An Introduction to WIT: Watson Implosion Technology

(Second Edition)

Robert Wittrock
Department of Mathematical Sciences
IBM T.J. Watson Research Center
P.O. Box 218
Yorktown Heights, NY 10598

January 8, 2013

Abstract

Watson Implosion Technology (WIT) is a software tool developed at IBM Research that solves a class of resource-constrained production planning problems called "implosion" problems. WIT provides two alternative approaches to solving implosion problems: one based on mathematical optimization and one based on a specialized heuristic. This paper presents a fairly detailed introduction to WIT: its essential model, solution approaches, optional capabilities, application program interfaces, and visualization tool, concluding with a brief discussion of several instances of productive use of WIT in solving a variety of resource-constrained planning problems, including some that fall outside the domain of production planning.

1 Introduction

The kind of production planning problem addressed by WIT is called an "implosion" problem: The data for an implosion problem includes a listing of the basic data objects of the problem: capacity parts, raw material parts, subassemblies, final products, and demands for the products, as well as a multi-level Bill-Of-Manufacturing network that defines how the products can be produced from subassemblies, raw materials and capacities. It also includes various attributes on these objects such as supply volumes, demand volumes, production usage rates, and either economic data or priorities. In addition, the problem data is defined with respect to a set of discrete time periods. Given these inputs, WIT computes a production plan that specifies how much of each product is to be produced in each time period and how much of each demand is to be fulfilled in each time period. The production plan is computed by either of two techniques, at the user's option: (a) "optimizing implosion", in which economic data is defined and the implosion problem is formulated and solved as a mathematical optimization problem, or (b) "heuristic implosion", in which priorities are associated with the demands and the implosion problem is solved by a specialized greedy heuristic.

While the implosion problem is nominally stated as a production planning problem, historically WIT's users have actually found this model flexible enough to be applied effectively to a variety of resource-constrained planning problems, including distribution planning, available-to-promise, reverse logistics, and even workforce allocation. WIT is applied to practical problems by building an application program in either C++, C, or Java, which accesses WIT's modeling and solving capabilities by making function calls to one of WIT's application program interfaces (APIs). Is it available on the Linux, Windows, and AIX platforms.

There a number of other documents describing various aspects of WIT. Dietrich and Escudero (1990) present an early predecessor of the implosion model used in WIT. The optimizing implosion and heuristic implosion techniques have each been patented (Dietrich and Wittrock, 1997, and Dietrich and Wittrock, 1996). Dietrich, et. al. (2005) describe the implosion model and discuss how it has been used in the context of IBM manufacturing. The WIT User's Guide and Reference (IBM 2012) (hereafter referred to as the WIT Guide) provides full details on WIT's capabilities and how to access them through the C/C++ API. Access through the Java API is explained in Wittrock (2012).

The intent of this paper is to provide a rather detailed introduction to WIT for new and potential users, including the following topics:

- The implosion model
- Optimizing implosion
- Heuristic implosion
- Numerous additional capabilities that extend these basic implosion approaches
- The application program interfaces
- A visualization tool for WIT models
- A discussion of some the WIT application programs that are currently in productive use

WIT has benefited from the contributions of quite a number of people. Brenda Dietrich was responsible for the initial formulation of WIT's implosion linear programming model (based on previous work of hers with Laure Escudero), as well as the design of the basic (default) heuristic implosion algorithm. The design and early implementation of the C/C++ API were due to J.P. Fasano. Most of the additional capabilities of WIT discussed in this paper were due to Bob Wittrock. Altogether, the following people have contributed directly to WIT's design and implementation: Dan Connors, Brenda Dietrich, Tom Ervolina, J.P. Fasano, Jean Gagnon, Juan Lafuente, Grzegorz Swirszcz, and Bob Wittrock.

2 The Essential Implosion Model

We begin by defining the "essential" implosion model: a simplified version of the implosion model consisting only of aspects relevant both to optimizing implosion and heuristic implosion. Various extensions to this essential model will be discussed in later sections. (Also, there are some minor aspects of the model that are discussed only in the WIT Guide.)

The essential implosion model involves the following concepts: data objects, input data attributes associated with each of the data objects, the output data attributes that make up a solution to an implosion problem, and the constraints that apply to such a solution. Each of these concepts will be described below.

Data Objects

An implosion problem is built up out of several kinds of data objects:

Periods: The periods of an implosion problem are a set of non-overlapping intervals of time. The duration of a period may be a day, a week, two months, a year, etc., and in fact the periods in an implosion problem are allowed to be differing durations. Let nPeriods be the number of periods. The periods are represented by the integers from 0 to nPeriods $-$ 1.

Parts: These represent capacity resources, raw materials, subassemblies, and even the final products themselves.

Materials: These are parts that remain available in the next period, if they are not used in a given period.

Capacities: These are parts that do not remain available in the next period, if they are not used in a given period. Each part is either a material or a capacity, but not both.

Products: These are the parts that can be produced. They may be either materials or capacities.

Demands: Each demand is associated with one part and represents demand for that part. Each part is associated with zero or more demands.

Operations: These represent the means by which products are produced. Each product is produced by one or more operations. If a product is produced by more than one operation, this indicates that there is more than one way to produce that product. Each operation produces any number of products. In most cases, an operation produces exactly one product. If an operation produces more than one product, this is called co-production. When an operation is "executed", some quantity of parts are consumed and some quantity of products are produced.

BOM Entries: Associated with each operation, there is a list called a Bill-Of-Manufacturing (BOM). The entries in this list are called BOM entries. Each BOM entry represents the consumption of one part as a consequence of executing one operation. The part associated with a BOM entry is called its "consumed part", and the operation associated with a BOM entry is called its "consuming operation". The BOM of an operation is a list of all BOM entries whose consuming operation is that operation.

Substitutes: A substitute is associated with one BOM entry (called its "replaced BOM entry") and one part (called its "consumed part"). A substitute represents the consumption of its consumed part instead of the consumed part of its replaced BOM entry an optional consequence of executing the consuming operation of the replaced BOM entry.

BOP Entries: Associated with each operation, there is a list called a Bill-Of-Products (BOP). The entries in this list are called BOP entries. Each BOP entry represents the production of one part as a consequence of executing one operation. The part associated with a BOP entry is called its "produced part", and the operation associated with a BOP entry is called its "producing operation". The BOP of an operation is a list of all BOP entries whose producing operation is that operation.

Notation

The following notation will be useful in discussing implosion problems:

Let

allPers	= The set of all periods: the "time horizon"
	$= \{0, 1, \ldots, \text{nPeriods} - 1\}$
allParts	= The set of all parts
allMats	= The set of all materials
allDems	= The set of all demands
allOpns	= The set of all operations
allBomEnts	= The set of all BOM entries
allSubs	= The set of all substitutes
allBopEnts	= The set of all BOP entries

$\forall\, d \in \text{allDems}$:	Let d.myPart	= The part demanded by demand d.
$\forall\, b \in \text{allBomEnts}$:	Let b.myPart	= The consumed part for BOM entry b.
	Let b.myOpn	= The consuming operation for BOM entry b.
$\forall\, s \in \text{allSubs}$:	Let s.myPart	= The consumed part for substitute s.
	Let s.myBomEnt	= The replaced BOM entry for substitute s.
$\forall\, k \in \text{allBopEnts}$:	Let k.myPart	= The produced part for BOP entry k.
	Let k.myOpn	= The producing operation for BOP entry k.

$\forall\, p \in \text{allParts}$:

Let $p.\text{myDems} = \{d \in \text{allDems} \mid d.\text{myPart} = p\}$

This is the set of all demands for part p.

$\forall\, b \in \text{allBomEnts}$:

Let $b.\text{mySubs} = \{s \in \text{allSubs} \mid s.\text{myBomEnt} = b\}$

This is the set of substitutes for BOM entry b.

Cycles

There is one restriction on the structure of an essential implosion problem. A "cycle" in an implosion problem is an ordered list of parts such that each part in the list can be built by consuming the next part in the list and where the last part in the list is the same as the first. An essential implosion problem is not allowed to have cycles. (This restriction will be partially relaxed in a subsequent section.)

Input Data Attributes

Associated with each data object in an essential implosion problem, there are various data attributes that specify aspects of the problem:

nPeriods (global): A positive integer.
As mentioned above, this is the number of periods in the problem. A vector of length = nPeriods will be called a "horizon-length" vector.

supplyVol (for parts): A horizon-length non-negative real-valued vector.
$\forall\, p \in$ allParts, $\forall\, t \in$ allPers, p.supplyVol[t] specifies the number of units of part p that are newly available in period t.

demandVol (for demands): A horizon-length non-negative real-valued vector.
$\forall\, d \in$ allDems, $\forall\, t \in$ allPers, d.demandVol[t] specifies the number of units of part d.myPart that are demanded by demand d in period t.

consRate (for BOM entries): A horizon-length non-negative real-valued vector.
$\forall\, b \in$ allBomEnts, $\forall\, t \in$ allPers, b.consRate[t] specifies the number of units of part b.myPart that must consumed due to BOM entry b for each unit of operation b.myOpn that is executed in period t without substitution.

consRate (for substitutes): A horizon-length non-negative real-valued vector.
$\forall\, s \in$ allSubs, $\forall\, t \in$ allPers, s.consRate[t] specifies the number of units of part s.myPart that must be consumed due to substitute s for each unit of operation s.myBomEnt.myOpn that is executed in period t using substitute s.

productRate (for BOP entries): A horizon-length non-negative real-valued vector.
$\forall\, k \in$ allBopEnts, $\forall\, t \in$ allPers, k.productRate[t] specifies the number of units of part k.myPart that are produced due to BOP entry k for each unit of operation k.myOpn that is executed in period t.

offset (for BOM entries): A horizon-length integer-valued vector.
$\forall\, b \in$ allBomEnts, $\forall\, t \in$ allPers, whenever operation b.myOpn is executed in period t, the consumption of part b.myPart that is due to BOM entry b occurs in period $t - b$.offset[t]. If $t - b$.offset[t] falls outside of the time horizon, then operation b.myOpn cannot be executed in period t. Also, $\forall\, s \in$ allSubs, $\forall\, t \in$ allPers, whenever operation s.myBomEnt.myOpn is executed in period t, the consumption of part s.myPart that is due to substitute s occurs in period $t - s$.myBomEnt.offset[t].

offset (for BOP entries): A horizon-length integer-valued vector.
$\forall\, k \in$ allBopEnts, $\forall\, t \in$ allPers, whenever operation k.myOpn is executed in period t, the production of part k.myPart that is due to BOP entry k occurs in period $t - k$.offset[t]. If $t - k$.offset[t] falls outside of the time horizon, then operation k.myOpn cannot be executed in period t.

Derived Relations

The following derived relations are useful for defining the essential implosion model:

$\forall\, p \in$ allParts, $\forall\, t \in$ allPers:

p.consBomPairs$[t]$ =

$\{(b, t') \mid b \in \text{allBomEnts}, t' \in \text{allPers}, b.\text{myPart} = p, \text{ and } t' - b.\text{offset}[t'] = t\}$

This is the set of all pairs (b, t') such that executing operation b.myOpn in period t' causes part p to be consumed in period t, due to BOM entry b.

$\forall\, p \in$ allParts, $\forall\, t \in$ allPers:

p.consSubPairs$[t]$ =

$\{(s, t') \mid s \in \text{allSubs}, t' \in \text{allPers}, s.\text{myPart} = p, \text{ and } t' - s.\text{myBomEnt.offset}[t'] = t\}$

This is the set of all pairs (s, t') such that using substitute s in period t' causes part p to be consumed in period t.

$\forall\, p \in$ allParts, $\forall\, t \in$ allPers:

p.prodBopPairs$[t]$ =

$\{(k, t') \mid k \in \text{allBopEnts}, t' \in \text{allPers}, k.\text{myPart} = p, \text{ and } t' - k.\text{offset}[t'] = t\}$

This is the set of all pairs (k, t') such that executing operation k.myOpn in period t' causes part p to be produced in period t, due to BOP entry k.

As mentioned above, an operation cannot be executed in a period if the offset on any of its BOM entries or BOP entries implies consumption or production outside of the time horizon. Specifically, a period t is a "disallowed execution period" for operation j, if and only if:

- there is a BOM entry b such that $b.\text{myOpn} = j$ and
  $t - b.\text{offset}[t] < 0$ or
  $t - b.\text{offset}[t] \geq \text{nPeriods}$, or
- there is a BOP entry k such that $k.\text{myOpn} = j$ and
  $t - k.\text{offset}[t] < 0$ or
  $t - k.\text{offset}[t] \geq \text{nPeriods}$.

From this definition, the following set may be defined:

$\forall\, j \in$ allOpns:

Let $j.\text{disPers} = \{t \in \text{allPers} \mid t \text{ is a disallowed period for operation } j\}$

Implosion Solution

A solution to an essential implosion problem consists of the following output data attributes:

shipVol (for demands): A horizon-length real-valued vector. $\forall\, d \in$ allDems, $\forall\, t \in$ allPers, d.shipVol$[t]$ specifies the number of units of part d.myPart that are to be "shipped" in period t in satisfaction of demand d.

execVol (for operations): A horizon-length real-valued vector. $\forall\, j \in$ allOpns, $\forall\, t \in$ allPers, j.execVol$[t]$ specifies the number of units of operation j that are to be executed in period t.

subVol (for substitutes): A horizon-length real-valued vector. $\forall\, s \in$ allSubs, $\forall\, t \in$ allPers, s.subVol$[t]$ specifies the number of units of operation s.myBomEnt.myOpn that are to be executed in period t using substitute s instead of BOM entry s.myBomEnt.

stockVol (for materials): A horizon-length real-valued vector. $\forall\, m \in \text{allMats}$, $\forall\, t \in \text{allPers}$, $m.\text{stockVol}[t]$ specifies the number of units of material m that are to be carried-over from period t to period $t+1$.

scrapVol (for parts): A horizon-length real-valued vector. $\forall\, p \in \text{allParts}$, $\forall\, t \in \text{allPers}$, $p.\text{scrapVol}[t]$ specifies the number of units of part p that are to be discarded in period t.

Implosion Constraints

A solution to a essential implosion problem is required to satisfy the following constraints:

Non-negativity Constraints:

$$\begin{array}{llll}
\forall\, d \in \text{allDems}, & \forall\, t \in \text{allPers}: & d.\text{shipVol}[t] & \geq 0 \\
\forall\, j \in \text{allParts}, & \forall\, t \in \text{allPers}: & j.\text{execVol}[t] & \geq 0 \\
\forall\, s \in \text{allSubs}, & \forall\, t \in \text{allPers}: & s.\text{subVol}[t] & \geq 0 \\
\forall\, m \in \text{allMats}, & \forall\, t \in \text{allPers}: & m.\text{stockVol}[t] & \geq 0 \\
\forall\, p \in \text{allParts}, & \forall\, t \in \text{allPers}: & p.\text{scrapVol}[t] & \geq 0
\end{array}$$

Disallowed Execution Constraints:

$\forall\, j \in \text{allOpns}$, $\forall\, t \in j.\text{disPers}$: $j.\text{execVol}[t] = 0$

These constraints ensure that an operation will not be executed when an offset on any of its BOM entries or BOP entries implies consumption or production outside of the time horizon.

Cumulative Shipment Constraints:

$\forall\, d \in \text{allDems}$, $\forall\, t \in \text{allPers}$:

$$\sum_{t'=0}^{t} d.\text{shipVol}[t'] \leq \sum_{t'=0}^{t} d.\text{demandVol}[t']$$

These constraints ensure that cumulative shipment does not exceed cumulative demand. This is equivalent to disallowing early shipment, i.e., shipping in a period earlier than the period in which the corresponding demand volume occurs.

Note that late shipment is allowed. It will be shown later that each of the two implosion techniques (optimizing and heuristic) provides a means of discouraging late shipment, but it is not prohibited by the constraints.

Substitution Constraints:

$\forall\, b \in \text{allBomEnts}$, $\forall\, t \in \text{allPers}$:

$$\sum_{s \in b.\text{mySubs}} s.\text{subVol}[t] \leq b.\text{myOpn}.\text{execVol}[t]$$

These constraints ensure that, for each BOM entry in each period, the total substitution volume does not exceed the execution volume of the corresponding operation.

Resource Allocation Constraints:

$\forall\, p \in \text{allParts},\ \forall\, t \in \text{allPers}$:

$$
\begin{aligned}
& \sum_{(b,t') \in p.\text{consBomPairs}[t]} b.\text{consRate}[t'] \bullet \left[b.\text{myOpn.execVol}[t'] - \sum_{s \in b.\text{mySubs}} s.\text{subVol}[t'] \right] \\
+ & \sum_{(s,t') \in p.\text{consSubPairs}[t]} s.\text{consRate}[t'] \bullet s.\text{subVol}[t'] \\
+ & \sum_{d \in p.\text{myDems}} d.\text{shipVol}[t] \\
+ & \quad p.\text{stockVol}[t] \\
+ & \quad p.\text{scrapVol}[t] \\
= & \quad p.\text{supplyVol}[t] \\
+ & \quad p.\text{stockVol}[t-1] \\
+ & \sum_{(k,t') \in p.\text{prodBopPairs}[t]} k.\text{prodRate}[t'] \bullet k.\text{myOpn.execVol}[t']
\end{aligned}
$$

(For notational simplicity, we assume $p.\text{stockVol}[-1] = 0$, and if p is a capacity, $p.\text{stockVol}[t] = 0$.)

The resource allocation constraints ensure that for each part in each period, the total allocated amount of the part in the period is equal to the total available amount of the part in the period. The total allocated amount is the sum of the total amount consumed due to BOM entries, the total amount consumed due to substitution, the total amount shipped, the amount carried over to the next period, and the amount scrapped. The total available amount is the sum of the supply, the amount carried over from the previous period, and the total amount produced due to BOP entries.

A feasible essential implosion solution is one that satisfies all of the above constraints.

Problem Statement

With the all of the necessary concepts now defined, the essential implosion problem can be stated as follows: Given a set of periods, parts, demands, operations, BOM entries, substitutes, and BOP entries, along with their associated input data attributes, find a feasible essential implosion solution that is, in some suitable sense, "desirable". At this generic level, the concept of a "desirable" implosion solution is left undefined. WIT's two methods of solving the implosion problem (optimizing implosion and heuristic implosion) define the concept of a "desirable" implosion solution quite differently and will be discussed in subsequent sections.

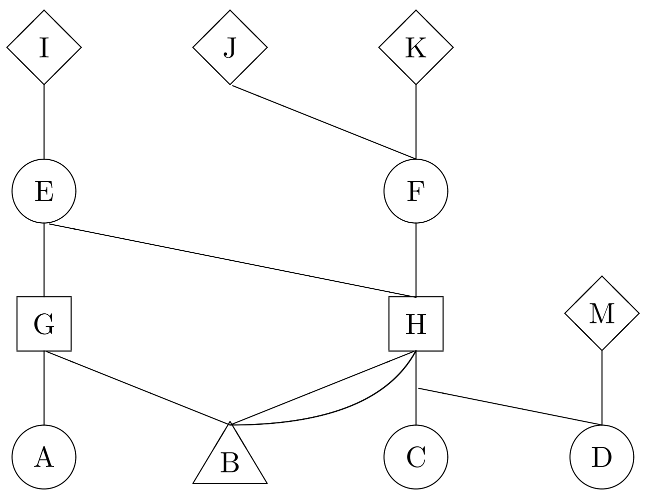
Note: Discrete Part Scheduling

One class of problems that WIT is not suited for is that of discrete part scheduling. A key aspect of the implosion model is the partitioning of the time horizon into disjoint periods, and the modeling of any activity by counting the number of units occurring in each period. Such an approach usually doesn't lend itself to solving problems in which one is concerned with the chronology of each item on an individual basis. Generally WIT is appropriate for somewhat aggregate planning, but not detailed scheduling.

3 WIT Diagrams

In many cases, it is helpful to visualize a small implosion problem (or a small portion or a large implosion problem) by using a diagram. For this purpose, a graphical notation has been defined which enables one to construct a "WIT Diagram" of an implosion problem. To illustrate, the following is a WIT diagram of a very small implosion problem:

Figure 1: Example WIT Diagram



The elements of a WIT diagram correspond to the data objects of an implosion problem:

- A material is indicated by a circle containing the name of the material.
- A capacity is indicated by a triangle containing the name of the capacity.
- A demand is indicated by a diamond containing the name of the demand with a stem extending downward to the associated part.
- An operation is indicated by a box containing the name of the operation.
- A BOM entry is indicated by a line extending from the bottom of the consuming operation to the top of the consumed part.
- A substitute is indicated by a line extending from the middle of the replaced BOM entry to the top of the consumed part.
- A BOP entry is indicated by a line extending from the top of the producing operation to the bottom of the produced part.

In figure 1:

- C is a material.
- B is a capacity.
- M is a demand for part D.
- H is an operation.
- There are two BOM entries with consuming operation H and consumed part B.
- Let b be the BOM entry whose consuming operation is H and whose consumed part is C. Then there is a substitute whose replaced BOM entry is b and whose consumed part is D.
- There is a BOP entry whose producing operation is H and whose produced part is F.

4 Optimizing Implosion

WIT's "optimizing implosion" technique solves the implosion problem by the following approach: The essential implosion problem is extended to include additional input data attributes which specify economic costs and rewards associated with each aspect of the implosion solution. A mathematical optimization problem is constructed that models this extended version of the implosion problem. This optimization problem is usually structured as a linear programming (LP) problem. The LP problem is then solved by invoking IBM ILOG CPLEX, IBM's software tool for solving various kinds of optimization problems. Finally, the resulting solution to the LP problem is converted into a solution to the implosion problem.

Most of the variables of the LP formulation correspond to the output data attributes that form a solution to an implosion problem:

$d.\text{shipVol}[t], \quad \forall\, d \in \text{allDems}, \quad \forall\, t \in \text{allPers}$

$j.\text{execVol}[t], \quad \forall\, j \in \text{allOpns}, \quad \forall\, t \in \text{allPers}$

$s.\text{subVol}[t], \quad \forall\, s \in \text{allSubs}, \quad \forall\, t \in \text{allPers}$

$m.\text{stockVol}[t], \quad \forall\, m \in \text{allMats}, \quad \forall\, t \in \text{allPers}$

$p.\text{scrapVol}[t], \quad \forall\, p \in \text{allParts}, \quad \forall\, t \in \text{allPers}$

One more set of variables is added:

$d.\text{cumShipVol}[t], \forall\, d \in \text{allDems}, \forall\, t \in \text{allPers}$

This represents the cumulative number of units of part d.myPart that are to be shipped in periods $0, \ldots, t$ in satisfaction of demand d.

The constraints of the LP formulation include most of the essential implosion constraints: non-negativity, disallowed execution, substitution, and resource allocation. However, in place of the cumulative shipment constraints from the essential implosion model, the following two sets of constraints are introduced:

LP Shipment Cumulation Constraints:

$\forall\, d \in$ allDems, $\forall\, t \in$ allPers:

$$d.\text{cumShipVol}[t] = d.\text{cumShipVol}[t-1] + d.\text{shipVol}[t]$$

where, for notational simplicity, $d.\text{cumShipVol}[-1] = 0$. These constraints ensure that $d.\text{cumShipVol}[t]$ represents the cumulative shipment volume of demand d up through period t.

LP Cumulative Shipment Constraints:

$\forall\, d \in$ allDems, $\forall\, t \in$ allPers:

$$d.\text{cumShipVol}[t] \leq \sum_{t'=0}^{t} d.\text{demandVol}[t']$$

These constraints ensure that cumulative shipment does not exceed cumulative demand. Together, the LP shipment cumulation constraints and the LP cumulative shipment constraints are equivalent to the cumulative shipment constraints of the essential implosion model, but use explicit variables to model cumulative shipment.

The objective function of the LP formulation is specified by the following additional input data attributes:

shipReward (for demands): A horizon-length real-valued vector.
$\forall\, d \in$ allDems, $\forall\, t \in$ allPers, $d.\text{cumShipReward}[t]$ specifies the reward received for each unit of part $d.\text{myPart}$ that is shipped to demand d in period t.

cumShipReward (for demands): A horizon-length real-valued vector.
$\forall\, d \in$ allDems, $\forall\, t \in$ allPers, $d.\text{cumShipReward}[t]$ specifies the reward received for each unit of part $d.\text{myPart}$ that is shipped to demand d in periods $1, \ldots, t$.

execCost (for operations): A horizon-length real-valued vector.
$\forall\, j \in$ allOpns, $\forall\, t \in$ allPers, $j.\text{execCost}[t]$ specifies the cost incurred for each unit of operation j that is executed in period t.

subCost (for substitutes): A horizon-length real-valued vector.
$\forall\, s \in$ allSubs, $\forall\, t \in$ allPers, $s.\text{subCost}[t]$ specifies the cost incurred for each unit of operation $s.\text{myBomEnt.myOpn}$ that are to be executed in period t using substitute s instead of BOM entry $s.\text{myBomEnt}$.

stockCost (for materials): A horizon-length real-valued vector.
$\forall\, m \in$ allMats, $\forall\, t \in$ allPers, $m.\text{stockCost}[t]$ specifies the cost incurred for each unit of material m that is carried over from period t to period $t+1$.

scrapCost (for parts): A horizon-length real-valued vector.
$\forall\, p \in$ allParts, $\forall\, t \in$ allPers, $p.\text{scrapCost}[t]$ specifies the cost incurred for each unit of part p that is discarded in period t.

Given the above cost and reward attributes, the LP formulation for optimizing implosion is as follows:

Maximize:

$$\begin{aligned}
&\sum_{\substack{d \in \text{allDems} \\ t \in \text{allPers}}} d.\text{shipReward}[t] \bullet d.\text{shipVol}[t] \\
&+ \sum_{\substack{d \in \text{allDems} \\ t \in \text{allPers}}} d.\text{cumShipReward}[t] \bullet d.\text{cumShipVol}[t] \\
&- \sum_{\substack{j \in \text{allOpns} \\ t \in \text{allPers}}} j.\text{execCost}[t] \bullet j.\text{execVol}[t] \\
&- \sum_{\substack{s \in \text{allSubs} \\ t \in \text{allPers}}} s.\text{subCost}[t] \bullet s.\text{subVol}[t] \\
&- \sum_{\substack{m \in \text{allMats} \\ t \in \text{allPers}}} m.\text{stockCost}[t] \bullet m.\text{stockVol}[t] \\
&- \sum_{\substack{p \in \text{allParts} \\ t \in \text{allPers}}} p.\text{scrapCost}[t] \bullet p.\text{scrapVol}[t]
\end{aligned}$$

Subject To:

- The Non-negativity constraints
- The disallowed execution constraints
- The LP shipment cumulation constraints
- The LP cumulative shipment constraints
- The substitution constraints
- The resource allocation constraints

Notice the effect of cumShipReward on the objective. Specifically, suppose x units of shipment to demand d in period t are delayed until period $t+1$, and suppose $d.\text{shipReward}[t+1] = d.\text{shipReward}[t]$. Then the effect of this delay is to decrease the objective function by $d.\text{cumShipReward}[t] \bullet x$ units. In this sense, $d.\text{cumShipReward}[t]$ functions as the penalty for delaying shipments to demand d from period t to period $t+1$, and this is how optimizing implosion discourages late shipment.

5 Additional Capabilities of Optimizing Implosion

In addition to the essential aspects of described in the previous section, optimizing implosion includes a number of additional capabilities, some of which are described here:

- Bounds
- Accelerated Optimizing Implosion
- MIP Mode

- External Optimizing Implosion
- Multiple Objectives
- Stochastic Implosion

Bounds

The linear programming formulation for optimizing implosion can optionally be extended to include bounds on the following three classes of variables:

j.execVol[t], $\forall\, j \in$ allOpns, $\forall\, t \in$ allPers

s.subVol[t], $\forall\, s \in$ allSubs, $\forall\, t \in$ allPers

d.cumShipVol[t], $\forall\, d \in$ allDems, $\forall\, t \in$ allPers

For each variable, there are three bounds: a hard lower bound, a soft lower bound, and a hard upper bound. In the case of the execVol variables, these bounds are specified by the following three input data attributes:

execBounds.hardLowerBound (for operations):
A horizon-length non-negative real-valued vector. (default value = 0)

execBounds.softLowerBound (for operations):
A horizon-length non-negative real-valued vector. (default value = 0)

execBounds.hardUpperBound (for operations):
A horizon-length non-negative real-valued vector or $+\infty$. (default value = $+\infty$)

The values of these attributes must also satisfy the following condition:

hardLowerBound[t] $\leq$ softLowerBound[t] $\leq$ hardUpperBound[t], $\forall\, t \in$ AllPers

For any operation, j, and period, t, j.execBounds.hardLowerBound[t] defines a lower bound on the value of j.execVol[t], i.e., the following constraint is added to the linear programming formulation:

j.execVol[t] $\geq$ j.execBounds.hardLowerBound[t]

Similarly j.execBounds.hardUpperBound[t] defines a upper bound on the value of j.execVol[t].

And finally, j.execBounds.softLowerBound[t] defines a value such that, if j.execVol[t] takes on a lower value than this, a linear penalty is incurred.

The bounds on the subVol variables and cumShipVol variable are defined in the same way as for the execVol variables, i.e., they are specified by the following six input data attributes, which have a similar interpretation:

subBounds.hardLowerBound (for substitutes):
A horizon-length non-negative real-valued vector. (default value = 0)

subBounds.softLowerBound (for substitutes):
A horizon-length non-negative real-valued vector. (default value = 0)

subBounds.hardUpperBound (for substitutes):
A horizon-length non-negative real-valued vector or $+\infty$. (default value = $+\infty$)

cumShipBounds.hardLowerBound (for demands):
A horizon-length non-negative real-valued vector. (default value = 0)

cumShipBounds.softLowerBound (for demands):
A horizon-length non-negative real-valued vector. (default value = 0)

cumShipBounds.hardUpperBound (for demands):
A horizon-length non-negative real-valued vector or $+\infty$. (default value = $+\infty$)

Finally, there is one additional input data attribute pertaining to bounds:

wbounds (global):
A scalar non-negative real number. (default value = 10000)

wbounds is the per-unit penalty on the sum of all violations of soft lower bounds. It is recommended that this be set to a large enough value that at an optimal solution, all soft lower bounds will be satisfied, if there is any feasible solution that does so.

In general it recommended that applications use soft lower bounds rather than hard lower bounds, because hard lower bounds can cause the implosion problem to be infeasible, while soft lower bounds (and hard upper bounds) always allow the implosion problem to be feasible.

Accelerated Optimizing Implosion

In some applications, optimizing implosion is used repetitively, for example using the following sequence of steps:

1. Specify an implosion problem to WIT.
2. Invoke optimizing implosion.
3. Alter the input data.
4. Re-invoke optimizing implosion.
5. Repeat this process numerous times.

In many cases, all but the first call to optimizing implosion can be made to run much faster than ordinary optimizing implosion by using accelerated optimizing implosion.

Accelerated optimizing implosion works as follows: At the end of the first optimizing implosion, the LP model, LP solution, and LP solver environment are kept in memory. Then, when the second optimizing implosion is invoked, the LP model and LP solver environment are updated to coincide with the new implosion problem and the LP solution from the first optimizing implosion is used as the starting solution for the second optimizing implosion. This process is repeated with each subsequent call to optimizing implosion. In tests, we found that this approach tends to run much faster than ordinary optimizing implosion.

When an application program uses accelerated optimizing implosion, two restrictions apply:

- All calls to optimizing implosion must be made from the same run of the application program.
- There are restrictions on what alterations can be made to the input data:
  - No data objects (e.g. parts) can be added.

  - Some attributes (e.g. offset) cannot be altered.

A complete list of the attributes that are allowed to be altered when using accelerated optimizing implosion is given in the WIT Guide. Some of the attributes that may be altered are:

- supplyVol
- demandVol
- costs and rewards for the objective function

Accelerated optimizing implosion uses the following additional input data attribute:

accAfterOptImp (global): A scalar boolean. (default value = FALSE)

To use accelerated optimizing implosion, the application program sets accAfterOptImp to TRUE. The first optimizing implosion invoked after accAfterOptImp has been set will be an ordinary optimizing implosion, but at its conclusion, it will leave the necessary data in memory. All subsequent invocations of optimizing implosion will be accelerated optimizing implosion, provided the input data has not been altered in a way that is disallowed for accelerated optimizing implosion.

MIP Mode

As explained previously, WIT normally formulates the optimizing implosion problem as a linear programming problem. In particular, this means that all the decision variables are continuous variables: they are allowed to taken on values that are either integer or non-integer. As an alternative, WIT can formulate the optimizing implosion problem as a mixed integer programming (MIP) problem. This is called "MIP mode".

When WIT is in MIP mode, the following aspects of the implosion solution can be required to take on integer values only:

- The execVol of any operation
- The subVol of any substitute
- The shipVol of any demand

Specifically, there is a boolean attribute for operations that specifies whether or not the execVol of the operation is required to be integer and there are similar boolean attributes for substitutes and demands. There is also a global boolean attribute, mipMode, that specifies whether or not MIP mode is to be used at all.

WIT solves the MIP problem by invoking CPLEX.

MIP mode is considered to be an experimental aspect of WIT and should be used with caution: The resulting MIP problem may take vastly longer to solve than the corresponding LP problem.

External Optimizing Implosion

Normally, optimizing implosion proceeds as follows:

1. WIT formulates the implosion problem as an LP or MIP problem.

2. WIT invokes CPLEX to solve the LP/MIP problem.
3. WIT constructs the implosion solution from the solution to the LP/MIP problem.

In external optimizing implosion, Step 2 above is replaced with a series of steps that allow the LP/MIP problem to be solved by any means the application developer chooses. Specifically, external optimizing implosion consists of the following steps:

1. WIT formulates the implosion problem as an LP or MIP problem.
2. The application program obtains the LP/MIP problem from WIT.
3. The application program solves the LP/MIP problem, using any suitable optimization software and using whatever function calls, parameters, etc. are appropriate.
4. The application program loads the solution to the LP/MIP problem into WIT.
5. WIT constructs the implosion solution from the solution to the LP/MIP problem.

From the application program's point of view, this is a much more complicated process than ordinary optimizing implosion: The application program must make explicit calls to WIT in order for steps 1, 2, 4, and 5 to be carried out, and the application program takes full responsibility for step 3. Nevertheless, external optimizing implosion can be worth using in cases where the LP/MIP problem is particularly difficult to solve by WIT's predetermined calls to CPLEX. For example, it could be helpful in MIP mode, if the resulting MIP problem requires the use of special techniques in order to obtain a high quality integer solution.

Multiple Objectives

Ordinarily, optimizing implosion works with a single objective function: It finds an implosion solution that achieves the maximum feasible value of a single linear function of the decision variables that make up the solution. In some cases, it may be desirable to have optimizing implosion find an implosion solution that achieves the maximum feasible value of several linear objective functions, where the objective functions are treated as a strict hierarchy. This can be done in WIT by using "multiple objectives mode".

In multiple objectives mode, each input data attribute that specifies an aspect of the objective function is allowed to have a different value for each objective. For example, if there are three objectives, then the execVol of a given operation in a given period is allowed to take on three different values: one associated with each objective.

The objectives are given a rank ordering. That is, there's a first objective, a second objective, etc. When optimizing implosion is invoked, an implosion solution is found that achieves the "lexicographic maximum value" of the objective functions with respect to this rank ordering. Specifically, this means the following:

1. The implosion solution will achieve the maximum feasible value of the first objective.
2. The implosion solution will achieve the maximum feasible value of the second objective, subject to condition 1.
3. The implosion solution will achieve the maximum feasible value of the third objective, subject to conditions 1 and 2.

4. etc.

WIT solves the optimizing implosion problem with multiple objectives by formulating a sequence of LP (or MIP) problems and solving them with CPLEX.

Stochastic Implosion

Stochastic implosion is a variant of the optimizing implosion in which some of the input data is treated probabilistically, resulting in an implosion problem structured as a "two-stage stochastic linear programming problem with recourse". Specifically, a stochastic implosion problem models a situation in which the decisions are to be made in two distinct stages:

- Some of the decisions must be made during stage 0, in which much of the input data is known deterministically, but the values of some of the input data attributes are specified probabilistically, as random variables.
- The remaining decisions are to be made during stage 1, at which point the values of all probabilistic data are known with certainty.

Normally, the stages of a stochastic implosion problem correspond to groups a time periods: the early periods are in stage 0 and the later periods are in stage 1. WIT has an attribute that specifies which periods belong to each stage.

The probabilistic aspects of a stochastic implosion problem are specified via a finite set of "scenarios". A scenario is a complete specification of the values of all of the probabilistic data in the problem. The set of all scenarios corresponds to the set of all possible outcomes of the random variables in the problem data. Each scenario has a probability associated with it and the sum of all scenario probabilities must be equal to 1.

Some of the input attributes of a stochastic implosion problem are "scenario-specific", i.e., they have a potentially different value in each scenario during stage 1. For example, supplyVol is a scenario-specific attribute, so a given part in a given period (in stage 1) can have a different supplyVol in each scenario. Other scenario-specific attributes include demandVol, bounds, and the objective function attributes, such as shipReward. However, even a scenario-specific attribute must have the same value in all scenarios during stage 0 periods.

A stochastic implosion problem is a model of the following decision process:

1. All stage 0 decisions are made.
2. One scenario occurs at random, according to the scenario probabilities.
3. All stage 1 decisions are made.

The solution of a stochastic implosion problem is a fully-specified contingency plan for the above decision process. It specifies:

- The value of each stage 0 decision variable.
- For each scenario, s, the value that each stage 1 decision variable is to take, if (hypothetically) scenario s occurs.

This contingency plan is given through WIT's solution attributes, such as execVol, and these attributes are scenario-specific. The stochastic implosion solution is required to be feasible in both stages in all scenarios. The objective function is taken to be the expected value of the objective function over all of the scenarios.

WIT solves the stochastic implosion problem by formulating it as a single LP problem and solving the LP problem with CPLEX.

6 Heuristic Implosion: The Default Heuristic

WIT's "heuristic implosion" technique solves the implosion problem by the following approach: The essential implosion problem is extended to include an additional input data attribute which specifies priorities for the demands. This extended version of the implosion problem is solved by the use of a specially designed algorithm. This algorithm is considered to be 'heuristic": The solution that it constructs is not guaranteed to be "optimal" according to some objective function; instead, the approach is intended to construct a solution that the user will consider to be "desirable" or "satisfactory".

In its most general form, heuristic implosion has many advanced capabilities, which are controlled by numerous additional input data attributes. However, for simplicity, we will begin with a description of a highly restricted version of heuristic implosion, called the "default heuristic".

Top level Logic

The top level logic of the default heuristic uses the following additional input data attribute:

priority (for demands): A horizon-length integer-valued vector,
where $d.\text{priority}[t] \geq 1, \forall\, d \in \text{allDems}, \forall\, t \in \text{allPers}$ (default value = 1)

$\forall\, d \in \text{allDems}$, $\forall\, t \in \text{allPers}$, $d.\text{priority}[t]$ specifies the priority assigned to the meeting of any quantity of the demand associated with demand d in period t. 1 is interpreted as the highest priority, 2 is the second highest, and so on.

To see how the default heuristic solves the implosion problem, consider following set of all "allocation triples":

$\text{allocTrips} = \{(d, td, ts) \mid d \in \text{allDems},\ td \in \text{allPers},\ td \leq ts < \text{nPeriods}\}$

For any allocation triple, (d, td, ts), the values d, td, ts, and $d.\text{priority}[td]$ are considered to be the "demand", "demand period", "shipment period", and "priority" (respectively) of the allocation triple.

The heuristic implosion technique is a "greedy" heuristic approach that proceeds as follows: Initially the implosion solution is all zeros. The heuristic then processes each triple in allocTrips sequentially. (The order in which the triples are processed will be specified later.) For each triple, $(d, td, ts) \in$ allocTrips, the heuristic attempts to fulfill as much of the demand $d.\text{demandVol}[td]$ by shipping in period ts as possible. That is, the heuristic attempts to increase $d.\text{shipVol}[ts]$ by as much as possible, subject to the implosion constraints and with the additional convention that the increase to $d.\text{shipVol}[ts]$ is considered to be in fulfillment of demand d in period td, where the total fulfillment of demand d in period td must never exceed $d.\text{demandVol}[td]$. Note that the heuristic only *attempts* to achieve a maximal increase in $d.\text{shipVol}[ts]$; the increase is not actually guaranteed to be maximal.

To understand the order in which the allocation triples are processed, consider two allocation triples (d, td, ts) and (d', td', ts').

1. If the demand periods are distinct, the triple with the earlier demand period is processed first.
2. Otherwise, if the shipment periods are distinct, the triple with the earlier shipment period is processed first.
3. Otherwise, if the priorities are distinct, the triple with the "higher" priority (lower value of d.priority$[td]$) is processed first.
4. Otherwise, the triple whose demand was entered earlier in the input data is processed first.

The above four cases may be called the default heuristic's "prioritization scheme". Note that case 2 is the heuristic's way of discouraging late shipment.

Explosion

Once a particular allocation triple, (d, td, ts), has been selected, the default heuristic attempts to fulfill as much of the demand d.demandVol$[td]$ by shipping in period ts as possible. This is accomplished by an "explosion" based technique related to the calculations of Material Requirements Planning (MRP) (see Vollmann, et. al., 1992). As with MRP, the heuristic's explosion process works with "requirements". A requirement on part p in period t is a specified quantity of part p that (according to the algorithm) is needed in period t for some purpose, either for shipment or for consumption. A requirement can be "met" in any of the following ways:

- Using supply of part p.
- Using production of part p that's already been generated.
- Generating new production of part p by executing an operation that produces it (i.e., "building" part p).

This last case tends to cause more consumption, which, in turn, creates more requirements. In this way, requirements are propagated throughout the implosion problem and this propagation of requirements is called an explosion. The default heuristic uses a binary search approach to select a shipment volume that will allow all of the requirements eventually to be met.

Substitute Netting

The default heuristic makes use of substitutes by means of a technique called "substitute netting". This technique uses the following additional input data attributes:

netAllowed (for substitutes): A scalar boolean. (default value = TRUE)

expNetAversion (for substitutes): A scalar real number. (default value = 0)

Consider that situation in which the explosion determines that an operation is to be executed that has some BOM entry, b, that has at least one substitute. In this case, BOM entry b and its substitutes are used according to the following rule:

1. First, availability of part b.myPart is used, without producing more of part b.myPart.

2. Next, availability of part s_1.myPart is used, without producing more of part s_1.myPart, where s_1 is the substitute in b.mySubs for which netAllowed is TRUE that has the lowest value of expNetAversion. (Ties are broken in favor of whichever substitute was entered first in the input data.)
3. Next, availability of part s_2.myPart is used, without producing more of part s_2.myPart, where s_2 is the substitute in b.mySubs for which netAllowed is TRUE that has the second lowest value of expNetAversion.
4. And so on.
5. Finally, more of part b.myPart is produced and the resulting availability is used.

7 Additional Capabilities of Heuristic Implosion

Heuristic implosion in its most general form consists of the default heuristic described above augmented by optional capabilities that extend this basic technique in various useful directions. While a thorough explanation of all of the heuristic's optional capabilities can be found in the WIT Guide, the following seem to be the most essential:

- Build-Ahead
- Multiple Routes
- Multiple Execution Periods
- Stock Reallocation
- Lot Sizes
- Equitable Allocation
- Heuristic Allocation
- Proportionate Routing

Build-Ahead

The build-ahead capability of heuristic implosion uses the following additional input data attributes:

buildNstn (for materials): A scalar boolean. (default value = FALSE)

buildAsap (for materials): A scalar boolean. (default value = FALSE)

buildAheadUB (for materials): A horizon-length integer-valued vector,
where $0 \leq m.\text{buildAheadUB}[t] \leq \text{nPeriods} - 1, \forall\, m \in \text{allMats}, \forall\, t \in \text{allPers}$,
(default value = nPeriods - 1)

By default, when the explosion tries to meet a requirement in period t by generating new production, the production is always done in period t. It never builds in an period earlier than the period in which the requirement occurs. In contrast, when build-ahead is being used, the heuristic will also consider doing such production in periods earlier than t.

The heuristic will use build-ahead on a material m, if and only if either m.buildNstn is TRUE or m.buildAsap is TRUE. Consider a material, m, and a period, t, in which a requirement for material m is to be met by new production. If m.buildNstn is TRUE, the heuristic will attempt to build the material in the following sequence of periods: $t, t-1, \ldots, 0$. This is called "No Sooner Than Necessary" (NSTN) build-ahead. If m.buildAsap is TRUE, the heuristic will attempt to build the material in the following sequence of periods: $0, 1, \ldots, t$. This is called "As Soon As Possible" (ASAP) build-ahead. Setting m.buildNstn to TRUE and m.buildAsap to TRUE for the same material m is not allowed. The buildAheadUB attribute provides an upper bound on build-ahead: The heuristic will never build in a period earlier than $t - m$.buildAheadUB$[t]$.

In many implosion problems, NSTN build-ahead seems to be a particularly effective form of build-ahead to use: supply in earlier periods is generally more flexible than supply in later periods, since there are normally more demands that can make use of supply in earlier periods. NSTN build-ahead tends to leave the earlier, more flexible supply available to be used in order to meet other demands.

Multiple Routes

The multiple routes capability of heuristic implosion uses the following additional input data attributes:

multiRoute (global): A scalar boolean. (default value = FALSE)

expAllowed (for BOP entries): A scalar boolean. (default value = TRUE)

expAllowed (for substitutes): A scalar boolean. (default value = TRUE)

expAversion (for BOP entries): A scalar real number. (default value = 0)

expNetAversion (for substitutes): A scalar real number. (default value = 0)

The heuristic will use the multiple routes technique, if and only if the multiRoute attribute is TRUE. The multiple route technique applies in two distinct cases:

- If multiRoute is FALSE, when the explosion determines that a part is to be produced in a period t, it selects only one BOP entry to produce the part. Specifically, it considers those BOP entries that are capable of producing the part in period t and for which expAllowed ("explosion allowed") is TRUE and selects the one with the smallest expAversion ("explosion aversion"), with ties broken in favor of BOP entries entered earlier in the input data.

  If multiRoute is TRUE, the heuristic will consider the same set of BOP entries as in the multiRoute = FALSE case, but it will use potentially any or all of these BOP entries as necessary, and it will use them in order of increasing expAversion.

- Now consider the situation in which the heuristic has decided to execute an operation with a BOM entry, b, that has one or more substitutes, but there is not enough availability of the part consumed by BOM entry b and of the parts consumed by the substitutes for BOM entry b to execute the operation at the specified execution volume. If multiRoute is TRUE, then the heuristic will not only consider building more of the part consumed by BOM entry b; it will also consider building more of the part consumed by each of the substitutes for BOM entry b. Specifically, it will only build the parts consumed by substitutes for which expAllowed is TRUE, and it will build the parts consumed by these substitutes in order of increasing expNetAversion, with ties broken in favor of whichever substitute was entered earlier in the input data. However, it will always build the part consumed by BOM entry b itself before building the parts consumed by its substitutes.

Multiple Execution Periods

The multiple execution periods capability of heuristic implosion uses the following additional input data attribute:

multiExec (global): A scalar boolean. (default value = FALSE)

The heuristic will use the multiple execution periods technique, if and only if the multiExec attribute is TRUE. Consider the situation in which the heuristic has decided to produce some part, p, in some period, t, using a particular BOP entry, k, but there is more than one period, t', in which operation k.myOpn can be executed in order to produce part p in period t using BOP entry k. In such a case, if multiExec is FALSE, the heuristic will execute operation k.myOpn in only one period: the latest period t' that results in production of part p in period t using BOP entry k. If multiExec is TRUE, the heuristic will consider executing operation k.myOpn in any or all of the periods, t', that result in production of part p in period t using BOP entry k. Specifically, it will consider these periods in decreasing order.

Stock Reallocation

The stock reallocation capability of heuristic implosion uses the following additional input data attribute:

stockReallocation (global): A scalar boolean. (default value = FALSE)

The heuristic will use the stock reallocation technique, if and only if the stockReallocation attribute is TRUE. Consider the following scenario: There is a requirement for some material in some period, t_3. This requirement could be met either by consuming stock of the material that was originally available (either as supply or previously generated production) in period t_1, where $t_1 < t_3$, or by building the material in period t_3. Since the heuristic uses available quantities before generating new production, it will use the stock carried over from period t_1. Later, a requirement for the material arrives in period t_2, where $t_1 \leq t_2 < t_3$, but now there is no more supply or production available in period t_2 or earlier and furthermore, no additional quantities of the material can be produced in period t_2 or earlier. In the default heuristic, this requirement could not be met. In contrast, when stockReallocation is TRUE, the heuristic can take the availability from period t_1 that was previously used to meet the period t_3 requirement and use it to meet the period t_2 requirement and then build the material in period t_3 to re-meet the period t_3 requirement. Since the availability from period t_1 that was originally used to meet the period t_3 requirement did so by carrying over to period t_3 as stock, and then this availability was later reallocated to a different requirement, this is called “stock reallocation”.

The Core Heuristic

The four optional capabilities described above have tended to be so broadly beneficial that it now seems to be a good idea to simply use all of them in most applications of heuristic implosion.

In effect, the capabilities described so far constitutes a “core” heuristic, which may be defined as follows:

- Start with the default heuristic.
- Specify d.priority, $\forall\, d \in$ allDems.
- Set m.buildNstn = TRUE, $\forall\, m \in$ allMats.
- Set multiRoute = TRUE.
- Set multiExec = TRUE.
- Set stockReallocation = TRUE.

All other heuristic attributes (e.g., buildAheadUB) would be at left at their default values. The remaining aspects of heuristic implosion may be considered add-ons to this basic core heuristic, to be used based on the specific needs of the application.

Lot Sizes

The lot size capability of heuristic implosion uses the following additional input data attributes:

minLotSize (for operations): A horizon-length non-negative real-valued vector.
(default value = 0)

incLotSize (for operations): A horizon-length non-negative real-valued vector.
(default value = 0)

These two attributes are used to define the lot size constraints for heuristic implosion:

Lot Size Constraints:

$\forall\, j \in$ allOpns, $\forall\, t \in$ allPers:

Let $mls = j.\text{minLotSize}[t]$ and $ils = j.\text{incLotSize}[t]$.

If $j.\text{incLotSize}[t] > 0$:

$$j.\text{execVol}[t] \in \{0,\ mls,\ mls + ils,\ mls + 2 \bullet ils,\ mls + 3 \bullet ils,\ \ldots\}$$

Heuristic implosion computes an implosion solution that satisfies the lot size constraints. j.minLotSize[t] is the “minimum lot size” of operation j in period t, while j.incLotSize[t] is the “incremental lot size” of operation j in period t. Note that if j.incLotSize[t] = 0, then the lot size constraints have no effect on operation j in period t and j.execVol[t] may take on any non-negative value. (This is the default situation.)

Equitable Allocation

The equitable allocation technique of heuristic implosion uses the following additional input data attribute:

equitability (global): A scalar integer.
where $1 \leq$ equitability ≤ 100 (default value = 1)

The heuristic will use the equitable allocation technique if and only if equitability > 1. Heuristic allocation is an alternative way of breaking ties between demands of equal priority. Recall the concept of an allocation triple defined in section 5:

(d, td, ts), where $d \in$ AllDems, $td \in$ AllPers, $td \leq ts <$ nPeriods

Suppose the heuristic encounters two or more allocation triples that are tied in the heuristic's default ordering, i.e.:

- The demand period, td, is the same for each triple.
- The shipment period, ts, is the same for each triple.
- The priority, d.priority[td] is the same for each triple.

In this case, the default heuristic will process the triples in the order in which the demands were entered in the input data.

When the equitable allocation technique is being used, the heuristic attempts to treat such tied allocation triples an an equal basis. Let N be the value of the equitability attribute. For each tied allocation triple, the heuristic divides the corresponding unmet demand volume into N approximately equal portions, "partial demands". It then makes N passes through the tied allocation triples, processing one partial demand for each allocation triple during each pass.

Note that the equitable allocation technique is an approximation technique: higher values of equitability result in more precise approximations, but require longer run times.

Heuristic Allocation

Recall that the top level logic of the default heuristic involves processing the allocation triples for the implosion problem according to a fixed prioritization scheme, in which the demand period has precedence over the shipment period, which has precedence over the priority. In some cases, this prioritization scheme can be too restrictive. For example, sometimes it may be more appropriate to assign highest precedence to the priority attribute. When the heuristic's standard prioritization scheme is too limited for the specific application, the "heuristic allocation" capability can be invoked, which allows the application program to implement its own customized prioritization scheme.

In order to use the heuristic allocation capability, the application program must make calls to special functions in WIT's application program interface (API). The approach can be summarized as follows:

1. First, the application program calls the function `witStartHeurAlloc`, which initiates the heuristic allocation process.
2. Next, the application program makes many calls to the function `witIncHeurAlloc`, which "increments" the heuristic allocation process. The arguments to this function include the following:
    - A demand, d
    - A shipment period, ts
    - A real number, desIncVol ("desired incremental volume")

This function attempts to increase d.shipVol[ts] by as much as it can up to desIncVol units, while maintaining a feasible implosion solution. This is accomplished by employing the same explosion based technique that's used by ordinary heuristic implosion when it processes an allocation triple.

3. Finally, the application program calls the function `witFinishHeurAlloc`, which concludes the heuristic allocation process.

The sequence of calls to `witIncHeurAlloc` define the customized prioritization scheme that is being used by the application program.

Proportionate Routing

The proportionate routing capability of heuristic implosion uses the following additional input data attributes:

propRtg (for parts): A horizon-length boolean-valued vector. (default value = FALSE)

propRtg (for BOM entries): A horizon-length boolean-valued vector. (default value = FALSE)

routingShare (for BOP entries): A horizon-length real-valued vector,
where k.routingshare[t] $\geq 1, \forall\, k \in$ allBopEnts$, \forall\, t \in$ allPers (default value = 1)

routingShare (for BOM entries): A horizon-length real-valued vector,
where b.routingshare[t] $\geq 1, \forall\, b \in$ allBomEnts$, \forall\, t \in$ allPers (default value = 1)

routingShare (for substitutes): A horizon-length real-valued vector,
where s.routingshare[t] $\geq 1, \forall\, s \in$ allSubs$, \forall\, t \in$ allPers (default value = 1)

The proportionate routing technique applies to the same kinds of cases to which the multiple routes applies: parts with multiple BOP entries and BOM entries with one or more substitutes. The part case will be described first.

For any part, p, and period, t, the heuristic will use proportionate routing at part p in period t if and only if p.propRtg[t] is TRUE. Suppose the heuristic determines that part p is to be produced in period t. Then, if p.propRtg[t] is TRUE, the heuristic will consider all of the eligible BOP entries capable of producing part p in period t and attempt to use all of them at the same time. More precisely it will initially attempt to use the eligible BOP entries in proportion to their routingShare values in period t. However, suppose that, due to various constraints, the heuristic is not successful in building the entire amount in strict proportion. In this case, for each BOP entry through which production was found to be constrained, the production is fixed at the maximum feasible level found, and the strictly proportionate production is applied only to those BOP entries that were found to be unconstrained.

For example, suppose the heuristic is attempting to build 36 units of part p in period t through BOP entries k_1, k_3, and k_2, where:

- k_1.routingShare[t] = 1
- k_2.routingShare[t] = 2
- k_3.routingShare[t] = 3

Then the heuristic will initially attempt to build in strict proportion:

- 6 units through BOP entry k_1
- 12 units through BOP entry k_2
- 18 units through BOP entry k_3

Now suppose that the heuristic was not able to build more than 4 units through BOP entry k_2. Then the production through BOP entry k_2 would be fixed at 4 units, while BOP entries k_1 and k_3 would produce in strict proportion:

- 8 units through BOP entry k_1
- 4 units through BOP entry k_2
- 24 units through BOP entry k_3

The BOM entry case is similar. Consider the situation in which the heuristic has decided to execute an operation in period t with a BOM entry, b, that has one or more substitutes, where b.propRtg[t] is TRUE. Then the heuristic will attempt to use BOM entry b and its substitutes in proportion to their routingShare values in period t, allowing the use of constrained BOM entries and substitutes to be fixed at the maximum feasible level found.

8 Other WIT Capabilities

In addition to optimizing and heuristic implosion, WIT also provides a number of capabilities that can be used in conjunction with either of the two implosion techniques. The most essential of these will be described below:

- WIT-MRP
- Focussed Shortage Schedule
- Post-Implosion Pegging
- Unexplodeable Cycles

WIT-MRP

As an alternative or adjunct to solving the implosion problem, WIT can perform a limited form of Material Requirements Planning, called WIT-MRP. The primary output of WIT-MRP is the requirements schedule, given by the following output data attribute:

reqVol (for parts): A horizon-length non-negative real-valued vector

For any part, p, and any period, t, p.reqVol[t] is the net requirement for part p in period t. This is an amount such that, if p.supplyVol[t] were increased by p.reqVol[t] units $\forall\, p \in$ allParts, $\forall\, t \in$ allPers, then a feasible implosion solution would exist that meets all demands on time. WIT-MRP also computes and provides this hypothetical feasible implosion solution.

Focussed Shortage Schedule

WIT's "focussed shortage schedule" capability (FSS) is used after implosion has been performed and works with the resulting implosion solution to provide additional information about the relationship between the supplies and demands in the problem. FSS uses the following additional input data attribute:

fssShipVol (for demands): A horizon-length non-negative real-valued vector.
(default value = shipVol)

The fssShipVol values for all demands constitute the "FSS shipment schedule".

The main output of FSS is the following output data attribute:

focusShortageVol (for parts): A horizon-length non-negative real-valued vector.

For any part, p, and any period, t, p.focusShortageVol[t] is considered to be the "shortage" of part p in period t. This is an amount such that, if p.supplyVol[t] were increased by p.focusShortageVol[t] units $\forall\, p \in$ allParts, $\forall\, t \in$ allPers, then a feasible implosion solution would exist that meets the FSS shipment schedule. FSS also computes and provides this hypothetical feasible implosion solution.

The process by which FSS computes its hypothetical implosion solution uses the actual implosion solution as its starting point. In particular, if the FSS shipment schedule is left at its default value (i.e., equal to the implosion shipment schedule), the resulting hypothetical implosion solution will be the same as the actual implosion solution and all of the shortages will be 0. The set of demands, d, and periods, t, such that d.fssShipVol[t] $>$ d.shipVol[t] can be thought of as the "focus" of the FSS computation, i.e., the additional shipments that are desired beyond those in the implosion solution. The shortages constitute a set of additional supplies that would be sufficient to make these additional shipments feasible.

Post-Implosion Pegging

A "pegging" of an implosion solution considers each resource allocated in the solution and designates a set of shipments to which that resource is considered to have been allocated. Note that this designation is somewhat arbitrary, since heuristic implosion and optimizing implosion do not explicitly allocate resources to specific shipments; they merely generate a feasible solution to the implosion problem (guided by the priorities or objective function).

WIT provides two kinds of pegging: "post-implosion pegging" and "concurrent pegging". Post-implosion pegging will described here; for information on concurrent pegging, see the WIT Guide.

Post-implosion pegging (PIP) can be applied to any feasible implosion solution, regardless of whether the solution was computed by heuristic implosion, optimizing implosion or by any other means. It computes peggings for several attributes, including the following:

- j.execVol, $\forall\, j \in$ allOpns
- s.subVol, $\forall\, s \in$ allSubs
- p.supplyVol, $\forall\, p \in$ allParts

PIP requires two inputs:

- A feasible implosion solution
- A "shipment sequence" for the implosion solution (see below)

A shipment sequence for an implosion solution takes the shipments in the solution

$\{(d, t, d.\text{shipVol}[t]) \mid d \in \text{allDems}, t \in \text{allPers}\}$

and arranges them in some specific order, possibly breaking the shipments into partial shipments and interlacing them. A good example of a shipment sequence is the ordered list of triples $(d, t, \text{incShipVol})$ that are used at the top level of heuristic implosion and in fact, PIP provides a means of using this sequence as its shipment sequence.

The PIP technique builds up its pegging by reconstructing the implosion solution through an explosion of all the shipments in the shipment sequence. As the explosion proceeds, it passes through each of the resources (supplies, etc.) that were allocated in the implosion solution. Each allocated resource found by the explosion is pegged to one or more corresponding shipments, with precedence given to the earlier elements of the shipment sequence, i.e., the first resources found are pegged to the first corresponding elements in the shipment sequence, etc.

To use PIP, the application program calls the WIT API function `witBuildPip`, which constructs the pegging. The pegging for each pegged attribute can then be retrieved by calling an API function corresponding to the attribute. For example, the supplyVol pegging for a particular demand, d, and a particular shipment period, ts, can be retrieved by calling the function `witGetDemandSupplyVolPip`, passing d and ts as arguments. This function returns a list of triples, $(p, t, \text{peggedSupplyVol})$, where $p \in \text{allParts}$, $t \in \text{allPers}$, and peggedSupplyVol is the portion of $p.\text{supplyVol}[t]$ that was pegged to demand d in shipment period ts.

In addition to pegging resources to shipments, PIP can optionally peg resources to operations. An operation pegging answers the question, "Which resources were used to enable a particular operation to be executed in a particular period?". The operation pegging is computed so that it makes sense with the demand pegging: Individual resources are considered to "pass through" the operations to which they are pegged, while they are on their way to the demands to which they are pegged.

Unexplodeable Cycles

At stated in section 2, an essential implosion problem is not allowed to have cycles. For general implosion problems, this restriction is partially relaxed. Recall the following attribute:

expAllowed (for BOP entries): A scalar boolean. (default value = TRUE)

Define an "explodeable BOP entry" to be a BOP entry for which expAllowed = TRUE. An *explodeable* cycle is an ordered list of parts such that each part in the list can be built by consuming the next part in the list using an explodeable BOP entry and where the last part in the list is the same as the first. In the general setting, an implosion problem is allowed to have cycles; it is just not allowed to have explodeable cycles. In other words, every cycle in an implosion problem must include at least one part that can be built from the next part in the cycle only by using an unexplodeable BOP entry.

There is a disadvantage to setting expAllowed to FALSE for a BOP entry. The following three WIT algorithms are "explosion based":

- Heuristic implosion
- MRP
- PIP

These algorithms will never deliberately build a part using an unexplodeable BOP entry. Thus in these algorithms, the only production through an unexplodeable BOP entry is that due to the execution of the operation for some other (explodeable) BOP entry. Because of this disadvantage, it is advisable to leave expAllowed defaulted to TRUE for most BOP entries in an implosion problem, setting it to FALSE only as needed in order to break up explodeable cycles.

9 The WIT Application Program Interface

In order to use WIT to solve actual problems, one builds an application program. Such a program, written in either C++, C, or Java, accesses WIT's modeling and solving capabilities by accessing one of WIT's application program interfaces (APIs). WIT has two APIs: One accessible from C or C++ and one accessible from Java. The C/C++ API (also called "The WIT API") will be described here; the Java API will be described in a later section.

WIT's C/C++ API is organized as a function library: a collection of C functions that are available to be called from an application program. All of the WIT API functions are declared in a single header file: `wit.h`. In addition to the API functions themselves, `wit.h` contains declarations of a number of special data types to be used in conjunction with these functions. The most important of these data types is a struct called "`WitRun`". An instance of struct `WitRun` contains all of the data used by WIT that's associated with a particular implosion problem: the problem itself, any parameters that control how WIT is to solve the problem, and ultimately, the implosion solution. Almost all of WIT's API functions have a pointer to a `WitRun` as their first argument. Some of the more sophisticated WIT application programs actually create two or more `WitRun`s and use them in different ways.

The following are a few examples of WIT API functions:

```
witReturnCode witNewRun (
   WitRun * * theWitRunPtr);
```

When this function returns, `(* theWitRunPtr)` will point to a newly constructed instance of struct `WitRun`.

```
witReturnCode witDeleteRun (
   WitRun * theWitRun);
```

Deletes `theWitRun`.

```
witReturnCode witAddOperation (
   WitRun * const     theWitRun,
   const char * const operationName);
```

Causes `theWitRun` to create and store a new operation whose name is given by `operationName`.

```
witReturnCode witSetPartSupplyVol (
   WitRun * const      theWitRun,
   const char * const  partName,
   const float * const supplyVol);
```

Looks up the part in `theWitRun` whose name matches `partName` and sets the value of its supplyVol attribute to the vector stored in `supplyVol`.

```
   witReturnCode witGetOperationExecVol (
      WitRun * const     theWitRun,
      const char * const operationName,
      float * *          execVol);
```

When this function returns, `(* execVol)` will point to a newly allocated vector storing a copy of the execVol of the operation in `theWitRun` whose name matches `operationName`.

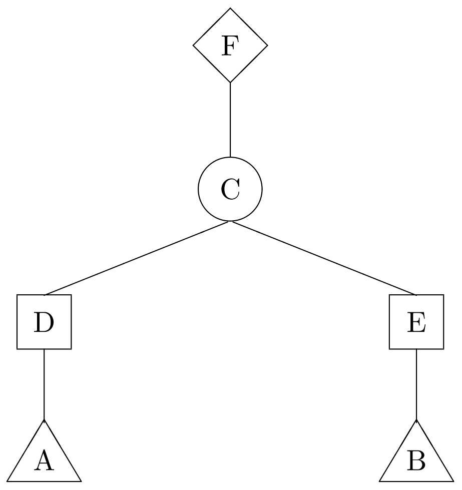
```
   witReturnCode witHeurImplode (
      WitRun * const theWitRun);
```

Invokes heuristic implosion on `theWitRun`.

10 Example of a WIT Application Program

For an example of a very simple WIT application program, consider an implosion problem illustrated in the following WIT diagram:

Figure 2: Implosion Problem for the Example WIT Application Program



This problem also includes the following attribute data:

```
   nPeriods          = 2
   multiRoute        = TRUE
   multiExec         = TRUE
   stockReallocation = TRUE
   A.supplyVol       = (10, 10)
   B.supplyVol       = (10, 10)
   F.demandVol       = ( 0, 35)
   C.buildNstn       = TRUE
```

The following C++ WIT application program constructs this implosion problem, invokes the core heuristic on it, and prints out the execution volumes.

```
//------------------------------------------------------------------------------
// Example WIT Application Program
//------------------------------------------------------------------------------

#include <wit.h>

#include <iostream>

int main ()
   {
   WitRun * theWitRun;
   float    supplyVolA[] = {10.0, 10.0};
   float    supplyVolB[] = {10.0, 10.0};
   float    demandVolF[] = { 0.0, 35.0};
   float *  execVolD;
   float *  execVolE;

   //---------------------------------------------------------------------------
   // Initial set up.
   //---------------------------------------------------------------------------

   witNewRun             (& theWitRun);

   witSetMesgFileAccessMode (theWitRun, WitTRUE, "w");
   witSetMesgFileName       (theWitRun, WitTRUE, "ex1.log");

   witInitialize            (theWitRun);
```

```
//----------------------------------------------------------------------------
// Set global attributes.
//----------------------------------------------------------------------------

witSetNPeriods           (theWitRun, 2);
witSetMultiRoute         (theWitRun, WitTRUE);
witSetMultiExec          (theWitRun, WitTRUE);
witSetStockReallocation  (theWitRun, WitTRUE);

//----------------------------------------------------------------------------
// Create data objects.
//----------------------------------------------------------------------------

witAddPart               (theWitRun, "A", WitCAPACITY);
witAddPart               (theWitRun, "B", WitCAPACITY);
witAddPart               (theWitRun, "C", WitMATERIAL);

witAddOperation          (theWitRun, "D");
witAddOperation          (theWitRun, "E");

witAddBomEntry           (theWitRun, "D", "A");
witAddBomEntry           (theWitRun, "E", "B");

witAddBopEntry           (theWitRun, "D", "C");
witAddBopEntry           (theWitRun, "E", "C");

witAddDemand             (theWitRun, "C", "F");

//----------------------------------------------------------------------------
// Set attributes associated with specific data objects.
//----------------------------------------------------------------------------

witSetPartSupplyVol      (theWitRun, "A", supplyVolA);
witSetPartSupplyVol      (theWitRun, "B", supplyVolB);

witSetDemandDemandVol    (theWitRun, "C", "F", demandVolF);

witSetPartBuildNstn      (theWitRun, "C", WitTRUE);

//----------------------------------------------------------------------------
// Invoke heuristic implosion.
//----------------------------------------------------------------------------

witHeurImplode           (theWitRun);
```

```
   //------------------------------------------------------------------------
   // Retrieve and print the execution volumes.
   //------------------------------------------------------------------------

   witGetOperationExecVol   (theWitRun, "D", & execVolD);
   witGetOperationExecVol   (theWitRun, "E", & execVolE);

   std::cout << "D.execVol[0] = " << execVolD[0] << std::endl;
   std::cout << "D.execVol[1] = " << execVolD[1] << std::endl;
   std::cout << "E.execVol[0] = " << execVolE[0] << std::endl;
   std::cout << "E.execVol[1] = " << execVolE[1] << std::endl;

   free (execVolD);
   free (execVolE);

   //------------------------------------------------------------------------
   // Final shut down.
   //------------------------------------------------------------------------

   witDeleteRun             (theWitRun);
   }
```

This program prints the following output to stdout:

```
D.execVol[0] = 10
D.execVol[1] = 10
E.execVol[0] = 5
E.execVol[1] = 10
```

11 WIT-J

WIT's Java API is called "WIT-J": "Watson Implosion Technology - Java Interface". It is implemented as a wrapper for the C/C++ API using the Java Native Interface.

WIT-J is organized as a class library. Its main classes represent the various kinds of objects that make up an implosion problem; classes such as: `Part`, `Operation`, `BomEntry`, etc. There is also a class `Problem`, that represents the implosion problem as a whole. (The Java class `Problem` corresponds to the C struct `WitRun`.)

Another class, `Attribute`, represents the data attributes of an implosion problem, such as nPeriods or execVol. Each instance of class `Attribute` is automatically created by WIT-J and stored as a static final field of class `Attribute`, where the name of the field matches the name of the represented attribute translated to upper case. For example, class `Attribute` has a static field called `SUPPLY_VOL` whose value is an `Attribute` that represents the supplyVol attribute of WIT.

Consider the following snippet of WIT-J application code:

```
import        com.ibm.witj.*;
import static com.ibm.witj.Attribute.*;

...

Problem   theProblem;
Operation theOperationA;

theProblem = Problem.newInstance ();

theProblem.set (N_PERIODS, 3);

theOperationA = Operation.newInstance (theProblem, "A");

theOperationA.set (EXEC_COST, new double[] {10., 9., 8.,});
```

This code:

- Imports the WIT-J package and statically imports the `Attribute` class.
- Creates a WIT-J `Problem`.
- Sets the number of periods in the implosion problem to 3.
- Creates a WIT-J `Operation` whose operationName is “A”.
- Sets the execution cost for the newly created WIT operation to the vector (10, 9, 8).

Full documentation on WIT-J is given in “Watson Implosion Technology - Java Interface, Application Developer’s Guide” (Wittrock, 2012).

12 WitViz

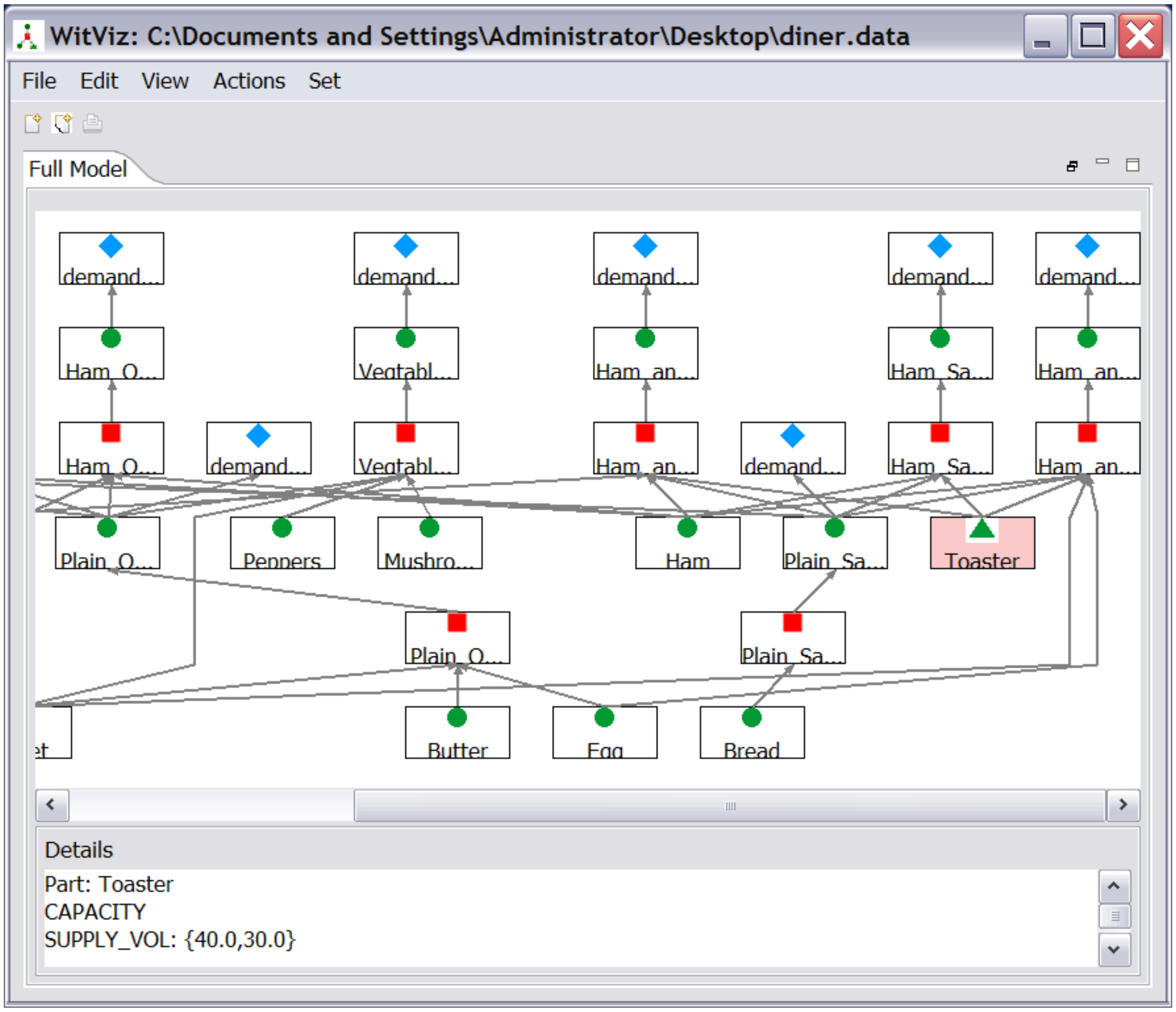
To aid in the development of WIT application programs, a visualization tool for WIT models has been developed: “WitViz”.

The input to WitViz is a ”WIT data file”, a file that specifies all of the data that defines an implosion problem. A WIT application program can generate a WIT data file for the current implosion problem at any time, simply by calling the WIT API function `witWriteData`, or the WIT-J method `writeData` of class `Problem`.

Given a WIT data file, WitViz displays a WIT diagram of either the whole implosion problem or a portion of it. Since real implosion problems are generally much too large to visualize all at once, WitViz provides ways to navigate through the implosion problem, either by scrolling, or by showing the aspects of the problem that are “near” the currently selected data object. It also displays attribute values of the selected data object.

Figure 3 shows a screenshot of WitViz. (The implosion problem shown is a small demo problem that models the filling of orders at a diner.)

Figure 3: WitViz Screenshot



Part: Toaster
CAPACITY
SUPPLY_VOL: {40.0,30.0}

WitViz can be a very helpful tool for debugging a WIT application program: It gives the application developer a view of the structure and details of the implosion problem that was built-up by the application code. In many cases, this can make deviations from the intended structure visually apparent.

WitViz was developed by Donna Gresh of IBM Research (see Gresh 2005). It invokes WIT-J to interpret the WIT data file and uses the Draw2d and GEF components of Eclipse to do the drawing.

13 Current Applications of WIT

WIT has been incorporated into numerous application programs which have used its capabilities to solve various resource-constrained planning problems. WIT's current applications (as of this writing) include the following:

SCE and The Integrated Demand/Supply Process

The Supply Capability Engine (SCE) is a major WIT application program developed by IBM Research and tailored to the specific needs of IBM's hardware supply chain. Like WIT itself, SCE is used in a number of distinct applications. One of its main applications is in the context of the IBM Server Group's "Integrated Demand/Supply" (IDS) process. IBM's Server Group is responsible for most of the company's computer hardware business and IDS is the elaborate business process by which the Server Group performs its supply chain planning. SCE functions as a key technical component of this process, providing the Server Group's managers and analysts with a rational basis to plan production within the company's manufacturing facilities ("sites"), to co-ordinate production between sites, and to co-ordinate with major suppliers. SCE's IDS application models this supply chain as a highly complex implosion problem which it then solves using heuristic implosion with many of the optional capabilities listed in this paper: build-ahead, multiple routes, stock reallocation, lot sizes, equitable allocation, heuristic allocation, proportionate routing, WIT-MRP, and post-implosion pegging.

SCE and the Available-To-Sell Process

Another application of SCE is in the context of the IBM Server Group's "Available-To-Sell" (ATS) business process. The ATS process seeks to identify economically favorable opportunities to reduce excess inventories by combining them with the purchase of additional supply (if necessary) and building products for which there is no predetermined demand but which could potentially be sold. The ATS business process is based on the ATS application of SCE, which formulates an implosion model of the ATS problem, solves it using optimizing implosion, pegs the solution with post-implosion pegging, and provides the pegged solution to the ATS analysts.

GIView Planner

IBM Global Integrated View Planner is a comprehensive supply chain management tool built and deployed by IBM GBS in Japan. It consists of a number of customizable modules for solving a variety of supply-chain management problems. Four of its modules are WIT applications:

- Production Planning
- Order Promising
- Product Allocation
- Distribution Planning

The GIView team has built and delivered customized WIT-based GIView Planner applications for more than 20 Japanese companies in such industries as semiconductors, consumer electronics, cutting tools, and others. GIView Planner uses heuristic implosion with many of its extensions and post-implosion pegging.

Gap/Glut Analyzer

The Gap/Glut Analyzer is a WIT application developed by IBM Research and used by IBM GBS Capacity Planning. It identifies shortages and excesses in the GBS workforce, taking into account the pipeline of service engagements to be fulfilled, while fully exploiting the flexibilities GBS's highly skilled workforce: Each worker tends to have multiple diverse skills and also can perform tasks that are below his/her nominal skill level. The problem is solved as an optimizing implosion problem.

As can be seen from these cases, WIT has been applied productively to a variety of resource-constrained planning problems, ranging from production planning (SCE/IDS, GIView production Planning) to more wide-ranging aspects of supply-chain planning (SCE/ATS, other GIView modules) to a workforce application (Gap/Glut Analyzer).

Appendix: Why Is It Called "Implosion"?

Historically, when the model that eventually became known as the implosion model was first being formulated, one of its first users observed that it seemed to have a reverse or complementary relationship to an MRP explosion: where MRP held the meeting of demands as fixed and allowed the availability of supplies to vary, the new approach held the availability of supplies as fixed and allowed the meeting of demands to vary. It was suggested that the new approach might be called "implosion", to reflect this reverse relationship.

References

Dietrich, B., D. Connors, T. Ervolina, J. P. Fasano, R. Lougee-Heimer, and R. Wittrock. 2005. Applications of Implosion in Manufacturing, in *Supply Chain Management on Demand,* An and Fromm, Eds., Springer-Verlag.

Dietrich, B. and L. Escudero. 1990. A Single-Level Implosion Tool with Replacement Parts and Effectivity Dates. *Proceedings of the 1990 IBM Manufacturing Productivity Symposium.* Thornwood, New York.

Dietrich, B., R. Wittrock. 1996. Allocation Method for Generating a Production Schedule U.S. Patent #5,548,518.

Dietrich, B., R. Wittrock. 1997. Optimization of Manufacturing Resource Planning U.S. Patent #5,630,070.

Gresh, D. 2005. Visualizing Supply Chains: A Design Study. IBM Research Report RC23584.

`http://domino.research.ibm.com/library/cyberdig.nsf/papers?SearchView&Query=RC23584`

IBM. 2012. Watson Implosion Technology, User's Guide and Reference. Release 8.0. (Available from this author on request)

Vollmann, T. E., W. L. Berry, and D. C. Whybark. 1992. *Manufacturing Planning and Control Systems,* 3rd Ed., Irwin, Homewood, IL.

Wittrock, R. 2012. Watson Implosion Technology - Java Interface, Application Developer's Guide (Available from this author on request)