0.580912 | 8.99326 | feas | 19 | 0.71875 | feas | 0.461538 | 19 | 60.6875 | feas | 0.461538 | 0 | 10 | 304 | 2519.34 |
| 124 | 1 | 10 | 4 | 13 | 0.800878 | 10.7113 | feas | 19 | 0.875 | feas | 0.461538 | 18 | 60.80078 | feas | 0.384615 | 0.055555 | 8 | 323 | 2586.12 |
| 125 | 1 | 11 | 1 | 14 | 1.22981 | 12.9452 | feas | 20 | 0.796875 | feas | 0.428571 | 19 | 60.42578 | feas | 0.357143 | 0.052631 | 11 | 340 | 2668.32 |
| 126 | 1 | 11 | 2 | 13 | 0.701894 | 11.3021 | feas | 21 | 0.828125 | feas | 0.615385 | 20 | 60.25781 | feas | 0.538462 | 0.05 | 10 | 330 | 2677.26 |
| 127 | 1 | 11 | 3 | 14 | 0.653901 | 10.9986 | feas | 21 | 0.921875 | feas | 0.5 | 20 | 59.89453 | feas | 0.428571 | 0.05 | 10 | 341 | 2779.68 |
| 128 | 1 | 12 | 1 | 13 | 1.45578 | 15.682 | feas | 22 | 0.96875 | feas | 0.692308 | 21 | 60.79296 | feas | 0.615385 | 0.047619 | 12 | 358 | 2848.8 |
| 129 | 1 | 12 | 2 | 14 | 1.12983 | 16.0204 | feas | 21 | 1.027343 | feas | 0.5 | 21 | 36.1875 | feas | 0.5 | 0 | 10 | 367 | 2937.6 |
| 130 | 1 | 13 | 1 | 15 | 1.90571 | 20.4352 | feas | 23 | 1.121093 | feas | 0.533333 | 23 | 60.15625 | feas | 0.533333 | 0 | 12 | 395 | 3109.14 |

Table 20. Experimental Results, Problems 106-130

# CHAPTER SIX

# 6 HORIZONTAL BALANCE METRIC FOR THE OPTIONS-MIX PARADIGM

Mass customization systems manufacture variations of a common base product that differ according to a set of customizable options (Pine), and have been subject to a massive amount of production planning in recent decades. The *model-mix* paradigm is nearly universal within this literature, in which a *model* consists of all output products with identical customization attributes. Each unique model may be considered as a batch of identical product with individual and independent production volume, resource usage, and other problem variables. Models are largely independent from one another, as there is little interaction between models in many production environments. Inter-model setup times are the notable exception, and the lot-sizing and scheduling modeling methods in such environments are typically very focused on this interaction.

As the number of configurable options increases, the size of the model set grows at a combinatorial rate in response resulting to several difficulties when applying the model-mix paradigm in environments with a large number of options. For example, (Meyr) discusses a modern automobile assembly line that features on the order of $10^{32}$ unique models. At this scale computational methods that iterate over the model set will be faced with exceptional memory and processing time requirements. Further, it may prove impossible to collect the necessary input data for each model. For example, demand for each model is very difficult to estimate when the model mix is many orders of magnitude larger than production volume.

The *option-mix* paradigm, as discussed by (Roder and Tibken) offers an alternative information model in which individual variables are assigned for each option rather than for each model. Reliable estimates of option-mix frequencies remain feasible even with high product variety, e.g. the fraction of cars with optional heads-up display systems. The primary difficulty with the options-mix paradigm is that options are not direct abstractions of production units, as models are. Each production unit may possess zero, one, or many options, according to its configuration. Moreover, options may exhibit strong interaction with each other, in contrast with the relatively weak interaction between models noted above. Consider the case of stereo speakers in a vehicle: while basic, premium, and perhaps several other types of speaker options may be available, only one of these options may be installed in any given vehicle. Such interaction information is necessary to options-mix methods, and herein we will assume that a rules database exists that documents these interactions. Such databases usually are maintained by product design or marketing departments within the organization, and facilitate the translation of options into feasible product configurations.

Note that while the options-mix paradigm can be applied to any problem in the model-mix domain, each problem presents unique challenges. Herein attention is devoted to an assembly line balancing (ALB) problem in which assembly tasks may be associated with a large number of optional parts. In section 6.1 the ALB problem is introduced, along with the horizontal balancing problem that motivates the options-mix problem. In section 6.3 the data environment that comprises the options-mix information model for this problem is presented. Section 6.4 describes the application of instances of

the Boolean satisfiability problem to evaluate a metric for the horizontal line balancing problem in this environment.

## 6.1 Assembly Line Balancing

The traditional form of an assembly line, as described by (Scholl), is a production system consisting of a configuration of consecutive workstations, typically using a conveyor or similar to transport production units down the line. The total work to be performed along the assembly line is subdivided into the smallest indivisible elements of work, called tasks, each of which possesses an associated task time ($t_i$). Tasks are related to one another by precedence attributes, i.e. some tasks must be finished before others can begin, usually due to the physical architecture of the product. These individual precedence relationships between tasks are collected and summarized by a precedence graph, an acyclic graph with each task as a node and arcs representing precedence.

Stations are spaced along the line such that there is one production unit present at each station, and all stations are allotted a fixed cycle time ($c$) to execute all assigned tasks before the conveyor moves the product to the next station. The ALB problem is to assign the set of tasks to stations, such that all work is performed upon the product as it traverses the line.

Assembly lines were originally constructed for mass production of standardized assembly products, to increase average worker productivity and overall throughput by leveraging labor specialization along the line (Shtub and Dar-El). Modern assembly lines designed for make-to-order and mass customization production permit fast and flexible responses to customer demand (Mather), but are associated with significant automation

and facility capital costs. See (Boysen, Fliedner and Scholl, Production Planning of Mixed-Model Assembly Lines: Overview and Extensions) for a recent survey of modern mixed-model methods.

Tool setup times and the subsequent lot sizing problem are typically avoided by assembly lines via application of universal machinery or the like, in order to maintain consistent flow of production cycles (Dolgui, Guschinsky and Levin). Lines that do require setup time to transition from one model to another are referred to as multi-model lines, and encourage batches of each model to be produced consecutively. Such lines require an additional lot sizing problem extension, as discussed by (Burns and Daganzo) and (Dobson and Yano).

## 6.2 Horizontal Line Balancing

Optimization of the traditional ALB problem seeks to minimize total idle time by minimizing of the number of stations (or workers) used, given a fixed cycle time. The problem is NP-hard, as shown by (Wee and Magazine). (Thomopoulos) and (Macaskill) transformed the mixed-model problem into a single model version by taking the demand-averaged time for each task. Such methods ignore the piece-to-piece variability in work content, and may result in disruptions in line operation.

Horizontal balancing seeks to equalize the work content at a station across all model alternatives, such that the resulting balance is more robust to changes in model demand and production sequencing. See Figure 45 for a visualization of two alternative solutions with the same average utilization, but drastically varying horizontal balance. In an early form of horizontal balancing, (Thomopoulos) attempted to compensate for this

effect by minimizing a secondary objective of the sum of absolute deviations of actual station times of each model to the average station time across models. (Domschke, Klein and Scholl) proposed a refined horizontal balancing objective that seeks to minimize the sum of work overload time, i.e. the work content in excess of the cycle time, across all models and stations. (Vilarinho and Simaria) developed a simulated annealing solution approach that incorporated both horizontal and vertical balancing objectives, within a model with parallel stations and additional assignment constraints.

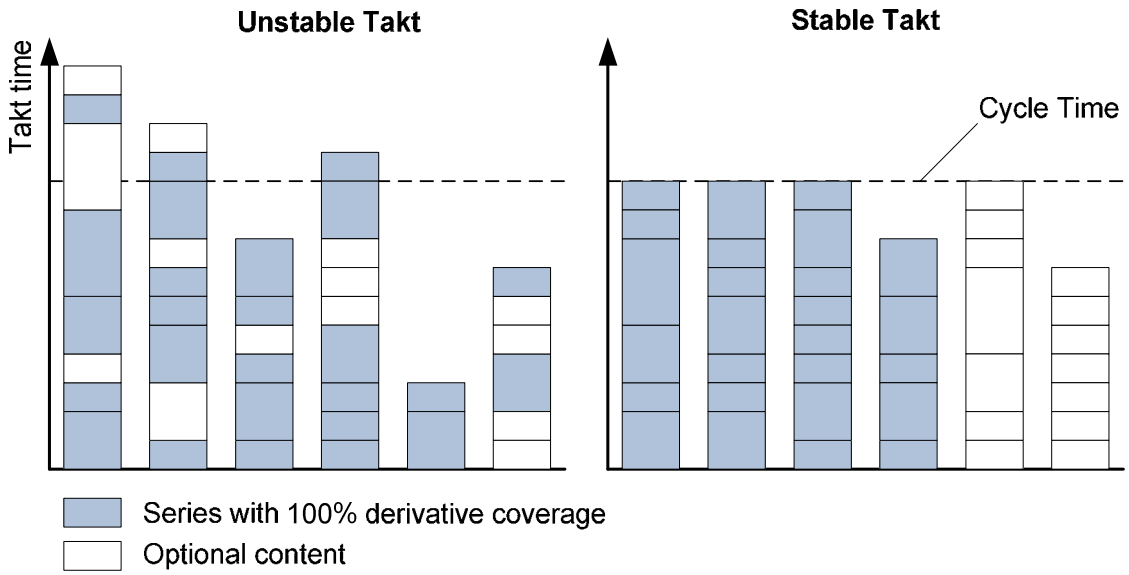


Figure 45: Horizontal Smoothing

The production sequencing problem that emerges from mixed-model environments can be solved in a staged fashion, subsequent to the ALB problem (Yano and Bolat); (Sumichrast and Russell, Evaluating Mixed-model Assembly Line Sequencing Heuristics for Just-in-time Production Systems); (Sumichrast, Russell and Taylor, A Comparative Analysis of Sequencing Procedures for Mixed-model Assembly

Lines in a Just-in-time Production System); (Bard, Dar-El and Shtub), or the two problems can be solved simultaneously (Merengo, Nava and Pozetti).

Demand is commonly realized at a shorter time horizon than is applicable to the ALB problem, however, suggesting that ALB methods that produce solutions that are robust to demand may be more applicable than those that simultaneously sequence the production units. To test the effectiveness of different horizontal line balancing metrics to this end, (Emde, Boysen and Scholl) conducted extensive computational experiments to detect differences in line disruption due to product variety on assembly lines balanced with an array of different horizontal metrics. (Emde, Boysen and Scholl) also mentions in closure the reliance of all tested methods on the mixed-model paradigm, and calls for methods that are robust to product variety size.

## 6.3 Data Environment

The necessary data inputs to the options-mix horizontal line balancing problem are two-fold: 1) the set of tasks (denoted $I$) and associated task attributes, 2) the set of options (denoted $\Omega$) and associated attributes, and 3) the database of object relations that defines option interactions. Before developing the attributes of these data inputs further it will be necessary to introduce the concept of a *derivative*.

### 6.3.1 Derivatives

It is assumed that each production unit is assembled to the specification of a single derivative, selected from the set of derivatives (denoted $\Psi$). Each derivative represents a partially configured product, unique in name, to assist both customers and internal operations in differentiating the vast array of product configurations into a more

manageable subset of categories. For customers, specification of product configuration begins with selection of derivative, which serves to assign a default subset of high-profile option content to the product while leaving low-profile options undetermined. As an example, customer choice of the motorsports derivative preselects engine, transmission, drivetrain, and brakes, while reducing the subset of available options for wheels, paint, and external trim. Full product configuration is then completed by the customer via specifying the remaining subset of options.

This approach affords the organization a tighter control of branding via the offering of several distinctive derivatives, rather than a single amorphous product. An addition organizational benefit is reduced effort in creation and maintenance of the database of object relations.

### 6.3.2 Object Interaction Types

Options are linked directly to each derivative category with flags to indicate that derivative selection either 1) requires the option, 2) forbids the option, or 3) leaves the option undetermined. See Table 21 for an example of how object relations between options and derivatives is recorded in the database.

| | Derivative 1 | Derivative 2 | Derivative 3 |
|---|---|---|---|
| Option 1 | M | M | F |
| Option 2 | F | M | F |
| Option 3 | O | O | O |

Table 21. Option / derivative relations: (M)andatory, (F)orbidden, and (O)ptional

Further control of relations between undetermined options is achieved via Boolean statements of the rule to be enforced within the database. Each rule is flagged

for the derivatives toward which it applies, and must be obeyed for products within the flagged derivatives. For example, to enforce the rule "IF option 1 THEN NOT option 2" for derivatives 1 and 2, but not derivative 3, then the database would encode the rule as seen in Table 22.

| | Derivative 1 | Derivative 2 | Derivative 3 |
|---|---|---|---|
| IF option 1 THEN NOT option 2 | T | T | |

Table 22: Rule relations: T = rule applies for derivative

Each task is also flagged for the derivatives towards which it is necessary. Task applicability to derivatives are encoded into one of three classifications: 1) the task is applied universally across all derivatives, 2) the task will only apply to a subset of the derivatives, but will be necessary for all production units within the subset, and 3) the task applies to a subset of derivatives, but only if some option is present. Table 23 presents an example of this encoding scheme in which task 1 is universal, task 2 is dependent on derivative only, and task 3 is dependent on both derivative and option. The code word SERIES is used to indicate that a task applies to all production units in the flagged derivatives.

| | Condition | Derivative 1 | Derivative 2 | Derivative 3 |
|---|---|---|---|---|
| Task 1 | SERIES | A | A | A |
| Task 2 | SERIES | | A | |
| Task 3 | Option 1 | A | A | |

Table 23: Task relations: A = task applies to derivative if condition met

### 6.3.3 Demand

Demand is specified on the derivative level as the relative frequency of each derivative $d_\psi, \psi \in \Psi$. Derivatives are mutually exclusive to one another, and each product must be of a single derivative, hence the total probability across all derivatives is equal to one, $\sum_\Psi d_\psi = 1$. This characterization of demand alone is insufficient, however, as it does not capture the demand of options that at not determined by selection of derivative. To completely specify demand it is necessary to also introduce the probability of each option conditioned upon each derivative: $d_{\omega,\psi} = P(\omega|\psi), \forall\, \omega \in \Omega, \psi \in \Psi$. Of course, forbidden option / derivative combinations result in $d_{\omega,\psi} = 0$, and mandatory combinations have $d_{\omega,\psi} = 1$. It is only optional combinations that must be specified by the input data.

The proportional demand for each task, $d_i$, can be derived as a function of the $d_{\omega,\psi}$ and $d_\psi$ data, by application of Eq 13.

$$d_i = \sum_{\psi \in \Psi} G1_{i,\psi} d_\psi + \sum_{\omega \in \Omega, \psi \in \Psi} G2_{i,\omega,\psi} d_{\omega,\psi} \qquad \forall i \in I \qquad \text{Eq 13}$$

In Eq 13, $G1_{i,\psi}$ is an indicator variable equal to 1 if task *i* is SERIES for derivative $\psi$ and 0 otherwise, and $G2_{i,\omega,\psi}$is an indicator variable equal to 1 if task *i* is requires option $\omega$ for derivative $\psi$ and 0 otherwise.

## 6.4 Maximum Bound on Cycle Time

Using the data inputs described in Section 6.3, we now have sufficient information to derive a horizontal line balancing metric for the options-mix ALB. Given a line balancing solution with all tasks assigned to *M* stations, the average utilization of

station $m$ is calculated by considering the subset of tasks assigned to station $m$, $I_m$, and then taking the weighted average of task times proportional to the cycle time, as seen in Eq 14.

$$Utilization_m = \sum_{i \epsilon I_m} \frac{d_i t_i}{C} \quad \text{Eq 14}$$

Utilization is a valuable metric for the classic ALB problem, and must be restricted to be $\leq 1$ for every station for the solution to be feasible to the cycle time. However, product variety may result in variability in the realized time usage from one product to the next.

In the following a method is developed to calculate the maximum amount of time that might be required of station $m$ to complete its assigned tasks. This upper bound is not necessarily simply the sum of all task times. If some tasks are linked to options content then it is possible that rules within the object relation database forbid execution of all tasks on any single product. Evaluation of whether a subset of tasks may co-exist upon any single product is achieved by first parsing all rules in the database into a Boolean encoding scheme, and then solving the resultant Boolean satisfiability problem (SAT).

The SAT problem considers a given set of primitives that are related by a given set of Boolean statements and determines whether there is any possible combination of true/false values that may be assigned to the primitives such that all statements are true. In Section 6.5 we show how to derive Boolean statements from the configuration data of the line balancing problem, and then parse those statements into binary parse trees.

Section 6.6 discusses outputting the binary trees to a SAT solver to determine which tasks assigned to a station may occur simultaneously on some unknown product to be assembled.

## 6.5 Logical Statement Construction

Before the SAT problem it is first necessary to translate all of the configuration information from database fields and rule strings show in Section 2 into a set of Boolean statements. Considering the example data provided in Table 23 above, equivalent Boolean statements are:

(IF derivative1 THEN option1) AND (IF derivative2 THEN option1) AND (IF derivative3 THEN NOT option1)

(IF derivative1 THEN NOT option2) AND (IF derivative2 THEN option2) AND (IF derivative3 THEN NOT option2)

Notice each derivative is treated with a separate IF clause, and the all of the derivative clauses are joined with AND conjunctions. No information within Table 23 relates option3 with any of the derivatives, and thus no Boolean statement is made regarding option3 as a result.

Considering the example data provided in Table 22, the equivalent Boolean statement is:

(IF derivative1 THEN (IF option1 THEN NOT option2)) AND (IF derivative2 THEN (IF option1 THEN NOT option2))

Considering the example data provided in Table 23 above, equivalent Boolean statements are:

IF derivative1 THEN (task1 AND NOT task2 AND (IF option1 THEN task3))

IF derivative2 THEN (task1 AND task2 AND (IF option1 THEN task3))

IF derivative3 THEN (task1 AND NOT task2 AND NOT task3)

Next it is necessary to add rules dictating that a product can be of one and only one derivative. Commonly referred to as "pick one" or "one hot lead" conditions, such rules are difficult to express using Boolean algebra expressions. It is sufficient to enumerate these rules pairwise, e.g. "IF derivative1 THEN NOT derivative2," and so on. Although the number of rules required to enforce this constraint grows combinatorially with respect to the number of derivatives, at the scale of this example problem (~20 derivatives) the size of this rule set is still feasible.

A binary parse tree is then used to translate each derived Boolean statement string into a more manageable data structure via depth-first recursive parsing of the strings. Within the tree structure each node is a logical operator or a primitive, with the following node types being sufficient to encapsulate the information: AND, OR, NOT, IF, and PRIMITIVE. Each node may possess up to two children nodes, as needed to complete the logical operator. PRIMITIVES are the codes that represent options, derivatives, or tasks, and are Boolean values that can be assigned true or false. PRIMITIVES exist only in leaf nodes and comprise all leaf nodes. Table 24 summarizes each of the node types within the parse tree.

| Node type | # Children | Description |
|---|---|---|
| AND | 2 | Both children evaluate to TRUE |
| OR | 2 | Either child must evaluate to TRUE |
| NOT | 1 | Child must be FALSE |

| | | |
|---|---|---|
| IF | 2 | If left child is TRUE, then right child must be TRUE |
| PRIMITIVE | 0 | Object (option, derivative, or task) |

Table 24: Nodes in binary parse tree

Figure 46 shows an illustration of the binary tree representation of a Boolean statement.

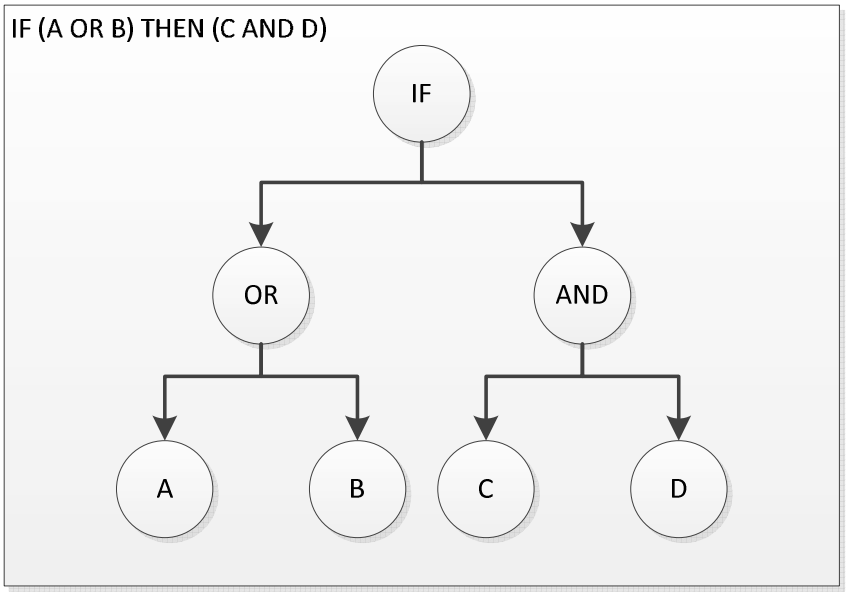


Figure 46: Binary Parse Tree Example (not CNF)

Most SAT solvers operate on Boolean statements that are in conjunctive normal form (CNF). Application of double negative and distributive laws to the parse trees are applied to achieve CNF.

## 6.6 SAT with Task Subsets

The information encoded in the object relation database defines all configurable products. Thus far we have derived a set of Boolean statements, each of which may evaluate to true or false depending on the true/false value of the related primitives. Let us call this total rule set Φ. It remains only to specify which tasks we wish to evaluate for

satisfiability. If the entire set of tasks $I_m$ assigned to the station can occur on a single product, then it follows that the primitives associated with those tasks can be set to true and the SAT can still be solved. In other words, in this case there would be some combination of true/false values of the option and derivative variables that would result in all of these tasks being necessary for a single product. Using this concept, we present the following algorithm to determine the maximum time subset of tasks that may occur on a single product.

```
1   Algorithm MaximumSubsetTime
2   Set J=Im
3   Set Φ̂ = Φ
4   Append rule j = true to Φ̂ for each j∈J
5   Solve SAT for Φ̂. If SAT is true, stop. Else, continue
6   Save J to remember that it has been tested
7   Set J= the next smallest set of tasks from Im
8   GoTo Step 3.
```

Figure 47: Maximum Time Subset Algorithm

In this algorithm task sets are considered and tested in a sequence determined by the sum all task times included in the set. Initially all tasks at the station are considered, $I_m$. If that set is not feasible then the task with the smallest time is removed from the set, and the test is repeated. Step 5 remembers past tests to facilitate the search mechanism that must be performed in Step 6.

## 6.7 Conclusion

As the number of configuration options grows, production modeling methods that rely on the mixed-model paradigm increasingly struggle to enumerate the total number of unique models that might be produced. Further, specification of necessary input parameters such as demand may become infeasible for these large model sets. The

options-mix paradigm offers hope for alternative modeling methods, as options sets can easily be enumerated and assigned e.g. demand. One of the primary challenges to using options-mix information is that the information model no longer contains direct analogues of the production units. A horizontal line balancing problem is introduced in which examination of various production units is mandatory for evaluation of the needed objective function. A procedure for modeling options information is then presented that permits application of a SAT to the options-mix configuration data in order to deliver the necessary maximum time bound for the horizontal line balancing metric.

CHAPTER SEVEN

# 7 CONCLUSIONS AND FUTURE WORK

## 7.1 Summary and Conclusions

The Modified Ranked Positional Weight constructive heuristic is developed, introducing a unique prioritization scheme driven by measuring constraint satisfaction scarcity. Responsibility sets are introduced to encapsulate task-to-task precedence and assignment linkage constraints. Urgency score are introduced to measure assignment limitations due to resource constraints. Tasks are weighted by a composite prioritization score based on these new metrics, and assigned according to a first-fit-decreasing single-pass heuristic. The MRPW heuristic is oriented toward creation of feasible solutions, with efficiency being a secondary consideration.

The Last Fit Increasing improvement heuristic leverages the task prioritization rankings of MRPW, and consolidates flexible tasks into otherwise lightly-packed workers. The goal of the LFI improvement heuristic is to improve the efficiency of a feasible solution.

The Work Zone Blocking heuristic focuses on the first work zone selection sub-problem of the bifurcated ALB problem. The purpose of this approach is to address zoning difficulties encountered in the MRPW heuristic. Two new metrics are introduced to support the heuristic, measuring work zone flexibility and uniqueness. The metrics are developed in consideration of each WZ's offerings in terms of satisfying task needs with respect to zoning, tooling, and accessibility constraints.

A binary integer programming formulation of the problem is developed with several unique approaches to manage the zoning and worker parallelization aspects of the

problem. Preprocessing transformations render several complex facets of the problem into representations amenable for a tight BIP formulation.

Each solution methodology is applied to a testbed of 130 instances derived from real ALB data collected in conjunction with our industrial partner. The IP is benchmarked primarily according to the runtime required relative to the size of the instance, to which it performs surprisingly well, needing only 22 seconds at most to solve an instance. The IP solution is used to benchmark heuristic performance. The heuristics were able to find feasible solutions for 83.2% of problem instances, conditioned on a feasible solution existing. Among that subset of feasible solutions, the heuristics averaged an optimality gap of approximately 20-22%, depending on which heuristic was applied, and found the optimal solution for 25% of the instances. Due to superior performance and adaptability, the IP is heartily recommended for industrial application. The heuristics appear to be a much poorer choice for implementation, excepting extenuating scenarios in which the IP is unsolvable.

The final contribution gives momentum to the movement towards the options-mix paradigm for modeling option-heavy mixed-model environment. A particular options-mix ontology is presented with an accompanying SAT-iterative algorithm for measuring worst-case takt time. The methods described deliver the ability to compute a horizontal line balancing metric for this ontology, which is otherwise unavailable using the model-mix paradigm.

## 7.2 Future Research

### 7.2.1 Penalization of Constraint Violation

As noted in section 5.1.1, our industrial partner occasionally deploys ALB solutions that are not feasible with respect to the zoning constraints specified. In particular, WZs were occasionally allowed to absorb tasks with PZs that would not ordinarily be allowed. Such an assignment violates management guidelines, but does not result in an ALB solution that is infeasible for technical reasons. It is speculated that such solutions are permitted by our industrial partner due to efficiency advantages gained.

There is an opportunity for future work to separate all constraints into management-derived guidelines and technical requirements. Violations to management guidelines might then be permitted via a constraint penalization function, wherein each violation accumulates a penalty in the objective function. If a violation offers sufficient benefit to the objective in return for the cost of violation, then it may be allowed. This future work will require careful consideration of methods to measure the degree of constraint violation, as well as the appropriate weighting function for aggregating and applying the penalty.

### 7.2.2 IP Extension: Task Sequencing

Implementing task sequencing constraints would require adding new decision variables to the IP to ensure that task start/stop times are properly managed. Adding these variables and associated sequencing constraints to the IP is relatively direct in terms of formulation, but might present significant consequences in terms of runtime. Adding decision variables might always be expected to add runtime, but in particular start/stop time variables are quantified over the real numbers. All other variables currently in the

IP are binary, significantly restricting the size of the solution space. Timing variables changes the IP from a BIP to a MILP, and runtime penalties should be expected.

### 7.2.3 Robustness of Solutions to Uncertain Demand

Each workpiece is custom ordered with limited lead time. At the time of line balancing the demand for each option is forecasted, but uncertain. Variation in demand can lead to infeasibility of the solution generated, via overloading average task loading for a worker. At the current time there are very few robust optimization approaches for ALB in the literature, and none that use the options-mix paradigm. A 2-stage scenario-based robust optimization model might be constructed to attack this problem. In stage 1 the primary problem is solved, with an efficiency based objective function. In stage 2 the secondary horizontal smoothing objective is used, using the metric discussed in section 2.1.36.4, seeking to minimize the maximum load time.

Construction of the scenarios will be an important lead-in step to prepare a robustness model. Each option has a given (forecasted) demand, which can be perturbed to create the uncertainty set. Options may be correlated, however, either through rule or customer preference. Historical production data may possibly be mined to examine option demand correlation.

The robust optimization ALB problem must reconcile both workforce costs (i.e. the cost associated with lower efficiency) and disruption costs (i.e. the cost incurred when a station is overloaded). In addition there is a rebalancing cost associated with changing the work content of a station, as the worker must learn the new assembly process.

## 7.3 Tools Developed as Part of Research Project

Several prototype software tools were developed during the course of this research. The heuristics were implemented in VBA, and are partially documented in Appendix A. The IP formulation was implemented in AMPL, with the model file and an example data file shown in Appendix B. Appendices C and D document configuration management methods developed upon the ontological scheme described in chapter 6, and implement SAT-iterative methods that are very similar in spirit to the worst-case takt time application.

APPENDICES

APPENDIX A

Prototype Software Documentation: MRPW heuristic

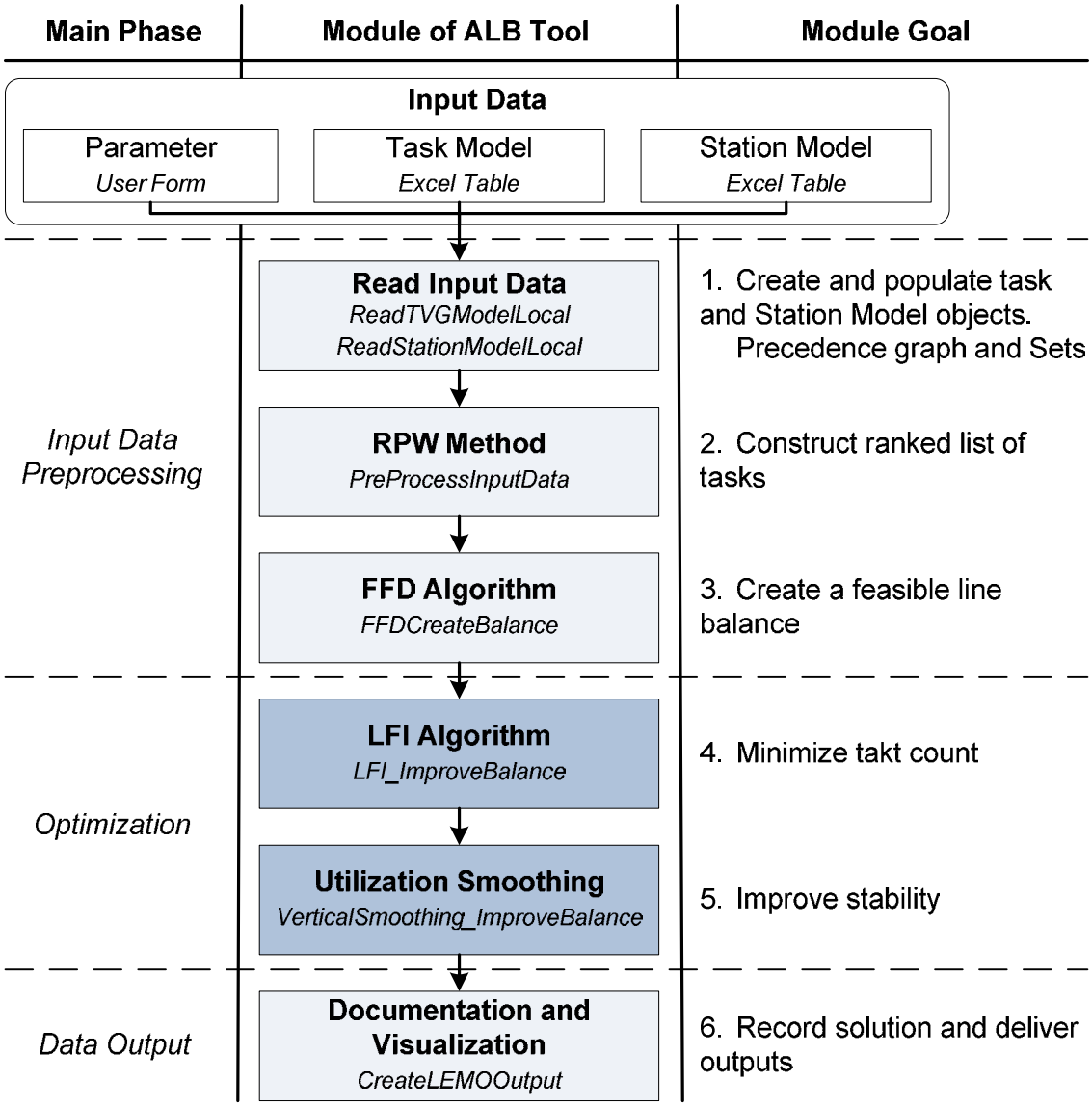


Figure 48: Heuristic Prototype, Main Function

7.3.1 Function Main

1. Input of parameters on form. Store parameters as global variables.

2. Read input data from source spreadsheets, corresponding to the band selected by user.
    a. Function ReadStationModelLocal constructs the station model object in memory. The data for each station is parsed and stored in an array of station objects.
    b. Function ReadTVGModelLocal constructs the task model object in memory. The data for each task is parsed and stored in a group of task objects.
        i. Cross reference the product references witnessed within the set of tasks to the derivative information on sheet Deriv_Map. Store unique ProdRef codes from into global variable PubProdRefList and corresponding product volume into global variable PubProdRefMaxVol.
3. Function PreProcessInputData performs a number of preprocessing steps to prepare the data environment.
    a. Set task volume by derivative. For series tasks this will be the same as the max volume for each derivative. Else, the task's total volume is divided among its applicable derivatives according to the relative size of each derivative's max volume.
    b. Tooling (station and task model). Verify that all tools needed by tasks exist on some station. Link each tool object to every task that needs it and to all stations that provide it.

c. Construct the full precedence graph. The algorithm to accomplish this is embedded in function PrecedenceBuilder, detailed in the appendix of this document.

d. Create task sets by calling TVGModel.BuildSets. After input data parse the task objects have only the strings corresponding to their set membership.

    i. Create an object of type CTVGSet for each uniquely named set.

    ii. Link the object to task members.

    iii. Derive implied sets (adjacency implies same takt) to check input data consistency.

    iv. Construct an extended precedence graph for each set .

        1. The extended, or indirect precedence graph for a task (say, $task_i$) is all tasks that are predecessors or in a set relationship with $task_i$, or with any other task in the extended precedence graph of $task_i$.

        2. The extended, or indirect successor set is constructed in the same way.

e. For tasks with eligible station constraints, link the corresponding stations to the task's .EligibleStation property.

f. Calculate 3 RPW values for each task. These formulas and algorithms are shown in function CalcRPW in the Appendix.

i. Classic RPW is calculated for each task by summing the shifttime of itself and all direct successor tasks.

ii. Set related RPW is calculated for each task by summing the shifttime of itself and all tasks that are extended successors. Note that tasks that share a set will have the same value for this score.

iii. The tooling/eligible station RPW score is constructed in two stages.

1. Calculate the urgency score of each task as a function of tooling and eligible station properties of the task.
2. The ToolRPW of a task is the maximum urgency score amongst the task itself and all tasks that are extended successors. Note that tasks that share a set will have the same value for this score.

g. Sort the tasks in 3 stages by RPW scores. Ties at a given stage are broken by considering the next-stage RPW score.

i. ToolRPW

ii. AdjRPW

iii. RPW

4. FFD algorithm. Search for a feasible balance by assigning tasks as early as possible on the band. tasks are assigned one-at-a-time according to RPW scores such that the tasks that are most important to assign early on the band are assigned first.

a. Take the task from the top of the RPW list. If this task must be assigned concurrently with other tasks due to set relations then collect the set of tasks.
b. Examine each station in turn, from the start of the band. Consider station-level constraints:
    i. Precedence. All tasks that are predecessors to the task (s) currently under assignment have already been assigned, due to having higher RPW scores. Fail if any of these predecessors are assigned to stations after this station.
    ii. Eligible station. Fail if the task (s) is/are not eligible at this station.
    iii. Tooling. Fail if this station does not provide required tool coverage at to the task (s) product zone.
    iv. PZ accessibility. Fail if this station is blocked for the product zone (for any) of the task (s).
    v. If any of these constraints fail then consider the next station in sequence. If all are satisfied, then consider takt-level constraints. Construct the list of all takts at this station that are zone compatible with the task(s).
        1. Shift time. Fail if this takt does not have sufficient shift time remaining to add the task(s).

2. Derivative-utilization. Fail if any derivative-utilization at this takt would exceed the optional user-derived utilization value if the task(s) were added.
3. If multiple takts pass these constraints, then prefer to assign the task(s) to a takt that is already active. If no active takt is available then a currently inactive takt may be opened.
    a. If multiple takts still persist, then choose arbitrarily via sequence L, R, V, H, I.
    b. If a valid station and takt have been identified, then assign the task(s).

c. If no station satisfies all constraints, then create a dummy station at the end of the band. Give the dummy station all tools and maximum accessibility. The final balance will not be feasible, but it will have all tasks assigned.

5. LFI algorithm. If the FFD algorithm successfully created a feasible balance, then attempt to increase average utilization by removing takts. LFI operates on a similar principal as FFD, but in reverse RPW sequence and moving tasks as far toward the end of the band as possible. A notable difference between the two heuristics is that LFI is not permitted to open new takts. The points at which the LFI algorithm differs from FFD are italicized in the following summary:

a. Take the task from the *bottom* of the RPW list. If this task must be assigned concurrently with other tasks due to set relations then collect the set of tasks.

*b. Remove the task(s) from current station and takt assignment. This algorithm's goal is to empty a takt here, finding another takt that may receive the task(s).*

c. Examine each station in turn, from the *end* of the band. Consider station-level constraints:

    i. Precedence. Fail if predecessors are assigned to stations after this station.

    ii. Eligible station. Fail if the task(s) is/are not eligible at this station.

    iii. Tooling. Fail if this station does not provide required tool coverage at to the task(s) product zones.

    iv. PZ accessibility. Fail if this station is blocked for the product zone (for any) of the task(s).

    v. If any of these constraints fail then consider the *previous* station in sequence. If all are satisfied, then consider takt-level constraints. Construct the list of all takts at this station that are zone compatible with the task(s).

        1. Shift time. Fail if this takt does not have sufficient shift time remaining to add the task(s).

2. Derivative-utilization. Fail if any derivative-utilization at this takt would exceed the optional user-derived utilization value if the task(s) were added.
3. *Activity. Fail if this takt is not currently active.*
    a. If multiple takts pass these constraints, then choose arbitrarily via sequence L, R, V, H, I.
    b. If a valid station and takt have been identified, then assign the task(s).

6. Smoothing algorithm.
    a. Competing balance solutions may be compared by development of a scoring process to measure smoothness. One such metric is to sum the squares of derivative-utilization across all takts. The minimum theoretical value for this score is achieved if the derivative-utilization figures are equal across and within all takts.
        i. Perform a neighborhood search upon the existing line balance solution. One likely neighborhood is to focus upon the highest derivative-utilization takts, and attempt to move the task(s) out of these takts.
        ii. If any neighboring solution improves the score, then move to that solution and repeat until some termination criteria is met.
    b. Alternatively, line balance solutions with derivative-utilization limits may be created with the existing FFD and LFI heuristics with the optional

derivative-utilization constraint active. Iterative reduction of the derivative-utilization cap parameter suffices to reduce the worst-case derivative-utilization takt. As the parameter is increasingly reduced the FFD algorithm will at some point cease to find a feasible balance.

7. Output and Visualization
    a. Create Sulzer-compatible .csv file of balance.
    b. Create spreadsheet view of balance similar to LEMO.

### 7.3.2 Object Modeling and Data Composition

The line balancing tool maintains an assortment of data necessary to the line balancing algorithm. This data is classified into three categories: production system data, line balancing output, and run-time parameters for the tool. Object model hierarchies are used extensively to structure the production system and line balancing data, and a collection of global variables are used to store run-time parameters. Descriptions of these object models and variables are shown in the following subsections.

#### 7.3.2.1 Production System Data

The category production system data contains all data from the real system that are necessary inputs to the line balancing process. Under this umbrella three subcategories are defined: task, station, and environment. Task data consists of the list of tasks to be balanced and all relevant task properties, e.g. time, precedence, etc. Station data describes the physical characteristics of each station within the band and associated intra-station attributes. Environmental data includes a variety of inputs, external to the more distinctly defined task and station data, such as cycle time. Specification of all

production system data for a band is necessary and sufficient to perform line balancing on that band.

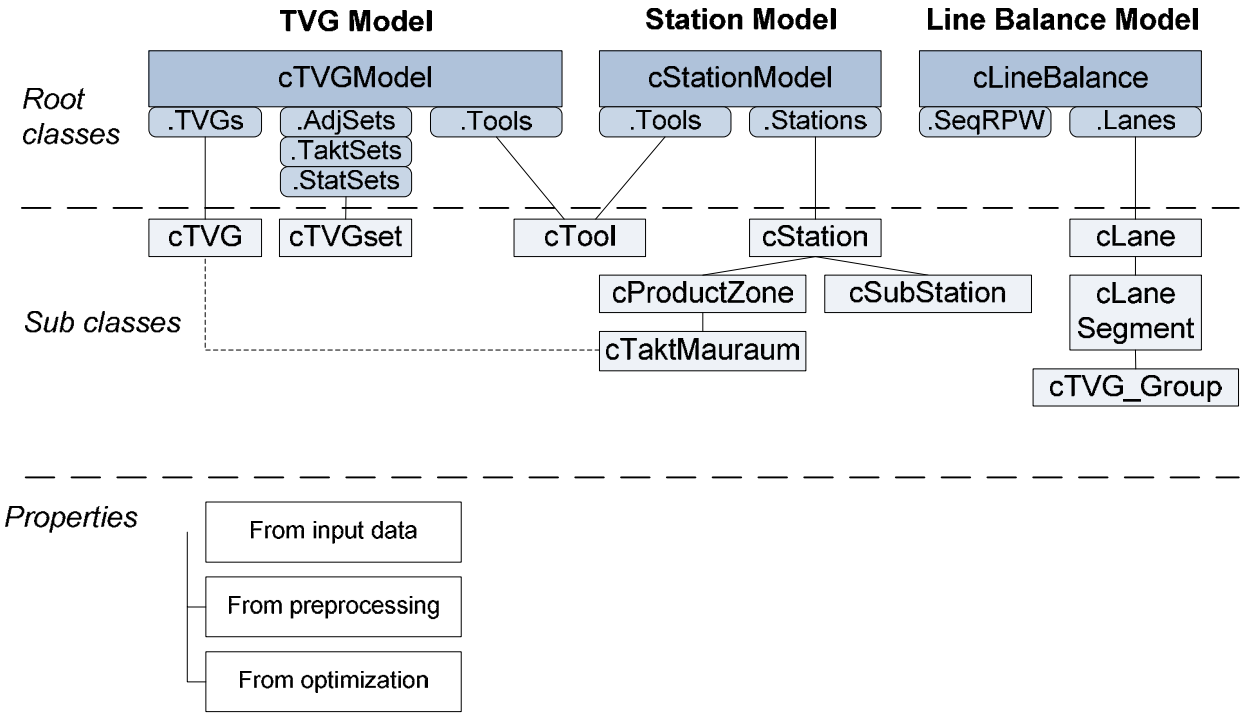


Figure 49: Properties of and Relations Between Top-level Objects

7.3.2.1.1 Task Model

Task data is sourced locally within the VBA tool, with task data for each band stored on separate spreadsheets named e.g. “Band 30 tasks.” These input worksheets are formatted in flat form, such that each task is given a single row in which its information is encoded. The data is read from source, parsed, preprocessed, and stored hierarchically within the top-level class CTVGModel. The structure of the CTVGModel object model hierarchy is shown in the following list, with brief summaries of each data element.

- Class CTVGModel. There is only a single instance of the task model class, created by the Main function.
    a) CTVGModel.TVGs. A group of objects of class CTVG; these are the task objects to be balanced. Each task is instanced once as a CTVG object. Each task within this set is referenced by its *ID* property. The following are the properties of the CTVG class.
        i) Properties that contain the data directly read from the input data sheet
            (1) *ID*. An integer for each task that is equal to the sequence of the task on the input data sheet (row number – 1). Any ZW tasks created by the tool will have a unique negative ID number.
            (2) *Name*. String containing the task name, e.g. S 5121 001 602 A 01.
            (3) *Signature*. String containing the task name concatenated with the task's ProdRef. Used for derivative specific balancing purposes, in case any task within the input data must be split amongst applicable ProdRef to create multiple tasks, so that each output task may have a unique identifier.
            (4) *OPR*. String containing OPR class, e.g. ZH, ZW, M, PF.
            *(5) Description*. String containing short text description of task.
            *(6) ImmedPredNames*. StrList containing the names of the tasks that are immediate predecessors to this task. This property is used in preprocessing to develop the full precedence graph, once all task data has been read off of the input data sheet.

(7) *Time_min*. Double containing the time, in minutes, that the task requires to perform.

(8) *Vol*. Double containing the shift volume of the task.

(9) *ProdBauraum*. Product zone of the task, encoded as Enum Bauraumen.

(10) *Ergo*. Double containing EBI score of task.

(11) *ProdRef*. StrList containing the ProdRef codes that determine the derivatives that the task applies towards.

(12) *StatSet*. String containing the same station set membership of the task. This property is used in preprocessing to establish linkage to all matching same station set tasks.

(13) *TaktSet*. String containing the same takt set membership of the task. This property is used in preprocessing to establish linkage to all matching same takt set tasks.

(14) *AdjSet*. String containing the adjacency set membership of the task. This property is used in preprocessing to establish linkage to all matching adjacency set tasks.

(15) *Tools*. Dictionary containing the tool objects that this task requires. Note that only a single tool object exists for each uniquely named tool.

(16) *EligStationNames*. StrList of the names of each station that this task may be assigned to. If empty, then the task may be assigned to any station.

(17) *OptionsList.* StrList containing the object reference codes for the task.

ii) Properties derived during preprocessing of data

(1) *Vol_ProdRef.* Array of doubles containing the volume of this task for each ProdRef. The sequencing of this array matches the public array PubProdRefList.

(2) *PredsImmed.* Group of task objects that are immediate predecessors. During preprocessing the strings in the ImmedPredNames property are read and these links to the task objects are established.

(3) *Preds.* Group of task objects that comprise all direct predecessors. During preprocessing the precedence graph for this task is completed by reading the contents of the PredsImmed property, and links established to predecessors of predecessors.

(4) *Succs.* Group of task objects that comprise all direct successors. During preprocessing the complete successance for this task is found by reading the contents of the Preds property, and establishing backward links established to successors.

(5) *AdjacentTVGs.* CTVGset object that contains the adjacency set of this task.

(6) *STaktTVGs.* CTVGset object that contains the same takt set of this task.

(7) *SStatTVGs.* CTVGset object that contains the same station set of this task.

(8) *Preds_Extended.* Group of task objects that comprise all direct and indirect predecessors. After finding all direct predecessors, indirect predecessors are tasks that are related by adjacency, same takt, and/or same station to any predecessor. This group also contains all tasks that are set-related with the local task, some of which may be successors to the local task.

(9) *Succs_Extended*. Group of task objects that comprise all direct and indirect successors. After finding all direct successors, indirect successors are tasks that are related by adjacency, same takt, and/or same station to any successors. This group also contains all tasks that are set-related with the local task, some of which may be predecessors to the local task.

(10) *EligStations*. Dictionary of Station objects that are eligible for the task to be assigned. During preprocessing the strings in the EligStationNames property are read and these links to the Station objects are established.

*iii)* Properties to support line balancing methods. These properties are set by the RPW module or during the subsequent line balancing algorithm.

(1) *Assigned.* Boolean containing whether the task has been successfully assigned to a takt.

(2) *RPW.* The ranked positional weight score of this task, based off of direct successance only, as determined by summing the cumulative shift time of the tasks in property Succs.

(3) *AdjRPW.* The ranked positional weight score of this task, based off of extended successance, as determined by summing the cumulative shift time of the tasks in property Succs_Extended.

(4) *ToolWt.* The "urgency score" of this task's tool requirements. An individual tool urgency score is the total number of stations in the band minus the index number of the last station that possesses the tool. A higher score indicates that the tool last appears closer to the beginning of the band. The ToolWt property is then calculated as the maximum urgency score amongst the task's tool requirements.

(5) *EligStWt.* The "urgency score" of this task's eligible station requirements. Scoring is performed identically as for tooling.

(6) *ToolRPW.* The ranked positional weight score of this task, based off of extended successance, as determined by taking the maximum ToolWt or EligStWt property of all extended successors.

(7) *Taktmauraum.* Link to the class TaktMauraum object that the task is assigned to.

b) CTVGModel.AdjSets. A group of objects of class CTVGset; these are the adjacency sets. Each adjacency set is instanced once as a CTVGset. The following are the properties of the CTVGset class:

i) *Name.* The name of the set, e.g. Träger TV KOM F30. This is the unique string that has been used on the input data form for all tasks in the set, indicating set membership. In the case of implicit sets (adjacency implies

same takt, and same takt implies same station) unique strings are created internally by the algorithm.

ii) *TVGs.* A group of objects of class CTVG; these are the member tasks of the set. For more information on CTVG objects, see CTVGModel.TVGs section.

iii) *Preds.* A group of objects of class CTVG; these are the predecessor tasks of the set. This predecessor group is constructed during preprocessing. The contents are all predecessors from all task members of the set, except for tasks that are themselves members of the set. For example, task A is a predecessor of task B, and both are in an adjacency set. In this case, task A will not appear as a predecessor of the adjacency set.

iv) *Succs.* A group of objects of class CTVG; these are the successor tasks of the set. The same guidelines described in the above *Preds* section applies here.

v) *Assigned.* Boolean to mark whether the set has been assigned to a takt.

vi) *Tools.* Dictionary containing the tool objects that this set requires, cumulative across all tasks in the set. Note that only a single tool object exists for each uniquely named tool.

vii) *Totaltime_sec.* Double containing the total time requirement of tasks in the set. Currently unused.

c) CTVGModel.TaktSets. A group of objects of class CTVGset; these are the same takt sets. Each same takt set is instanced once as a CTVGset. See CTVGModel.AdjSets section above for more information on CTVGset contents.

d) CTVGModel.StatSets. A group of objects of class CTVGset; these are the same station sets. Each same station set is instanced once as a CTVGset. See CTVGModel.AdjSets section above for more information on CTVGset contents.

e) CTVGModel.Tools. A group of objects of class CTool; these are all the tools that are needed by the tasks. Each unique tool is instanced once as a CTool, regardless of how many times that tool might appear in the station model, or how many tasks require the tool. CTVGModel.Tools requires cross-referencing the station model for construction.

7.3.2.1.2 Station Model

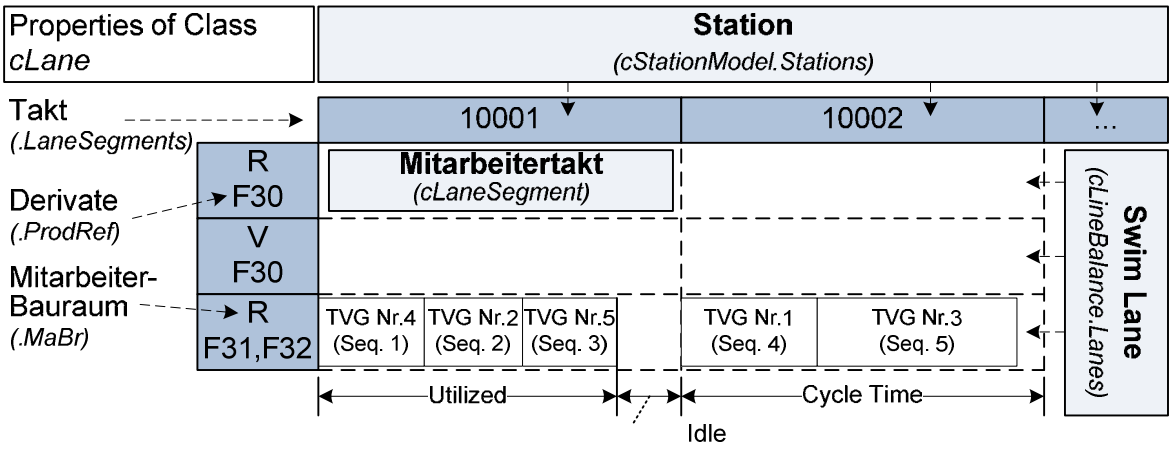


Figure 50. Station Model Objects

Station data is sourced locally within the prototype heuristic tool, with station data for each band stored on separate spreadsheets named e.g. “Band 30 Stations.” These input worksheets are formatted in flat form, such that each station on the band is given 16 rows in which its information is encoded. The data is read from source, parsed, preprocessed, and stored hierarchically within the top-level class CStationModel. The

structure of the CStationModel object model hierarchy is shown in the following list, with brief summaries of each data element.

- Class CStationModel. There is only a single instance of the station model class, created by the Main function.
    a) CStationModel.Stations. An array of objects of class CStation; these represent the physical stations, sequenced as they appear on the input data sheet. The following are the properties of the CStation class.
        i) *ID.* Integer corresponding to the sequence of the station in the input data sheet.
        ii) *Name.* String containing the name of the station in the input data sheet, e.g. 01001.
        iii) *Orientation.* Custom type VehicleOrientationList; contains an indicator variable that maps to the orientation of the vehicle within the station. There are eight vehicle orientations currently supported. E.g. "R-Leading" indicates that the right side of the vehicle is oriented to the front relative to the flow of the line. Orientation is used to determine the mapping between product zones and work zones, as well as the mapping of tool coverage to product zones. See the Substation class documentation below for more information on taktmauraums and tooling coverage mapping.
        iv) *Tools.* Dictionary containing the tool objects that this station provides, cumulative across all member substations. Note that only a single tool object exists for each uniquely named tool.

v) *WZMax.* Integer of the maximum number of workers permitted to work at this station simultaneously.

vi) *SubStations.* An array of objects of class CSubStation; these are sections of the physical station. Each substation is analogous to a work zone, and are named as such (L, R, V, H, I). This distinction between substations and work zones is enforced to clarify that the substations are simply physical zones, with attendant physical properties. The following are the properties of the CSubStation class.

(1) *Name.* String containing the name of the substation, e.g. "01001V."

(2) *Station.* Object of class CStation; the station that contains this substation.

(3) *Zone.* Custom type SubstationZoneList; contains an indicator variable that maps to which substation this is, e.g. V, H, L, R, or I.

(4) *Accessibility.* Custom type ZoneAccessibilityList; contains an indicator variable that maps to the accessibility status of the substation. Currently only "available" and "blocked" are supported. If "available" then an associate may be placed in the corresponding taktmauraum at this station, but not if "blocked."

(5) *Tools.* Dictionary containing the tool objects that this substation provides. Note that only a single tool object exists for each uniquely named tool.

vii) *ProductZones*. An array of objects of class CProductZone; these are virtual zones in the same pattern at product zone, except oriented to the station

regardless of the orientation of the vehicle (i.e. these are always V-leading). The following are the properties of the CProductZone class.

(1) *Station*. Object of class CStation; the station that contains this product zone.

(2) *PZ*. Custom type Bauraumen (e.g. LV, MH) containing the real product zone of the vehicle that will be present in this product zone. This property will change in response to changes in the vehicle orientation.

(3) *PZNatural*. Custom type Bauraumen containing the product zone of the vehicle that would be present in this product zone if the orientation was the default V-leading. This property will not change in response to changes in the vehicle orientation.

(4) *Accessibility*. Custom type ZoneAccessibilityList; contains an indicator variable that maps to the accessibility status of the product zone. Currently only "available" and "blocked" are supported. If "available" then tasks of matching product zone may be placed at this station, but not if "blocked."

*(5) Tools*. Dictionary containing the tool objects that provide coverage to this product zone. Tool coverage is determined during preprocessing in consultation with orientation and default tool coverage maps.

*(6) TaktMauraumAssigned.* Object of class CTaktmauraum containing the takt that has been assigned to tasks at this product zone at this station, if any.

b) CStationModel.Tools. A group of objects of class CTool; these are all the tools that are provided by the stations. Each unique tool is instanced once as a CTool, regardless of how many times that tool might appear in the station model. CTVGModel.Tools requires cross-referencing the task model for construction.

7.3.2.1.3 Environment data

The environmental data inputs are stored in global variables.

1. *Pubtakt_time_sec*. Double containing the cycle time of the assembly line, in seconds. User may edit this input on the launch form.
2. *PubProdRefList.* StrList containing all product reference (derivative) strings appearing within the tasks. This list is constructed as tasks are read from the input sheet. If a task is read that possesses product reference that has not yet been encountered, then that new product reference is stored here.
3. *PubProdMaxVol.* AList containing the maximum volume for each derivative. The elements within this list are sequenced to match the sequence of derivative strings in PubProdRefList. This list is constructed during preprocessing, after PubProdRefList is complete. The values contained are read from the table on the spreadsheet "Deriv_Map."

7.3.2.2 Line Balancing Output

- Class CLineBalance. This class contains all information related to a line balance solution. This class may have more than one instance, in case of multiple alternative line balances. The following is a hierarchy of the class contents.

*a)* *SeqRPW*. Array of integers. Each integer in the array maps to the ID number of a task. This array contains all task ID numbers, sorted by RPW score. The highest scoring tasks by RPW (i.e. the most important to place early on the line) are listed first.

b) *Lanes*. Array of objects of class CLane. Lanes are determined by the cross of the derivatives present and the taktmauraums, e.g. F32-R. The following are the properties of the CLane class.

i) *Name*. String containing the name of the lane, e.g. F32-R.

ii) *WZ*. Custom type MABauraum, containing an indicator variable that maps to the work zone of this lane, e.g. "R".

iii) *ProdRef*. String containing the ProdRef (derivative) of this lane.

**iv)** *LaneSegments*. Array of objects of class CLaneSegment. Each lane segment is determined by the cross of the lane and the station, e.g. F32-R, 01001. The following are the properties of the class CLaneSegment.

#### 7.3.2.3 Function PrecedenceBuilder

Constructs the complete precedence graph for tasks. The input data document delivers, at minimum, all of the *immediate* predecessors of each task. For any given task, the predecessors that are not immediate may be found by examining the implicit chains of precedence formed by linking the immediate predecessors of immediate predecessors, and so on. The algorithm below performs a depth-first stack trace to accomplish this function. Note that any valid precedence graph must contain root nodes, i.e. tasks that have no predecessors. The precedence chain of a task can be constructed by adding immediate predecessors to the stack until root nodes are found. As an additional benefit, the algorithm also detects precedence cycles that would otherwise lead to degenerate cycling.

It is permissible for non-immediate (implicit) predecessors to be delivered in the input data document. Such additional data will not harm the algorithm below, but may tend to clutter the input data form unnecessarily.

The algorithm uses three structures to trace each of the *N* tasks back to precedence roots:

1) *mpreds*, a *N*x*N* matrix, to store precedence relationships. If *mpreds(i,j)* = 1 then task *j* is a predecessor of task *i*. Upon initialization, all elements of *mpreds* are 0. Elements are changed to 1 as precedence relationships are discovered.

2) *stack*, a generic stack data structure, to hold the tasks involved in each trace. Upon initialization the stack is empty.

3) *flags* , a *N*x1 array, , to track the status of each task. There are 3 states tracked by *flags*: {0= untraced, 1= on the stack, 2= fully traced}. Upon initialization all elements of *flags* are 0.

Mark immediate precedence relationships in *mpreds*, as shown in Figure 51.

```
1   For i = 1:N
2       For j = 1:N
3           If task j is an immediate predecessor of task i
4               mpreds(i,j) = 1
```

Figure 51. Tracking Immediate Predecessors

Now we wish to examine each task, in arbitrary order, and construct the stack trace of predecessors towards root nodes. For speed savings, it is only necessary to stack predecessor tasks that have not undergone stack trace previously. Tasks that have previously undergone stack track will already have their complete precedence graph constructed, so it is not necessary to duplicate this work. This point is relevant to a situation in which a predecessor task is shared between multiple successors (only one successor must trace through this predecessor), or, alternatively, a situation in which a predecessor is examined earlier than one of its successors (due to arbitrary ordering of tasks).

```
1   For i = 1:N
2       If flags(i) = 0
3           stack.push(i)
4           flags(i) = 1
5           While stack.size > 0
6               index = stack.top
7               done = true
8               For j = 1:N
9                   If mpreds(index, j) = 1
10                      If flags(j) = 0
11                          stack.push(j)
12                          flags(j) = 1
13                          done = false
14                      Elseif flags(j) = 1
15                          Log error
16                          Exit program
17                      Else
18                          flags(j) = 2
19                  If done = true
20                      For j = 1:N
21                          If mpreds(index, j) = 1
22                              For k = 1:N
23                                  If mpreds(j, k) = 1
24                                      mpreds(index, k) = 1
25                      stack.pop
26                      flags(index) = 2
```

Figure 52. Trace Implicit Predecessors and Detect Precedence Cycles

Table 25 describes the logic undertaken at each line of this algorithm.

| LINE | COMMENT |
|---|---|
| 1 | i is the task undergoing trace |
| 2 | Is task i untraced? |
| 3 | Push i onto stack |
| 4 | flag i as on stack |
| 5 | stack not empty |
| 6 | index is task on top of stack |
| 7 | changes to false if index has untraced preds |
| 8 | j is potential predecessor of index |
| 9 | Is j immediate predecessor to index? |
| 10 | Task j has not yet been traced |
| 11 | Push j onto stack |

| | | |
|---|---|---|
| 2 | 1 | flag j as on stack |
| 4 | 1 | Task j is currently on the stack |
| 5 | 1 | A cycle has been detected |
| 7 | 1 | Task j already traced – no action necessary |
| 8 | 1 | Task index had no untraced predecessors. The trace is complete for index, the top of the stack. Either index is a root or all predecessors of index have already been traced. For the second case, we wish to inherit precedence to index from all of its previously-traced predecessors. |
| 0 | 2 | j is potential predecessor of index |
| 1 | 2 | Is j an immediate predecessor of index? |
| 2 | 2 | k is potential predecessor task to j |
| 3 | 2 | Is k a predecessor to j? |
| 4 | 2 | index inherits precedence from j |
| 5 | 2 | remove index from top of stack |
| 6 | 2 | flag index as traced |

Table 25. Comments on Predecessor Trace and Cycle Detection

*mpreds* now contains the full precedence graph. All that remains is to copy this information into the object model. If *mpreds(i,j)* = 1 then task *j* is a predecessor of task *i*; add the object for task *j* to task *i*'s predecessor set.

7.3.2.4 Function CalcRPW

RPW scores are designed to facilitate one-at-a-time assignment strategies. As such, predecessors will always have a higher score than their successors. Note that each of the RPW scoring methods below exhibit inheritance of scores from successor tasks.

This pool of successor tasks from which to inherit is expanded to include indirect precedence relationships via set relations, i.e. task groupings. The tooling and eligible station constraints of the expanded successor pool are prioritized first, followed by shift time requirements.

First calculate the classic RPW score TVG.RPW by summing the shifttime of itself and all direct successor tasks.

```
1   For i = 1:N
2       i.RPW = i.Time_min * i.Vol
3       For j = 1:N
4           If j is a successor of i
5               i.RPW += j.Time_min * j.Vol
```

Figure 53. Computing Classic RPW

| LINE | COMMENTS |
|---|---|
| 1 | Task *i* to find RPW for |
| 2 | Shifttime of *i* |
| 4 | Shifttime of *j* |

Table 26. Comments on Classic RPW Computation

To calculate the set related RPW score TVG.AdjRPW, sum the shifttime of itself and all tasks that are extended successors.

```
1   For i = 1:N
2       For j = 1:N
3           If j is an extended successor of i
4               i.AdjRPW += j.Time_min * j.Vol
```

Figure 54. Computing Extended RPW

| LINE | COMMENTS |
|---|---|

| | |
|---|---|
| 1 | Task *i* to find AdjRPW for |
| 4 | Shifttime of *j* |

Table 27. Comments on Extended RPW Computation

To calculte the tooling RPW score, TVG.ToolRPW, first calculate the urgency score for each task individually, TVG.ToolWt and TVG.EligStWt. To calculate TVG.ToolWt:

```
1   For i = 1:N
2       latest = StationModel.Numstations
3       For each tool t required by i
4           location = t.laststation
5           If location < latest
6               i.ToolWt = StationModel.Numstations – location
7               latest = location
```

Figure 55. Computing Tool Urgency

| LINE | COMMENTS |
|---|---|
| 1 | Task *i* to find ToolWt for |
| 2 | tracks the latest station that satisfies tooling |
| 4 | last station tool *t* appears on |
| 5 | tool *t* is more urgent than any previously seen. Urgency score defined to be the number of stations at the end of the band that cannot satisfy this tool. Higher scores correspond to tools that last appear early in the band |
| 7 | update latest |

Table 28. Comments on Tool Urgency Computation

To calculate TVG.EligStWt a similar process is applied:

```
1   For i = 1:N
2       latest = 0
3       For each eligible station s
4           location = s.ID
5           If location > latest
6               i.EligstWt = StationModel.Numstations – location
7               latest = location
```

Figure 56. Computing Eligibility Urgency

| LINE | COMMENTS |
|---|---|
| 1 | Task *i* to find EligStWt for |
| 2 | tracks the latest station that is eligible |
| 4 | sequence number of station *s* |
| 5 | station *s* is later than any previously seen. Urgency score defined to be the number of stations at the end of the band that cannot satisfy station eligibility. Higher scores correspond to the latest eligible station appearing early in the band. |
| 7 | update latest |

Table 29. Comments on Eligibility Urgency Computation

Now the TVG.ToolRPW score may be calculated by taking the maximum urgency score amongst the task itself and all tasks that are extended successors.

```
1   For i = 1:N
2       i.ToolRPW = max(i.ToolWt, i.EligStWt)
3       For j = 1:N
4           If j is an extended successor of i
5               i.ToolRPW = max(i.ToolRPW, j.ToolWt, j.EligStWt)
```

Figure 57. Computing Tool and Eligibility RPW

| LINE | COMMENTS |
|---|---|
| 1 | Task *i* to find ToolRPW for |
| 2 | Maximum of *i*'s urgency scores |
| 4 | Take the most urgent score in extended successors. |

Table 30. Comments on Tool and Eligibility RPW Computation

The tasks are then sequenced with a 3-stage hierarchical sort. TVG.ToolRPW is the highest priority of the sort, followed by TVG.AdjRPW, and finally TVG.RPW.

# APPENDIX B:

## IP Model Technical Documentation

```
option gurobi_options 'presolve 2';
# SETS
set TVG;
set STATION ordered;
set MABR;
set PRBR;
set TOOL;
#PARAMETERS
param c > 0;                            #cycle time
param t {TVG} >= 0;                     #tvg time (direct)
param v {TVG} >= 0;                     #tvg volume
param v_max > 0;                        #max volume
param tbar {i in TVG} = t[i] * v[i] / v_max;     #tvg time, average piece
param b {TVG} > 0;                      #tvg prbr?
param Lmax {STATION} >= 0 integer;      #max headcount / station
param P {TVG,TVG} binary;               #precedence
param Ra {TVG,TVG} binary;              #adjacency
param Rst {TVG,TVG} binary;             #sametakt
param Rss {TVG,TVG} binary;             #samestation
param Rnt {TVG,TVG} binary;             #not sametakt
param Qu {TVG,TOOL} binary;             #tool need
param Qc {STATION,PRBR,TOOL} binary;    #tool support
param Am {MABR,STATION} binary;         #mabr access
param Ap {PRBR,STATION} binary;         #prbr access
param B {STATION,MABR,PRBR} binary;     #zoning compatibility
param w {TVG,TVG} binary;
#DECISION VARIABLES
var x {TVG,STATION,MABR} binary;        #tvg assignment
var y {STATION,MABR} binary;            #mabr active
#IMPLICIT DECISION VARIABLES
#var y {k in STATION,m in MABR} = if sum {i in TVG} x[i,k,m] > 0 then 1 else 0;
#OBJECTIVE
minimize HeadCount:  sum {k in STATION, m in MABR} y[k,m];
#CONSTRAINTS
Assign_all {i in TVG}:
   sum {k in STATION,m in MABR} x[i,k,m] = 1;
Cycle_time {k in STATION,m in MABR}:
   sum {i in TVG} x[i,k,m] * tbar[i] <= y[k,m] * c;
Assoc_per_station {k in STATION}:
   sum {m in MABR} y[k,m] <= Lmax[k];
Precedence {vv in STATION, i in TVG,j in TVG: P[i,j] = 1 and vv < last(STATION)}:
   sum {k in STATION, m in MABR: k > vv} x[j,k,m] <= 1 - sum {k in STATION, m in MABR: k <= vv} x[i,k,m];
Adj_or_Sametakt {i in TVG,j in TVG,k in STATION,m in MABR: Ra[i,j] = 1 or Rst[i,j] = 1}:
   x[i,k,m] = x[j,k,m];
SameStation {i in TVG,j in TVG,k in STATION: Rss[i,j] = 1}:
   sum {m in MABR} x[i,k,m] = sum {m in MABR} x[j,k,m];
NotSameTakt {i in TVG,j in TVG,k in STATION,m in MABR: Rnt[i,j] = 1}:
   x[i,k,m] + x[j,k,m] <= 1;
Tooling {i in TVG,k in STATION,o in TOOL}:
   Qu[i,o] * sum {m in MABR} x[i,k,m] <= Qc[k,b[i],o];
Access_MABR {i in TVG,k in STATION,m in MABR: Am[m,k] = 0}:
   x[i,k,m] = 0;
Access_PRBR {i in TVG,k in STATION: Ap[b[i],k] = 0}:
   sum {m in MABR} x[i,k,m] = 0;
Zoning_compatible {i in TVG,k in STATION,m in MABR: B[k,m,b[i]] = 0}:
   x[i,k,m] = 0;
Zone_assignment {i in TVG,j in TVG,k in STATION,m in MABR: w[i,j] = 1}:
   x[i,k,m] + sum {m2 in MABR: m2 <> m} x[j,k,m2] <= 1;
```

Figure 58. AMPL Model File of BIP Formulation

```
set TVG := 1 2 3;
set STATION := 1 2 3;
set MABR := 1 2 3 4 5;
set PRBR := 1 2 3 4 5 6 7 8 9;
set TOOL :=
   PNEUMATICRIVETGUN
   VEHICLEACCESS
   SUPERMARKETDELIVERY
   RVDOORSEALROBOT
   LVDOORSEALROBOT
;
param c := 104;
param Lmax :=
   1 2
   2 2
   3 2
;
param v_max := 342;
param: TVG:          t     v   b :=
       1  0.06  320  3
       2  1.5  320  9
       3  1.8  320  3
;
param P: 1 2 3 :=
     1 0 0 0
     2 1 0 0
     3 0 0 0
;
param Ra: 1 2 3 :=
     1 0 0 0
     2 0 1 0
     3 0 0 1
;
param Rst: 1 2 3 :=
     1 0 0 0
     2 0 0 0
     3 0 0 0
;
param Rss: 1 2 3 :=
     1 0 0 0
     2 0 0 0
     3 0 0 0
;
param Rnt: 1 2 3 :=
     1 0 0 0
     2 0 0 0
     3 0 0 0
;
param  Qu:  PNEUMATICRIVETGUN  VEHICLEACCESS  SUPERMARKETDELIVERY  RVDOORSEALROBOT
   LVDOORSEALROBOT :=
     1 0 1 0 0 0
     2 0 1 0 0 0
     3 0 1 0 0 0
;
param Qc :=
[*,*,PNEUMATICRIVETGUN]: 1 2 3 4 5 6 7 8 9 10 :=
   1 1 1 1 1 1 1 1 1 1
   2 0 0 0 0 0 0 0 0 0
   3 0 0 0 0 0 0 0 0 0
[*,*,VEHICLEACCESS]: 1 2 3 4 5 6 7 8 9 10 :=
   1 1 1 1 1 1 1 1 1 1
   2 1 1 1 1 1 1 1 1 1
   3 0 0 0 0 0 0 0 0 0
[*,*,SUPERMARKETDELIVERY]: 1 2 3 4 5 6 7 8 9 10 :=
   1 1 1 1 1 1 1 1 1 1
   2 0 0 0 0 0 0 0 0 0
   3 0 0 0 0 0 0 0 0 0
```

```
[*,*,RVDOORSEALROBOT]: 1 2 3 4 5 6 7 8 9 10 :=
   1 0 0 0 0 0 0 0 0 0
   2 0 0 0 0 0 0 0 0 0
   3 0 1 1 0 1 1 0 1 1 1
[*,*,LVDOORSEALROBOT]: 1 2 3 4 5 6 7 8 9 10 :=
   1 0 0 0 0 0 0 0 0 0
   2 0 0 0 0 0 0 0 0 0
   3 1 1 0 1 1 0 1 1 0 1
;
param Am (tr):
    1 2 3 4 5 6 :=
   1 1 1 1 1 1 0
   2 1 1 1 1 1 0
   3 1 1 1 1 0 0
;
param Ap (tr):
    1 2 3 4 5 6 7 8 9 10 :=
   1 1 1 1 1 1 1 1 1 1
   2 1 1 1 1 1 1 1 1 1
   3 1 1 1 1 1 1 1 1 1
;
param B :=
[1,*,*]: 1 2 3 4 5 6 7 8 9 :=
   1 0 0 1 0 0 1 0 0 1 1
   2 1 1 1 0 0 0 0 0 0 1
   3 1 0 0 1 0 0 1 0 0 1
   4 0 0 0 0 0 0 0 1 1 1 1
   5 0 0 0 1 1 1 0 1 0 1
   6 0 0 0 0 0 0 0 0 0 0 0
[2,*,*]: 1 2 3 4 5 6 7 8 9 :=
   1 0 0 1 0 0 1 0 0 1 1
   2 1 1 1 0 0 0 0 0 0 1
   3 1 0 0 1 0 0 1 0 0 1
   4 0 0 0 0 0 0 0 1 1 1 1
   5 0 0 0 1 1 1 0 1 0 1
   6 0 0 0 0 0 0 0 0 0 0 0
[3,*,*]: 1 2 3 4 5 6 7 8 9 :=
   1 0 0 1 0 0 1 0 0 1 1
   2 1 1 1 0 0 0 0 0 0 1
   3 1 0 0 1 0 0 1 0 0 1
   4 0 0 0 0 0 0 0 1 1 1 1
   5 0 0 0 0 0 0 0 0 0 0 0
   6 0 0 0 0 0 0 0 0 0 0 0
;
param w: 1 2 3 :=
    1 1 0 1
    2 0 1 0
    3 1 0 1
;
```

Figure 59. AMPL Data File Example

# APPENDIX C

## Prototype Software Documentation: Object Relationships

The purpose of this research is to develop methods to extract logical content from VRM and TAIS reports, and store the logical content in structure amenable for subsequent algorithmic manipulation. Subsequent methods are developed to filter the total information base into smaller subsets of interacting objects and constraints.

### 7.3.3 Context: Configuration Change Management

Technology and market conditions change through time, inducing changes to product offerings. Such changes are pushed by a Project Nachtrag (PN) change request, which prompts the configuration management (CM) team to investigate adaptations to operational policies to reflect the desired change. In practical terms, this entails making changes to the OKA and TAIS databases, which house the system constraints that control the customization choices available to the customer in the configurator interface, and the part allocation processes that create the bill of materials (BOM), respectively. Configuration change management requires finding a set of alterations to the OKA and TAIS databases that correctly maps to the intended change, and validating that the changes induced don't create unintended side effects, e.g. incorrect BOM for any particular vehicle configuration.

The first step in creating software support tools for the CM process is to render the information content of OKA and TAIS in suitable programmatic data structures, as a foundation for subsequent algorithmic approaches. A Boolean logic paradigm is taken for understanding and interpreting the total information content. Each OKA rule or TAIS

release line may be understood as a constraint that conditionally enforces relationships between objects. Section 7.3.4 briefly summarizes the Boolean logics applied and corresponding grammar. Section 7.3.5 describes the structure of the system information within database reports, parse methods for extracting this information, and the programmatic data structures used to hold the information.

Section 7.3.6 develops an algorithm for filtering the data set. This filtering method is referred to as the Model Refinement module in the high-level report ("Configuration Management Project Wrapper Report," Technical Report 2015-CEDAR-BMW-Configuration-000). The purpose of the method is to isolate interacting, tightly coupled subsets of rules and objects. It takes as input the full set of configuration information, representing all configuration literals and their relationship constraints, from the active data model or any of the sandbox models within the shell. From this complete set of information, a "small world" subset of objects and constraints is isolated. The user must initially specify a set of options or packages for investigation. The isolated subset is centered upon these options or packages, extending outward to related objects via relationship constraints. The degree to which relationships propagate outward is controlled by interaction depth, a user-chosen parameter.

### 7.3.4 Background: Boolean Logic

First-order Boolean logic is a mathematical model and formal grammar, used for reasoning about the truth of logical expressions. The grammar is composed of *operators* and *literals.* The operators used herein are the logical AND, OR, NOT, IMPLICATION, and BICONDITIONAL. Literals are Boolean objects that take either true or false values.

In this application, the option codes, model codes, parts, etc. are literals, as they must be either present for a configuration (true) or absent (false). *Expressions*, e.g. OKA rules, are formed from combining literals and operators. Each expression represents a *constraint*, the logical content of which must be satisfied by any valid configuration. Table 31 lists all components in Boolean expressions.

| *Expression component* | *Syntax* | *Example* |
|---|---|---|
| *AND* | & | (A & B) means "both A and B" |
| *OR* | / | (A / B) means "A or B" |
| *NOT* | ¬ or - | (-A) means "not A" |
| *IMPLICATION* | → | (A → B) means "if A, then B" |
| *BICONDITIONAL* | ↔ | (A ↔ B) means "A if and only if B" |
| *Literal* | <name> | S323A is itself |

Table 31. Components of Boolean expressions

If all literals within a Boolean expression are assigned a truth value, then the expression itself can be evaluated as either true or false. For example, if A and B are TRUE, then the expression (A / B) evaluates TRUE.

### 7.3.5 Constraint Construction

The scope of the configuration management problems under consideration includes conceptual objects (options, packages, etc.) and physical objects (parts). The constraints that relate these objects are encoded in two databases at BMW: 1) VRM, which contains information on the relationships between options, packages, and models, and 2) TAIS, which contains information on the relationships between parts and the conceptual objects. These databases are not directly accessible as data inputs. Instead, three standardized reports generated from the databases are used as input. Section 7.3.5.1 discusses the reintegration of data from these reports into a single, local database.

After extraction from reports, two top-level methods are applied to transform configuration constraint data into a programmatic data structures suitable for analysis and

experimentation, performed serially. First, the component fields from the reports are queried from the local database, and a Boolean expression string is written to represent each constraint. The process for constructing Boolean expressions varies for each type of constraint. Section 7.3.5.2 presents the details for this process across all constraint types. Next, as discussed in Section 7.3.5.3, the Boolean expressions are transformed from strings to binary tree data structures, to support subsequent analysis.

#### 7.3.5.1 Integration of BMW Data Sources

Data inputs are taken from three BMW system reports: the VRM report, the AG Usage report, and the TAIS report. Taken together, these reports contain all system-level constraints relevant to configuration. Each of these reports is cleansed, parsed and integrated into a single, local Access database. The purpose of the local database is to stabilize any inconsistencies arising during data acquisition. For example, many of the reports acquired by the team reflect slight differences in the model codes included, or inconsistencies due to differing dates of report generation. Hence, the local database should be considered an artifact of the decoupled development phase, which may be replaced by direct coupling to live data in a more mature future iteration. An ER diagram of the local database is provided in Figure 60. The relationships reflected in the ER diagram are correct insofar as they match the object patterns witnessed in available BMW reports.

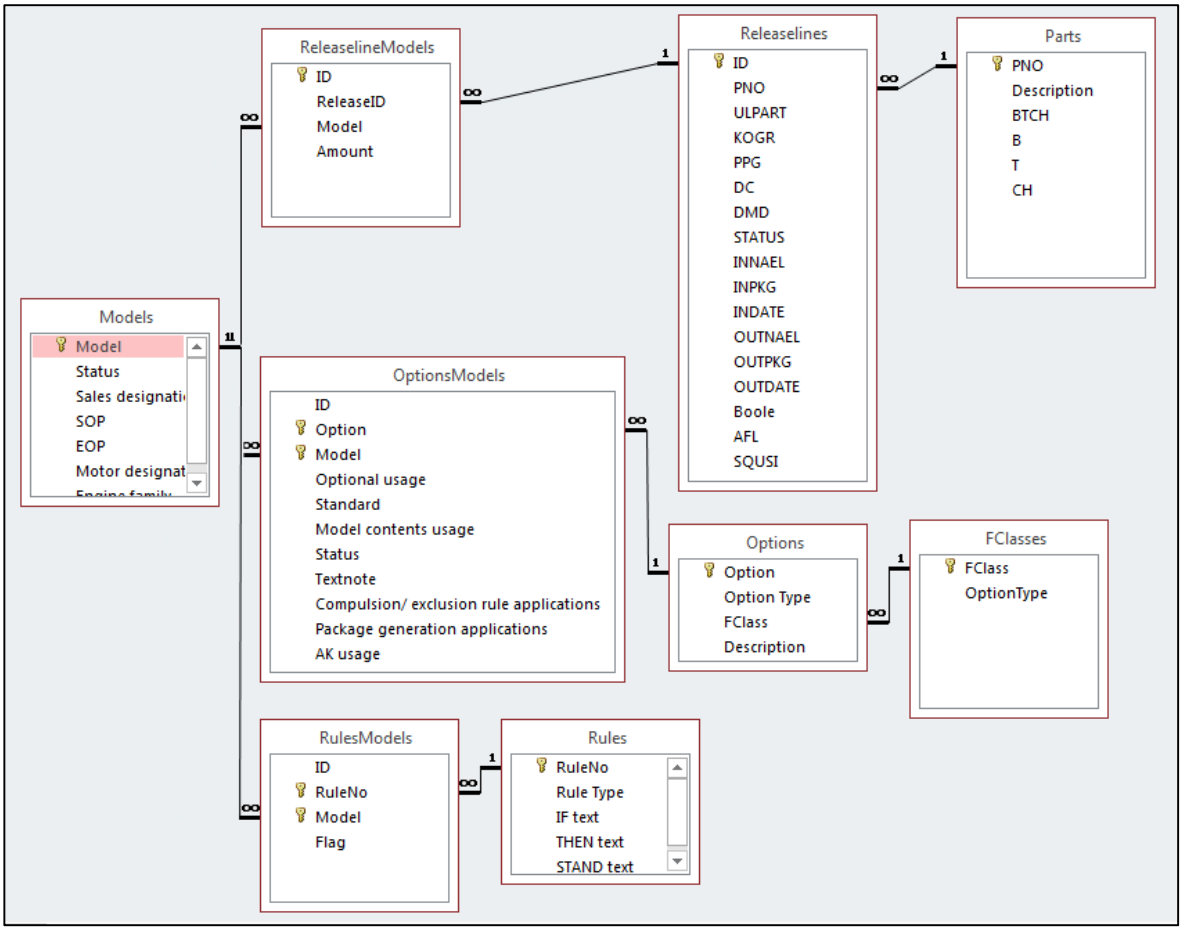


Figure 60. ER diagram for integrated VRM/AG/TAIS database

The VRM report is comprised of two sections: object declarations in the upper portion and OKA rules in the lower. The declarations section lists all options and packages, the models with which they are associated, and, if associated, the nature of the association (standard or optional). The AG usage report duplicates this content, as well being the sole source for FClass information for each option. The lower section of the VRM report delivers all OKA rules, and the models to which the each rule applies. The TAIS report is structured as a series of release lines. Each release allocates a single part,

with five conditions that control whether the release is activated for a vehicle. A summary of the source report for each database table is provided in Table 32.

| *Database Table* | *Source* | *Description* |
|---|---|---|
| *Models* | AG, VRM, TAIS | Model code. Models must appear in all reports for inclusion. |
| *ReleaselineModels* | TAIS | Linkage of releases to models. |
| *Releaselines* | TAIS | Release details. |
| *Parts* | TAIS | Part details. |
| *OptionsModels* | AG, VRM | Linkage of options to models. |
| *Options* | AG, VRM | Option details. |
| *FClasses* | AG | FClass membership of option. |
| *RulesModels* | VRM | Linkage of OKA rules to models. |
| *Rules* | VRM | OKA rule details. |

Table 32. Tables in integrated database

There is a small subset of configuration-relevant constraints that are known to exist, but are not explicitly expressed in BMW databases. These are paint, interior, and land relationships. Simply stated, there may only be one paint, interior, or land code on any vehicle. Further, every vehicle requires one paint and interior code. These constraints are implemented as special cases within the methods to process the BMW system reports.

#### 7.3.5.2 Boolean Expression Strings

##### 7.3.5.2.1 FClasses

FClasses (Feature Classes) are collections of related, mutually exclusive options. FClass membership is derived from the "Option group usage" field in the AG usage report. For example, the DAREL FClass contains 3 options related to roof rails: S3MCA, S3ATA, and S3AAA. Only one, if any, of these options may be active for a valid configuration.

The local database is initially queried to obtain the name and membership collection for each FClass. For each FClass, a Boolean expression is generated to

represent membership, via $FClass \leftrightarrow member_1/member_2/\ldots/member_n$. Next, a set of Boolean expressions is generated to enforce exclusion constraints between members, taking the form $member_i \rightarrow \neg member_j, \forall i \neq j$. Note that the exclusion expressions alone are insufficient, due to FClass names occasionally appearing in OKA rules. Without the membership expressions, any FClass name literals are undefined. The expressions generated for the DAREL example are DAREL ↔ S3MCA / S3ATA / S3AAA for membership, and S3MCA → ¬S3ATA & ¬S3AAA, S3ATA → ¬S3MCA & ¬S3AAA, and S3AAA → ¬S3MCA & ¬S3ATA for exclusion.

In addition to these explicit FClasses, the paint (L), interior (P), and land (LA) option types are treated as implicit FClasses, as only one option from each of these categories may be active on a valid configuration. Additional Boolean expressions are generated to enforce these exclusions, in the same fashion as for explicit FClasses above. Membership expressions are generated for the L and P option type codes as well. Though not strictly necessary, as nowhere else are the codes L or P referenced within any rule, these membership rules support a convenient mechanism to enforce the idea that that every vehicle has one paint and one interior. When appending two activation rules, ¬L → L and ¬P → P, this logic is achieved.

For type P interior options, additional membership expressions are generated for all 2-character (material only, no color) option codes, e.g. KC ↔ KCB4 / KCSW. These additional expressions support no-color upper level part allocation, used in section 7.3.5.2.3.

```
1  Function ParseClassRules(className As String, members As
   Collection, bMemberRules As Boolean, bExclusionRules As Boolean)
2      If bMemberRules Then
3          If members.count = 0 Then
4              strRuletext = "-"  & className
5          Else
6              For Each aMember In members
7                  If Len(RHS) = 0 Then
8                      RHS = aMember
9                  Else
10                     RHS = RHS & "/" & aMember
11                 End If
12             Next
13             strRuletext = className & "=(" & RHS & ")"
14         End If
15         aNode = New CBinNode
16         Call aNode.ParseExpression(strRuletext)
17         Call RuleForest.Add(aNode)
18     End If
19     If (bExclusionRules = True) And (members.count >= 2) Then
20         strRuletext = ""
21         For i = LBound(members.Array) To UBound(members.Array) -
   1
22             For j = i + 1 To UBound(members.Array)
23                 RHS = "-(" & members.Array(i) & "&" &
   members.Array(j) & ")"
24                 If Len(strRuletext) = 0 Then
25                     strRuletext = RHS
26                 Else
27                     strRuletext = strRuletext & "&" & RHS
28                 End If
29             Next
30         Next
31         aNode = New CBinNode
32         Call aNode.ParseExpression(strRuletext)
33         Call RuleForest.Add(aNode)
34     End If
35     Return RuleForest
36 End Function
```

Figure 61. Construction of Class Membership and Exclusion Boolean Expressions

7.3.5.2.2 OKA rules

The VRM report delivers OKA rules in a format that is nearly a Boolean expression already, needing only minor manipulation to achieve the desired form. Each rule is delivered as a logical implication, with the “IF” column containing the antecedent,

and the "THEN" column containing the consequent. Table 33 shows some example OKA rules for reference.

| Type | IF | THEN | STANDARD |
|---|---|---|---|
| Z | & + S205A & - P337A | / + S255A / + S2XAA / + S7XAA | + S240A |
| A | & + S212A | / + L801A | |
| PK | P7S2A | (!S255A) & (!S4CKA / S4ADA / S4B8A) | |

Table 33. Example OKA rules

As an initial step, each field of the OKA rule is cleansed to remove undesired characters. Using the example shown in Table 33, the leading operator strings "/" and "&" are removed from each component. The Boolean grammar adopted does not permit binary operators such as OR or AND to appear as leading characters. The "+" characters are simply removed, as our grammar assumes any literal is positive unless a negation operator is present. Lastly, all white space is removed from the string.

The Boolean expression for type Z rules is constructed by concatenation of the IF and THEN components with an IF operator, e.g. $antecedent \rightarrow consequent$. Ignoring the STANDARD column for a moment, the example Z rule in Table 33 becomes $S205A \,\&\, \neg P337A \rightarrow S255A \,/\, S2XAA \,/\, S7XAA$. Type A rules negate the consequent, and the concatenated Boolean string is of the form $antecedent \rightarrow \neg\, consequent$. The example A rule in Table 33 is interpreted $S212A \rightarrow \neg L801A$.

The STANDARD field in an OKA rule indicates options that are included by default during the customer configuration process. The customer may have opportunity to upgrade the standard option to an alternative, but at a price premium. This research is unconcerned with pricing or default configurations, only valid configurations. If upgrade alternatives exist for a standard option, then the alternatives are found in either the FClass

of the standard option or in the OKA rule that declares the standard. The process to transform OKA rules with standard options into Boolean expressions is as follows.

- Query whether the option belongs to any FClass. If so, replace the standard option with the name of its FClass. The Boolean expressions developed for FClass membership link to all alternative options.
- Append this resultant to the consequent of the OKA rule with an OR operator.
- Complete construction of the Boolean expression using the methods for Z/A rules.

For example, the Z rule in Table 33 contains S240A as standard equipment. S240A is not a member of any FClass. S240A is appended to the rule consequent, yielding $\boxed{\text{S255A / S2XAA / S7XAA / S240A}}$. The new consequent is finally merged with the antecedent to yield $\boxed{\text{S205A \& } \neg \text{ P337A} \rightarrow \text{S255A / S2XAA / S7XAA / S240A}}$.

PK rules differ from Z/A rules in root cause, being inspired by marketing instead of engineering purposes. However, the two types of rules are interpreted identically by this tool, with one caveat. PK rule consequents possess the "!"operator, which declares a definite Horn clause (multi-input XOR), such that exactly one of the literals within the parentheses must be true. The tool internally reduces each horn clause, transforming it into a logically equivalent set of AND and OR clauses. The Boolean expression for PK rules is of the form $\boxed{antecedent \rightarrow consequent}$, and the example PK rule in Table 33 becomes

$$\boxed{\text{P7S2A} \rightarrow \begin{array}{c}(\text{S255A}) \,\&\, (\text{S4CKA / S4ADA / S4B8A}) \,\&\, \neg(\text{S4CKA \& S4ADA}) \\ \&\, \neg(\text{S4CKA \& S4B8A}) \,\&\, \neg(\text{S4ADA \& S4B8A})\end{array}}$$

```
1  Function BuildBinTree_OKArule(rTYPE As String, rIF As String,
   rTHEN As String, rSTAND As String)
2      rIF = CleanseArtifactStrings(rIF)
3      rTHEN = CleanseArtifactStrings(rTHEN)
4      rSTAND = CleanseArtifactStrings(rSTAND)
5      If Len(rTHEN) > 0 And Len(rSTAND) > 0 Then
6          RHS = "(" & rTHEN & ")/($" & rSTAND & ")"
7      ElseIf Len(rSTAND) > 0 Then
8          RHS = "$" & rSTAND
9      Else
10         RHS = rTHEN
11     End If
12     If rTYPE = "A" Then
13         RHS = "-(" & RHS & ")"
14     End If
15     strRuletext = "(" & rIF & ")>(" & RHS & ")"
16     rootNode = New CBinNode
17     Call rootNode.ParseExpression(strRuletext)
18     Return rootNode
19 End Method
```

Figure 62. Construction of OKA Rule Boolean Expressions

7.3.5.2.3 TAIS Part Allocation Rules

There are several criteria to meet for a part to be allocated for a given configuration, all of which are sourced from the TAIS report. This report is organized as a series of release lines, with potentially several release lines allocating any part. The necessary condition to determine whether a part is allocated to a vehicle, then, is that *at least one* of the part's release lines is activated. The expression $\boxed{(release_1/\ release_2/\ \dots/\ release_n) \leftrightarrow part}$ encapsulates this logic in Boolean terms, where releases 1-n are all of the releases that allocate the part. It is undesirable to include each release as a literal, as this would flood the model with variables of little interest. Instead, each release is broken into the components that determine whether the release is activated. There are five conditions that collectively determine whether a release is

active, allocating the associated part. The conditions are found in the ULPART, BOOLE, AFL, Model code, and INDATE/OUTDATE columns of TAIS.

1) The vehicle model must be associated to the release. The Model Code column for the release must have a non-zero, non-blank quantity. The value of the numbers in this column are not considered. The method considers only whether a part is allocated, and does not consider the quantity allocated.
2) If the release has a Boolean condition, it must be satisfied. These conditions are located in the TAIS column labeled 'Boole' or 'Effective_SA'. With the exception of spring constraints, these are already in propositional Boolean algebra form.
3) If the release has an AFL (interior) condition, it must be satisfied. These conditions are located in the TAIS column labeled 'AFL'. Note that upper no-color parts have only 2-character AFL codes, corresponding to the material of the interior only. The derivation of 2-character AFL codes is described in section 0.
4) If the release has an upper-level part, then that part must also be allocated for the vehicle. These conditions are located in the TAIS column labeled 'ULPART'
5) The current date must be inside the release's activity window, as determined from the TAIS columns INDATE and OUTDATE.

A release must meet all five conditions to be active. Putting these pieces together builds a Boolean expression to determine release activation:

$$release \leftrightarrow (MC > 0)\ \&\ AFL\ \&\ BOOLE\ \&\ ULPART\ \&\ (INDATE < now < OUTDATE)$$

Substituting this expression in the part allocation expression resolves the issue of release literals in the SAT. Table 34 shows some example TAIS data to illustrate this process.

| ID | PNO | ULPART | Boole | AFL | KR01 |
|---|---|---|---|---|---|
| r1 | 1180625 | | (S300A) | | 1 |
| r2 | 2907905 | 2907904 | | LCSW | 1 |
| r3 | 2907905 | 2907904 | | AVAT KCSW LCSW | |
| r4 | 2907904 | | NOT(S6NSA/S775A/S776A) | KC LC NA | 1 |

Table 34. Example TAIS excerpt

Using the example data in Table 34, the following release activation expressions are constructed, assuming the model KR01 is active and the INDATE/OUTDATE time window is active for all releases:

$$\boxed{\text{r1} \leftrightarrow (true)\ \&\ \text{S300A}}$$

$$\boxed{\text{r2} \leftrightarrow (true)\ \&\ \text{LCSW}\ \&\ 2907904}$$

$$\boxed{\text{r3} \leftrightarrow (false)\ \&\ (\text{AVAT}\ /\ \text{KCSW}\ /\ \text{LCSW})\ \&\ 2907904}$$

$$\boxed{\text{r4} \leftrightarrow (true)\ \&\ (\text{KC}\ /\ \text{LC}\ /\ \text{NA})\ \&\ \neg(\text{S6NSA}\ /\ \text{S775A}\ /\ \text{S776A})}$$

Assuming that there are no other releases except those shown, substituting these release activation expressions yields the following part allocation expressions.

$$\boxed{1180625 \leftrightarrow (true)\ \&\ \text{S300A}}$$

$$\boxed{2907905 \leftrightarrow ((true)\&\text{LCSW}\&2907904)\ /\ ((false)\&(\text{AVAT}/\text{KCSW}/\text{LCSW})\&2907904)}$$

$$\boxed{2907904 \leftrightarrow (true)\ \&\ (\text{KC}\ /\ \text{LC}\ /\ \text{NA})\ \&\ \neg(\text{S6NSA}\ /\ \text{S775A}\ /\ \text{S776A})}$$

The Boolean expression strings for parts are created through serial application of two methods, each representing one conditional half of the biconditional expression. The first method, shown in Figure 63, constructs an expression to require that the part is allocated if the release conditions are met. The first condition checks INDATE and OUTDATE of the release. If the release is out of date, then no expression is created. Else, each subsequent condition is appended into a single Boolean expression held in the variable `LHS`, joined with the "&" character to ensure that all conditions must be met. The AFL condition must be specially formatted before it is appended, as AFL conditions list a set of interior codes, any of which suffice. After concatenating all conditions, the Boolean expression is completed by setting "LHS > PNO," such that if all conditions are

met, then the PNO is allocated. If there are no conditions at all, then the part is series, and must be allocated for every vehicle. The expression in this case is simply "PNO".

```
1  Function BuildBinTree_TAISrelease(rPNO As String, rULPART As
   String, rAFL As String, rBOOLE As String, rSTART as String, rEND
   as String)
2      If (rSTART > Now) Or (rEND < Now) Then Exit Function
3      LHS = rULPART
4      rAFL = FormatAFLString(rAFL)
5      If Len(rAFL) > 0 Then
6          If Len(LHS) = 0 Then
7              LHS = rAFL
8          Else
9              LHS = "(" & LHS & ")&(" & rAFL & ")"
10         End If
11     End If
12     rBOOLE = FormatBooleString(rBOOLE)
13     If Len(rBOOLE) > 0 Then
14         If Len(LHS) = 0 Then
15             LHS = rBOOLE
16         Else
17             LHS = "(" & LHS & ")&(" & rBOOLE & ")"
18         End If
19     End If
20     If Len(LHS) = 0 Then
21         strRuletext = rPNO
22     Else
23         strRuletext = "(" & LHS & ")>" & rPNO
24     End If
25     rootNode = New CBinNode
26     Call rootNode.ParseExpression(strRuletext)
27     Return rootNode
28 End Function
```

Figure 63. Construction of TAIS release Boolean expressions

The next step is to create Boolean conditions to enforce the other half of the biconditional, so that parts cannot be allocated unless one of the releases is active. The method for enforcing this logic is shown in Figure 64. The passed parameter `PartReleaseRules` contains the root nodes for all of the constraints constructed in the prior method. First, if there are no releases for a part, then that part must not be allocated on any vehicle, and the expression is simply "-PNO". Similarly, if any release

for the PNO is series, then the part must be allocated, and the expression is "PNO". Otherwise, release conditions are collected from the left child of each release's root node, concatenated with an OR between each release, and stored in the method-level variable RHS. The Boolean expression string is then "PNO > RHS," which enforces that if the PNO is allocated, then at least one of the releases must have fulfilled conditions.

```
1  Function BuildBinTree_TAISpart(rPNO As String, PartReleaseRules
   As Collection)
2      If PartReleaseRules.count = 0 Then
3          strRuletext = "-" & rPNO
4      Else
5          For Each root In PartReleaseRules
6              If root.Oper = "VAR" Then
7                  strRuletext = rPNO
8                  Exit For
9              ElseIf root.Operator = ">" Then
10                 bNode = root.LeftChild
11                 If Len(RHS) = 0 Then
12                     RHS = "(" & bNode.PrintExp & ")"
13                 Else
14                     RHS = RHS & "/(" & bNode.PrintExp & ")"
15                 End If
16                 strRuletext = rPNO & ">" & RHS
17             End If
18         Next
19         End If
20     End If
21     rootNode = New CBinNode
22     Call rootNode.ParseExpression(strRuletext)
23     Return rootNode
24 End Function
```

Figure 64. Construction of TAIS part Boolean expressions

7.3.5.3 Parsing: Boolean Expression Strings to Binary Parse Trees

Configuration constraint information from BMW reports is delivered in string form. Parsing is a computational process that extracts the semantic content of these strings, and stores the content in an internal (memory-resident) data structure. The purpose of the parsing approach is twofold. First, parsing validates the grammar used in

each constraint string. If a constraint is e.g. missing a parenthesis, then the parsing method will log the error. Second, semantic content is notoriously difficult to analyze, transform, or otherwise manipulate when stored in a string representation. The data structures used are designed to support these tasks.

To review, Boolean expressions are composed of a series of *literals* and *operators*, arranged in a grammatical structure. Literals are objects that may take either true or false values. In this project, each individual part, option, package, and FClass is rendered as its own literal. For any single configuration, each of these objects is either true (included in the configuration) or false (absent). Operators are logical functions like AND, OR, or NOT. Each operator has either one or two *operands*, sometimes called inputs or arguments. *Binary* operators require two operands, e.g. the AND operator in the expression $\boxed{A \,\&\, B}$ has operands *A* and *B*. Binary operators appear between their operands in text expressions. *Unary* operators have only one operand, e.g. the NOT operator in the expression $\boxed{\neg C}$ has operand *C*. Unary operators appear before their operand in text expressions.

Section 7.3.5.3.1 describes the parse tree data structure used to hold Boolean constraint information. Section 7.3.5.3.2 describes the parsing algorithm for creating a parse tree from each Boolean expression string. The Boolean expression strings used as parse input are created from BMW sources by the methods described in section 7.3.5.2.

7.3.5.3.1 Binary Parse Tree

Boolean expressions are held in a *binary parse tree* data structure, which emulates the semantics of Boolean grammar. Trees are hierarchically arranged node networks,

with one (mandatory) root node. Links between node pairs indicate a relationship between the nodes. The tree is *binary*, meaning that each node may have up to two child nodes. When two nodes are linked, the higher-level (i.e. closer to the root) node is the *parent*, and the lower-level node is the *child*. The nodes that populate the tree each represent either a literal or an operator from the Boolean expression modeled.

The child node(s) of an operator node are its operand(s). As all operators used are either unary or binary, no operator may be a *leaf node*, i.e. a node with zero children. All leaf nodes must be literals, and all literals must be leaf nodes. The tree form of $A \,\&\, B$ would have an AND root node, with one child node *A* and another child node *B*. The tree form of $\neg C$ would have a NOT root node, with one child node *C*. See Figure 63 for examples of simple expressions rendered as binary trees.

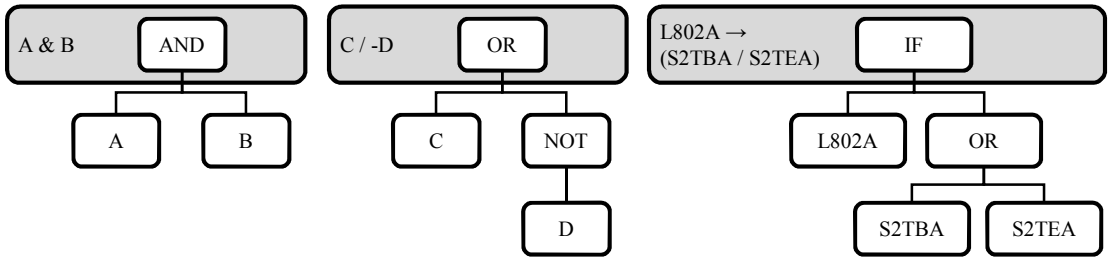


Figure 65. Example constraint strings and corresponding rule trees

Trees have any arbitrary depth, determined by the complexity of the expression modeled. For example, the expression $(\mathrm{A} \,\&\, \mathrm{B}) \,/\, (\mathrm{C} \,\&\, D) \rightarrow (E/F/G) \,\&\, (H/I \,\&\, (J/K))$ has depth of 5 on its deepest branch.

An object-oriented programming approach is used to model the binary tree data structures. Instances of class CBinNode represent each node, with properties, methods, and functions as shown in Table 35.

| *Characteristic* | *Type* | *Description* |
|---|---|---|
| *.LChild* | Property | Link to left child CBinNode instance (if exists) |
| *.RChild* | Property | Link to right child CBinNode instance (if exists) |
| *.Oper* | Property | Operator type |
| *.Varname* | Property | If node is a literal, name of the literal |
| *.RuleNo* | Property | Constraint source name (e.g. "ReleaseLine 21545") |
| *.RultCat* | Property | Constraint source type (e.g. "OKA_PK") |
| *.ParseExpression (etext)* | Method | Constructs tree from passed expression *etext* |
| *.TransformToCNF* | Method | Transforms tree to CNF |
| *.AllLiterals* | Function | Returns an dictionary containing all literals in tree |
| *.CopyNode* | Function | Returns an object copy of node |
| *.PrintNode* | Function | Returns text form of expression |

Table 35. Class CBinNode

The *.Oper* property is the logical operator modeled by the node. There are 8 operator types, shown in Table 36. Operator type determines the number of children for the node, as logical operators have a fixed number of operands. Note that the *EXC* and *STAN* operator types refer to BMW-specific constraint properties.

| *Operator* | *Symbol* | *#Children* | *Logical Description* |
|---|---|---|---|
| *VAR* | <name> | 0 | Literal |
| *NOT* | - | 1 | NOT (LChild) |
| *AND* | & | 2 | (LChild) AND RChild) |
| *OR* | / | 2 | (LChild) OR (RChild) |
| *IF* | > | 2 | IF (LChild) THEN (RChild) |
| *IFF* | = | 2 | (LChild) IFF (RChild) |
| *EXC* | ! | 1 | LChild is embedded in PK exclusion clause |
| *STAN* | $ | 1 | LChild is embedded in OKA Z/A standard clause |

Table 36. Operator types of class CBinNode

The *EXC* node type denotes that the child and all following lower-level nodes are within a package PK definite horn clause, e.g. (!A / B / C). These clauses define that one and only one of the parenthesized literals may be chosen. The *EXC* node represents the "!" in these expressions. All lower-level nodes must be of type *OR* or *VAR*.

Then *STAN* node may only appear for a Z or A OKA rule. It denotes that the child and any lower-level nodes are within the Standard column of the rule in VRM.

Only root nodes are instanced and retained at the outer programming level. Lower-level nodes instances are retained as links (.LChild and .RChild properties), chained from the root. After instancing the root for a constraint, the *rootnode.ParseExpression* method is called to construct the tree, passing as argument the Boolean expression text for the constraint.

7.3.5.3.2 Parsing Algorithm

A recursive descent parse is applied in method *.ParseExpression*. It requires a passed *etext* argument, which is the string expression of the Boolean constraint modeled. The method will define all local properties of the node, and then recursively define children nodes.

Figure 66 summarizes the parsing algorithm. Several string manipulation functions are used within, briefly described in Table 37. Note that there are three non-overlapping categories of characters that may appear in a valid Boolean expression: operator strings, parenthesis open and close, and alphanumeric characters used by literals. Using these categories, the algorithm seeks to break *etext* into 3 separate pieces: the operator of the current node (stored in Me.Oper), the text for the left child to inherit (stored in *ltext*), and the text for the right child to inherit (stored in *rtext*). After storing the operator in the local node, any needed child nodes are instanced and recursed upon, by calling *ParseExpression* for their instances and passing *ltext* or *rtext*.

The algorithm initially strips all fully-spanning parentheses from the *etext* expression string, removes all empty characters, then analyzes the leftmost character

(*char1*). In any valid Boolean expression, *char1* must be either a unary operator, a parenthesis open, or the beginning of a literal name (alphanumeric).

7.3.5.3.2.1 Unary operator

Unary operators include the negation operator ("-"), the mutual exclusion operator ("!"), and the standard operator ("$"). If *char1* is a unary operator, the parse method searches for the end of the unary operand clause. If the clause comprises the entirety of *etext*, then the local node becomes an operator node of the unary type. The remainder of *etext,* other than the leading operator character, is stored in *ltext*, and *rtext* is empty. For example, if $etext = "-(\mathrm{A}\ \&\ \mathrm{B})"$, then $char1 = "-"$. The negated clause begins with the open parenthesis. The clause ends with the end parenthesis, which is the final character in *etext*. The local node becomes a NOT node, $ltext = "(\mathrm{A}\ \&\ \mathrm{B})"$, and *rtext* is the empty string. The left child node is instanced, and recursion begins, passing *ltext*.

If the unary operand clause does not fully span *etext*, then the character following the clause's end must be a binary operator. The local node becomes an operator of the same type, with *ltext* containing all text left of the binary operator, and *rtext* all text to the right. For example, if $etext = "-(\mathrm{A}\ \&\ \mathrm{B})\ /\ (\mathrm{C}\ \&\ D)"$, then $char1 = "-"$. The negated clause begins with the first open parenthesis. The clause ends with the first end parenthesis, at position 6 in *etext*. Since *etext* is 12 characters long, the unary operand does not fully span. The character at position 7, an OR symbol, determines the operator of the local node. The left and right children are both instanced, and recursion begins with $ltext = "-(\mathrm{A}\ \&\ \mathrm{B})"$ and $rtext = "(\mathrm{C}\ \&\ \mathrm{D})"$.

7.3.5.3.2.2 Parenthesis open

If *char1* is a parenthesis open, then there must be a binary operator after the corresponding parenthesis close. All fully-spanning parentheses are stripped initially, so examples like $\boxed{(\text{A \& B})}$ cannot occur. The next character after the corresponding parenthesis close becomes the operator for the local node. All text to the left of this operator is stored in *ltext*, and text to the right of the operator is stored in *rtext*. For example, if $\boxed{etext="(\text{A \& } B) \ / \ (C \ \& \ D)"}$, then the central OR becomes the operator for the local node. Recursion begins with $\boxed{ltext = "(\text{A \& } B)"}$, and $\boxed{rtext = "(\text{C \& } D)"}$.

7.3.5.3.2.3 Literal name

If *char1* is the beginning of a literal name, the parse method searches for the end of the literal name by finding the next character that is not alphanumeric. If the literal name comprises the entirety of etext, then the local node becomes type VAR, and the literal name is stored in the local property varname. Recursion terminates in such leaf nodes, and both children are null.

If the literal name does span the entirety of *etext*, then the character following the literal name must be a binary operator. The local node becomes an operator node of this type. All text to the left of this operator is stored in *ltext*, and text to the right of the operator is stored in *rtext*. For example, if $\boxed{etext="\text{A \& B}"}$, then the local node becomes the central AND operator. The left and right children are both instanced, and recursion begins with $\boxed{ltext = "\text{A}"}$ and $\boxed{rtext = "\text{B}"}$.

| *Function* | *Input* | *Output* | *Description* |
|---|---|---|---|
| *Len(text)* | *text*: "A&(B/C)" | 7 | Returns number of characters in *text.* |
| *Left(text,length)* | *text*: "A&(B/C)"<br>*length*: 1 | "A" | Returns left-most characters from *text* of length *length.* |
| *Right(text,length)* | *text*: "A&(B/C)"<br>*length*: 1 | ")" | Returns right-most characters from *text* of length *length.* |
| *Mid(text,start,length)* | *text*: "A&(B/C)" | "B/C" | Returns middle characters from *text* beginning at index |

| | | | |
|---|---|---|---|
| | *start*: 4<br>*length*: 3 | | *start*, of length *length.* |
| *IsCharAnOperator(char)* | *char*: "-" | TRUE | Returns whether *char* is an operator string |
| *ParenStrip(text)* | *text*: "((A/B))" | "A/B" | Removes all outer, fully-spanning parentheses from string. |
| *ParenEnd(text,start)* | *text*: "A&(B/C)"<br>*start*: 3 | 5 | Returns index number of ")", matching to the "(" indicated by argument *start.* |
| *CodeEnd(text,start)* | *text*: "S323A"<br>*start*: 1 | 5 | Returns index number of terminus of literal name beginning at index *start.* |
| *ClauseEnd(text,start)* | *text*: "(A/B/C)&D"<br>*start*: 1 | 7 | Returns index number of terminus of clause beginning at index *start.* A clause is either a parenthetical statement or a literal name. |

Table 37. Parse support functions

```
1   Method ParseExpression
2   etext = ParenStrip(etext)
3   char1 = Left(etext, 1)
4   Select char1
5       Case "!"
6           Me.Oper = "!"
7           ltext = Right(etext, Len(etext) – 1)
8           rtext = ""
9       Case "$"
10          subtext = Right(etext, Len(etext) - 1)
11          cNext = ClauseEnd(subtext, 1)
12          If cNext = Len(subtext) Then
13              Me.Oper = "$"
14              ltext = subtext
15              rtext = ""
16          Else
17              Me.Oper = Mid(etext, cNext + 2, 1)
18              ltext = Left(etext, cNext + 1)
19              rtext = Right(etext, Len(etext) - Len(ltext) - 1)
20          End If
21      Case "-"
22          subtext = Right(etext, Len(etext) - 1)
23          cNext = ClauseEnd(subtext, 1)
24          If cNext = Len(subtext) Then
25              Me.Oper = "-"
26              ltext = subtext
27              rtext = ""
28          Else
29              Me.Oper = Mid(etext, cNext + 2, 1)
30              ltext = Left(etext, cNext + 1)
31              rtext = Right(etext, Len(etext) - Len(ltext) - 1)
32          End If
33      Case "("
34          cNext = ParenEnd(etext, 1)
35          Me.Oper = Mid(etext, cNext + 1, 1)
36          ltext = Mid(etext, 2, cNext - 2)
37          rtext = Right(etext, Len(etext) - cNext - 1)
38      Case Else
39          cNext = CodeEnd(etext, 1)
40          If cNext = Len(etext) Then
41              Me.Oper = VAR
42              Me.Varname = etext
43          Else
44              Me.Oper = Mid(etext, cNext + 1, 1)
45              ltext = Left(etext, cNext)
46              rtext = Right(etext, Len(etext) - cNext - 1)
47          End If
48  End Select
49  If Len(ltext) > 0 Then
50      Lchild = New CBinNode
51      Call Lchild.ParseExpression(ltext)
52  End If
53  If Len(rtext) > 0 Then
54      Rchild = New CBinNode
55      Call Rchild.ParseExpression(rtext)
56  End If
57  End Method
```

Figure 66. Recursive Descent Parse Algorithm

### 7.3.6 Local Object Interactions

Change initiatives are motivated by a change *intention*, which states in words the desired end state of the configuration space after application of the change. The intention is a vision of some future state of affairs, where the consumer is offered new technological or aesthetic choices for vehicle customization. Implementation of a change requires altering system constraints. New options or packages may be offered for existing models, or may be removed. The set of models available may be updated to add or remove items. Relations between options, packages, and models may be altered. Further, for any change, the correct parts must be allocated to valid configurations. Validation of a set of changes requires checking that altered system constraints correctly map to the intended outcome.

Importantly, change intentions are typically observed to be oriented toward modifying particular vehicle subsystem(s). Disparate, unrelated changes are not managed under a single change initiative. Instead, when such cases arise, each unrelated change would be managed under a separate change initiative. The conceptual reliance on un-relatedness, or independence between subsystems and change initiatives, motivates the following definition of an *interaction* between objects.

Definition: Objects are interacting (with degree-1) if they are mutually present in an OKA rule.

OKA rules encode constraints that forbid certain combinations of object Boolean values. For example, the rule $\boxed{\text{S205A \& } \neg\text{P337A} \rightarrow \text{S255A / S2XAA / S7XAA}}$ forbids any of the options in the consequent if the configuration in the antecedent is true. These objects

are related by this dependence, and are defined to interact. The degree of interaction between these objects is 1, as they are directly related by this OKA rule.

Of course, objects may be present in more than one OKA rule. Complexly entwined subsystems typically encode constraints across an array of rules, such that each individual rule encapsulates only one facet of the larger interaction. This distinction is arbitrary from a purely constraint-logical point of view. That is, to say that every valid configuration must obey all OKA rules is identical to saying that all object values must satisfy the conjunction of OKA rule Boolean expressions. To a human reader, however, the complete set of conjoined OKA rules would be an opaque, incomprehensible mess. Hence, several separate rules are used to encode a complex concept. Under this approach, rules cannot be assumed to be truly independent from all other rules, as sometimes rule batches are used to divide up complex interactions into chunks manageable to a human reader.

A degree-*N* interaction exists between two objects if there exists a path of length *N* between the objects, where each link in the path is a degree-1 interaction. For example, consider a batch of OKA rules such as $\boxed{\mathrm{A} \,\&\, \mathrm{B} \rightarrow \mathrm{C}}$ and $\boxed{\mathrm{C} \,/\, \mathrm{D} \rightarrow \mathrm{B} \,\&\, \mathrm{E}}$. *A* and *C* interact with degree-1. *C* and *D* interact with degree-1. The interaction between *A* and *D* is degree-2, as there is a set of two degree-1 linkages between *A* and *D*. Further, there may be several different interaction levels between objects. In the example above, *A* and *B* interact at degree-1 (directly from the first rule), and also at degree-2 (from the path *A* to *C* in the first rule, and then *C* to *B* in the second rule). In these cases, which are many, only the smallest interaction level is retained between two objects. As OKA rule

construction exhibits logically arbitrary divisions to support human comprehension, this preference enforces that only the tightest coupling between objects is considered.

To support the investigation task required by validation, an algorithmic method is designed to drill down into the complex whole of system constraints, to isolate subsets of constraints that are tightly coupled to some objects of interest. The user supplies two inputs. The first is a set of objects to be investigated, which represent the objects involved in the subsystem in the change initiative. The second input is a depth parameter, which controls determining the depth of interaction to include around those objects. The algorithm finds all objects that have interaction depth less than or equal to the depth parameter, relative to the source objects. This subset of interacting objects, along with the OKA rules that define the interactions, are returned from the algorithm. This output defines the *small world* of objects and constraints that are likely to require modification or investigation during the validation process.

Figure 67 presents the algorithm. The `options` parameter is the set of starting options to be investigated, and the `depth` parameter is the desired interaction depth. The first step is to acquire the set of all OKA rules that contain any object in `options`, which are parsed into binary trees and stored locally in the `RuleForest` structure. This initial set of OKA rules is the smallest possible small world surrounding the initial objects. Any degree-*N* interaction to the initial objects must have a path that begins in this initial set of OKA rules. Next, a loop begins iterating from 1 to `depth`. During each iteration, both the `options` and `RuleForest` data structures will grow to include new objects/rules found at the depth level. During a single iteration, first all objects are

extracted from the subset of rules currently in `RuleForest`, by invoking the rule.AllObjects procedure. All objects found are appended to the batch of interacting `options`. Then, `RuleForest` is updated to append any rules which contain the new objects just-added to `options`. The loop then continues to the next iteration. Upon completion, the function returns the `RuleForest` data structure. The set of objects participating in these rules may later be extracted by invocation of the rule.AllObjects procedure, if desired.

```
1  Function InteractingConstraints(options, depth)
2
3      RuleForest = ParseOKARules(options)
4      For i = 1 To depth
5          For Each rule In RuleForest
6              options = Union(options, rule.AllObjects)
7          Next
8          RuleForest = ParseOKARules(options)
9      Next
10     Return RuleForest
11 End Function
```

Figure 67. Interaction Search Algorithm

# APPENDIX D

## Prototype Software Documentation: Conflict Detection

Support is provided to configuration management (CM) through development of algorithmic methods to detect potentially faulty object relationships within the VRM and TAIS databases. These methods are intended for use by launch and change control, to validate system changes induced by a Project Nachtrag (PN). A suite of distinct anomaly/conflict events is established to classify different failure modes, with accompanying Boolean algebra proof-checking approaches for each.

### 7.3.7 Context: Option Change Management

PNs entail altering system constraint located in the OKA and/or TAIS databases. Before implementation, investigation is required to validate whether the proposed changes will induce problems. This research develops a tool to support this validation process, which operates by first establishing a class of potential failure mode, then applying satisfiability methods to detect whether any of the failure modes may occur. The following list briefly summarizes the failure modes captured by the tool.

1. "Rule conflict." Is there a subset of two or more VRM rules such that no possible configuration may satisfy them?
2. "Object activation." Can all options/parts/etc. that are declared as being available for selection actually be selected?
3. "Antecedent satisfiability." Are there any rules for which the antecedent (IF-part) of the rule cannot be satisfied? If so, then the effects of the rule are inconsequential, as the rule is never active.
4. "Implicit relationships." Are there any binary inclusion/exclusion object relationships that are implicitly enforced, through the collected effects of explicit constraints?

5. “Part family allocation.” For a given family of alternative parts (e.g. all windshields), will one (and only one) of the parts be allocated for every configuration?
6. “Part family matching.” Consider a suite of several part families, some of which are intended to match to others for geometry or color reasons. Are the rules correctly implemented, or is there a configuration that mismatches parts?

The tool takes as input a set of constraints derived from TAIS and VRM, which relate part, option, FClass, model, and package objects. These constraints are collected by methods described in report (“Configuration Management Model Refinement,” Technical Report 2015-CEDAR-BMW-Configuration-005). The tool makes no assumptions about whether PN changes have been implemented within this source data. As described in the (“Configuration Management Project Wrapper Report,” Technical Report 2015-CEDAR-BMW-Configuration-000), any version of constraint set data may be checked with this tool, from the active model or any sandbox model. The system state pre-change may also be checked, if desired, as it will presumably (but not necessarily) contain no conflicts.

Section 7.3.8 presents an overview of Boolean satisfiability techniques used to check for each conflict class. In section 7.3.9, each conflict class is described in detail, along with the test protocols used to search for each. Section 7.3.10 describes the technical steps for preparing constraint data for satisfiability testing.

### 7.3.8 Background: Satisfiability

First-order Boolean logic provides techniques for reasoning about logical expressions. These expressions are constructed as a series of operators and literals, assembled according to a formal grammar. Literals are Boolean objects, which may take

values of either TRUE or FALSE. Within this project, objects like options, parts, etc. are literals, as they are either present on a vehicle instance (TRUE) or not (FALSE). Operators are functions like OR, AND, and NOT, used to conjoin literals into expressions that represent system constraints.

If all literals are assigned a truth value, then a Boolean expression containing them may be resolved to either true or false. If, on the other hand, some or all of the literals are unassigned, then the truth of the expression may be unresolved. Indeed, within CM literals do not typically take explicit values, as it is the discretion of the customer to choose which options, etc. are chosen for the vehicle. The task for CM is to manage the set of system constraints, such that all valid, user-selectable configurations result in correctly specified, buildable vehicles. When working with Boolean expressions that contain unspecified literals, a pertinent question may be "Is there any set of true/false values for literals that results in the expression resolving to true?" This question is known as the Boolean satisfiability problem (SAT). SAT approaches have been successfully applied in non-automotive sectors, for problems ranging from software configuration validation, electronic circuit design validation, and mathematical proof-checking.

Solvers for the SAT problem have been developed by several independent researchers. These solvers take as input a set of Boolean expressions, and search for some combination of literal values that will "satisfy," i.e. result in all expressions evaluating true. For this work the MiniSAT solver is adopted, available at http://minisat.se/. If the solver finds that no possible combination of literal values can

satisfy all expressions, then the solver returns UNSATISFIABLE. Otherwise, the solver returns SATISFIABLE along with the satisfying values found for each literal. There may be many different sets of satisfying literal values; the solver returns only the first set found within the search.

SAT does not strictly require that all literals be unspecified at the start. Partially configured vehicles are discussed in the conflict detection methods used below. A partially specified SAT problem is accomplished by appending an additional expression for each literal that has an initial value. For example, appending the expression "-A" will force solutions where A=false.

The research questions in section 7.3.3 are addressed by a general strategy of constructing a SAT problem instance (or suite of parallel SAT instances) that is(are) equivalent to a conflict. The SAT problem is then evaluated via the SAT solver. If the solver finds a solution, then this means that there exists a set of literal true/false values with which the conflict emerges.

#### 7.3.8.1 BASESAT Constraint Set

Prior to testing for conflicts, first all system-level constraints, e.g. OKA and TAIS rules, are collected. See ("Configuration Management Model Refinement," Technical Report 2015-CEDAR-BMW-Configuration-005) for description of the methods for constructing Boolean expressions from TAIS and VRM system information. This collection of constraints is herein referred to as the BASESAT, a set of constraints must be obeyed for any valid vehicle, regardless of the details of any particular experiment.

During the conflict detection routines, additional constraints are appended to BASESAT, as necessary for construction of the conflict conditions. Section 7.3.9 provides detailed descriptions for construction of the appended constraints for each conflict class. After performing a conflict test, it is necessary to remove the appended test constraints from the constraint pool before beginning another test. For this reason, the BASESAT is tracked independently, so that the state of the constraint pool may be restored after each test.

### 7.3.9 Methods

The following subsections describe independent experimental tests to check for potential conflicts, matching to the research questions posed in the section 7.3.3. The general methodology for each conflict detection method is to begin with the BASESAT constraint information discussed in section 7.3.8.1, append test expressions for the particular conflict detection class, solve via the SAT solver, and finally interpret the results to answer the question posed. If a suite of tests is performed serially then all test expressions are removed between tests, resetting the BASESAT to its original state. The tool reports all conflict/problems found, and, if applicable, the particular configuration found that caused the problem.

In each conflict class subsection below are details for constructing test expressions, and logic for interpretation of SAT solver results.

#### 7.3.9.1 Rule Conflict

Rule conflict tests check whether the change has created a conflict that prevents all rules from being satisfied simultaneously. This test addresses research question 1 in

section 7.3.3. The method to perform this test is to append the proposed change onto the BASESAT, and then apply the solver. If the solver returns UNSATISFIABLE, then the change has created a conflict.

If, on the other hand, the solver returns SATISFIABLE, this is not sufficient evidence to conclude that there are no problems with the change. There might still be issues that could be detected by performing some of the other conflict detection tests. For example, consider rule 0000037796 from VRM, and suppose the change is to induce a new rule that is identical except that the Z/A type is switched, as shown in Table 38.

| Rule | Type | IF | THEN |
|---|---|---|---|
| 0000037796 (current) | Z | & + L807A & + S609A | / + S6AEA |
| New, (0000037796 inverted) | A | & + L807A & + S609A | / + S6AEA |

Table 38. New rule example

Although this pair of rules is clearly nonsense, the solver will return SATISFIABLE if this experiment is performed. If either rule is "active," i.e. its antecedent is true, then the other rule will be violated. However, the apparent conflict can be avoided by the satisfiability routine, and a valid configuration found, if both rules are inactive. In this case, setting either option L807A or S609A to false will deactivate both rules. Adding this new OKA rule would effectively forbid any valid configuration from having both L807A and S609A. See section 7.3.9.2.2 for details on resolving this issue.

#### 7.3.9.2 Object Activation

Object activation tests iteratively check each individual object, to determine whether there exists a configuration for which the object is active. An object is disabled

if no possible configuration activates the object. Disabled objects may be present in the constraint databases for legacy reasons, or they may be evidence of errors arising from the PN changes. The types of objects considered may be divided into two categories, literals and constraints. The following subsections discuss these in turn.

7.3.9.2.1 Literal Activation

Literals include parts, options, packages, and FClasses. Each literal is checked individually, to determine whether the literal is active on some configuration. This test addresses research question 3 in section 7.3.3. A single test expression is appended to the BASESAT during each iteration, forcing the literal to be active, e.g. $\boxed{\neg \mathrm{S323A} \rightarrow \mathrm{S323A}}$. If the solver returns SATISFIABLE then the literal may be active. Else, if UNSATISFIABLE is returned, then the literal is not permitted on any configuration. Disabled literals are evidence of potential errors in the implemented PN changes.

7.3.9.2.2 Antecedent Satisfiability

The antecedent is the "IF" portion of a rule. Testing antecedent satisfiability verifies that a rule can be activated. This test addresses research question 2 in section 7.3.3.

The scenario presented in section 7.3.9.1 showed a latent conflict, causing rule deactivation. In cases such as this, activating the rule results in no valid configurations. This test iteratively tests each OKA rule, by appending a forcing expression for the rule to the BASESAT. For example, Table 39 shows rules 0000027332, and a corresponding test rule forcing rule 0000027332 to be active. If the solver returns SATISFIABLE for

this test, then the rule can be activated. Else, the proposed change has forced the rule to be inert, and is evidence of a potential error.

| Rule | Type | IF | THEN |
|---|---|---|---|
| 0000027332 | Z | S2D4A / S2H4A / S2H7A / S2LSA / S2T2A / S2TZA / S2WFA | S258A |
| Force | Z | - S2D4A & - S2H4A & - S2H7A & - S2LSA & - S2T2A & - S2TZA & - S2WFA | S2D4A / S2H4A / S2H7A / S2LSA / S2T2A / S2TZA / S2WFA |

Table 39. Testing antecedent satisfiability

Antecedent satisfiability tests are performed only on OKA rules. The constraints imposed by FClasses and PK rules are tested using the methods in section 7.3.9.2.1, and do not need to be reproduced here.

There is a small set of existing OKA rules that are designed such that the antecedent is unsatisfiable, e.g. $(\neg DVD \rightarrow DVD)$. These rules exist to force application of options. The antecedent satisfiability test routine will always report these rules as potential errors. It is the discretion of the user to ignore or act on the report.

#### 7.3.9.3 Implicit Inclusion / Exclusion

Implicit relationships between option pairs can be either an inclusion, if the options must always occur together, or an exclusion, if the options may never occur together. This test addresses research question 4 in section 7.3.3. The test operates iteratively, checking each pair of options in turn.

For a given pair of options, inclusions are found by testing the contradiction, "Can one option be active, but not the other?" A pair of test expressions similar to those in Table 40 are appended to the BASESAT. If the solver returns UNSATISFIABLE, then the option pair has an inclusion relationship.

| Rule | Type | IF | THEN |
|---|---|---|---|
| Force + | Z | -S323A | S323A |
| Force - | Z | S4FFA | -S4FFA |

Table 40. Testing implicit inclusion

Exclusions between the option pair are also found via contradiction, by testing the question, "Can both options be active simultaneously?" A pair of test expressions similar to those in Table 41 are appended to the BASESAT. If the solver returns UNSATISFIABLE then the option pair has an exclusion relationship.

| Rule | Type | IF | THEN |
|---|---|---|---|
| Force + | Z | -S5DPA | + S5DPA |
| Force + | Z | -S645A | + S645A |

Table 41. Testing implicit exclusion

#### 7.3.9.4 Part Families

Parts may be tested upon as solitary objects, as seen in the activation tests in section 7.3.9.2.1. To address questions pertaining to collections of parts, instead of singletons, the concept of a part *family* is introduced. A part family is a set of related parts, such as windshields, for which one and only one of the member parts should be allocated for any valid vehicle. At the current time there are no BMW repositories that list part families and their member parts, so the family content must be provided by the user. An example family including windshields is shown in Table 42.

| PNO | Description |
|---|---|
| 7292394 | COVERING WINDSCREEN LAMINATED SAFETY GLA |
| 7292399 | ASSY WINDSCREEN GREEN WITH RLSBS |
| 7292400 | ASSY WINDSCREEN GREEN AND HUD WITH RLSBS |

| | |
|---|---|
| 7292401 | ASSY WINDSCREEN IR AND HUD WITH RLSBS |
| 7292402 | ASSY WINDSCREEN GREEN MT HUD RLSBS CAM-B |
| 7292403 | ASSY WINDSCREEN GREEN WTH GREY SHADE A R |
| 7308905 | ASSY WINDSCREEN IR WTH GREY SHADE RLSBS |

Table 42. Example part family (Windscreen)

There are two classes of conflict tests that use part families. The first checks whether 1-and-only-1 condition holds for all part families, as discussed in section 7.3.9.4.1. The second tests user-defined part family interaction rules, as discussed in section 7.3.9.4.2.

7.3.9.4.1 Part Family Allocation

The part family allocation test verifies whether exactly one of the parts in the family is allocated for every vehicle. This test addresses research question 5 in section 7.3.3. This result is achieved in two stages, first testing whether zero of the parts may be allocated, then testing whether two or more of the parts may be allocated.

The first stage tests for contradiction, "Can all of the parts in the family be inactive?" A test expression similar to that in Table 43 is appended to the BASESAT, to force all parts in the family inactive. The example provided checks whether any configuration results in no windshield. If the solver returns SATISFIABLE, then it is possible to build a vehicle using none of them. If the solver returns UNSATISFIABLE then the stage 1 test is passed, and stage 2 tests begin.

| Rule | Type | IF | THEN |
|---|---|---|---|
| Force off | Z | 7292394 / +7292399 / +7292400 / +7292401 / +7292402 / +7292403 / +7308905 | -7292394 & -7292399 & -7292400 & -7292401 & -7292402 & -7292403 & -7308905 |

Table 43. Testing part family activation

The second stage tests whether more than one of the parts in the family may be allocated for any vehicle. Note that this test counts the number of PNOs allocated, not

the quantity of individual parts. A single PNO may call any non-zero quantity of parts, or any liquid volume, and here it is simply counted as 1 PNO.

This result is achieved iteratively, checking each part in the family in turn. A direct proof is offered by the question, "If PNO *i* (chosen by iteration) is active, can another PNO in the family be active as well?" A test expression similar to that in Table 44 is appended to the BASESAT, to force one of the other parts to be active along with part *i*. If the solver returns SATISFIABLE for any of the iterated tests, then a configuration has been identified that calls more than one of the parts. Else, if the solver returns UNSATISFIABLE, then the stage 2 test is passed for this iteration. If all iterations return UNSATISFIABLE, the stage 2 test is passed.

| Rule | Type | IF | THEN |
|---|---|---|---|
| Force *i* | Z | -7292394 | 7292394 |
| Force another | Z | 7292394 | / +7292399 / +7292400 / +7292401 / +7292402 / +7292403 / +7308905 |

Table 44. Testing multiple PNO inclusion from one part family

7.3.9.4.2 Part Family Matching

Parts are commonly designed to fit with other specific parts. In this section scenarios are considered where parts from one family are designed to match with another family. This test addresses research question 6 in section 7.3.3. Consider the example data in Table 45. There are two part families for exhaust tips, one for round profiles and one for rectangular. There are two part families for bumpers, reflecting the shape of the hole through which the exhaust tip fits. Geometry constraints require matching round bumpers with round exhaust tips, and square with square.

| Exhaust Tip, Round | Exhaust Tip, Rect. | Bumper, Round | Bumper, Rect. |
|---|---|---|---|
| $PNO(Ex, Ro)_1$ | $PNO(Ex, Re)_1$ | $PNO(Bp, Ro)_1$ | $PNO(Bp, Re)_1$ |

| … | … | … | … |
|---|---|---|---|
| $PNO(Ex, Ro)_n$ | $PNO(Ex, Re)_n$ | $PNO(Bp, Ro)_n$ | $PNO(Bp, Re)_n$ |

Table 45. Part families with geometry relationships

A contradiction test is performed to ensure that mismatches cannot occur. A test expression similar to that in Table 46 is appended to the BASESAT, forcing one of the part families to be active, and forcing its matching family inactive.

| Rule | Type | IF | THEN |
|---|---|---|---|
| Force *Round Bumper* | Z | -(RND BUMPER) | (RND BUMPER) |
| Force Not *Round Exhaust* | Z | (RND EXHAUST) | -(RND EXHAUST) |

Table 46. Testing part family matching

If the solver returns SATISFIABLE for this test, then a configuration has been identified with mismatched parts.

7.3.10 Implementation

This chapter describes the programming methods used to prepare constraint data into a form suitable for SAT experimentation. There are two necessary tasks, performed serially. The first task transforms the binary trees that represent constraint information into conjunctive normal form (CNF), as discussed in section 7.3.10.1. The second task, discussed in section 7.3.10.2, writes the content of all binary trees into a single text output file, in the DIMACS format required by the SAT solver.

7.3.10.1 Transformation to Conjunctive Normal Form

The SAT solver requires input formatted in CNF. CNF expressions contain only AND, OR, and NOT operators, and are written as conjunctions (ANDs) of disjunctive (OR) clauses. Further, all negation operators must be applied directly to literals, and not to parenthetical expressions. For example, the example expression $(A \,\&\, B) \,/\, -(C \,/\, D)$ is not in CNF. A series of logical transformations can be applied to rearrange this

expression into a logically equivalent expression in CNF. For the example given above, the result would be $\boxed{(\mathrm{A} \,/\, -\mathrm{C}) \,\&\, (A \,/\, -D) \,\&\, (\mathrm{B} \,/\, -\mathrm{C}) \,\&\, (B \,/\, -D)}$.

Recall that all constraint information is stored programmatically in a forest of binary trees, as documented in ("Configuration Management Model Refinement," Technical Report 2015-CEDAR-BMW-Configuration-005). Each binary trees is transformed to CNF using the process:

1. Transform "!" Exclusive-OR subtrees
2. Substitute STAN nodes with FClass
3. Reduce binconditionals to two conditionals
4. Reduce conditionals to AND, OR, NOT
5. Propagate NOT downward towards leaves
6. Distribute AND over OR

The following subsections describe each of these steps in turn.

7.3.10.1.1 Transform Exclusion Subtrees

Exclusion nodes indicate the start of a definite horn clause derived from a PK rule. Definite horn clauses require exactly one of the lower leaf nodes to be true. Before transformation, the subtree below the exclusion node will have a collection of OR operators, and literal objects at the leaves. After transformation, the exclusion node will be replaced by an AND node, with two child subtrees: one subtree to enforce that at least one of the literals is true, and one subtree to enforce that no more than one of the literals is true. The first subtree is simply a disjunction (OR) of all literals. The second subtree is a conjunction of pairwise exclusion statements.

Consider the example clause $\boxed{(!A \,/\, B \,/\, C)}$. After the initial parse, the root node is type EXC ("!"), and all lower-level nodes are either OR or VAR. After transformation,

the expression becomes $\boxed{(\mathrm{A} \,/\, \mathrm{B} \,/\, \mathrm{C}) \,\&\, -(A \,\&\, B) \,\&\, -(A \,\&\, C) \,\&\, -(B \,\&\, C)}$. The pre- and post-transformation trees for this expression are shown in Figure 66.

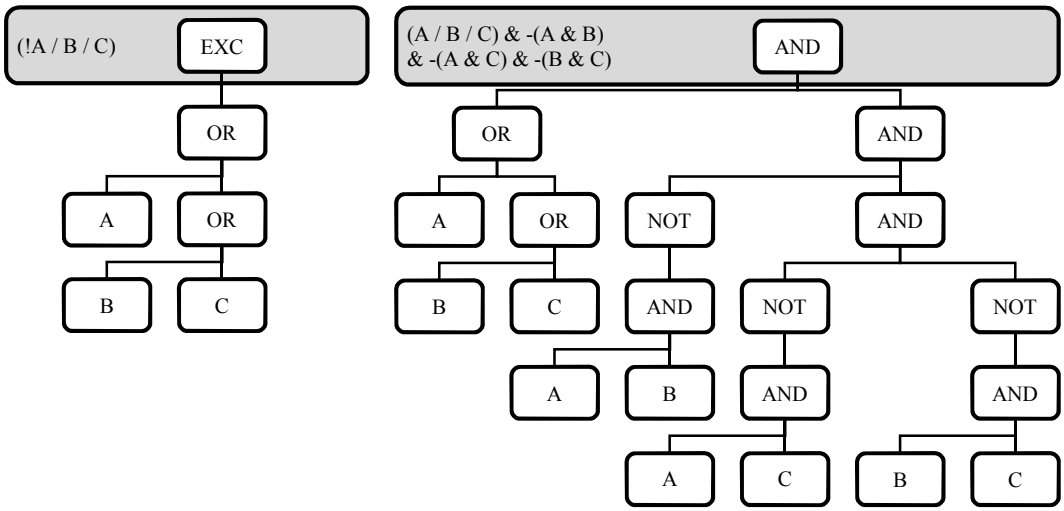


Figure 68. Example trees before and after transformation of exclusion node

The recursive method TransformEXC, detailed in Figure 69. CNF Step 1: Transform Exclusion, is initially called upon the root node of each binary tree. If the current node is not of type EXC, then the method is called upon any existing children nodes, to search deeper in the tree. If the current node is type EXC, the AllLiterals function is called to count all VAR-type leaf nodes in the subtree below. If only one leaf node exists, then that literal must be located in LChild, the only child node below. In this case the properties of the leaf node are simply copied to the current node, then the leaf node is deleted. The former EXC node has been effectively removed from the tree, replaced with the former LChild.

If there two or more leaf nodes below, several steps are undertaken to remodel the tree. Note that LChild already contains the subtree of ORs, from the initial parse. The

requirement that at least one literal is true is already fulfilled by the subtree contained in LChild. RChild is currently null, and will be instanced to contain a subtree enforcing the mutual exclusion logic between each literal pair. The current node is changed from type EXC to type AND, so that both the LChild and RChild subtree logics are required to be true.

A Boolean expression string (*strRuleText*) is written to contain the pair-wise mutual exclusion logic. For each unique pair of literals A and B contained in leaf nodes, an exclusion expression of form $\boxed{-(\mathrm{A}\ \&\ B)}$ is written. Each of these pairwise exclusion strings are concatenated into *strRuleText*, with "&" operators between each. Finally, RChild is instanced, and its subtree constructed by calling its .ParseExpression method with argument *strRuleText*. See ("Configuration Management Model Refinement," Technical Report 2015-CEDAR-BMW-Configuration-005) for details of the .ParseExpression method.

```
1   Method TransformEXC
2       If Me.Oper = "!" Then
3           If Me.AllLiterals.count = 1 Then
4               Me.Oper = VAR
5               Me.Varname = Lchild.VarName
6               Lchild = Nothing
7           Else
8               Me.Oper = "&"
9               litArray = Me.AllLiterals.array
10              strRuletext = "-(" & litArray(1) & "&" & litArray(2) & ")"
11              For i = 2 To UBound(litArray) - 1
12                  For j = i + 1 To UBound(litArray)
13                      RHS = "-(" & litArray(i) & "&" & litArray(j) & ")"
14                      strRuletext = strRuletext & "&" & RHS
15                  Next
16              Next
17              Rchild = New CBinNode
18              Call Rchild.ParseExpression(strRuletext)
19          End If
20      End If
21      If Not Lchild Is Nothing Then Call Lchild.TransformEXC
22      If Not Rchild Is Nothing Then Call Rchild.TransformEXC
23  End Method
```

Figure 69. CNF Step 1: Transform Exclusion

7.3.10.1.2 Substitute Standard Nodes

The Standard field of OKA rules is reserved for standard equipment, applied to any vehicle that satisfies the conditions of the rule. Standard equipment sometimes can be upgraded by the customer, at additional cost. The FClass of the standard option contains all upgrade alternatives. If the standard option is not a member of any FClass, then it is mandatory equipment with no alternatives.

The recursive method SubstituteStandard performs these FClass replacements, as detailed in Figure 70. The argument *upperStandard* tracks whether any upper-level node oftype STAN was previously detected during recursion. The method is initially called upon the root node of a rule tree, passing *upperStandard = false* .

If the current node is of type STAN, then the current node properties are removed and replaced by LChild's properties. Recursion continues on the current node's newly defined self, except this time passing $upperStandard = true$.

If the current node is of type VAR, and *upperStandard* is true, then the local node is a literal within a standard clause. An SQL query is performed to find the FClass membership of the local literal. If an FClass is found, then the name of the FClass replaces the name of the local literal. Else, no adjustment to name is taken.

Finally, the method recurses to deeper into the tree. The .SubstituteStandard method is called upon any existing children, passing the current value of *upperStandard.*

```
1   Method SubstituteStandard(upperStandard As Boolean)
2       If Me.Oper = "$" Then
3           Me.Oper = Lchild.Operator
4           Me.Varname = Lchild.VarName
5           Rchild = Lchild.RightChild
6           Lchild = Lchild.LeftChild
7           Call Me.SimplifyStandard(True)
8       ElseIf (Me.Oper = VAR) And (upperStandard = True) Then
9           RS = New ADODB.Recordset
10          SQL = SQL_GetFClassOfOption(Me.Varname)
11          RS.Open SQL, cn
12          If Not RS.eof Then
13              Me.Varname = RS.fields("FClass").value
14          End If
15          RS.Close
16      End If
17      If Not Lchild Is Nothing Then Call Lchild.SubstituteStandard(upperStandard)
18      If Not Rchild Is Nothing Then Call Rchild.SubstituteStandard(upperStandard)
19  End Method
```

Figure 70. CNF Step 2: Substitute Standard with FClass

7.3.10.1.3 Reduce Biconditionals

Biconditionals indicate "if and only if" logic, usually written $A \leftrightarrow B$. A pair of conditional nodes may replace the biconditional, with the antecedent/consequent swapped for the two conditionals, e.g. $(A \rightarrow B)\ \&\ (B \rightarrow A)$. The two conditional nodes are

connected with an AND. An example of this reduction in binary tree form is shown in Figure 69.

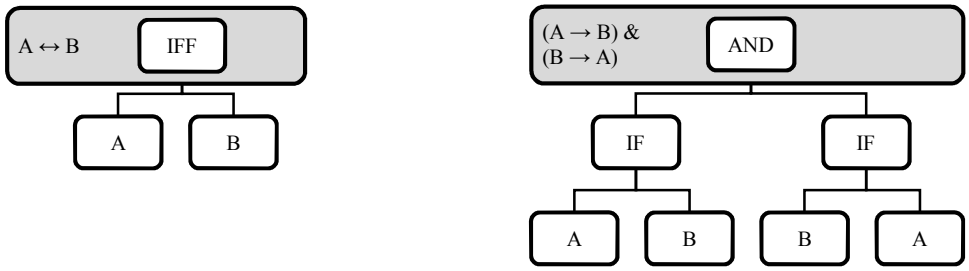


Figure 71. Example trees before and after biconditional reduction

The recursive method described in Figure 72 is initially called upon the root node of a rule tree. If the local node is type IFF, then it is a biconditional node. The operator of the current node is changed to AND. Next, both LChild and RChild subtrees are stored in method-level variables *childA* and *childB*, for later use. New instances are then created for LChild and RChild, and assigned operator type IF. The children of the newly created LChild and RChild are then copied from the previously stored *childA* and *childB*. Finally, the recursion proceeds deeper into the tree searching for any other biconditional nodes.

```
1   Method ReduceBiconditional()
2       If Me.Oper = "=" Then
3           childA = Lchild
4           childB = Rchild
5           Me.Oper = "&"
6           Lchild = New CBinNode
7           Rchild = New CBinNode
8           Lchild.Operator = ">"
9           Rchild.Operator = ">"
10          Lchild.LeftChild = childA
11          Lchild.RightChild = childB
12          Rchild.LeftChild = childB.CopyNode
13          Rchild.RightChild = childA.CopyNode
14      End If
15      If Not Lchild Is Nothing Then Call Lchild.ReduceBiconditional()
16      If Not Rchild Is Nothing Then Call Rchild.ReduceBiconditional()
17  End Method
```

Figure 72. CNF Step 3: Reduce Biconditionals

7.3.10.1.4 Reduce Conditionals

Conditionals indicate "if this, then that" logic, usually written $A \rightarrow B$. Note that if the antecedent is false, then no constraint is placed on the consequent, i.e. it may be true or false. This insight motivates the reduction logic applied for a conditional expression. Either the antecedent is false, or the consequent is true, e.g. $-A \ / \ B$. An example of this

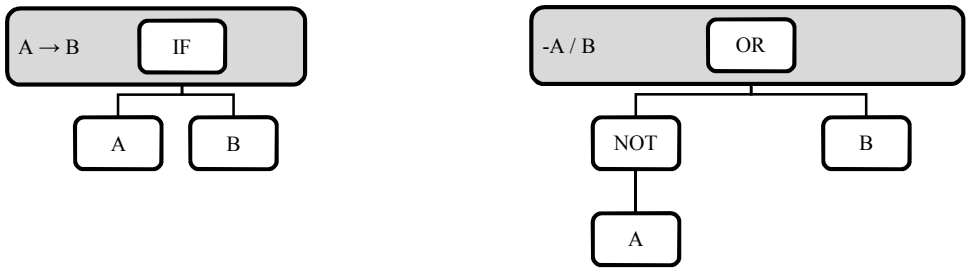


Figure 73. Example trees before and after conditional reduction

reduction in binary tree form is shown in Figure 71.

The recursive method described in Figure 74 is initially called upon the root node of a rule tree. If the local node is type IF, then it is a conditional node. The operator of the current node is changed to OR. Next, the LChild subtree is stored in a method-level variable *childA*, for later use. A new instance is then created for LChild, and assigned operator type NOT. The grandchild of the newly created LChild is then copied from the previously stored *childA*. Finally, the recursion proceeds deeper into the tree searching for any other conditional nodes.

```
1   Method ReduceConditional
2       If Me.Oper = ">" Then
3           Me.Oper = "/"
4           childA = Lchild
5           Lchild = New CBinNode
6           Lchild.Operator = "-"
7           Lchild.LeftChild = childA
8       End If
9       If Not Lchild Is Nothing Then Call Lchild.ReduceConditional
10      If Not Rchild Is Nothing Then Call Rchild.ReduceConditional
11  End Method
```

Figure 74. CNF Step 4: Reduce Conditionals

7.3.10.1.5 Propagate NOT

NOT nodes indicate that all logic in the child subtree is negated. In CNF, NOT operators apply only to literals, and not to clauses with other operators inside. To achieve this outcome, NOT nodes within binary trees are removed, recursively carried downward in their subtree until leaf nodes are reached, and then re-instantiated just above the leaf if applicable.

The downward propagation of negation modifies intermediate operator nodes. At this point in the overall CNF transformation process, only AND, OR, and NOT operators remain. The Boolean expression negation transformations for these three operator types is summarized in Table 47.

| *Operator negated* | *Initial expression* | *Expression after propagation* |
|---|---|---|
| *NOT* | $-(-A)$ | $A$ |
| *AND* | $-(A \,\&\, B)$ | $(-A \,/\, -B)$ |
| *OR* | $-(A \,/\, B)$ | $(-A \,\&\, -B)$ |

Table 47. Negation propagation across operators

The recursive method PropagateNegation is detailed in

Figure 75. The argument *upperNeg* tracks whether negation is propagating downward from a previous, higher recursion level. The method is initially called upon the root node of each binary tree, passing $\boxed{upperNeg = false}$. The behavior of the

method depends on three items: the passed argument *upperNeg*, whether the local node is type NOT, and whether the local node is type VAR. A summary of PropagateNegation behavior for each combination of these items is shown in Table 48.

Cases 1 and 2: The current node is type NOT. Flip the polarity of *upperNeg* to reflect the negation. Next, copy all of the child node's properties to the local node, to remove the NOT node from the tree. Recursion continues upon the newly-defined self node, passing *upperNeg*.

Case 3: The current node is type AND or OR, and *upperNeg* is true. If the current node is type AND, then change it to type OR, and vice-versa. Next, begin recursion on both children nodes, passing *upperNeg*.

Case 4: The current node is type VAR, and *upperNeg* is true. A NOT node was removed from the tree at a higher level of recursion. Here at the leaf, the NOT node is restored. Instance LChild, and copy all local properties to it. Next, set the local operator type to NOT.

Case 5: The current node is either and AND or OR. Since *upperNeg* is false, no changes are made to the local node. Recursion proceeds to both children nodes, passing *upperNeg*.

Case 6: Recursion terminates at leaf nodes. Since *upperNeg* is false, no changes are required.

| | **Argument** | **Local Node Type** | | |
|---|---|---|---|---|
| **Case #** | *upperNeg* | NOT | VAR | Action |
| **1** | F | T | F | Negation begin. Replace self with child. Recurse on (new) self, passing *upperNeg* = true. |
| **2** | T | T | F | Double negative. Replace self with child. Recurse on (new) self, passing *upperNeg* = false. |
| **3** | T | F | F | Transform operator. Swap self between AND/OR type. Recurse to all children, passing *upperNeg* = true. |
| **4** | T | F | T | Terminal negation. Insert negation node between self and parent. Recursion ends. |

| | | | | |
|---|---|---|---|---|
| **5** | F | F | F | Continue recursion to children. |
| **6** | F | F | T | Terminus. Recursion ends. |
| **7** | F | T | T | Impossible, self cannot be both NOT and VAR type. |
| **8** | T | T | T | Impossible, self cannot be both NOT and VAR type. |

Table 48. Propagate Negation criteria and behavior

```
1   Method PropagateNegation(upperNeg As Boolean)
2       If Me.Oper = "-" Then
3         If (Lchild.Operator = VAR) And (upperNeg = False) Then
4               'Do nothing, recursion terminates
5           Else
6               Me.Oper = Lchild.Operator
7               Me.Varname = Lchild.Varname
8               Rchild = Lchild.RightChild
9               Lchild = Lchild.LeftChild
10              Call Me.PropagateNegation(Not upperNeg)
11          End If
12      ElseIf (upperNeg = True) And (Not Me.Oper = VAR) Then
13          If Me.Oper = "/" Then
14              Me.Oper = "&"
15          ElseIf Me.Oper = "&" Then
16              Me.Oper = "/"
17          End If
18          If Not Lchild Is Nothing Then Call Lchild.PropagateNegation(upperNeg)
19          If Not Rchild Is Nothing Then Call Rchild.PropagateNegation(upperNeg)
20      ElseIf (upperNeg = True) And (Me.Oper = VAR) Then
21          Me.Lchild = Me.CopyNode
22          Me.Varname = ""
23          Me.Oper = "-"
24      ElseIf (Not upperNeg) And (Not Me.Oper = VAR) Then
25          If Not Lchild Is Nothing Then Call Lchild.PropagateNegation(upperNeg)
26          If Not Rchild Is Nothing Then Call Rchild.PropagateNegation(upperNeg)
27      End If
28  End Method
```

Figure 75. CNF Step 5: Propagate Negation

7.3.10.1.6 Distribute AND over OR

At the final stage of CNF transformation only type AND, OR, and VAR nodes remain in the binary trees. The last step is to distribute ANDs over ORs. The process is algebraically similar to multiplicative distribution, e.g. $\boxed{A / (B \,\&\, C)}$ becomes

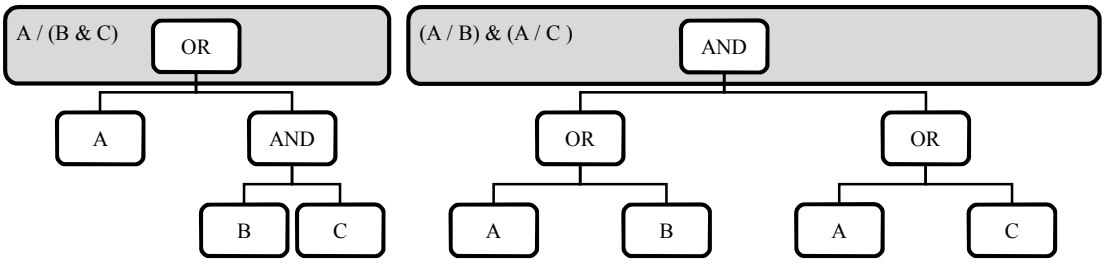


Figure 76. Example tree before and after AND distribution

(A / B) & (A / C). Two AND distribution examples are shown in Figure 74 and Figure 75. Algebraically these examples behave identically. They are shown separately to support the different coding approach used for each case.

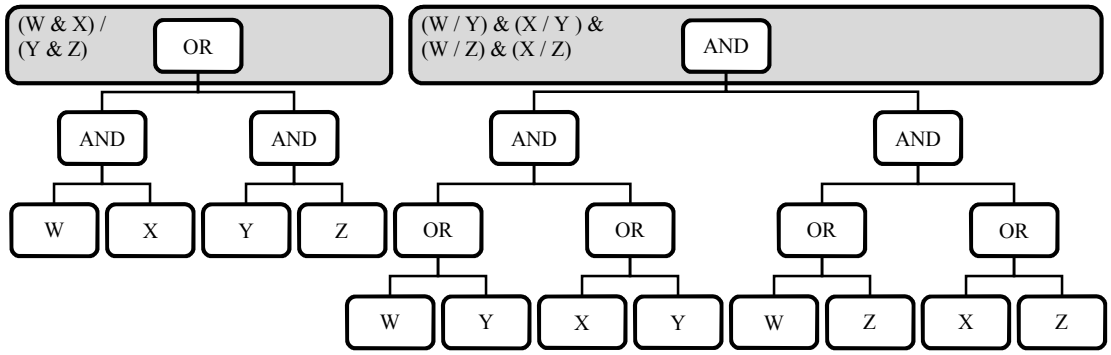


Figure 77. Example tree before and after AND distribution

The recursive method DistributeAND, detailed in Figure 78, is initially called upon the root node of each binary tree. If the local node has an OR operator, and at least one child node is an AND operator, then the local subtree must be remodeled to reflect distribution of the AND. Two separate cases are identified.

Case 1: The current node is an OR, and only one child is an AND node. Temporary variables *childA*, *gchildB*, and *gchildC* are instantiated to hold links to lower-level subtrees. Whichever child node is not the AND is assigned to *childA*. The children of the AND node are stored in *gchildB* and *gchildC*. The local node is reassigned to type AND, and new LChild and RChild nodes are instantiated, both of operator type OR. The grandchildren nodes are assigned from the temporary variables. LChild receives *childA* and *gchildB* as children, and RChild receives *childA* and *gchildC* as children.

Case 2: The current node is an OR, and both child nodes are type AND. Four temporary variables *gchildW*, *gchildX*, *gchildY*, and *gchildZ* are instantiated and assigned copies of the four existing grandchildren subtrees (e.g. LChild.RightChild). The local node is reassigned to type AND. LChild and RChild are already AND nodes, and remain so. All four existing grandchildren nodes are replaced with new node instances of type OR (note that the subtree content overwritten here is retained in the temporary variables). Finally, eight great-grandchildren nodes are assigned using the temporary variables. The assignment pattern depends on the original location of the subtrees. Each OR pairs two nodes that were not originally siblings. E.g. *gchildW* and *gchildX* were originally siblings, so this pairing does not appear.

```
1   Method DistributeAND()
2       If Me.Oper = "/" Then
3           If (Lchild.Operator = "&") And (Rchild.Operator = "&") Then
4               gchildW = Lchild.LeftChild
5               gchildX = Lchild.RightChild
6               gchildY = Rchild.LeftChild
7               gchildZ = Rchild.RightChild
8           '0 layers below (this node): change operator to AND
9               Me.Oper = "&"
10          '1 layer below: both AND...
11              'no changes needed, both are already ANDs.
12          '2 layers below: 4 ORs...
13              Lchild.LeftChild = New CBinNode
14              Lchild.RightChild = New CBinNode
15              Rchild.LeftChild = New CBinNode
16              Rchild.RightChild = New CBinNode
17              Lchild.LeftChild.Operator = "/"
18              Lchild.RightChild.Operator = "/":
19              Rchild.LeftChild.Operator = "/"
20              Rchild.RightChild.Operator = "/"
21          '3 layers below: W,X,Y,Z
22              Lchild.LeftChild.LeftChild = gchildW
23              Lchild.LeftChild.RightChild = gchildY
24              Lchild.RightChild.LeftChild = gchildX
25              Lchild.RightChild.RightChild = gchildY.CopyNode
26              Rchild.LeftChild.LeftChild = gchildW.CopyNode
27              Rchild.LeftChild.RightChild = gchildZ
28              Rchild.RightChild.LeftChild = gchildX.CopyNode
29              Rchild.RightChild.RightChild = gchildZ.CopyNode
30          Else
31              If Lchild.Operator = "&" Then
32                  childA = Rchild
33                  gchildB = Lchild.LeftChild
34                  gchildC = Lchild.RightChild
35              ElseIf Rchild.Operator = "&" Then
36                  childA = Lchild
37                  gchildB = Rchild.LeftChild
38                  gchildC = Rchild.RightChild
39              End If
40              If Not childA Is Nothing Then
41                  Me.Oper = "&"
42                  Lchild = New CBinNode
43                  Lchild.Operator = "/"
44                  Lchild.LeftChild = childA
45                  Lchild.RightChild = gchildB
46                  Rchild = New CBinNode
47                  Rchild.Operator = "/"
48                  Rchild.LeftChild = childA.CopyNode
49                  Rchild.RightChild = gchildC
50              End If
51          End If
52      End If
53      If Not Lchild Is Nothing Then Call Lchild.DistributeAND
54      If Not Rchild Is Nothing Then Call Rchild.DistributeAND
55  End Method
```

Figure 78. CNF Step 6: Distribute AND over OR

#### 7.3.10.2 DIMACS Output

After transforming the binary trees into CNF, the next step is to write the constraint content into a file compatible with the SAT solver. The MiniSAT solver

requires inputs as DIMACS formatted text files. The DIMACS format was created by, and named after the Center for Discrete Mathematics and Theoretical Computer Science (DIMACS). Originally created to standardize formats for organization-internal purposes, it has since been widely accepted as the standard for CNF Boolean expressions. For format specifications see ftp://dimacs.rutgers.edu/pub/challenge/satisfiability/. This section describes the method used to extract logical content from the CNF binary trees, and write it into an equivalent representation in a DIMACS-formatted text file.

7.3.10.2.1 DIMACS Format

In the DIMACS format, each literal is encoded as a unique integer, counting upward from one [1, 2, 3, …]. All operators are suppressed, not explicitly written to file. Instead, line breaks in the text file implicitly encode the original location of operators. All content on a single line of text comprises a single disjunctive (OR) clause. A line break implies a conjunction (AND), and the next disjunctive clause is written in the following line of text. Within a line of text, a literal's number is written if that literal appears in the corresponding disjunctive clause. If the literal is negated within the disjunctive clause, then the integer is negative. Otherwise, the integer is positive. Spaces are placed between each literal's number, to prevent ambiguity. The text line for each disjunctive clause is finished by appending a "0". This "0" is merely an end-of-line flag, and does not represent any literal.

Comments in the DIMACS format are denoted by lines that begin with the letter "c". The first non-comment line within the file begins with the letter "p", and contains metadata. Following the "p" flag, there are three pieces of metadata:

1 The format. This should always be "cnf"
2 The number of unique literals
3 The number of disjunctive clauses

Consider the Boolean expression $(x_1/-x_5/x_4)\&(x_5/x_2/x_4)\&(-x_3/-x_4)$. The expression is CNF, with 5 unique literals and 3 disjunctive clauses. In the DIMACs format, this expression is written as shown in Figure 79.

```
1   c Here is a comment.
2   p cnf 5 3
3   1 -5 4 0
4   5 2 4 0
5   -3 -4 0
```

Figure 79. Example DIMACS format

7.3.10.2.2 Converting Binary Trees to DIMACS

Recall that in CNF, expressions are written as conjunctions (ANDs) of disjunctions (ORs). After running the routines described in section 7.3.10.1, the program has a forest of binary trees stored in memory, each of which has been transformed to CNF. Each binary tree represents a constraint that must be obeyed, and all constraints must be obeyed simultaneously for any valid vehicle.

One logically equivalent approach would be to compile all binary trees into a single, enormous tree, by conjoining the roots of each tree with AND nodes. These new AND nodes would serve to simultaneously enforce all constraints within their subtrees. It is useful to store each constraint as a separate object, however, so that each constraint may retain reference properties of the source OKA or TAIS rules that it is based upon. Instead of conjoining each binary tree, instead the DIMACS file writing routine reads each tree sequentially, and simply inserts a line break after each tree. Line breaks

represent conjunctions in the DIMACS format, so this treatment delivers a file output that enforces all constraints to be obeyed.

Figure 80 presents the algorithm for writing a DIMACS-formatted text file from binary tree constraint information, which requires two passed parameters. The first, `RuleForest`, contains the root nodes for all binary trees. The second, `fullpath_MiniSAT_datafile`, is the path at which the text file output is to be written. First the `intmap` variable is initialized, which manages mapping each literal name to a unique integer. The following loop considers each binary tree root node one at a time. From each root node an array of disjunctive clauses is extracted, using the node function `DIMAC_Array`, as detailed in Figure 81. The literals within each disjunctive clause are re-aliased to integers, and then the clause is stored into the local variable `dlines`, which will accrue all lines of text to be written to file. Finally, a text file is created at the specified path, and all text stored in `dlines` is written into it.

```
1   Method WriteDIMACS(RuleForest, fullpath_MiniSAT_datafile)
2       intmap.Initialize(RuleForest)
3       For Each root In RuleForest
4           root.TransformToCNF
5           rootDIMAC_Array = root.DIMAC_Array
6           dlines.Add(" c RuleNo: " & root.RuleNo)
7           For i = LBound(rootDIMAC_Array) To UBound(rootDIMAC_Array)
8               line = rootDIMAC_Array[i]
9               EncodeLiteralsToInts(line, intmap)
10              line = line & " 0"
11              dlines.Add(line)
12              numclauses = numclauses + 1
13          Next
14      Next
15      Open fullpath_MiniSAT_datafile For Output As #1
16      Print #1, "p cnf " & intmap.count & " " & numclauses
17      For Each line In dlines
18          Print #1, line
19      Next
20      Close #1
21  End Method
```

Figure 80. Write DIMACS file

The function DIMAC_Array is located in the binary node class, as detailed in Figure 81. The tree must be formatted CNF prior to executing this function. The function's purpose is to collect all disjunctive clauses within the subtree below into a single array, such that each disjunctive clause is a separate element in the array. To this end, a recursive approach examines the operator type of the local node. There are 4 types of operators that may be present in CNF:

1 Literal. The returned array has one element, and that element is the name of the local literal.
2 NOT. The returned array has one element, and that element is the negated name of the child literal.
3 OR. OR nodes indicate that both child nodes are part of the same disjunctive clause. The returned array has one element, and that element is the concatenated elements from calling this function recursively upon both children.
4 AND. AND nodes indicate a conjunction between the child nodes. This function is called recursively upon both children, and the returned results are stacked into separate elements in the returned array.

```
1   Function DIMAC_Array() As Array
2       Select Me.Oper
3       Case VAR
4           retArray[1] = Me.Varname
5       Case NOT
6           retArray[1] = "-" & Lchild.LiteralName
7       Case OR
8           retArray[1] = Lchild. DIMAC_Array[1] & " " & Rchild. DIMAC_Array[1]
9       Case AND
10          retArray = ArrayUnion(Lchild.DIMAC_Array, Rchild.DIMAC_Array)
11      End Select
12      Return retArray
13  End Function
```

Figure 81. Collect disjunctive clauses from a binary tree

### 7.3.11 Open Issues

Using a SAT proof checking engine requires all logic to be performed in Boolean terms only, and prohibits arithmetic counting. Largely, this design decision meets the needs of CM problems, though there are a small number of circumstances in which counting may provide additional functionality. One example is provided by the TAIS spring constraints, which are written using upper and lower bounds on the weight permissible for each spring part. Currently these constraints are excluded from the conflict detection methods. Another example is provided by the part allocation quantities, given in the TAIS model code columns. These quantities are currently modeled in binary terms (>0 = true, else false). The current framework detects whether a PNO is allocated or not, but not the quantity allocated. To accommodate arithmetic constraints and terms will likely require replacing the SAT framework. Pseudo-Boolean satisfiability problems are a likely replacement candidate, as these problems generalize the satisfiability problem to include numeric inequality constraints.

# REFERENCES


Agrawal, P. "The Related Activity Concept in Assembly Line Balancing." *International Journal of Production Research* 23 (1985): 403-421.

Ammer, E. D. "Rechnerunterstützte Planung Von Montageablaufstrukturen Für Erzeugnisse Der Serienfertigung." *IPA-IAO Forschung Und Praxis* 81 (1985).

Arcus, A. "COMSOAL: A Computer Method of Sequencing Operations for Assembly Lines." *International Journal of Production Research* 4 (1966): 259-277.

Bard, JF, E Dar-El and A Shtub. "An Analytic Framework for Sequencing Mixed Model Assembly Lines." *International Journal of Production Research* 30 (1992): 35-48.

Bartholdi, JJ. "Balancing Two-Sided Assembly Lines: A Case Study." *International Journal of Production Research* 31 (1993): 2447-2461.

Bautista, J, et al. "Local Search Heuristics for the Assembly Line Balancing Problem with Incompatibilities Between Tasks." *Proceedings of the 2000 IEEE International Conference on Robotics and Automation*. San Francisco, CA, 2000. 2404-2409.

Baybars, I. "A Survey of Exact Algorithms for the Simple Assembly Line Balancing Problem." *Management Sciences* 32 (1986): 909-932.

—. "An Efficient Heuristic Method for the Simple Assembly Line Balancing Problem." *International Journal of Production Research* 24 (1986): 149-166.

Baykasoglu, A. and T. Dereli. "Two-sided assembly line balancing using ant-colony-based heuristic." *International Journal of Advanced Manufacturing Technology* 36.5-6 (2008): 582-588.

Becker, C and A Scholl. "A Survey on Problems and Methods in Generalized Assembly Line Balancing." *European Journal of Operational Research* 183 (2006): 694-715.

Boysen, N and M Flidener. "A versatile algorithm for assembly line balancing." *European Journal of Operations Research* 184.1 (2008): 39-56.

Boysen, N, M Fliedner and A Scholl. "A Classification of Assembly Line Balancing Problems." *European Journal of Operational Research* 183 (2007): 674-693.

—. "Production Planning of Mixed-Model Assembly Lines: Overview and Extensions." *Production Planning & Control: The Management of Operations* 20.5 (2009): 455-471.

Bukchin, J, E Dar-El and J Rubinovitz. "Mixed Model Assembly Line Design in a Make-to-order Environment." *Computers & Industrial Engineering* 41 (2002): 405-421.

Burns, L and C Daganzo. "Assembly Line Job Sequencing Principles." *International Journal of Production Research* 25 (1987): 71-99.

Carnahan, B, B Norman and M Redfern. "Incorporating Physical Demand Criteria into Assembly Line Balancing." *IIE Transactions* 33 (2001): 875-887.

Carraway, R. "A Dynamic Programming Approach to Stochastic Assembly Line Balancing." *Management Science* 35 (1989): 459-471.

Chase, R. "Survey of Paced Assembly Lines." *Industrial Engineering* 6.2 (1974): 14-18.

Chutima, P and P Chimklai. "Multi-Objective Two-Sided Mixed-Model Assembly Line Balancing Using Particle Swarm Optimization with Negative Knowledge." *Computers and Industrial Engineering* 62 (2012): 39-55.

Deutsch, D. "A Branch and Bound Technique for Mixed-Product Assembly Line Balancing." *Ph.D. Dissertation, Arizona State University* (1971).

Dobson, G and C Yano. "Cyclic Scheduling to Minimize Inventory in a Batch Flow Line." *European Journal of Operational Research* 75 (1994): 441-461.

Dolgui, A, N Guschinsky and G Levin. "A Special Case of Transfer Lines Balancing by Graph Approach." *European Journal of Operational Research* 168 (2006): 732-746.

Domm, Robert. *Michigan Yesterday and Today*. Minneapolis, MN: Voyageur Press, 2009.

Domschke, W, R Klein and A Scholl. "Antizipative Leistungabstimmung bei moderner Variantenfließfertigung." *Zeitschrift für Betriebswirtschaft* 66 (1996): 1465-1490.

Emde, S, N Boysen and A Scholl. "Balancing Mixed-Model Assembly Lines: A Computational Evaluation of Objectives to Smoothen Workload." *International Journal of Production Research* 48.11 (2010): 3173-3191.

Frenk, J and G Galambos. "Hybrid Next-Fit Algorithm for the Two-Dimensional Rectangle Bin-Packing Problem." *Computing* 39 (1987): 201-217.

Garey, M and D Johnson. *Computers and Intractability - A Guide to the Theory of NP-completeness*. New York: W.H. Freeman & Co, 1979.

Gökcen, H and Ö Baykoc. "A New Line Remedial Policy for the Paced Lines with Stochastic Task Times." *International Journal of Production Economics* 58 (1999): 191-197.

Helgeson, W and D Birnie. "Assembly Line Balancing using the Ranked Positional Weight Technique." *Journal of Industrial Engineering* 12 (1961): 394-398.

Huang, E and R Korf. "New Improvements in Optimal Rectangle Packing." *Proceedings of the 21st International Joint Conference on Artificial Intelligence*. 2009.

Johnson, R. "A Branch and Bound Algorithm for Assembly Line Balancing Problems with Formulation Irregularities." *Management Science* 29 (1983): 1309-1324.

Kao, E. "A Preference Order Dynamic Program for Stochastic Assembly Line Balancing." *Management Science* 22 (1976): 1097-1104.

Kim, YK, JY Kim and Y Kim. "A Coevolutionary Algorithm for Balancing and Sequencing in Mixed Model Assembly Lines." *Applied Intelligence* 13 (2000): 247-258.

—. "Two-sided Assembly Line Balancing: A Genetic Algorithm Approach." *Production Planning and Control* 11 (2000): 44-53.

Kottas, J and H Lau. "A Cost-oriented Approach to Stochastic Line Balancing." *AIIE Transactions* 5 (1973): 164-171.

—. "A Total Operating Cost Model for Paced Lines with Stochastic Task Times." *AIIE Transactions* 8 (1976): 234-240.

Kukchin, J, E Dar-El and J Rubinobitz. "Mixed-model Assembly Line Design in a Make-to-order Environment." *Computers and Industrial Engineering* 41 (2002): 405-421.

Lapierre, S. D. and A. D. Ruiz. "Balancing assembly lines: An industrial case study." *Journal of Operational Research Society* 55 (2004): 589-597.

Lau, H and A Shtub. "An Exploratory Study on Stopping a Paced Line when Incompletions Occur." *IIE Transactions* 19 (1987): 463-467.

Lee, T, Y Kim and YK Kim. "Two-sided Assembly Line Balancing to Maximize Work Relatedness and Slackness." *Computers and Industrial Engineering* 40 (2001): 273-292.

Macaskill, J. "Production-line Balances for Mixed-model Lines." *Management Science* 19 (1972): 423-434.

Mather, H. *Competitive Manufacturing*. Englewood Cliffs: Prentice Hall, 1989.

Merengo, C, F Nava and A Pozetti. "Balancing and Sequencing Manual Mixed-model Assembly Lines." *International Journal of Production Research* 37 (1999): 2835-2860.

Meyr, H. "Supply Chain Planning in the German Automotive Industry." *OR Spectrum* 26 (2004): 447-470.

Moodie, C and H Young. "A Heuristic Method of Assembly Line Balancing for Assumptions of Constant or Variable Work Element Times." *Journal of Industrial Engineering* 16 (1965): 23-29.

Nkasu, M and K Leung. "A Stochastic Approach to Assembly Line Balancing." *International Journal of Production Research* 33 (1995): 975-991.

Pastor, R and A Corominas. "Assembly Line Balancing with Incompatibilities and Bounded Workstations." *Ricerca Operativa* 30 (2000): 23-45.

Pine, B. *Mass Customization: The New Frontier in Business Competition*. Boston: Harvard Business School Press, 1993.

Pinto, P, D Dannenbring and B Khumawala. "A Heuristic Network Procedure for the Assembly Line Balancing Problem." *Naval Research Logistics Review* 25 (1978): 299-307.

Rao, D. "Single and Mixed-model Assembly Line Balancing Methods for both Deterministic and Normally Distributed Work Element Times." *M.S. Thesis, Industrial Engineering Department, Oregon State University* (1971).

Rekiek, Brahim and Alain Delchambre. *Assembly Line Design*. London: Springer-Verlag, 2006.

Roberts, S and C Villa. "On a Multiproduct Assembly Line Balancing Problem." *AIIE Transactions* 2 (1970): 361-364.

Roder, A and B Tibken. "A Methodology for Modeling Inter-company Supply Chains and for Evaluating a Method of Integrated Product and Process Documentation." *European Journal of Operational Research* 169 (2006): 1010-1029.

Roe, Joseph Wickham. *English and American Tool Builders*. New Haven, CT: Yale University Press, 1916.

Salveson, M. "The Assembly Line Balancing Problem." *The Journal of Industrial Engineering* 6.3 (1955): 18-25.

Sarin, S and E Erel. "Development of Cost Model for the Single-model Stochastic Assembly Line Balancing Problem." *International Joural of Production Research* 28 (1990): 1305-1316.

Sarin, S, E Erel and E Dar-El. "A Methodology for Solving Single-model, Stochastic Assembly Line Balancing Problems." *Omega* 27 (1999): 525-535.

Scholl, A. *Balancing and Sequencing Assembly Lines*. 2nd. Heidelberg: Physica-Verlag, 1999.

Schöniger, J and J Spingler. "Planung der Montageanlage." *Technica* 14 (1989): 27-32.

Shtub, A and E Dar-El. "A Methodology for the Selection of Assembly Systems." *International Journal of Production Research* 27 (1989): 175-186.

Silverman, F and J Carter. "A Cost-based Methodology for Stochastic Line Balancing with Intermittent Line Stoppages." *Management Science* 32 (1986): 455-463.

Simaria, A and P Vilarinho. "2-ANTBAL: An Ant Colony Optimization Algorithm for Balancing Two-sided Assembly Lines." *Computers and Industrial Engineering* 56 (2009): 489-506.

Smith, Adam. *The Wealth of Nations*. London: Methuen & Co., Ltd., 1776.

Sniedovich, M. "Analysis of a Preference Order Assembly Line Problem." *Management Science* 27 (1981): 1067-1080.

Sphicas, G and F Silverman. "Deterministic Equivalents for Stochastic Assembly Line Balancing." *AIIE Transactions* 8 (1976): 280-282.

Sumichrast, R and R Russell. "Evaluating Mixed-model Assembly Line Sequencing Heuristics for Just-in-time Production Systems." *Journal of Operations Management* 9 (1990): 371-390.

Sumichrast, R, R Russell and B Taylor. "A Comparative Analysis of Sequencing Procedures for Mixed-model Assembly Lines in a Just-in-time Production System." *International Journal of Production Research* 30 (1992): 199-214.

Thomopoulos, N. "Mixed Model Line Balancing with Smoothed Station Assignments." *Management Science* 16 (1970): 593-603.

Tonge, F. *A Heuristic Program for Assembly Line Balancing*. Prentice-Hall, 1961.

—. "Assembly Line Balancing Using Probabilistic Combinations of Heuristics." *Management Science* 11 (1965): 727-735.

—. "Summary of a Heuristic Line Balancing Procedure." *Management Science* 7 (1960): 21-42.

Townsend, Beverly. *The Basics of Line Balancing and JIT Kitting*. Boca Raton, FL: Taylor & Francis Group, 2012.

Vilarinho, P and A Simaria. "A Two-stage Heuristic Method for Balancing Mixed-model Assembly Lines with Parallel Workstations." *International Journal of Production Research* 40 (2002): 1405-1420.

Wee, T and M Magazine. "Assembly Line Balancing as Generalized Bin Packing." *Operations Research Letters* 1.2 (1982): 56-59.

Yano, C and A Bolat. "Survey, Development, and Application of Algorithms for Sequenced Paced Assembly Lines." *Journal of Manufacturing and Operations Management* 2 (1989): 172-198.