Implementing Reusable Solvers: An Object-Oriented Framework for Operations Research Algorithms

by

John Douglas Ruark

Bachelor of Arts, Harvard University (1993)

Submitted to the

Department of Electrical Engineering and Computer Science

in partial fulfillment of the requirements for the degree of

Doctor of Philosophy

in Operations Research

at the

Massachusetts Institute of Technology

June 1998

Copyright © Massachusetts Institute of Technology, 1998. All rights reserved.

Signature of Author ____

Department of Electrical Engineering and Computer Science

May 20, 1998

Certified by ____

Stephen C. Graves

Abraham J. Siegel Professor of Management

Co-director, Leaders for Manufacturing Program

Thesis Supervisor

Accepted by ____

Robert M. Freund

Seley Professor of Operations Research

Co-director, Operations Research Center

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

1998

LIBRARIES

Eng.

This page intentionally left blank.

Implementing Reusable Solvers: An Object-Oriented Framework for Operations Research Algorithms

by

John Douglas Ruark

Submitted to the Department of Electrical Engineering and Computer Science
on May 20, 1998, in partial fulfillment of the requirements for the degree of
Doctor of Philosophy in Operations Research

Abstract

The expression of algorithms as mathematical constructs and the implementation of algorithms as software are disjoint activities in operations research. Researchers develop non-robust prototypical implementations of cutting-edge algorithms, while programmers develop commercial-grade implementations of mainstream, aging algorithms. There has been no middle ground, due to a lack of suitable software tools, reusable components, and appropriate standards.

This thesis proposes such a middle ground, an object-oriented framework for implementing and using reusable solvers, and demonstrates through several solvers and applications of those solvers that the framework can reduce client-side development efforts and increase reusability.

Prior research on implementation has not focused on reusability as an important factor. Greater reusability improves subsequent development times, increases productivity, and enables many to create solutions that would not otherwise be possible. Focusing on reusability as a primary goal, this thesis presents the requirements for a framework for building reusable solvers (algorithm implementations) and applications in the context of solving real-world, small- to medium-sized problems faced by applied operations researchers.

Out of these requirements, the thesis develops an object-oriented framework for implementing solvers and the associated components that solvers use. The framework defines a set of interfaces, objects, and associated machinery that characterize the nature of data, solvers, and client interactions in solving applied operations research problems. The framework includes protocols for data flow, solver interactions, introspection, progress notifications, life cycle control, and building networks of solvers.

Several solvers and applications are examined in the context of the framework, to determine the cost and benefit of using the framework in developing solutions. These examples include wrapping the CPLEX Callable Library, implementing network optimization algorithms, and solving several real-world problems.

The framework benefits three essential target audiences: analysts who build solutions, researchers and developers who build solvers, and researchers and developers who build modeling environments. The framework reduces development effort for end users, increases and encourages reuse of solvers, reduces dependencies on existing modeling environments and solvers, and simplifies the creation of integrated modeling environments.

Thesis Supervisor: Stephen C. Graves
Title: Co-director, Leaders for Manufacturing Program

This page left intentionally blank.

Acknowledgments

When I embarked on this journey five years ago, I never planned on needing a fifth year. But I also didn't expect my thesis to change in the spring of my fourth year to something completely different. One can climb to amazing heights if only he digs a deep enough hole from which to begin. I did a lot of digging in my fourth year. And then I struck what could someday be gold.

Throughout the hectic life, the long hours, the anxiety, the angst, the uncertainty, the hopelessness, the futility, to the final redemption, relief, and joy, there were constants. My parents, Bill and Cynthia Ruark, provide eternal support, love, understanding, phone calls, and computer questions. The monthly checks, however, have reached their conclusion. My brother, Marcus Ruark, keeps me in line with where it's at, what's hip, and who's hot. If only I could keep up. Penny—Penster, Sporkinator, the furbeast—has been a constant source of amusement, purring, fur, and bushy tails. And, of course, Liz Stein, my companion, my best friend, and my love, has seen me through these five years and will for many more.

I received help from many people at MIT. Most important is my advisor, Professor Steve Graves, who provided support and wisdom, as well as understanding and leniency when I began this project. Hopefully, the investment has paid off. The ORC staff, particularly Laura Rose, Paulette Mosley, and Tom Magnanti, also with me these five years, have been utterly reliable, and without them, of course, no one graduates from MIT with an OR degree.

Many friends have come and gone, relationships influenced by distance and time. At MIT, my fellow classmates Arni, Beril, Brian, David, Edi, Hari, Jim, Keely, Martin, Mina, Rafael, Rebecca, Sean, Thalia, Tim, and Yi—they took the journey with me, learned the things I learned, and we knew each other better for it. Professors Bertsimas, Dellarocas, Freund, Magnanti, Orlin, and Rosenfield are friends and teachers. My other friends do not claim to understand operations research, but secretly I think they know. They include Aaron, Amanda, Amy, Brett, Deane, Jason, Jen, Max, Melissa, Victor, and Zack. Thanks also to the Stein family for many weekends of relaxation, excitement, and travel—capped by the two-minute total solar eclipse in Antigua in February, 1998; what a great way to end.

This page left blank intentionally.

Table of Contents

Acknowledgments....5
Table of Contents....7
Table of Figures....11
Table of Tables....13
About the Author....14

1 Introduction....17

1.1 The Target Audience....19
1.2 Outline of Introduction....19
1.3 The Evolution of Compound Documents....20
1.3.1 Method one—Pasting a picture....20
1.3.2 Method two—Out-of-place editing....21
1.3.3 Method three—In-place activation....22
1.3.4 The future of compound documents....23
1.3.5 Summary....24
1.4 The Evolution of Optimization Applications....24
1.4.1 Method one—Custom solution....25
1.4.2 Method two—Modeling language....26
1.4.3 Method three—Component-based modeling environments....27
1.4.4 The future of modeling environments....30
1.5 The Evolution of Application Implementation....31
1.5.1 The modeling environment literature....33
1.5.2 Future modeling application implementations....34
1.5.3 COM, CORBA, object models, and domain modeling....36
1.6 Overview of the Thesis....37
1.6.1 Requirements of a reusable solver....38
1.6.2 Features and goals of the framework....39
1.6.3 Proof of concept....40
1.7 Overview of Related Modeling Literature....41
1.7.1 Structured modeling and beyond....41
1.7.2 Model selection....43
1.7.3 Model integration....45
1.8 Epilogue....49

2 Requirements 51

2.1 Domain Analysis: The Nature of OR Solutions..52
2.1.1 The components of a solution archetype 53
2.1.2 Categorization of sample solutions..67
2.1.3 Discussion 69

2.2 Participants in the Solution Process 70

2.3 Requirements of a Solver Implementation...72
2.3.1 Executable...73
2.3.2 Invoking from different applications and environments 75
2.3.3 Documentation and introspection...76
2.3.4 Progress updates...81
2.3.5 Life cycle control..84
2.3.6 Dimension and type support...85
2.3.7 Testing and validation...88
2.3.8 Computer-based training...89

2.4 Requirements of Networks of Solvers 90
2.4.1 Integrity of data 91
2.4.2 Distribution and synchronization 96
2.4.3 Notifications..98
2.4.4 Global/local control 99

2.5 Requirements of the Core Services..101
2.5.1 Registry of available solvers...103
2.5.2 Dimension and type support...104

2.6 Conclusion...104

3 Framework...107

3.1 Organization of the Framework...107
3.1.1 General services and specifications 108
3.1.2 Solver services and specifications 109
3.1.3 Interconnection services and specifications 110
3.1.4 Intersecting services and specifications...111
3.1.5 Framework entities...111
3.1.6 Framework interfaces 112
3.1.7 Framework reference..115

3.2 The Solver Executable...115

3.3 Solver Interfaces 116

3.4 Data Flow...118
3.4.1 Data structure in the framework..120
3.4.2 Data elements 120
3.4.3 Data sources..132

3.5 The Primary Solver Interfaces....137
3.5.1 Solver structure....137
3.5.2 IRSolver....139
3.5.3 IRSolverInputs: Setting the inputs....140
3.5.4 IRSolverOutputs: Getting the outputs....140
3.5.5 IRSolverParameters: Parameterization....141
3.6 Introspection....141
3.6.1 SolverInfo: Paralleling the structure of the solver....142
3.6.2 SolverDescription: The capabilities of the solver....162
3.6.3 Discovering capabilities through QueryInterface....168
3.7 Progress Updates and Life Cycle Control....169
3.7.1 Progress notifications (push)....169
3.7.2 Progress queries (pull)....175
3.7.3 Solver life cycle control....177
3.8 Networking Solvers....178
3.8.1 Networks in traditional clients and global control....178
3.8.2 Moving to local control....180
3.8.3 Solver sites and mappings....184
3.8.4 Inbound solver sites....189
3.8.5 Outbound solver sites....193
3.8.6 Mappings....199
3.8.7 Putting it together....203
3.8.8 How this fulfills the networking requirements....211
3.9 Core Services....214
3.9.1 Solver database....215
3.9.2 Miscellany....216
3.10 Conclusion....217
4 Solvers and Applications....219
4.1 Solvers....220
4.1.1 Packaging a solver....220
4.1.2 RandVar module....223
4.1.3 RNetOpt module....224
4.1.4 RLPWrapper solver....226
4.1.5 Using a solver....228
4.2 Applications....229
4.2.1 Monsanto....229
4.2.2 FlexCap....239
4.2.3 M/M/k queueing model in Excel....244
4.2.4 SIPModel....250
4.2.5 ALCOA....253

4.3 Conclusion 258

5 Conclusion 261

5.1 Benefits of the Framework 261
5.1.1 For clients, analysts, and application developers 262
5.1.2 For solver developers 266
5.1.3 For modeling environment developers 270

5.2 Issues 272

5.3 For Future Research 281

5.4 Conclusion 284

Coda 286

A Sample Code and Extensions 289

A.1 Developing the RLPWrapper Solver 289
A.1.1 Generating the SolverInfo 289
A.1.2 The solver skeleton 291
A.1.3 Implementing solver inputs 295
A.1.4 Solving the linear program 298
A.1.5 Implementing the output methods 301
A.1.6 Implementing the parameters 302

A.2 Creating a Solver That Wraps Model Knowledge 308

A.3 Custom Extensions to the Framework: A Random Variable Specification 311
A.3.1 Specifying a data type 311
A.3.2 Using a custom data type in the framework 313
A.3.3 A random variable specification 315
A.3.4 A queueing system specification 317

A.4 M/M/k Queue Examples 319
A.4.1 VBA macros 319
A.4.2 Framework solver 323

Bibliography 325

Bibliography of Applied Solutions 333

Index of Authors 341

Index 343

Table of Figures

Figure 1.1: Pasting a picture ... 21
Figure 1.2: Out-of-place editing ... 22
Figure 1.3: In-place activation ... 23
Figure 1.4: Custom solution ... 26
Figure 1.5: Modeling language ... 26
Figure 1.6: Component-based modeling environments ... 28
Figure 1.7: Monolithic application architecture ... 31
Figure 1.8: Component-enabled application architecture ... 32
Figure 1.9: Future application architecture with modeling component services ... 34
Figure 1.10: Future component-based operating system architecture ... 35
Figure 2.1: Three subproblems hooked together to make a larger subproblem ... 54
Figure 2.2: Single-stage architecture ... 54
Figure 2.3: Directed acyclic graph architecture ... 55
Figure 2.4: Architecture of P&G supply chain model solution ... 57
Figure 2.5: Decision-based directed graph architecture ... 58
Figure 2.6: Architecture of SANTOS's SIPS planning solution ... 59
Figure 2.7: Real time directed graph architecture ... 60
Figure 2.8: Hierarchy of execution periodicity types ... 65
Figure 2.9: Sample two-stage solution demonstrating participants ... 71
Figure 2.10: A progress bar window ... 83
Figure 2.11: Progress notifications through the status bar ... 83
Figure 2.12: Traditional solver structure with decentralized dimension support ... 87
Figure 2.13: Solver structure with centralized dimension support ... 88
Figure 2.14: High-level solver state diagram ... 91
Figure 2.15: Event trace diagram for normal solver invocation ... 93
Figure 2.16: Event trace diagram of attempting to retrieve invalid outputs ... 94
Figure 2.17: Event trace diagram of attempting to retrieve invalid inputs ... 94
Figure 2.18: Event trace diagram of attempting to retrieve outputs too late ... 95
Figure 2.19: Example of global control in a 2-solver network ... 100
Figure 2.20: Example of local control in a 2-solver network ... 101
Figure 2.21: A common dialog ... 101
Figure 3.1: Overview of services and specifications of the framework ... 108
Figure 3.2: How the VB run-time wraps controls to expose complicated interfaces ... 117
Figure 3.3: Relationships of data elements, dimensions, and sets ... 121
Figure 3.4: State diagram for a data source ... 133
Figure 3.5: High-level class diagram of solver structure ... 138
Figure 3.6: Object diagram of knapsack solver structure ... 139

Figure 3.7: Class diagram of SolverInfo interfaces 142
Figure 3.8: Object diagram of knapsack SolverInfo structure 143
Figure 3.9: Sample solution network for framework development 178
Figure 3.10: Perspective of Solver 3 under client global control 181
Figure 3.11: Perspective of Solver 3 under local control 182
Figure 3.12: Sample solution network with wrapper components 182
Figure 3.13: Perspective of a Solver 3 wrapper under local control 183
Figure 3.14: Perspective from Solver 3 under local control with wrapper 184
Figure 3.15: Sample solution network with solver site components 186
Figure 3.16: Sample solution network with mappings and internal solver connections 187
Figure 3.17: Sample solution network with mappings, solver sites, and the client 188
Figure 3.18: Interaction diagram of outbound solver site locking mechanism 194
Figure 3.19: Iterating a solution network from the client 196
Figure 3.20: Adding intelligence to the outbound solver site to iterate within the network 197
Figure 3.21: State diagram for outbound solver site 198
Figure 3.22: DetermineWhoMaintainsData: Decision tree 202
Figure 3.23: Wiring Solver 3's solver sites 204
Figure 3.24: Interaction diagram showing wiring of Solver 3's site 205
Figure 3.25: Interaction diagram when the data source maintains input data 208
Figure 3.26: Interaction diagram when mapping maintains input data 209
Figure 3.27: Interaction diagram when solver maintains input data 210
Figure 4.1: Package diagram for original Monsanto solution 232
Figure 4.2: Package diagram for Monsanto solution with a special model solver 233
Figure 4.3: Package diagram of Monsanto solution with entire component-based solution 237
Figure 4.4: Screenshot of FlexCap 240
Figure 4.5: Screenshot of spreadsheet modeling of M/M/k queue 245
Figure 4.6: Screenshot of goal seeking in spreadsheet modeling 246
Figure 4.7: Screenshot of SIPModel 2.1 user interface 250
Figure 4.8: Sample property dialog from SIPModel 2.1 251
Figure 4.9: Screenshot of initial ALCOA client application 255
Figure 5.1: Screenshot of modeling environment solution network 282

TABLE OF TABLES

Table 1.1: Matrix of component modeling benefits 29
Table 2.1: Categorization of Edelman paper solutions 68
Table 2.2: Example knapsack problem with dimension information 86
Table 3.1: Summary of framework entities 112
Table 3.2: Data element entities and interfaces 113
Table 3.3: Solver interaction entities and interfaces 113
Table 3.4: Progress updates and life cycle control entities and interfaces 114
Table 3.5: Introspection entities and interfaces 114
Table 3.6: Solver registration and selection entities and interfaces 114
Table 3.7: Networking and interconnections entities and interfaces 115
Table 3.8: Bitmask values for IRSolverBaseInfo::GetFlags 144
Table 3.9: Bitmask values for IRSolverInfo::GetFlags 145
Table 3.10: Bitmask values for IRSolverInputInfo::GetAssignmentFlags 147
Table 3.11: Bitmask values for IRSolverInputInfo::GetChangeFlags 147
Table 3.12: Bitmask values for IRSolverOutputInfo::GetChangeFlags 148
Table 3.13: Custom attributes for SolverInfo Type Library 159
Table 3.14: Bitmask values for IRSolverDescription::GetSolutionFlags 163
Table 3.15: Pre-defined RDESCPROPIDs 164
Table 3.16: Parameters of IRSolverAdvise::OnSolveNotify 173
Table 3.17: Bitmask values for IRSolverAdvise::OnSolveNotify 173
Table 3.18: Action codes for IRSolverAdvise::OnSolveNotify 174
Table 3.19: Notification codes for IRSolverAdvise::OnSolveComplete 174
Table 3.20: Status codes for IRSolverStatus::GetStatus 176
Table 3.21: Creation flags for IRSolverSiteInSolverFactory::SpecifySolver 192
Table 3.22: Destruction flags for IRSolverSiteInSolverFactory::SpecifySolver 192
Table 3.23: Characterization of who maintains data in mappings 201
Table 3.24: Possible values for the MaintainPreference mapping attribute 201
Table 4.1: Network flow algorithm implementation sizes—original solvers 225
Table 4.2: Network flow algorithm implementation sizes—framework solvers 225
Table 4.3: Inputs to the CPLEX wrapper solver 227
Table 4.4: Outputs from the CPLEX wrapper solver 227
Table 4.5: Summary of Monsanto implementation techniques' code sizes 238
Table 4.6: Comparison of FlexCap implementations 243
Table 4.7: Summary of M/M/k queue implementation characteristics 249
Table 4.8: Summary of solvers and applications from Chapter 4 259

About the Author

John Ruark received a Bachelor of Arts in Mathematics at Harvard University in 1993 after attending high school in O'Fallon, Illinois. During the summers while at MIT, he worked with General Motors; Scudder, Stevens, and Clark; and ALCOA. Prior to his studies at MIT, he was assistant editor of the best-selling *Let's Go: USA & Canada, 1994,* travel guide. At Harvard, he was a lighting, set, and sound designer of numerous theatrical productions, and a member of the Harvard University marching and jazz bands and the Harvard-Radcliffe Orchestra. He lived in Dunster House.

John can be reached at ruark@post.harvard.edu.

On the eve of the fourth day, 220 runs to go with eight in hand, we sent in the night watchman. He fulfilled his duty, lasting the night, but fell slogging, early on the fifth day. They brought on lethal spin and buried the middle order in the first session, sending us reeling to 162 for seven, needing 140 to win. But we still had our top batsman at the crease, and he weathered the storm and hit a ton, digging out the boundaries even as the pitch deteriorated, securing a two-wicket victory in the final over off a shattering drive through the covers.

To my parents

illegitimum non carborundum

Chapter One

Introduction

In operations research there is a disparity between the creation, description, and validation of algorithms and the practical, useful implementation and distribution of those algorithms. As top researchers push the envelope of algorithm design and faster asymptotic run times, truly useful, widespread implementations of algorithms stagnate, advancing haltingly and mysteriously. Researchers develop implementations that are *good-enough*—good enough to prove viability, good enough to demonstrate, and good enough to benchmark. Only occasionally are these prototypical implementations good enough to distribute or sell. Rarely are they easy enough to incorporate into a larger project implementation.

Why is this the case? Primarily, what is good enough for the researcher is not at all sufficient for the software developer. In addition to viability and correctness, the developer needs ease of integration, simplicity of interface, and separation of orthogonal functionality. Too often, a prototypical implementation fails all three: it is a stand-alone application (hence difficult to integrate) that requires arcane or proprietary input and output formats (hence complicated interfaces) and that mixes file I/O operations with algorithm logic in the same processes (hence mixing orthogonal functionality).

In many cases, these implementations are also too difficult and complicated to be used by knowledgeable operations research modelers working on decision support projects. If a prototypical implementation requires too much time and effort to use, it will not be. Instead, the modeler will rely on more generic, and possibly less efficient or optimal, techniques, such as monolithic mathematical programming solvers or spreadsheets. As computers permeate all modeling applications, the needs of the applied operations researcher are converging to those of the software developer: ease of integration, simplicity of interface, separation of orthogonal functionality. An algorithm implementation that exhibits these requirements could be easily understood and quickly incorporated into an application project.

Unfortunately, the implementation of complicated algorithms in such a manner lies in a bleak no-man's-land. Researchers have neither the motivation nor desire to improve their prototypical implementations because they typically do not have the software engineering expertise or easy access to it, the rewards are unspecified at best, and it is not their area of interest. Normal software developers do not approach such projects because they lack the expertise to understand the algorithms and untangle the relevant strands from a messy prototype, and again, the rewards are unclear.

The rewards for the operations research community are perfectly clear. In a world where algorithms are easy to implement, quick to distribute, and simple to integrate, consultants, modelers, and students all stand to gain immensely. Shortened development times, freedom to choose algorithms, independence from a single vendor or modeling tool, lower costs, and reusable codes and models are just some of the benefits. The operations research community would gain broader exposure to the consulting and modeling community as the provider of a multitude of useful, specialized, optimized algorithms.

The goal, then, is to create rewards for the brave souls who tread into the no-man's-land. These rewards will be monetary and laudatory. Monetary rewards derive from generating sufficient revenue from the sales of an algorithm implementation to offset its costs of production. Simple economics, yes, but a powerful enough barrier to entry to keep developers and researchers from entering the market *en masse.* Mostly, researchers provide their prototypical implementations free for academic use, and often their research is funded by outside sources. There is no revenue stream to speak of. Usually, the users get what they pay for. The problem is generating sufficient revenues for a product. Laudatory rewards, the satisfaction and widespread recognition of a job-well-done, are less tangible but no less important. A large (and happy) customer base fuels these rewards. The problem is generating that customer base.

Both of these problems could be tackled with the right *killer application*[1] for the modeling community. Such an application would make it simple to incorporate well-implemented, reusable algorithms and generate larger decision support applications. Most likely, this killer app would be a graphical decision support system (DSS). There has been extensive research on the presentation, architecture, and knowledge of a DSS, but very little research on tying a DSS to disparate back-end algorithms. Typically, a DSS relies on a monolithic mathematical programming solver, and the task becomes representing any model in a common language that can map to the generic solver. This is not conducive to the successful production of specialized algorithms.

This thesis outlines a potential standard set of interfaces and protocols that would enable any appropriately designed DSS to communicate with any appropriately well-implemented,

---

1 A killer application is a "use of technology so attractive to consumers that it fuels market forces" that "change technological advances from curiosities into moneymaking essentials" (Gates [31] p. 74). Two examples are Lotus 1-2-3 for the IBM PC and the NCSA Mosaic browser for the World-Wide Web.

reusable algorithm. Such a standard could open the floodgates for algorithm implementations that do not depend on monolithic optimizing engines.

The key enabling technology is component software. Components are computer programs that encapsulate both data and operations in a single object. Well-designed components typically are self-sustaining, have few hidden dependencies on other components, and accomplish only as much as they need to and nothing more. Complicated tasks and problems are usually broken into multiple components, thereby keeping the duty of each component limited, manageable, and coherent.

1.1 The Target Audience

This thesis presents an approach to developing software for algorithm implementations. So, the primary audience of the thesis as a document is software developers. Three types of developers are of interest: *systems tools developers,* who create modeling environments and operating systems; *component tools developers,* who create the algorithms and related data structures; and *application developers,* who integrate components into systems to create customized solutions and applications. Operations researchers interested in the implementation of algorithms and how algorithms can be made more reusable should also find reading the thesis rewarding.

As applied technology, however, the results of the thesis have a different audience. This technology is designed to specifically address problems of a small or medium size; that is, problems of up to thousands or tens of thousands of elements. Anyone who works with these types of problems stands to benefit from the widespread adoption of policies and technologies that promote and enable reusable algorithms. This includes *students* and *teachers,* who typically deal with small problems for pedagogical purposes; *consultants,* who often mock-up medium problems in spreadsheets to solve strategic or tactical planning problems; and OR *practitioners* working on these types of problems. To the extent that developing algorithms (especially reusable ones) can be made easier, *algorithm designers* and developers can also benefit from the technology.

1.2 Outline of Introduction

Component software has been extremely successful in enabling compound documents, which are documents that contain multiple media types and objects, such as a word-processing document with charts, spreadsheets, and graphs embedded within it. Hence, an examination of the parallels between components in compound documents and components in modeling environments should prove enlightening. The next section summarizes the evolution of both compound documents and modeling tasks and explores how the lessons from one can translate to the other. The perspective here is the end-user's: how does the user accomplish common yet important tasks, and how do components make those tasks easier? Following that, an examination of the evolution of the implementations of

applications reveals how components are entering daily computer programming life, even when the end-user might not be aware of any components. The proposed framework is then inserted into the application architecture. The introduction concludes with an outline of the remainder of the thesis.

1.3 The Evolution of Compound Documents

Many users are familiar with the task of embedding a chart as an exhibit into a report. The report is created inside an appropriate word-processing application. How would a user embed a chart into the report within that application? Long ago, the user would create the chart separately from the report; the image of the chart would be literally cut-and-paste onto the printed page in the correct place or be appended to the report as an endnote. Of course, modern computing technology permits user to insert the chart prior to printing the document.

There are traditionally two eras of compound documents. The first, still prominent on some platforms and applications, is *application-centric*. In the application-centric era, the application is the fundamental unit of interaction. Documents exist within applications. To access a spreadsheet, the spreadsheet application must be located and started. This process—start the application, load the document—is the essence of application-centric computing. The second era, evolving on advanced, mainstream GUI operating systems like Windows 95 and the MacOS, is *document-centric*. In this era, the document is the fundamental unit of interaction. A spreadsheet exists as part of a document, and the spreadsheet application is started in order to load that document. To see this in action, simply right-click on a document in the Windows 95 shell to activate a context-menu that will include options such as "Open" and "Print."

There are three modern techniques for including a chart in a report. The first two are firmly application-centric. For each, the user creates the chart as an entirely separate document within the charting application, and then copies some representation of the chart, either as a picture or some native opaque data format into the report within the word-processing application. The third method is document-centric. The user can create the chart within the report, accessing charting functionality as a transparent extension to the word-processor without visibly running the chart application.

1.3.1 Method one—Pasting a picture

Integration based on a common denominator format.

To embed a chart in a report requires that the charting application and the word-processor speak the same language. In the earliest days of the MacOS and Windows, the lowest common denominator for applications was either a text format or a picture format on the clipboard. Transferring a chart from a charting application to a word-processor involved preparing the chart within the charting application, placing an image of the chart on the

clipboard (copying) and then switching to the word-processor application and placing that image into the report (pasting). In order to make changes to the chart within the report, the old image must be deleted and replaced by a newer one from the charting application (see Figure 1.1).

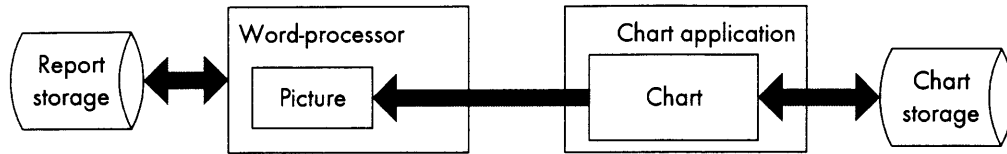


Figure 1.1: Pasting a picture

Clearly, there is significant overhead in putting the chart into the document. Activities such as layout and sizing must be done in the charting program, and it is impossible to access or modify the chart from the word-processing program. The document contains a static image of the chart but no data about the contents of the chart. The word-processing application does not even understand what a chart is; it understands only pictures. Scaling the chart within the word-processor causes the entire image to be scaled, without recalculating the layout of the chart; this can ruin font placement and sizing. This is a portable technique as long as a standard "picture" format is used (one called "metafile" is provided with Windows). The report will contain multiple files: one for the text itself, and one for each exhibit in the report. This will add some non-trivial administrative burden for maintaining and versioning the report.

1.3.2 Method two—Out-of-place editing

Integration based on proprietary suites and protocols.

The previous technique relies on a standard picture format, enabling the user to paste any picture from any application into any word-processor. Because of the absence of meaningful state data about the chart itself, this flexibility comes at the cost of not being able to modify the chart through the picture of the chart. More technically sophisticated word-processors, especially those that were pieces of productivity suites in the early- to mid-90s, defined their own proprietary chart formats that enabled those word-processors to specifically know that a particular object was a chart as opposed to just being a picture. Armed with this knowledge, these word-processors could activate the chart application when the user wanted to edit the chart (see Figure 1.2).

The improvements of this technique are better perceived integration between the word-processor and charting application, and a single file that contains both report and chart, improving maintainability and versioning. Better integration is perceived in several ways. For one, the action of resizing the image of the chart doesn't just scale the picture. Instead, it sends a message to the underlying chart object that causes it to resize itself; therefore, fonts

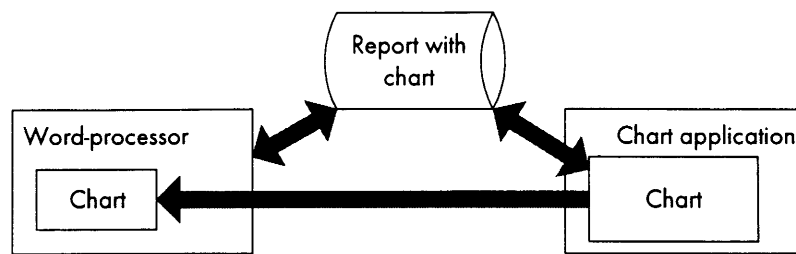


Figure 1.2: Out-of-place editing

can be repositioned instead of simply scaled. Secondly, the word-processor knows that the object is a chart, and thus can provide facilities for initiating the editing of the chart.

This is the "suite" way of preparing exhibits. Versions of Lotus', Novell's, and Microsoft's office suites from the early 1990's behaved this way. The primary disadvantage is compatibility. A vendor's word-processor can only embed full information about that same vendor's chart; mixing two vendor's applications requires reverting to the previous technique, pasting pictures.

1.3.3 Method three—In-place activation

Integration based on standard protocols and frameworks.

Currently, on Windows machines almost every word-processor supports a common technology created by Microsoft called Object Linking and Embedding (OLE) [11, 15]. OLE defines a standard for embedding one document or part of a document in another. The embedded chart is managed by a "server" application, while the word-processor is the "client." That is, the word-processor application asks the chart application to "serve" it in displaying and editing the chart. Any vendor that writes an OLE-compliant chart server application can embed a chart into *any* other OLE-compliant word-processing client application. Thus, vendor lock-in is removed.

Furthermore, these standards define "in-place activation," a series of protocols and user-interface guidelines for editing the chart object directly inside the word-processor. To the user, this provides a "document-centric" point of view, instead of the older, application-centric perspective. This technology is currently supported on most major platforms (OLE, OpenDoc, and JavaBeans). See Figure 1.3.

How does this all work? Basically, each particular compound document technology defines a specific standard of protocols and behaviors for every step of the embedding process. For OLE, Microsoft has designed no fewer than forty separate interfaces for in-place activation, linking, and embedding. These interfaces specify protocols for all sorts of activities, such as: telling the chart object to draw itself; managing menu and toolbar merging when the user edits the chart inside the word-processor; telling the word-processor to save the entire

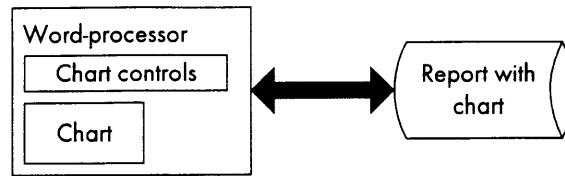


Figure 1.3: In-place activation

document even when the user is working inside the chart; persisting the chart inside the report document; notifying the word-processor whenever the image in the chart should be changed; notifying each other when the user drags and drops data from one place to another; and managing undo/repeat activities at the document-level.

In these scenarios, the word-processor is often called the "container" because it contains embedded objects. Because a container must manage an entire document that might contain many embedded objects, and each embedded object only communicates with one container, developing container applications is more difficult than developing server objects. As a result, there are significantly fewer containers than objects on the market. This is not unreasonable, however, because typically a container is simply a blank slate onto which objects are placed, so most of the value-added, constructive, and interactive work occurs with the embedded objects and not the container itself.

1.3.4 The future of compound documents

In today's implementations, embeddable objects such as charts, pictures, spreadsheets, sound clips, and videos, exist on a flat namespace. That is, there is no hierarchical organization of these disparate objects, and to the operating system and applications these objects are conceptually indistinguishable. They support the same interfaces, they can paint themselves when told to, and they can save themselves to disk. Within Microsoft Word, for instance, choosing "Insert Object" presents a complete list of all available objects that can be embedded. There is no content management, no filter to assist the user in selecting the appropriate object. There might be five charts available to select from, interspersed among the various objects, but the user has to know which those five are. This is, at the core, no better than the application-centric approach that requires the user to find the application in order to access the document. At the other extreme, from within Word selecting "Insert Chart" embeds Microsoft's particular chart without any choice by the user. There is no middle ground "allow me to choose a vendor's chart and insert it" between the extreme options "insert any generic object" and "insert the Microsoft chart."

Compound document technologies are heading towards content management, though. Microsoft has introduced a specification called "component categories" that will enable an object to register itself as a "chart." Then, containers that are aware of component categories could let the user filter all objects down to just the available charts. The OpenDoc compound document technology had a more sophisticated scheme that would have

provided for specification of default objects. By default, the same chart would be used in all OpenDoc containers. This extends even further the document-centric tendencies, because the container application plays a much less significant role.

Eventually the application and document will be subservient to the task at hand; call this *task-centric* computing. When a user wants to compose a memo, the operating system will know that for memos she likes to use WordPerfect with Memo Template #3 and spell-check with Microsoft Office's spell checker. For reports she prefers Word with a customized report template. Then, instead of running Word (application-centric) or choosing "Create new Word document" from a context menu (document-centric), she will say, "Create a memo." (And with speech recognition software becoming ubiquitous on many systems, that is a literal "say.")

1.3.5 Summary

Compound document technology has been enabled by:

- **Standardization.** Standards enable interoperability, document abstraction, and consistent behavior.

- **Objectification.** Each container and server is its own entirely self-contained entity called an *object*. They are transportable, modular, reusable, and easily replaceable.

- **Broad support.** For most platforms, vendors willingly support the standards. Almost all applications for Windows support the OLE standards (in most cases, this is required by the Windows 95 Logo program); many vendors pledged to support OpenDoc for MacOS and OS/2. Hundreds of vendors have announced their support for the JavaBeans specification for Java components.

1.4 The Evolution of Optimization Applications

Switch gears now to a seemingly entirely different task: invoking an optimizing solver within some operational modeling software. This could be calling the CPLEX Callable Library from within a home-grown application, or solving a network flow optimization problem with OSL, or solving a mixed-mode distribution network with a completely homegrown customized C subroutine. There has been an evolution of these tasks over the past decade that closely parallels the evolution of compound documents.

The key realization is that the solver itself, the actual part of the code that takes input data, crunches numbers, and provides output data, is a separable piece of the application. Or, at the very least, it can be designed as such. Solvers can, therefore, be components, and with the proper infrastructure, they can become embeddable, reusable, modular, and replaceable.

The parallel problem of placing a chart inside a report in the optimization arena is that of providing input to, invoking, and pulling outputs from, a solver in a repeatable manner. For instance, a scheduling system could require that the operator feed in a data set every morning, press a button "Go!" and then analyze and act upon the output data.

The traditional solutions to building these models and systems are application-centric. The modeler or developer begins the implementation process selecting a modeling language application and a destination solver application. These choices constrain the development process thereafter; as with any software development, decisions made earliest in the project are the most costly to reverse later. The parallel of document-centric computing in modeling might be called *model-centric*. The model is the driving force; modeling languages and solvers are selected dynamically based upon which ones are appropriate for the model.

The first two methods listed below are application-centric; the third method describes the future model-centric environment.

1.4.1 Method one—Custom solution

This is a highly traditional and pervasive technique, possibly because operations researchers look for efficiencies within other systems and not within themselves. In this scenario, the operator of the system is responsible for managing the data files and manually injecting the correct data sets into the system. Upon completion, the operator must take an output file and convert it into a more useable format.

For example, a research group at MIT developed a tactical planning application that fit within this scenario (see Ruark [91]). The users stored all of their data in spreadsheets. When they wanted to run the model, they had to save each data set as a separate text file, organize all the files into one folder, and then run the application. The application itself simply read in all these text files, created the internal LP model from the data files, invoked a commercial solver, and wrote the outputs directly into new text files. The user then would have to import these text files back into the spreadsheet program to create a new workbook containing all of the outputs. Then they could begin an analysis phase. The developers automated this process as much as possible, providing users with macros in the spreadsheet application to save and load the text files and to create a few important graphs from the raw data.

This method is similar to pasting a picture in that in order to make a change, the entire process must be repeated. (Remember that when pasting a picture of a chart, changing the chart requires that the old picture be deleted.) There is no facility to easily change a value within the context of the application. Also, there are many files to maintain—the application had forty-three input files and nineteen output files—which leads to versioning and maintainability problems.

However, this solution is, in some ways, slightly worse than the picture-pasting technique, because the users were locked into using the provided application and the provided solver

(LINDO). It's almost as if there was only one charting application in the first place, and even then it wasn't well integrated. See Figure 1.4.

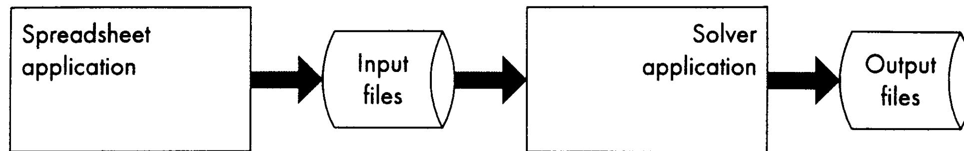


Figure 1.4: Custom solution

There is room for improvement. For example, the application could create MPS files, and then invoke the optimization engine as a separate step. This would make the optimization engine replaceable fairly easily; this is generalization. The application could have been more tightly integrated with Excel, or the input and output more tightly integrated into the application; this is specialization. However, the client is still tied into the custom application, because modeling logic was hard-coded into the code itself. That is, the constraints are created using procedural Pascal code; changing a constraint requires that the entire application be recompiled from the development machine. Therefore, the client cannot change the underlying model.

1.4.2 Method two—Modeling language

Two drawbacks of custom code, namely customized I/O and modeling code and hard-coded modeling, are addressed by using standard modeling languages and standard database packages. Many current modeling language drivers, like AMPL, can read and write to standard database formats, through ODBC or SQL, and they can invoke standard solver libraries, such as CPLEX, OSL, and MINOS. Furthermore, the developer describes the model not with compiled code but with interpreted textual representations of the model; modifications to the model thus happen at the point of delivery and not at the point of development. See Figure 1.5.

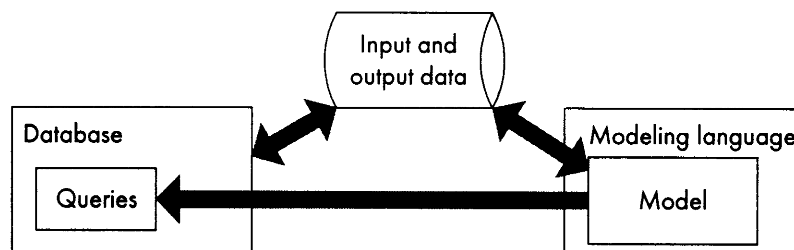


Figure 1.5: Modeling language

In general, this technique imposes fewer steps upon the end-user (it's easier to automate). A single data file (a large database) could contain all of the input and output data, easing the versioning and maintenance process. For operational modelers, the modeling language is typically easier to write, understand, and debug than the corresponding lower-level computer code.

This technique is similar to that of the second technique with charts in reports. Namely, there are separate environments—the word-processor is analogous to the modeling language environment while the chart is analogous to the data set and select solver. To change the data, the user must work within the database environment; to change solver parameters requires working in the modeling or solver environment. There is certainly vendor lock-in; using a particular modeling environment forces the modeler to use only those databases and solvers supported by that modeling environment. Often this problem is mitigated by standards like ODBC and MPS, but these standards are regularly not expressive enough, especially in setting solver parameters such as maximum number of iterations. In that case, the modeling environment must specifically know how to set those parameters.

Graphical modeling environments

A recent advancement on the textual modeling environment is the addition of graphical interfaces for the modeling process; see, for instance, Jones [51], Ma, Murphy, and Stohr [64], and Piela, McKelvey, and Westerberg [82]. The model itself is expressed in a graphical notation (possibly converted to or from an underlying textual representation), which the user can manipulate using a mouse. The primary benefit is also a potential disadvantage: the textual representation might be hidden from the modeler. This is an advantage because it allows the modeler to abstract the problem to a higher, graphical level while not worrying about the underlying representation, but it also might be a disadvantage if the modeler cannot easily access the underlying model to make sure the correct model is being generated. Generally, the modeler will not have to know a modeling language (GAMS or AMPL) or a solver language (CPLEX commands) to develop sophisticated models.

With these environments usually comes the added benefit of being able to manipulate problem data from within the environment. Thus, the modeling application, solver, and database appear as a unified environment. This is typical of *integrated modeling environments.*

The problems of vendor lock-in still remain, however. Graphical environments constrain the user even more to using those solvers and databases supported by the environment. Substituting between solvers supported by the environment might be simple, but it might be extremely difficult to insert a foreign solver with no native support.

1.4.3 Method three—Component-based modeling environments

Modeling environments do not yet have a parallel to the existing compound document technologies, where models, data sets, and solvers are interchangeable across different vendor implementations.

Examining the reasons compound documents have succeeded explains why:

- **Standardization.** The standards that exist in the modeling community are primarily text-based. Modeling languages and representations like MPS, SML, AMPL, and GAMS rely on textual descriptions of a problem. The query language SQL is also textual. Outside of ODBC for database access, there are few implementation standards. There are no implementation standards for modeling environments. At the binary level, two different environments and their associated solvers and data handlers cannot communicate; they must rely on textual standards. This is equivalent to relying on simple clipboard formats like rich-text format and pictures for inter-application communication: it might work, but it is inefficient and insufficiently robust. Before there is true cross-environment sharing of models, data, and solvers, there must be implementation protocol standards.

- **Objectification.** Conceptually, it is easy to imagine that the modeling environment is replete with objects. The solver, the mathematical program, databases, constraints, and variables all stand out as vague kinds of "objects." There are several papers on bringing math modeling into the object-oriented domain; see, for instance, Lenard [62], Muhanna [77], Lazimy [60], and Huh [45]. But there has not been a consolidated, concerted effort to bring portable objects to implementation. Popular libraries like the CPLEX Callable Library are still modestly difficult to use and do not exhibit sufficient behaviors that rank as object-oriented. The CPLEX Callable Library API is a single list of over a hundred functions in a single namespace.

- **Broad support.** This will be the largest challenge for the community. Vendors are generally resistant to removing vendor lock-in, especially with their own products. Most likely, it will take a "killer application" modeling environment that is so simple, extensible, and powerful that it can essentially force a standard on the market. For instance, the CPLEX API could theoretically be licensed to create a CPLEX-compatible system that is transparently interchangeable with CPLEX at the binary level, thereby elevating the CPLEX API to a pseudo-standard.

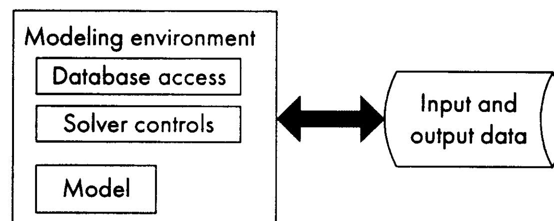


Figure 1.6: Component-based modeling environments

What will be the shape of these solver components and the modeling environments into which they integrate? Consider the previously stated needs: ease of integration, simplicity of interface, and separation of orthogonal functionality. Ease of integration enables an application developer to create a modeling environment where solvers, algorithms, data sets,

and rules can be dynamically and flexibly specified by the modeler with complete freedom. The best-crafted, manageable, and useable components exhibit simplicity of interface; they are easy to comprehend and use. They expose minimally sufficient functionality—enough to do the required task but no more. The separation of orthogonal functionality makes components more flexible. The following matrix compares these needs with the techniques.

	Standardization	Objectification
Ease of integration	Flexible, dynamic, unrestricted modeling.	Simpler integration of components.
Simplicity of interface	Focus knowledge and concentration on task at hand.	Simpler implementation of components.
Separation of functionality	Fine granularity of integration, interconnections, and modeling.	Object lifetimes and behavior tied to their mission.

Table 1.1: Matrix of component modeling benefits

Flexible, dynamic, unrestricted modeling. Two components that support the same standards and are easy to integrate will be indistinguishable in their appearance to a modeling environment, permitting dynamic swapping of the components. This is plug-and-play modeling, where one solver or query engine can be swapped for another with a simple click-and-drag operation.

Focus knowledge and concentration on task at hand. Rather than present a massive, flat list of all possible functionality, simple, standardized interfaces present smaller chunks in well-organized, comprehensible groupings. Operations that act on files are completely separated from operations that set parameters, create constraints, limit run times, etc. The developer or modeler can learn what is necessary for the interfaces at hand, and postpone understanding other parts of the system.

Fine granularity of integration, interconnections, and modeling. If a single component managed both file I/O and computation, that component is useless if the inputs cannot come from files. By creating two components, one that manages file operations and one that manages computation, either one can be swapped with another component to increase overall functionality. Consider a scenario where validation must be performed with random numbers. Where one component manages both file and computation, the random numbers must be shoehorned into whatever file format the component expects, thus adding work to the testing stage. If that component reads data from a complex corporate database, this could get complicated indeed. When a separate component manages the database lookup, and the connection between the two database component and the solver component abstract any specific implementations away, a new component could be created that generates random numbers and feeds them to the solver component without having to replicate the database schema.

Furthermore, instead of hooking together entire applications by specifying which output files from one application go to which input files to another, which requires batching, scripts, and pipes, individual components will be hooked together at the binary, procedural level.

Simpler integration of components. When vendors create components in manageable, encapsulated objects that can easily integrate, it becomes easier to link components. These linkages include static relationships, such as specification of sub-problems, constraints, units, and database locations, as well as temporal relationships, such as workflow, sequencing of algorithms, and pre- and post-conditions. Each relationship hooks together two well-defined, contained objects.

Simpler implementation of components. Objects with simple, minimally sufficient interfaces are easier to develop than objects with complex interfaces. Typically, the discipline required to implement objects well makes developing objects more difficult than developing procedural code, but in environments where there is a natural mapping of the domain into an object space, objects can actually make implementation simpler. Objects partition the scope and space of the problem, allowing a developer to concentrate on a particular, coherent piece of the puzzle while treating the remainder as a black box.

Object lifetimes and behavior tied to their mission. In high utilization, multitasking environments with limited resources, it is important to manage those resources effectively. At one extreme, imagine an application in limited memory space with two procedures: a file I/O procedure and a solver procedure. The file procedure takes little time but requires much memory for its code. The solver procedure takes more time and requires much memory for its data structures but not its code. If both procedures have to reside in memory concurrently, then there will be extensive disk activity as the virtual memory manager must page file code or solver data structures in and out of physical memory. This is because the file I/O procedure has no knowledge of when it is no longer needed, or knowledge of the self.

If these instead were two properly implemented components, then the file component could recognize that once it has completed its task, it could delete itself. This would free up the limited memory, thus resulting in less paging. In an environment where the code might be running hundreds of times simultaneously, as part of a massive parallel algorithm, this might be desirable behavior.

1.4.4 The future of modeling environments

Currently, modeling environment user interfaces typically lag behind the user interfaces for mainstream commercial software. As advances in interfaces trickle down to the massive software base, modeling environments will eventually acquire the interfaces that users expect and get from their other desktop applications.

Modeling environments will benefit from the dissemination of technologies for natural language processing, voice recognition, artificial intelligence, and agents. These will enable a

modeler to pose problems to an environment in a very natural way. To make a cultural comparison, eventually modelers will be able to invoke algorithms on their computers in much the same way that characters on *Star Trek* do. "Computer, how long can we maintain power at our reduced emergency levels while enabling certain subsystems?" "Computer, please display a list of all people who are likely to default on their loans in the next ten days." Simple questions that make many assumptions, disguise constraints, and soften criteria.

1.5 The Evolution of Application Implementation

Prior to the development of the Microsoft Component Object Model (COM) for Windows and its use by applications developers, a Windows application relied in a monolithic manner upon the operating system services, as illustrated below in Figure 1.7 (this is true for all similar operating systems):

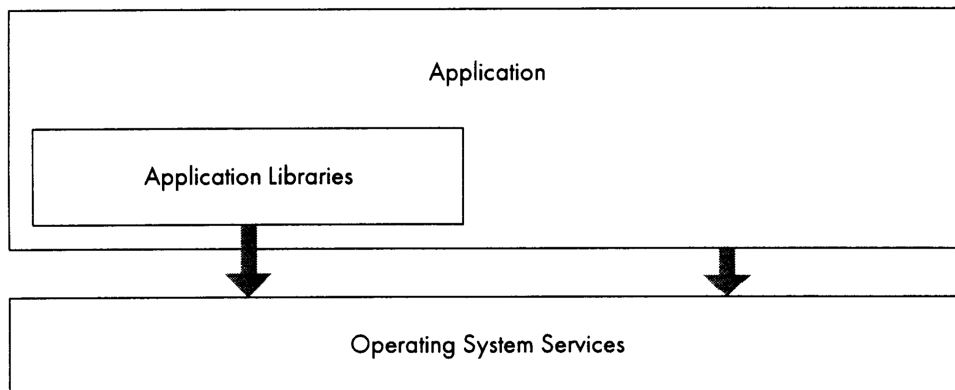


Figure 1.7: Monolithic application architecture

This picture shows the various pieces of this architecture. Each element is presented as a box with its function labeled inside. Where one element contains another element, the contained element is a sub-module of, and inherits the development, use, and distribution characteristics of, its container. For instance, application libraries are linked into an application and distributed with the application. When one element has an arrow to another element, there is a dependency of the vertically higher element upon the lower element; that is, the higher element requires and utilizes pieces of the lower element. Containers are dependent in the same way upon the elements they contain. The elements of this architecture are:

> **Operating system services.** Standard services provided by the operating system, such as GUI, file management, kernel object administration, and network support. These services are developed and distributed by the manufacturer of the operating system.

Application libraries. Reusable codes developed for this and other applications by the software company. Examples include linear programming solvers, high-precision numerical routines, differential equation solvers, and special graphics codes.

Application. The code specific to the modeling application. This is the perceived entity that the user interacts with.

Applications with this architecture are characterized by highly specialized and possibly non-conformant user-interfaces, proprietary file formats, particular modeling languages, and little interaction with other applications. These characteristics are due to the absence—especially in earlier operating systems—of sufficient operating system services such as standardized user-interface components, structured storage capabilities, and standardized scripting.

The advent of component technology support within operating systems mitigates many of these problems. Apart from the standard benefits of components and the "object-oriented way," this change has brought stability and consistency in the behavior of application subsystems. For example, Microsoft's Object Linking and Embedding (OLE) libraries introduced *structured storage* as a way to manage document storage. Applications that use structured storage acquire an additional feature set for free from the operating system, such as easier cross-application support and file summary information in Explorer.

Applications that use these component services have an architecture like this:

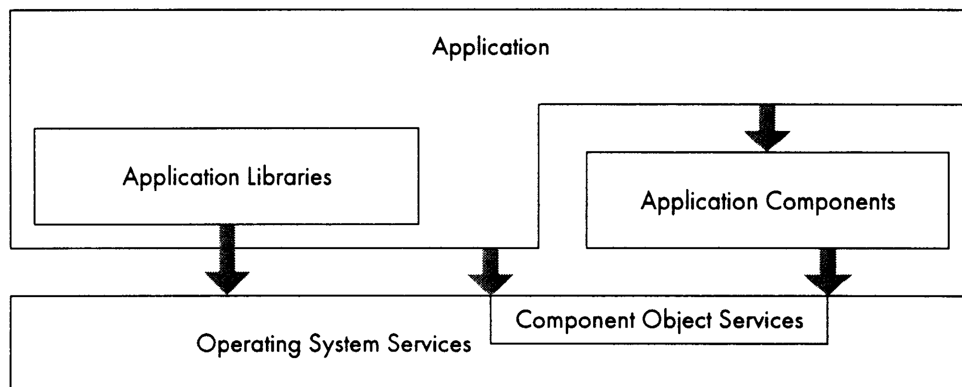


Figure 1.8: Component-enabled application architecture

The new elements of this architecture are:

Component object services. Component technology services provided by the operating system, such as component instantiation and administration, scripting, inter-process and remote procedure calls, typing, and storage.

Application components. Reusable components developed for this and other applications.

There are several differences between application components and application libraries. Components can leverage the existing component object services. Components can be upgraded or changed without recompiling the entire application, whereas a change to a library forces the application to rebuild and therefore re-ship. When two components perform the same function, which one to use can be chosen at run time; when two libraries perform the same function, which one to use must be chosen at link time[2].

1.5.1 The modeling environment literature

Most of the literature that describes modeling environments or potential modeling environment implementations is based on the monolithic application architecture. As such, these papers detail specific proposals or implementations for either application libraries or modeling applications themselves. While many of these papers propose their own modeling language and solver hooks to permit extending the application, the implementations are severely limited and restricted to the proposed framework or application. It is rare to find a proposal that provides both an extensible framework that can stand alone and also fit into existing modeling environments.

The usual modeling environment or implementation paper develops one of these theses:

1. It proposes a modeling environment or a methodology captured within a modeling environment, which is new and improved. The environment is extensible in a unique manner, and anything developed for this environment will not work in other environments. [2, 3, 35, 45, 57, 74, 75, 77, 81, 82, 84, 86]

2. It proposes a modeling language that satisfies some particular need. The modeling language expresses a solution technique for this need, but does not satisfy other needs of a similar scope, and it might be difficult to extend the language to account for other needs. [4, 5, 10, 26, 32, 35, 50, 60, 77, 81, 86]

3. It offers a solution technique or pseudo-code for a particular problem. The technique might be an algorithm, straight ugly C code, or a Unix binary, or it might be designed for a particular modeling environment. In most cases, using it requires recompilation, re-coding, and often re-designing; i.e., too much work that is not related to the modeling process.

The evolution of these modeling frameworks and applications mimics that of monolithic feature-rich applications before component technologies were available. Each vendor creates a full-scale application with unique customization and automation features; customers are usually locked into purchasing extensions from the same vendor to enhance the capabilities of the application. Special filters and applets are required to convert documents from one application to another (look at all of Word's and WordPerfect's available import filters).

---

2 In a sense, the components are *dynamically bound* while the libraries are *statically bound*.

Modeling applications exhibit this "feature" as well. Special filters are required for each particular optimization engine, and for each database type; if the vendor does not provide a filter for the optimizer of choice, that is reason enough not to use the product even if it has the best modeling capabilities.

The evolution of modeling frameworks must now follow the path other applications are taking, towards component-based architectures, leveraging smaller independent elements that can be customized, automated, and presented in consistent, standardized ways.

1.5.2 Future modeling application implementations

Just as the operating system provides specific component technology services for component-based applications, there will be a modeling component services system (MCSS) that provides for component-based modeling applications. The MCSS will provide a standard means for managing a model store, for storing and manipulating data types and conversions, and for defining solver, data, and application interactions. It fits into the architecture just above the operating system component services:

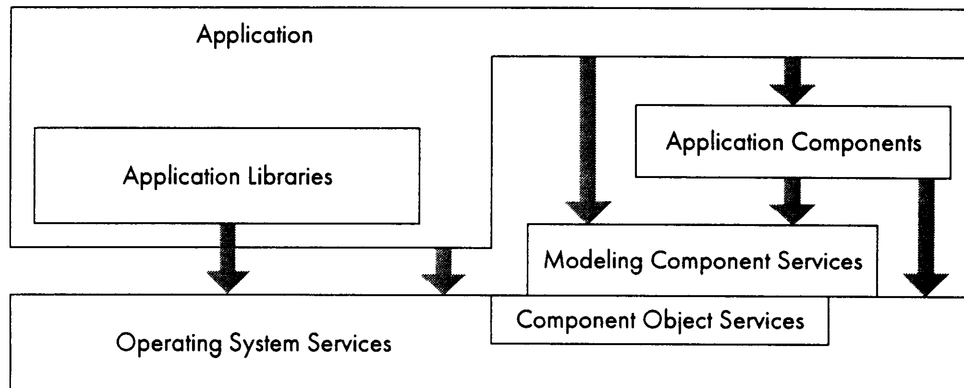


Figure 1.9: Future application architecture with modeling component services

The new element of this architecture is:

> **Modeling component services.** Modeling component services provided by an add-on library to the operating system, providing for standardized modeling services like typing, the model store, and defining solver, data, and application interaction. Typing encompasses the collection, classification, manipulation, and conversion of dimensions and units of variables, such as "cost per truck" and "feet per second squared." The model store is a database of models and solvers available on a machine or accessible on a network.

The MCSS has these advantages over traditional proposed library frameworks:

- It will be based on future technology. Component-based operating systems and object-based distributed network systems will be widespread within a few years. The MCSS will leverage these systems.

- Being based on standardized operating system-provided technology, the MCSS will use technology familiar to developers, thereby decreasing development learning curves, times, and costs.

- For that same reason, modeling components that use the MCSS will work in other productivity applications that use the same underlying object model, such as Microsoft Excel, Borland Delphi, etc.

- Modeling applications that use the MCSS could work with any modeling components that use the MCSS. So, rather than developing a component for a particular modeling environment, a component will run in all environments. This item and the previous have great appeal to the developer who wants the widest possible audience.

Geoffrion talks about the Structured Modeling Language as the *lingua franca* for modelers [32, 33], but this is at a conceptual, definitional level targeted towards the modelers themselves. The component-based system is for the developers who actually implement the environments. The building blocks of a component-based system will form the cornerstone of well-engineered modeling systems. Through a component-based, standardized service system, developers of modeling environments, model solvers, and model database systems will be able to communicate using a common *implementation* language.

Future operating systems will only increase the presence of the component technologies and the object-oriented paradigm, as shown in Figure 1.10.

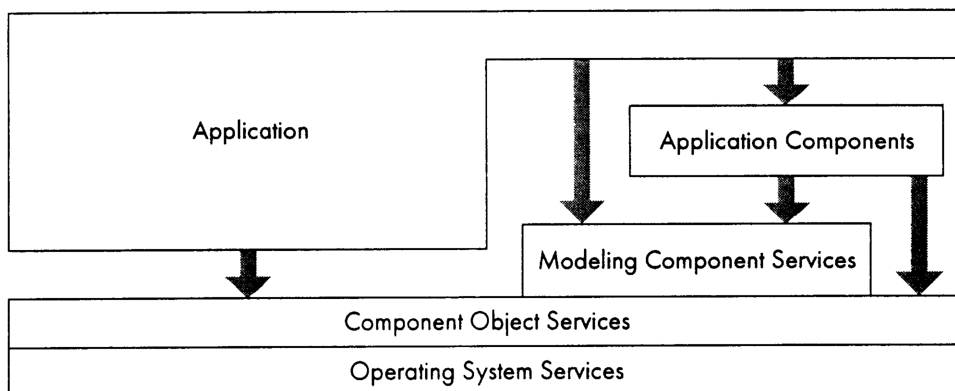


Figure 1.10: Future component-based operating system architecture

Now, application libraries are transformed into application components because the operating system services are accessible only through a component object services layer. The Java Virtual Machine is an example of an existing environment with this architecture.

This thesis presents a framework that is the first step towards a future modeling component services system.

1.5.3 COM, CORBA, object models, and domain modeling

The component object services described in the previous sections are effectively part of the operating system. These object services, and the models and philosophies imposed by them, are typically independent of any single application, but rather standardize the interaction, behavior, and structure of objects that use them. While many objects models have been proposed, the field of viable models has dwindled to two: COM and CORBA. Microsoft's Component Object Model (COM) was originally the underlying technology of Microsoft's Object Linking and Embedding (OLE), but is now pervasive throughout Microsoft Windows and applications that run on Windows. The Object Management Group's Common Object Request Broker Architecture (CORBA) is a cross-platform standard with significant industry input and support. Both of these object models have the backing of major industry players, significant legions of fanatical followers, market presence, and successfully implemented cases to guarantee that they both will be prevalent object models for years to come. Because this thesis will target the types of applications which today are routinely implemented in Windows, the thesis uses COM as the underlying object technology (for more information, see section 2.3.2, page 75). This choice is, effectively, arbitrary; the important fact is that there is a standard object technology on which the operation research modeling services are built.

COM, CORBA, and similar object models are horizontal technologies. They span all domains while not completely solving any of them (except the most trivial or those that are specifically within the COM or CORBA umbrella). COM and CORBA define how objects interact, life cycle issues, notification protocols, introspection and discovery, but they do not, for instance, even attempt to define a standard mechanism for financial transactions. Finance, banking, medical services, business workflow, and operations research problems are outside the scope of what the COM and CORBA specifications provide. Instead, these object models are extensible, so that others can create their own specifications for particular domains. For instance, Microsoft in conjunction with a number of other companies has defined an extension to COM called OLE DB that frames a database specification within COM. In other fields, Microsoft has created specifications for healthcare, finance, and insurance [73]. Similarly, the OMG is overseeing the creation of vertical industry specifications such as the Business Object Framework Facility, part of the CORBAfacilities specifications (See Mowbray and Ruh [76]). These specializations of the object technologies into vertical domains are intended to be used by application developers in those domains; they will use these specifications to build portable applications that can share data and control. In finance, for instance, two different banks' applications would exchange

transactions or a teller's terminal would communicate with the branch-office central computer using one of these protocols.

This thesis is essentially proposing a vertical, domain-based specification for operations research solvers and solutions, equivalent in spirit to the vertical standards that Microsoft and the OMG are creating. It will specify new COM interfaces, objects, enumerations, structures, etc., that apply to this domain, as well as the protocol for the behavior between the various objects.

1.6 Overview of the Thesis

The thesis is relatively lengthy, compared to many operations research theses. Hence, it is important to clarify what this thesis is not trying to do in relation to what has been discussed above. With definition-by-negation, this not only limits the expected scope for both author and reader but also brings into focus what the thesis will be accomplishing. This thesis will not:

- Propose a new modeling language. A modeling language is not an implementation device for software developers but for the modelers who use the software.

- Propose a new modeling and problem taxonomy. Instead, the thesis will propose a minimal mechanism for registering installed solvers on a computer. The flexibility to work with different hierarchical taxonomies on the global list of solvers will be managed by not imposing any one taxonomy upon the system.

- Propose a new model or solver selection methodology. The problem of choosing a model or solver to fit a particular problem is the topic of other theses. This thesis will present a unified method to administer a solver database while not imposing any selection technique.

- Propose a new modeling environment. The purpose of the thesis is to define an infrastructure for the developers of environments without limiting the presentation of that environment to the user.

This thesis should be viewed as a necessary albeit possibly unpleasant and at times boring presentation of a component-based modeling infrastructure, targeted towards the people who are developing, architecting, and programming modeling systems. This thesis *will:*

- Present a set of *requirements* for reusable algorithms. Namely, what are the behaviors expected of truly useful, highly crafted, successful, reusable implementations? These behaviors are focused solely on making implementations of algorithms better, and exist independently of any particular platform, language, operating system, or algorithm.

- Provide a *specification*[3] of a modeling component services system and algorithm object framework that can conceptually extend an operating system to enable diverse modeling applications and modeling environments.

- Provide at least a reasonable subset of the associated *implementation* of that specification for COM, particularly on Windows 95 and Windows NT (though the specification will not necessarily be limited to COM or Windows, with suitable modifications). It must be noted that the specification is the true deliverable, in that once a specification is "agreed upon," anyone is free to develop an implementation of that specification.

- Provide a set of illustrative *examples* that use the specification. Any example programs, algorithms, screen shots, etc., will serve as proper techniques for using the specification only, and not necessarily proper ways to develop modeling environments in the whole or to develop algorithms.

Essentially, the four sections of the thesis are (1) this introductory document with a literature review, (2) the requirements of reusable algorithms, (3) the framework, and (4) the proof-of-concept through implementation and examples. The last three are discussed in more detail below.

1.6.1 Requirements of a reusable solver

There are many aspects of an algorithm implementation that concern a developer. These range from fundamental issues such as correctness, efficiency, and robustness, to secondary, more esoteric concerns (in the operations research community) like maintainability, security, and reusability. While the first three are routinely studied and analyzed, the last three receive little attention in operations research literature. Maintainability affects the ease with which the implementation might be updated in the future, to fix bugs, upgrade performance, or add enhancements. Security addresses both the internal security and integrity of the implementation itself, as well as how well access to the data and process of the implementation is protected. Reusability measures how easily the implementation can be removed from its initial context and placed into another context.

From the perspective of operations researchers who need to employ algorithms to solve larger problems, reusability is easily the most important of the "secondary" aspects. Hence, the thesis will focus on examining *why* current implementations are insufficiently reusable in today's computational environments, *what* is required and what is possible for reusable implementations, and *how* to achieve more reusable implementations.

---

[3] Commonly referred to as an API, or *application programming interface,* although the specification will include elements that transcend the standard interpretation of API, which usually is limited to functions exported by libraries and operating systems.

The requirements chapter is intended to organize, define, explain, and defend these desirable behaviors in the context of operations research modeling implementations. This will exhibit an unusually high attention to detail to a topic that has been insufficiently addressed by the modeling literature.

The solution to the reusability problem has two dimensions. For one, software developers can extend their efforts to develop individually reusable implementations; that is, software components that by themselves satisfy as many of the requirements as possible. By selecting popular target environments for their implementations, such as Solaris or Windows or Excel or Mathematica, and by customizing their implementations to those environments, they can develop implementations which can receive the widest possible reuse within that environment. Through disciplined and careful analysis and design, they can make it relatively painless to retarget their implementations for other environments, as well.

Another method is to remove the dependence on selecting a target environment and introduce a layer of standards between the algorithm implementation and the environment in which the implementation will run. This has two immediate, significant advantages: the software developer has instant access to a wider audience, and the end user of an implementation that follows the standards can make reasonable assumptions about the behavior and capabilities of that implementation, thereby decreasing the user's learning curve and application effort.

The thesis will introduce such a standard for a particular target operating system, the Microsoft Windows family of products. Because of their ubiquity and availability within the thesis's target audience, Windows makes an ideal choice for an implementation platform.

1.6.2 Features and goals of the framework

The object framework and library will exhibit the following primary features:

- A pattern of solver invocation and a standard interface for implementing this pattern. Every algorithm implementation can support a standard interface for solver invocation (the steps taken to initiate execution of an algorithm). Client applications can then follow the invocation pattern to work with any available solvers.

- A specification for hooking many solvers together into a network of solvers that act in concert to solve a complex problem.

- Fundamental model-base support, including registration and enumeration of available solvers on a system or network. The library itself will include no logic about selecting an appropriate model or solver for a particular problem, but instead will enable easy extensibility of the model-base system to permit the addition of such intelligence.

- A mapping of the requirements of reusable algorithms into the component system. Additional patterns and interfaces will explain how the requirements can be easily incorporated into the model framework.

The framework and library has the following primary goals.

- Usefulness now and in the future. The system will be useful now because algorithms will be accessible to current client applications that can speak the common underlying object model (Microsoft's COM). The system will be useful in the future if modeling environments are created to leverage the features targeted specifically towards operations research modeling, such as the model-base and typing.

- Appropriateness and correlation to operations research modeling and implementation needs. Operations researcher modelers, often in the guise of management consultants and quantitative analysts, have particular needs. The thesis does not target those massive operations like airlines that have huge, recurring problems such as plane scheduling and yield management. Instead, it targets those professionals who have to cobble together quick solutions of small- to medium-sized problems.

- Extensibility without sacrifice of capability. A common trade-off to be made: the most extensible systems are often the least capable, while the most capable systems are the least extensible. The goal is to fall somewhere in the middle.

- Ease of implementation and of following the standards. As future software development packages advance the process of programming, this goal will follow naturally. In the mean time, it is important to make implementing an algorithm a not-impossible undertaking. While the thesis targets software developers with implementation details, the learning curve should not be too steep.

1.6.3 Proof of concept

In order to validate the proposed framework and standards, certain issues will need to be addressed. How much effort is required to implement algorithms within the new framework compared to previous implementation efforts? How much easier is it for the end user to work with implementations that follow the standards? What reusability requirements does the framework meet, and which need further improvement?

The thesis will present several examples in order to illustrate successful applications of the specification and requirements. These examples include:

1. An object model and sample implementations for random variables and simple queueing systems. These implementations allow a modeler within Excel to calculate distribution functions and moments for common distributions such as normal, exponential, Poisson, and binomial, as well as calculate performance distributions for simple queueing systems such as M/M/1 and M/M/k (section 4.1.2, page 223, and Appendix A.3, page 311).

2. A before and after look at a shipping application developed for clients of the Leaders for Manufacturing (LFM) program. The "before" version incorporate both a complex algorithm and all the user-interface code in a single application. The "after" version has separated the solver code enabling it to be used from within any appropriate client environment, such as Excel, modeling environments, and the customized, dedicated user-interface that ships with the application (section 4.2.4, page 250).

3. Wrapping existing algorithms with objects that fit within the proposed framework. These include some freely available network optimization codes, some of the random variable codes from the first example, and a proposal for a linear programming object model framework (section 4.1, page 220).

4. A description of an application with another LFM company that used a bin-packing algorithm as a sub-problem, and how it would have benefited from a standardized component model for the bin packing implementations (section 4.2.5, page 253).

1.7 Overview of Related Modeling Literature

Many of the ideas explored by the thesis have been examined or mentioned in previous works, particularly in the areas of structured modeling, model selection, and model integration. Blanning [6] provides a brief overview of these areas in the broader context of model management systems, along with an extended bibliography. Geoffrion [33] presents his view, as of 1989, of what a modeling environment should offer. Eight years later, his grand vision of conceptual unity for an environment has not been fully realized. Yet, there are plenty of new ideas about the modeling environment, mostly derived from the enormous success of graphical operating systems and their object-oriented flavor. Many aspects of the modeling environment have been prototyped but there has been no grand, especially cross-research, implementation.

1.7.1 Structured modeling and beyond

The primary source in structured modeling is Geoffrion [32, 35]. These papers present the structured modeling foundation: the basic framework is a hierarchical, directed acyclic graph that represents both the semantic and the mathematical content of a model. A structured model is a network composed of five basic types connected together by dependencies. The types are the *primitive entity* (usually a non-valued *a priori* postulation), the *compound entity* (a non-valued concept dependent on primitives), the *attribute* (a constant-valued property), the *function* (a derived-value based on a specific rule), and the *test* (a specialization of the function where the only valid results are true and false). A fully explicit structured model network has as many elements as there are variables, constants, functions, etc.; this is the *elemental structure.* By grouping similar elements together, more generalized networks are created, and herein is the hierarchical nature of the structured modeling method. A partition of the elemental structure groups elements into families of the same types—for example, all variables of the same letter over all its indices—called *genera;* the result is the *generic structure.* This continues

up the hierarchy through *modular structure,* the *structured model,* and the *model schema.* All levels up to the structured model represent specific *instances* of a model, replete with actual data. The model schema represents instead the class of all instances whose structured models are similar. Geoffrion presents a descriptive language, the Structured Modeling Language (SML), that can be parsed effectively by computers and understood by trained practitioners.

The primary benefits are: (a) a single model representation can be used in different modeling domains such as mathematical programming, simulation (with extensions to the SML to account for stochasticity and state dynamics), forecasting, and queueing; (b) a single model representation facilitates transmission of a model among various modeling environments, solvers, and data stores; (c) a computer-understandable modeling language permits computer-based manipulation of the model, so that the computer is no longer simply a text-editor but is directing and performing semantic manipulations (for example, automated model integration). The benefits of (a) are being realized with continuing research, as the basic SML is expanded and other modeling needs are shoehorned. Similarly, as research deals with model integration theory and issues (see below), (c) will come to fruition. The outcome of (b) is much less obvious; this is a focus of this thesis, because there has been little success in broad implementation of systems that feature SML and permit interaction with other systems. Many prototypes have been created, but there have not been any shipping commercial-grade applications that use SML to perform modeling life cycle tasks.

A much more recent paper by Geoffrion [37] summarizes research to date, including work being undertaken concerning model integration, simulation via structured modeling, and the existence of other structured modeling languages based on graph grammars and object-oriented methods. A telling feature is that as of early 1996, the flagship implementation of an SML-based system is still a system developed during the early 1990s (FW/SM), built on top of a now-defunct inherently non-graphical product (Ashton-Tate's Framework IV). Jones [49] has developed a graphical-based modeling system that is also in a prototype stage (Networks/SM), but it seems unclear how this might be integrated with other application environments as well.

Bhargava and Krishnan [5] discuss an example of the dramatic departure from structured modeling provided by first-order logic (FOL) methods. They tackle the process of converting a user-composed problem statement into a domain-specific language and then into a mathematical model from the perspective of FOL-based model formulation languages. This technique seems language-heavy: they introduce the language PM^* to model the domain of production problems, then break out the qualitative components of this into another language QPM^*, and describe how PM^* and QPM^* are related to two existing (meta-) modeling languages, $L\uparrow$ and $L\downarrow$. Mapping user-stated problems into QPM^* and $L\downarrow$ is facilitated through axioms and statements in $L\uparrow$. An automated model management system would generate a schema from the user's problem and then map data to that schema to generate a problem instance. While intriguing, methods such as this are not as amenable to implementation as the structured approach of the SML and the object-oriented approach of the Unified Modeling Language, discussed next.

An exciting and promising development in software engineering is the Unified Modeling Language [29, 87, 102]. Originally called the Unified Method, this notation is an object-oriented approach to modeling any process, from enterprise processes to a telephone or car to even a mathematical program. The UML is so vast and encompassing that justice cannot be done here to its capabilities. The Structured Modeling Language can be mapped into the UML fairly simply. Since the UML also adds object communication and dynamic state information, it is a much richer environment for modeling. Furthermore, just as structured analysis and design—the prevalent techniques of the 1980s and the momentum of which must have fueled structured modeling—has given way to object-oriented analysis and design, so too will structured modeling give way to object-oriented modeling. The UML has the added benefits of becoming an industry standard [80] for software engineering process notation and of being a language understood by many software developers and architects.

1.7.2 Model selection

Model selection is fundamentally the process of assigning a known model representation (or creating a new one) to a user's problem, and of assigning an available solver to the selected model representation. Increasingly it is understood that flexibility and automation are the driving forces behind ongoing research. The user must be given flexibility in describing the problem she wants solved and in choosing a model and solver that best fit her needs. The modeling environment should automate the process of filtering inconsistent model representations and incompatible solvers, it should automate the presentation of suitable models and solvers, and it should automate the interconnections between data, model representation, and solver.

An important aspect of model selection (and integration) is how to structure and maintain the database of available models. This model database is usually called the *modelbase* to reflect the original suppositions that model manipulation theory paralleled data manipulation theory. At times this thesis might use *modelstore* to highlight the distinction that model manipulation theory has been observed to be richer than data manipulation theory—even though terms such as datastore, datamart, and data warehouse are now becoming common to database implementations.

Numerous papers have proposed techniques both for the model selection process and for the structure of the modelbase. Often the two go together because a proposed modelbase structure will impose a selection heuristic, and vice versa.

Eck, Philippakis, and Ramirez [26] propose a solver representation language paralleling SML, which is presumed as the modeling language, that enables automatically matching solvers to models. The solver representation captures all genera classes that a particular solver will manipulate, the preconditions and postconditions, if any, on those genera classes, and a list of input/output structures that the solver uses. The proposed selection process is to provide the end-user, who has a particular model schema in mind, with a list of all solvers in the solver store that by virtue of the genera classes it manipulates and its preconditions, could act upon the model schema in whole or in part. The user then selects a solver from the list

that has the desired effect and postconditions, and the solver is invoked. The solver representation captures genera class states, which are key-value pairs. The key is an ordered pair of a particular genera class and a property, and the value is the state of the property for that class. So, a condition might be {(<INDEXi>,ENUMERATION), enumerated} which means that the genera class INDEXi is enumerated. The list of properties is by definition extensible. This approach holds much promise, because it imbues a textual language with object-oriented characteristics and an extensible documentation mechanism.

Mili and Cioch [74] propose a documentation framework for mapping decision models to generic problem classes. There, the emphasis is on justifying to an end-user why a particular model is or is not suited for the user's particular problem. The documentation framework is presented as, but not limited to, a prose-based end-user document that is delivered alongside a modelbase. While the concept is important, not enough implementation or automation details are provided, and this paper is dominated by another documentation paper, Mili and Szoke [75].

Mili and Szoke [75] propose both a selection process and a modelbase structure within a more formal documentation framework. This framework documents known problem classes, available decision models, and accessible applications (solvers). Links among the three groups are also stored, including the relative strengths of applicability for models to problems and applications to models. For the user, a distinction is made between the task required and the tools available; mappings are generated between task and tool to measure fit. They have created a system that gathers information about a user's problem, creates a web of relationships between the problem and known models, and examines potential models for appropriateness. Finally, they note that their system facilitates activities beyond model selection, including searching and browsing of models and problems.

Banerjee and Basu [2, 3] propose a knowledgebase tree of model classes. This hierarchy is based on the concept of "frames" and "slots;" each element in the tree is a frame, and each frame can contain any number of slots. The slots contain the properties and parameters of the frame. Frames are variable in size and purpose; so, model types and model solvers are stored as frames at different hierarchical levels. The intent is to assist the user in selecting a model (solver) for a particular problem. In their methodology, there would have to be significant up-front analysis to determine the classifications and taxonomies. Furthermore, the proposed slot structures do not seem sufficiently formal to permit automation. Finally, the knowledgebase limits its use realistically to the selection process only. The techniques of both Eck, Philippakis, and Ramirez [26] and Huh [45], discussed below, are better choices for the broader tasks of selection and integration.

Lenard [62] introduces the effectiveness of object-oriented techniques to structured modeling. The author proposes a set of classes and protocols (communication schemes) that capture structured modeling types and their interaction. This object hierarchy can also be interpreted as an early proposal for a modelstore implementation.

Using SML as the model representation, Huh [45] proposes an implementation using object-oriented constructs within an object-oriented database system (ODBMS). The components of a structured model are stored as objects within various tables; these include a table of all the modules, one of all the models, and a table of ports. Ports are the input and output points for a model; primitive entities and attributes might be input ports, and functions and tests would be output ports. A benefit of this approach is that the objects have functionality; a model object has the Solve operation that is tailored to solve models of its type. Thus, the database environment becomes a complete modeling environment; commands are composed in an SQL-based language. Huh abstracts three layers of the modeling process. At the top are *model types,* such as linear programming, forecasting, and max-flow. This roughly corresponds to the modeling paradigms of Geoffrion [35]. Below a model type are *model structures,* specific known problems that specialize a single model type. The ProductMix model is a structure of the Linear Programming model type, and the PlantLocation model is of the Integer Programming type. When actual data are attached to a structure, a *model instance* is formed. The Solve operation can be invoked on an instance; Solve in turn invokes the appropriate optimization engine for the model structure. The primary drawback of Huh's implementation is that the mapping between solvers and model structure appears rigid, and it is unclear how such a mapping would be extended for new solvers or models. A goal of the paper is to shift "much of the burden of determining an appropriate and compatible solver onto the model management system" so that the "user's possibility of misapplication of irrelevant solvers to the model is clearly excluded." This is accomplished by having all models expose only a Solve command. This command does take one argument to specify which of the appropriate solvers to use, but the user is limited to the list of solvers already specified for that model structure. However, Huh does not discuss the selection process; perhaps through judicious model selection, the solver selection problem is trivialized.

1.7.3 Model integration

Model integration is simply the process of making at least two models interact. They can interact at several levels (see Geoffrion [35]), but of particular interest are how models integrate structurally and substantively. That is, how are multiple model schemas integrated into a single schema, and how are multiple models integrated to act on the same data?

An early paper by Geoffrion [34] proposes an integration algorithm for creating a single schema from multiple schemas using SML. The five-step procedure is user-intensive; it is not apparent how to automate the integration process. Given the limitations of the SML, much of the interpretation of genera and elements is left to the user, so the user must specifically indicate which genera can be integrated and which must be renamed. However, the notions of integrability, reusability, and modularity are proposed as applicable to the realm of modeling. Dolk [23] highlights some of the problems of integration, including the problems of identifying join points for two schemas, manipulating an integrated model, determining what processes and algorithms are suitable for an integrated model, and mapping the data structures of an integrated model to available solvers. Geoffrion also

discusses software integration; an integrated modeling environment should support the user's preferred interface.

A pair of papers by Dolk and Kottemann (Dolk and Kottemann [24], Kottemann and Dolk [57]) present modeling integration from the perspective of the process: how are the activities performed on an integrated model themselves integrated? For example, if two transportation models of similar structure but different underlying data are to be integrated, the result might be a transportation model with the same structure but combined data; the same solvers can be used for the integrated model as the original models. However, if two different models are to be integrated, suppose a transportation model with a scheduling model, then none of the solvers for the original models might be applicable directly to the integrated model. Instead, some hybrid solver must be generated. These two papers suggest how these hybrids are created and how they communicate. Two particularly important forms of process communication are *direct data exchange* and *generative communication.* The former is a synchronous data exchange; that is, the ordering of events is important. An example is when the outputs of one process are passed as inputs to another solver; the second solver is blocked from operating (it must wait) until the first solver is complete. Generative communication is an asynchronous form of communication; these are events where a component notifies any interested components, called listeners, that some state has changed. The listeners are free to ignore the notification or act on it. The component that provides the notification does not wait for nor expect results from the listeners, and so continues about its work. The authors propose a form of "demon constructs" whose purpose is to monitor the states of various components in order to provide the notifications.

A substantial treatment of synchronous and asynchronous control, as well as resource flows and more general control flows, is provided by Dellarocas [20, 21]. These papers characterize the nature of component interactions in the context of coordination theory, and provide a taxonomy of flow dependencies of which the Dolk and Kottemann assumptions are a subset. This thesis also assumes a subset of the multitude of possibilities described in these papers.

Ramirez, Ching, and St. Louis [86] examine the problem of separating a model from its data (model/data independence) and from its solvers (model/solver independence). In effect, each is entirely isolated from the other. Thus, a mapping is required to attach any data to a model and to invoke any solver on a model-data pair. They define sufficient conditions for model/data independence, which roughly are these. (1) *Value independence:* the value of any variable or constant can change without changing the model representation. (2) *Dimension independence:* the number of variables or similar dimensions can change without changing the model representation. (3) *Data structure independence:* the names, locations, types, and formats of the data sets can change without affecting the model representation. Data structure independence is the most difficult to achieve practically; it requires that the model have neither prior knowledge nor prior prescription of how the data are persistently stored. Instead, only the interface to the model can be concretely specified. A similar set of conditions exists for model/solver independence. Using a home-grown application called the Data and Algebraic Management System (DAMS), the authors show how models, data,

and solvers are linked together through mappings. The mappings are part of a special scripting language within DAMS that specifies how any data set will be mapped onto a variable name in a model, and how that is mapped into an input or output for a solver. While not directly related to model integration, independence and mappings are important ways to facilitate flexible integration techniques. This thesis will make significant use of independence and mappings; it is the core of flexibility in selecting different models and making database access transparent to solvers.

The separation of models, data, and solvers into isolated, independent components is vaguely object-oriented. With this in mind, Tung, Ramirez, and St. Louis [100] in an earlier paper discuss integration within an object-oriented environment, notably an object-oriented database (ODBMS). The ODBMS provides three benefits: it stores models as abstract objects relatively effortlessly, it provides unique object identifiers to help distinguish models from each other, and through manipulation objects it helps semantic identity, i.e., selection. Their system presents a rather simplistic view of model integration, however, through four types of model relationships. One model can be identical to, can be a complete subset of, can have some intersection but not be contained in, or can be disjoint with another model. The integration rules are statically defined based on schema integration as found in Geoffrion [34]. As such, there is no dynamic component to model integration, no notion of solving one model before or after another. Another interesting feature is the discussion of semantic identity and object (model) equality. This is the problem of names: how to determine when two models are or are not referring to the same object (data) when using the same or different names. There are two types of naming problems: *synonyms* (same name, different data) and *homonyms* (different names, same data). The authors claim that one solution would be to enforce a unique name on models and genera, but that this represents too great a burden on models of realistic size. Finally, the authors present a technique for validating model equality by analyzing the type, value, and purpose of the underlying data attached to a model. The methods of Bhargava, Kimbrough, and Krishnan [4] and Bradley and Clemence [10], both discussed below, might be more effective solutions to this problem.

Muhanna [77] presents an object-oriented approach to managing the model life cycle, model representation, and model integration. The author presents a compelling case for the object-oriented approach. The author smartly divides a model into its public interface with the world and its internal behavior and specification, represented as a "glass box" that has a common exterior but can be examined inside as well. By separating these two elements (public interface and internal behavior), the author applies the fundamental object-oriented notions of polymorphism and protocol classes in the context of model management systems. Polymorphism is the property of different objects supporting different functionality through the same method. That is, sending the message "Play" to a sound clip will play the sound; sending the message "Play" to a child will evoke entirely different behavior. A protocol class is simply a type of object that only defines an interface without any implementation. Thus, the protocol class "Playable" might have a public interface method called "Play." By deriving a sound clip class from "Playable," anyone can call "Play" on the sound clip without knowing it is a sound clip as long as it supports the "Playable" protocol. Thus the protocol class is precisely the *model type,* and polymorphism enables the *model versioning* in Muhanna

[77]. The integration capabilities are as limited as those of Tung, Ramirez, and St. Louis [100]: the outputs of one model can be the inputs to another. Again, the richness of the Dolk and Kottemann (Dolk and Kottemann [24], Kottemann and Dolk [57], discussed above) approaches is preferred. The notion of using a formal language such as structured modeling to describe the atomic components and then using object-oriented constructs to model in the large is worthwhile.

1.7.3.1 Naming and typing

Two papers address a special problem of textual modeling languages: the naming of elements. The fundamental issues are: when do two named elements mean the same or different conceptual elements, and when do two named elements mean the same or different instances (data). In the first paper, Bradley and Clemence [10] present a new modeling language that integrates features of earlier languages with a typing scheme that permits type checking. First, each element is assigned a type comprising three elements: a *concept* that represents the element's essence (apples), a *quantity* that is the measurable feature (weight), and a *unit* that scales the quantity (tens of bushels). Second, each concept exists within a user-specified conceptual relationships graph that describes how concepts can be aggregated. So, if the concepts of apple and oranges can be combined into fruit, this is explicitly added to a conceptual graph that precedes the model proper. Similarly, conversions between quantities must be user-specified per problem (weight/volume to density). Conversions among units (feet to inches) are supplied by the system. This system enables type conversions, checking, and equation reductions. The overhead induced by this system appears manageable, although it does add to the list of modeling languages.

In the second paper, Bhargava, Kimbrough, and Krishnan [4] extend the quantity and concept elements of Bradley and Clemence [10] into a single expression called *quiddity*. The quiddity measures the whole essence of a model element, such as the cost of labor in the production of a truck. Quiddities are expressed as logical, functional relationships; they present a calculus imposed upon the quiddity relationships that mimics the traditional calculus used with dimensions. This paper does not introduce a new modeling language, but extensions similar to the first paper might be appropriate. The result is that with well-defined initial models, it is possible to use the quiddities of the elements in each model to see if there are any name collisions and type problems when the two models are integrated.

There are several concerns with each approach. First, much of the underlying conversions of units are provided by the system and not easily extensible. If the system does not provide conversions from liter to gallon, then that conversion might never be possible. Second, generally a single human language for modeling is assumed. The procedure for integrating a model expressed using English and one using Russian is unclear. Again, the system might have to provide mappings. Even within the English language, how is the system to know that "car" and "automobile" are the same, or "IC" and "chip" are? Finally, neither paper addresses the issue of mapping the model elements into model instance data. It might be that two different names with different quiddities refer to the same data set. This might be

an error, and certainly needs to be examined by a user. This can be viewed as an enterprise-wide modeling technique when enterprise data is shared among multiple modelers.

1.8 Epilogue

With the stage set, the play unfolds just so. The second act details the requirements of a framework for implementing reusable solvers. The third presents the framework itself. The fourth describes several solvers and applications that use the framework, to examine the costs and benefits of the framework. The final act describes qualitatively the benefits of, issues concerning, and future research areas for the framework, and concludes the thesis.

This page blank, intentionally left.

CHAPTER TWO

REQUIREMENTS

The first step in developing anything is to state the requirements.
Rumbaugh et al., *Object-Oriented Modeling and Design* [93]

Before setting out on the path of specification, it is helpful to know the destination[4]. Requirements analysis distills the project goal into a specification that expresses the necessary features, capabilities, restrictions, and expectations of the project. Requirements are the expression of the destination, to which analysis and design determine the path.

This chapter begins to lay the foundation for requirements of solvers and solutions for applied problems in operations research. It attempts to capture the most significant requirements of any desirable solver. In a sense, it is a "prolegomena to any future solver[5]," in that the requirements herein extend beyond the framework that follows and beyond the particular technology decisions made or discussed.

Requirement specification begins, in this case, with a *domain analysis,* examining actual operations research solutions and classifying them to gain insight into recurring patterns of implementation techniques, decisions, and possibilities. A categorization of applied solutions will allow the framework to target particularly common or receptive types of problems. Categories that manifest repeatedly in solutions are ripe for inclusion in the framework.

---

[4] Some might argue that the journey itself is the reason and the reward, but it is difficult to convince customers of this.

[5] Borrowed shamelessly from Kant's *Prolegomena to Any Future Metaphysics.*

Domain analysis will show that identifying individual subproblems as separate, modular components simplifies implementations. This modularization leads to a classification of the various participants in an applied solution. These participants include the user or program that acts on behalf of the user, the problem data the user wishes to manipulate, the solvers the user employs for the manipulation, and the various flows of data and control that actually link the user, solvers, and data. This classification will be essential to developing a proper object model framework for solutions.

A given solution might rely on numerous, independent algorithms and solvers interacting to solve a larger problem. Nevertheless, many problems can be solved by a single algorithm. Standardizing these entities and developing requirements for a solver as a stand-alone component is half the journey. Requirements for a solver implementation will detail expectations of a solver, so that it is widely applicable, easily reusable, and effective. For solutions that rely on many solvers, the interconnections between solvers is an important specification. Requirements for a network of solvers describe the primary concerns and needs when two or more solvers work together on a single problem.

Some desirable features are independent of any one implementation. For instance, no one solver should be responsible for converting dimensions. Dimension conversion is orthogonal to the optimization of a linear program; no linear programming code should also embody dimension conversion code. Instead, dimension support should be centralized. Such features are requirements of a core, operating-system level, service. Just as the management of file systems is part of an operating system, management of global activities such as dimension support should effectively be part of the operating system.

2.1 Domain Analysis: The Nature of OR Solutions

Most applied operations research modeling solutions fall within a small set of *solution archetypes*. A sampling of applied solutions presented in the INFORMS journal *Interfaces* provides an adequate representation of the solution archetypes. The Franz Edelman finalist papers, printed in the first issue of each volume of *Interfaces,* are particularly worthwhile. Finalist papers represent "outstanding accomplishments in the practice of our profession" with "verifiable results that had a major impact on a client organization" and a "commitment to practice excellence" (Finkelberg and Graves [27]). Along with two implementation projects in which the author participated, the seventy Edelman papers published in *Interfaces* between 1987 and 1997 form the sample set for the domain analysis. All of these projects are cited in the Bibliography of Applied Solutions (see page 333). Projects are identified by a related word in all capital letters. For instance, the project AAASAS references the American Airlines Arrival Slot Allocation System project.

To the extent that the proposed framework can capture the mechanisms of an archetype, it will be suitable for easing the implementation of solutions within that archetype. Hence, it is important to characterize these archetypes, to identify the requirements and desirable features of the framework.

2.1.1 The components of a solution archetype

There are three relevant dimensions that partition solutions. These are the *solution architecture,* the *execution periodicity,* and the *solution reuse.* Solution architecture characterizes the structure of the solution—how different solvers connect to each other, and how they react to their environment to solve the problem. Execution periodicity captures how often the solution is used—is it run for strategic planning, frequent scheduling, or real-time response? Solution reuse indicates the proven flexibility and adaptability of a system to be applied in different environments—that is, whether the solution is custom and embedded to serve a single role in its existence, or whether it is used in different companies or to solve different problems with little effort.

2.1.1.1 Solution architecture

Solution architecture captures the interaction between modeling components in a solution. The architecture express the flow of data (inputs, outputs, and parameters) and control in a network of processors or subproblems[6]. Simple problems will have solutions with correspondingly simple architectures. Solutions that are more complex might exhibit branching flows of data, parallel paths of execution, and reentrant information flows, and they might broadcast control signals to other systems.

There are four types of solution architectures: single-stage, directed acyclic graph, decision-based directed graph, and real-time directed graph. These are distinguished by the organization of the related subproblems and the behavior of the interconnectivity between them.

What is a subproblem?

An important question is, what is a subproblem? For the domain analysis, a subproblem is a contained, complete, independently consistent problem, which might or might not be embedded in a larger solution implementation. A subproblem represents the atomic level of being and control in the solution architecture. It takes inputs and transforms them into outputs, perhaps according to some parameterization. A client of a subproblem has no control over the internal implementation of a subproblem. An example of a subproblem is a simplex-algorithm linear programming solver. It takes inputs and generates outputs, and can be controlled through a parameterization that includes the maximum number of pivots and the number of pivots at a single basic feasible solution that will be tolerated before degeneracy is declared.

Subproblems are an abstraction of functionality, interface, and implementation, much as classes are in object-oriented design. As shown in Figure 2.1, a collection of interacting

---

[6] The term *subproblem* is used instead of *component* because a component will be a well-defined entity within the modeling framework.

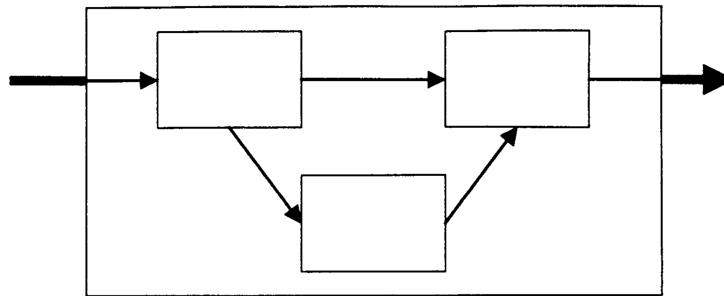


Figure 2.1: Three subproblems hooked together to make a larger subproblem

subproblems can be viewed, if desired, as another subproblem, with its own interface and implementation; the level of abstraction is not bounded. A subproblem could range from calculating the mean waiting time for an M/M/m/k queue or solving a network min-cost flow problem to solving ODEs or optimizing a non-linear program.

A primary goal of this thesis is to standardize the communication, interconnection, and control of subproblems; all algorithm implementations should be expressible and useable as subproblems that can be embedded in larger, customized networks of subproblems and solutions.

2.1.1.1.a Single-stage architecture

This simplest solution architecture consists of a single processing step. An implementation gathers inputs, transforms the inputs, and generates outputs (see Figure 2.2).

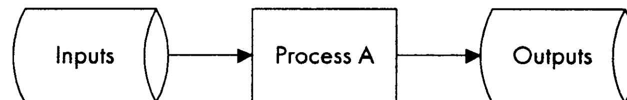


Figure 2.2: Single-stage architecture

The canonical example of a single-stage solution is a generic linear programming solver, such as the LINDO or CPLEX software packages. The expressive power of linear programming or mixed integer programming formulations is sufficient to model many real-world problems, and commercial solvers have significant performance capabilities to optimize or approximately solve these problems to satisfaction. But the single-stage architecture even encompasses simple problems such as "calculate mean queue length" or "solve set of simultaneous equations" to the extent that these problems are final results by themselves.

The processing of a single-stage system can be arbitrarily complex; the feature that distinguishes a single-stage system with a complex processing step from a networked system

with complex interactions of simple processing steps is the amount of control the modeler has over the interconnections within the steps. For example, the CPLEX integer solver is very complex, with branch and bound or cut capabilities that can be enhanced with callbacks (calls from CPLEX into the client code during optimization). Used without modification, the CPLEX integer solver represents a single-stage system, even though within itself it might solve multiple subproblems. It is single stage because the client has control over the internal implementation of the CPLEX solver only through parameterization of the algorithm. However, with the addition of external code and logic in the client, such as routines to generate cuts or solve relaxations, the CPLEX solver becomes a subproblem within a larger, networked system that is no longer single-stage. (The clients of the resulting system might still consider the entire system single-stage if they have only parameterized control over the algorithm, as the modeler does with CPLEX.)

The single-stage system is probably the most prevalent solution architecture. Almost any simple modeling in spreadsheet packages constitutes a single-stage solution. Examples of particular interest include using the built-in spreadsheet solver to solve network flow problems, solving queueing formulas in spreadsheet form, and performing what-if analyses. The nearly universal presence and availability of spreadsheet packages, as well as the preference and familiarity many users have with spreadsheets, mandates that any relevant framework that addresses the single-stage architecture must work within a spreadsheet environment.

For examples of single-stage solutions, see AT&TCLS, TEXACO, and MONSANTO.

2.1.1.1.b Directed acyclic graph architecture

More complex solutions can usually be expressed as a collection of independent, simpler subproblems whose outputs and inputs are chained together. The collection of subproblems and the collection of links from the output of one subproblem to the input of another form the nodes and arcs of a solution network. When all of the arcs in a solution's network represent only the flow of data (as opposed to the flow of control or real-time events, see sections below), and when there are no cycles present as a result of the arcs, then the solution has a *directed acyclic graph architecture* (see Figure 2.3).

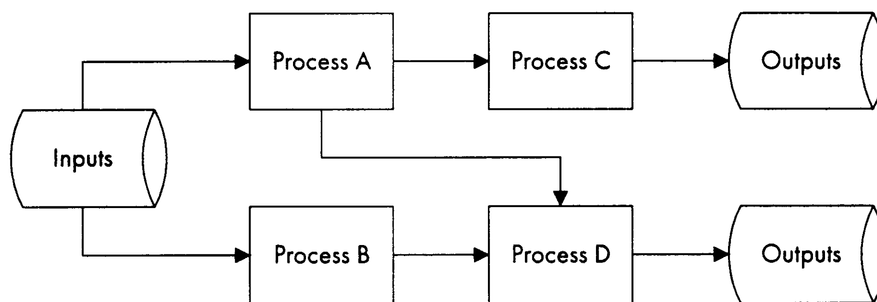


Figure 2.3: Directed acyclic graph architecture

Each step in the network is a subproblem of arbitrary complexity. One could be a simple computation (such as a database query) while another could be a complex optimization algorithm. On their own, the subproblems might be sufficient in some circumstances to work as single-stage architectures; it is their dependencies on other subproblems that engender the network structure.

The boundaries between subproblems are also arbitrary. Different subproblems could be entirely different applications, such as CPLEX and Excel. They could be a single application used in different configurations, such as using CPLEX to solve a transportation problem and then using it again to solve an assignment problem. Different subproblems could also be separate modules or subroutines within a single application, such as a spreadsheet's financial functions, its database capabilities, and its solver.

The restriction that the interconnections between subproblems represent only flows of data and not flows of control or other information is crucial to simplifying the analysis of directed acyclic graphs. When the interconnections are all data flows, the algorithm for the entire solution is trivial: simply execute each subproblem in its order in the topological ordering of the network of subproblems, storing its outputs as necessary for input into subproblems that have yet to be executed. The absence of control flows, which might specify that some subproblems should be skipped or repeated, guarantees that the execution path through the solution is the same every time. This deterministic execution can perhaps be leveraged in simplifying the design and implementation of a framework to address these solution architectures.

The directed acyclic graph is the essential structure of much integrated modeling research; see Geoffrion [35] and Muhanna [77]. The chief advantage of acyclic graphs is that a topological ordering of the network provides a simple sequence of subproblems that ensures that each subproblem is not executed before its required inputs have been determined by previous subproblems. Compared to networks with cycles, acyclic networks are relatively easy to analyze, and it becomes much simpler to add multiprocessing or distributed processing to the solution when the precise dependencies and orderings of the steps is known.

Example. The solution used to restructure Procter & Gamble's supply chain (P&G) provides an excellent example of a directed acyclic graph architecture. The problem P&G examines is product sourcing—choosing the best location for making and distributing each product. Their strategy is to decompose the supply chain model into two phases. In the first phase, they solve a distribution-location problem (a 0-1 integer program) to place distribution centers and assign distribution centers to customers. Because of the heuristic nature of decomposing this problem into two phases, the optimal distribution locations in the first phase might lead to choices in the second phase that yield a result that is not globally optimal. One technique to overcome this problem might be iteration between the two phases (which would create a network with a cycle; see the decision-based directed graph architecture, below). The technique the authors use is to generate many good solutions in the first phase, in order to give the second phase more opportunities to find good solutions. So, in the first phase they also determine a fixed number of near-optimal solutions; a near-

optimal solution is another assignment of distribution centers to customers that performs almost as well as the optimal assignment.

Given the locations of the distribution centers, the second phase solves a separate transportation problem for each product. This transportation problem for a given product assigns flow of that product from a plant either to a distribution center or directly to a customer. For a given product, the number of transportation problems is the number of solutions generated by the first phase. The best result obtained over those problems is used as the result for that product. Each product remains independent of the other products.

The solution architecture for this problem is shown in Figure 2.4. The graph clearly shows that each separate Phase 2 step is independent of the other Phase 2 steps. Hence, each of these steps can be processed on separate processors (or separate computers) once the first phase is complete. In this case, decomposition leads trivially to potentials for multiprocessing.

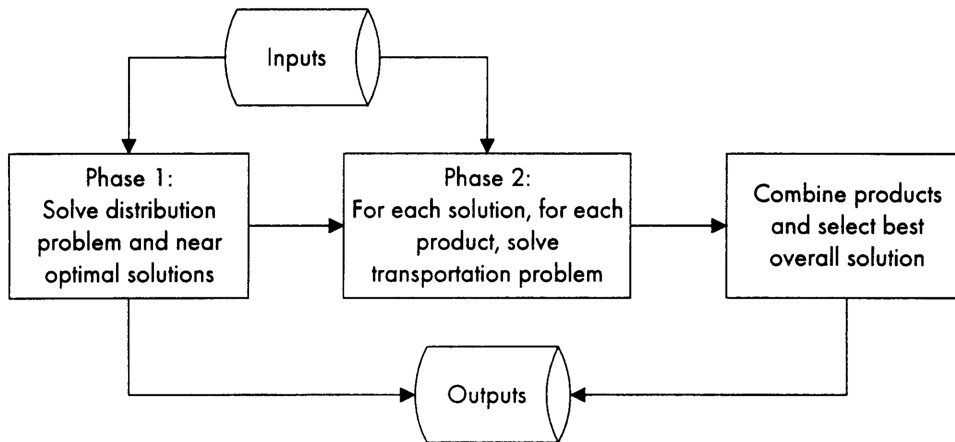


Figure 2.4: Architecture of P&G supply chain model solution

For more examples of solutions with a directed acyclic graph architecture, see EPRI (particularly Figure 3), SPAIN, GM, and SANFRAN (particularly Figure 2).

2.1.1.1.c Decision-based directed graph architecture

Often, a solution will require some iteration and branching decisions at the level of the subproblems. For example, branch and bound algorithms might use any of a number of relaxations as subproblems for the bounding step; there is an intrinsic repetition of branch and bound that eventually ends due to some predefined termination criteria. While it might be possible to model the entire algorithm as single-stage architecture, this sacrifices the opportunity to insert different relaxations into the subproblem mix. Because of its repetition,

hence its cyclical nature, a directed acyclic graph is also insufficient to model this large solution.

The third architecture, the *decision-based directed graph,* addresses this concern by extending the directed acyclic graph architecture in two ways. First, there can be cycles of data flow. Second, there can be subproblems whose output flows represent either-or results, i.e., flows of control and data instead of just flows of data. Essentially, these subproblems are decision nodes in the graph, from which control of the network travels along one of the outgoing arcs (see Figure 2.5).

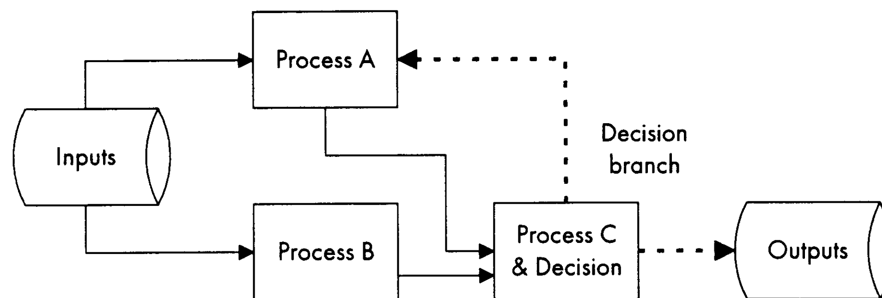


Figure 2.5: Decision-based directed graph architecture

While the primary analytical challenge of directed cycles is verifying convergence and eventual or timely termination, directed cycles of data and control flow introduce a variety of architectural and implementation challenges in general systems. These include control timing and synchronization, and data sharing, availability, and synchronization. For example, in Figure 2.5, the execution of Process A and C must be serialized (control synchronization) so that Process C is never working with invalid outputs from Process A (data synchronization). In single-processor systems (or single-threaded code), these problems are usually resolved by the inherent single process nature of the system (or code). But, in general unrestricted environments more care must be taken with directed cyclic graphs than with directed acyclic graphs to ensure the integrity of the data.

Focusing on models as communicating processes, Kottemann and Dolk [57] address directly the problems of data availability and synchronization. They propose a global workflow manager (a message router) that controls the various subproblems in an integrated model. Each subproblem must be designed to receive appropriate control messages and generate appropriate control events. They also propose a class of daemons[7] (demons) that activate when conditions about data or execution flow are met. In theory, the daemons enable completely indeterminate, real-time processing (see the next architecture), although they do not discuss implementation problems specific to real-time reactions.

[7] "A program that is not invoked explicitly, but lies dormant waiting for some condition(s) to occur. The idea is that the perpetrator of the condition need not be aware that a daemon is lurking [88]."

Example. An optimization tool used by SANTOS, Ltd., a publicly owned mineral exploration and production company in Australia (SANTOS), provides an example of a decision-based directed graph. SANTOS developed an application called SIPS to assist in the scheduling of oil well and refinery operations for a number of companies working in central Australia. SIPS determines investment (capacity planning) decisions and reservoir production schedules to maximize net present value of the system over a twenty-five year planning horizon. The tool implements the Wolfe-Dantzig generalized programming method, which breaks the program into a master problem and a series of subproblems. The system alternates between solving the master problem and then solving the series of subproblems, terminating when an optimal solution for the master problem is found. Hence, each time the master problem is solved, the system decides whether to continue or terminate. If the system elects to continue, it cycles back through the subproblems and into the master problem again as shown in Figure 2.6.

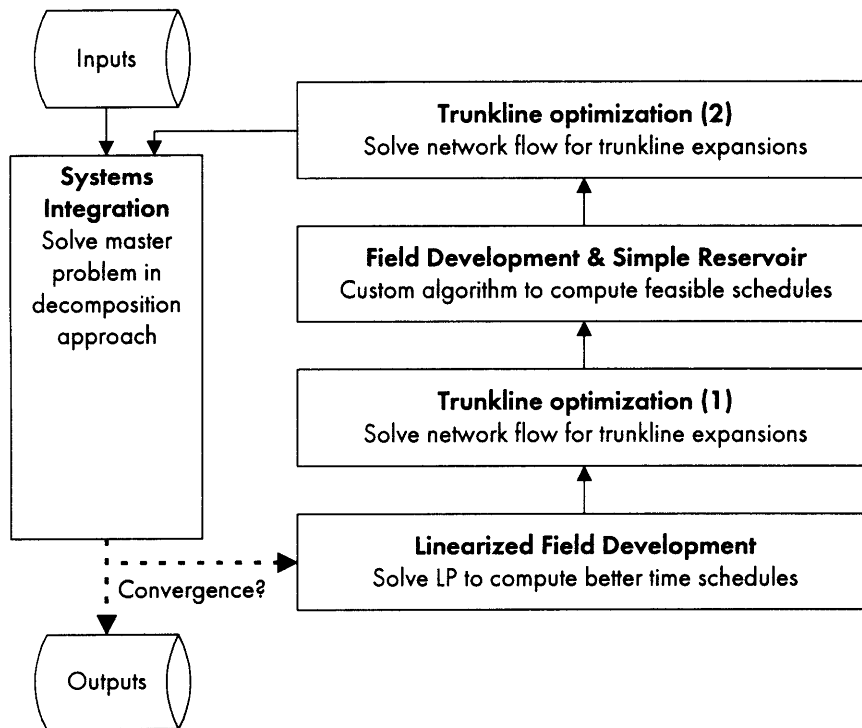


Figure 2.6: Architecture of SANTOS's SIPS planning solution

For other examples of decision-based directed graph architectures, see IBMOPT, AAASAS, GTE, and SADIA.

2.1.1.1.d Real-time directed graph architecture

The architectures explored so far are, essentially, batch processes. Control starts at the Inputs and proceeds, with possibly different flow paths depending on the inputs and decisions, until data flow onto the Outputs. These architectures cannot respond to real-time changes to data[8]. Hence, they are not always suitable for online optimization and operational planning problems that require the algorithm to adapt to changes in live data. For example, a system that interactively schedules a job shop in real time might tie into a shop floor control package that provides instantaneous updates of piece-flow through the shop. When a piece leaves one workstation and is placed in an inventory bin, the shop floor control system signals an event to the optimization engine, which can then adapt its view of the world to account for this piece-flow.

The fourth solution architecture, the *real-time directed graph,* extends the decision-based directed graph architecture with the addition of real-time, exogenous signals (see Figure 2.7). An example of such a signal might be a call to 911 for emergency assistance; the 911 operators must dispatch and possibly re-route medical and police teams to various sites based on these random calls.

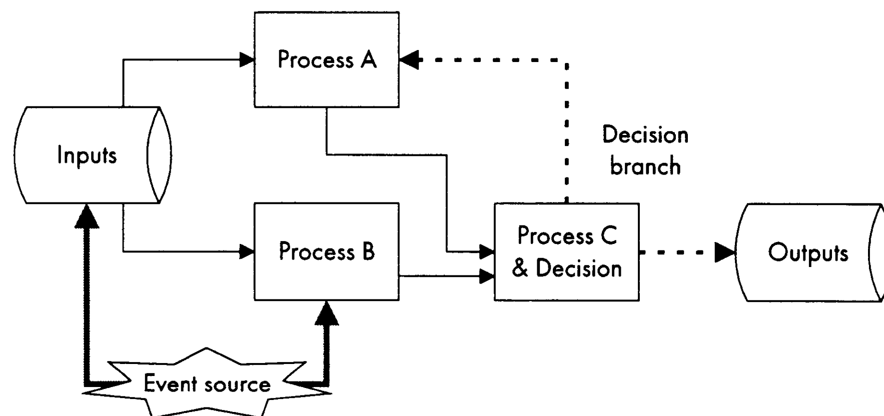


Figure 2.7: Real time directed graph architecture

Subproblems within these architectures can be designed to respond to these events, allowing them to leverage the additional information provided by a real-time system. These subproblems can also be used in architectures without real-time events; responding to events is an additional feature but not a requirement.

Real-time architectures require the most careful design and forethought towards implementation. The presence of exogenous, possibly random and varying events raises issues of synchronization, queueing, and reliability that are more complex than in the

[8] They can be designed to respond to simple interactive control events such as "pause" or "cancel."

previous architectures. In architectures without these events, a global entity acting as a controller can manage the entire system with complete certainty; in a sense, the space of activities is known deterministically. This is not possible in a real-time system.

Typically, real-time architectures are useful only in circumstances where decisions must also be made in real time. For a strategic planning system, there is no need to respond to real-time events; instead, these events are aggregated or queued in a database prior to running the system. This aggregation removes any system dependencies on real-time events.

Example. Part of the traffic control system developed for the Hanshin Expressway in Japan exhibits real-time structures (HANSHIN). With the goal of maximizing the total traffic flowing into an expressway network, the system limits the flow of cars onto the expressway at each entrance ramp to relieve congestion. The system comprises two phases. In the first phase, representing normal operations, a linear program is solved every five minutes with the latest traffic data to determine the flow-rate of traffic onto the expressway. When traffic volumes and fluctuations remain within a control range, the LP solution is used to control traffic. When volume or fluctuations are out of range, a special subsystem overrides the LP solution and actually closes entrance ramps using predetermined rules; these rules are computed off-line with analyses and simulations. In the second phase, representing extreme operating conditions such as accidents, another subsystem will actually force vehicles on the expressway to exit, relieving congestion around the trouble spots. As is typical of such real-time systems, the actual "operation research" component of this system is small relative to the database and information technology infrastructure.

Examples of other real-time architectures include AAYIELD, IBMLMS, and HARRIS.

2.1.1.2 Execution periodicity

Besides the architecture of a solution, another component is its *execution periodicity*. Execution periodicity captures how frequently the solution executes; or, to put it another way, it specifies at what level of the decision process for a company the solution participates. Nahmias [79] calls this the "time horizon" and partitions the horizon into three ranges: long-term or strategic, medium-range or tactical, and short-term or operational. Solutions are classified into three similar groups, as strategic, tactical, or operational solutions. Some implementations might be used at multiple levels, perhaps in a different mode or configuration.

2.1.1.2.a Strategic solutions

Strategic solutions are systems that solve long-term or long-horizon problems. Examples include capital asset allocation, facility location, and network design. A solution might be executed many times for a single problem (as in what-if scenario analyses), so even though the problem is long-term, the solution might still need to exhibit responsive run times.

Solutions of this nature also encompass problems that only need to be solved once; these are *one-shot solutions.* An excellent example is resolution of the web of debt following the crash of Kuwait's al-Manakh stock market in August of 1982 (KUWAIT). This crash resulted in $94 billion (U.S.) of debt between traders; most traders owed and were owed money. The "entanglement" of debt among the traders was so severe that courts could not resolve the problem case by case. A team of researches applied LP models to determine the optimal pay-off amounts to maximize the total payback of debt. Barring any future crashes, this problem is clearly solved only once, although the LP itself was solved many times during validation and sensitivity analysis.

There are typically two approaches to implementing these solutions. One follows the philosophy "because we will only use the solution one time, or rarely, let's use off-the-shelf algorithms, and concentrate on data collection and prioritizing the goals and constraints instead of run time." With this philosophy, the implementation team might model their strategic problem in a spreadsheet package or use commercially available, standard optimization packages. The option of investing significant modeling resources to design a customized algorithm for the problem might not be cost-effective if the solution is used only once a year. The MONSANTO and SANDF projects, which used a standard LP solver and spreadsheet package, are examples of this approach. This approach also encompasses most strategic analyses that management consultants perform: spreadsheet-based models that leverage standard database protocols.

The other approach is to develop a highly-customized solution, under the belief that "this solution addresses such crucial strategic issues and will be employed and scrutinized by high-level managers that a custom solution is warranted." Furthermore, this custom solution, if designed appropriately, might then be sold to other companies or departments within a company. In the NYNEX project, for instance, custom algorithms were created to solve all but one of the many subproblems.

For both of these approaches, a capability to reuse existing solvers easily can greatly reduce development time of a strategic solution. For the off-the-shelf technique, reusable solvers are another item to add to the toolkit. Rather than model an interesting problem using a spreadsheet's monolithic solver, a preexisting solver for that problem could be used with little additional effort. For the custom-solution technique, an existing solver might be embedded within a larger framework.

Strategic solution implementations typically have fairly loose requirements on their components, relative to other solutions. Usually, components do not need to respond to real-time events. They might not need to have a bounded run time, and thus can run to full completion. However, given that the developers of the solution might have limited resources, the reusability must be greater; developers will not have time to learn arcane command-line syntax or file formats or APIs.

2.1.1.2.b Tactical solutions

Tactical solutions solve mid-term problems, at the planning or scheduling levels. Examples include monthly distribution schedules, energy planning, fleet scheduling, and maintenance planning. These solutions are usually run on a fixed time schedule (such as once a month), but this is not a strict requirement.

Most tactical solutions fall into one of two categories. The first includes *production-environment schedulers*. These systems run on a regular basis to generate schedules or assist a regular decision-making process. They normally have automated data inputs and outputs and intimate knowledge of their environment. Solutions generated by these systems might be directly implementable. That is, the data outputs might directly feed shop-floor control systems, for instance.

The second category includes *informal schedulers*. These systems are used primarily for what-if analyses to indirectly assist a decision-making process. Data input might be manual (such as entering values into a spreadsheet), and solutions generated by these systems must be reformulated into plans. They might provide approximate solutions, or solutions to relaxed problems that do not address all of the constraints in the environment. Consequently, the (human) scheduler uses these systems for guidance, not for actual schedule generation.

Production-environment schedulers rely on significant information accessibility, normally through a corporate information technology infrastructure. These systems have direct access to company databases, via batch reports or dynamic queries. The operations research component can be a large part of the solution, consuming most of the processing resources (CPU time) and development time. Texaco's OMEGA blending software (TEXACO) is an example of production-environment scheduling. Implemented at all of their domestic locations, OMEGA ties into the plant databases to provide substantial decision support. OMEGA permits extensive what-if analyses in generating a schedule, which can then be turned over to the operators.

Informal schedulers have less restrictive data requirements. Users might enter data by hand, or manually query the company's databases. They might save outputs directly in a spreadsheet, rather than uploading them back into large databases. The typical outputs of these systems are monthly reports, tables, and charts. The operations research contribution is the most important piece of these systems.

Production-environment schedulers are often complex software applications, developed by a company's IT group. Reusable components can reduce development times of these applications, but the modelers need to acquire sufficient input in the development of these systems. Informal schedulers can also be complex software applications, but usually are spreadsheet models, often created by the modeler or planner. Reusable solvers enable the modeler to enhance the planning, reduce development times, and use customized solvers instead of monolithic solvers.

2.1.1.2.c Operational solutions

Operational solutions solve short-term and real-time problems. These systems might run overnight, instantaneously, or even continuously. They manage the day-to-day and moment-to-moment operations of automated activities, such as yield management, job shop scheduling, traffic control, and emergency dispatch.

There are three types of operational solutions:

- *Batch solutions* are overnight batch jobs or daily planning cycles that schedule the next day (or time period) or analyze the previous one. The KODAK scheduling solution is an example of a planning cycle that occurs three to six times a week.

- *Responsive solutions* run many times during the day at the request of operators; these include yield management systems that must respond quickly to a request for resources. AAYIELD and NATIONAL, both reservation systems, are responsive solutions.

- *Live solutions* run continuously and respond to events in the environment; these include emergency dispatching system, control of traffic systems. The HANSHIN expressway control solution is live; it monitors the conditions of an expressway and can react quickly to detrimental effects.

Operational solutions are typically highly automated. They can even be operator-independent, especially batch or live solutions. These systems must be intimately tied into corporate data systems and have fast turn-around times on requests for information. Speed is critical in responsive and live solutions. Hence, they must be designed with greater care than a spreadsheet model used for less frequent tactical solutions.

From an implementation standpoint, the operations research contribution is a smaller proportion of these systems. The optimization provided by operations research methods might be the driving force behind a solution, but often the challenges of providing real-time data access, suitable user interfaces, and sufficiently powerful, concurrent systems overwhelm the development effort of the operations research components.

It is much more difficult to create good, reusable components for this class of problems; the real-time requirements and run-time requirements are much more stringent than for other solutions. These systems typically have custom algorithms and decision support code. Therefore, the value of non-monolithic, reusable solvers in this area might be limited.

2.1.1.2.d Conclusion

A summary of the types of execution periodicity appears in Figure 2.8. Each box in the figure is a category of solutions, grouped by execution periodicity and sub-classification or sub-functionality within that periodicity. Links in the figure represent "is-a" relationships (generalizations), where the category at the tail of the link "is a" type of the category at the

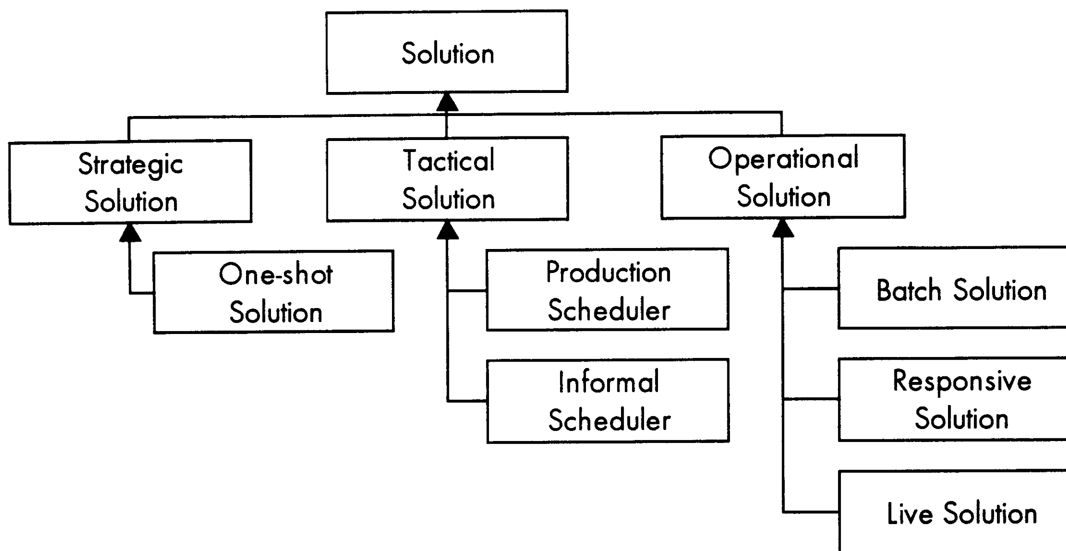


Figure 2.8: Hierarchy of execution periodicity types

head of the link. So, a batch solution is an operational solution, is a solution. These types are not mutually exclusive; some solutions are so complex as to span several categories. Furthermore, some solutions might exist outside this classification.

It appears that reusable components are best targeted towards strategic and tactical solutions, especially those that use spreadsheet packages. Reusable components are suited for such interactive environments. They can enable modelers to solve problems with special algorithms rather than the monolithic spreadsheet solvers. And they can reduce development time by alleviating the need for the modeler to map a specific problem into a generic solver.

Reusable solvers also have a place in custom-built solutions, where they can reduce development time by providing an encapsulated, standardized method to solve small and known subproblems.

Currently, real-time markets are not so appropriate for focus, given (a) the small part operations research models often play in real-time systems, and (b) the need for highly specified, custom solutions to satisfy the special needs of real-time solutions. Hence, the returns on effort in this area will be more difficult to realize.

2.1.1.3 Solution reuse

Solution reuse expresses whether a particular solution is implemented once or multiple times. As explored below, there are different implications for the design of an implementation depending on whether the entire solution will be reused.

2.1.1.3.a Single use solutions

Many solutions are designed and implemented to solve a single problem. These problems can range from strategic one-shot scenarios to real-time data-intensive scheduling systems. They can be simple models that are used once and then discarded, or highly complex, embedded systems that run a factory.

There are several reasons solutions are limited to a single use:

- The problem is often complex or unusual enough to warrant a custom solution that cannot be applied somewhere else. The fix of the Kuwaiti stock market crash (KUWAIT) is an example of an unusual problem. The search for the *SS Central America* (METRON) is another[9].

- Data requirements might be tremendously complex, making it difficult to extract the solution from its environment. The Hanshin expressway (HANSHIN) and fleet assignment at Delta (DELTA) are two examples of problems with massive or highly-specialized data requirements.

- Proprietary technologies might make it impossible for the solution to be used again. National Car Rental (NATIONAL) developed a yield management system that could theoretically, be applied at other rental agencies, but it gives National a competitive advantage over companies that lack a sophisticated yield management system.

- The system might have been designed without the proper measure to enable more than one use, so that it cannot be used again.

Single use solutions can be somewhat lax in their data requirements. Customized "hacks" can be applied if necessary to save time and money in development. Of course, removing the hack at a future time in order to use the solution to solve another problem might destroy any initial gains—hacks rarely are a better alternative to good design from the beginning.

2.1.1.3.b Multiple use solutions

Other solutions are designed and implemented to solve many problems. These often yield generic software applications that are easily transportable. Multiple use solutions are typically created for strategic and scheduling solutions; because of the complex requirements of operational solutions, deploying one to solve multiple problems is difficult.

[9] This solution was presented in the Edelman paper as a technique that could be applied in the future, although no other solutions were given; the technique was later turned into a generic software solution at Metron.

There are several justifications for creating a multiple use solution:

- The problem might recur frequently enough within a company to warrant the extra development effort to create a standardized package. This is true of General Motor's PLANETS package (PLANETS), Texaco's OMEGA product (TEXACO), and the SONET toolkit (SONET).

- The company, especially if it is a consulting firm, might be able to sell the package to other companies in the same market. PONTIS and EPRI are examples of software created for specific markets.

- The per-use cost of a standardized solution is dramatically less after the first use. Therefore, the deployment costs are decreased; this provides an incentive—or at least, removes a barrier—to try the solution in other areas.

Multiple use solutions must be designed with more care than single use solutions. They must be more flexible, accept a wider variety of data input, and have more robust user interfaces.

2.1.2 Categorization of sample solutions

Table 2.1 on page 68 categorizes sixty-three of the seventy Edelman papers for the eleven years from 1987 through 1997. The other seven papers (ABB, EGYPT, GRI, MARRIOTT, NEWHAVEN, NHFIRE, and SAINSBURYS) either described applications that did not have significant computational components or summarized the impact of management science and operations research efforts at a company without providing sufficient details for classification. The matrix also includes two non-Edelman projects of personal interest to the author, namely MONSANTO and SIPMODEL, and to which the author contributed significantly.

Down the side of the matrix are the solution architecture types: single-stage, directed acyclic graph, decision-based directed graph, and real-time directed graph. Across the top of the matrix are the execution periodicity types: strategic, tactical, and operational solutions. At the juncture of an architecture and an execution periodicity are two cells. The upper cell lists the solutions that are single use; the lower cell lists those that are multiple use. The solution reuse was determined solely from information provided in the Edelman paper itself.

These categorizations are open to interpretation. In some cases, articles presented insufficient detail to ascertain the precise nature of an implementation. In other cases, a solution might be used at different planning levels; the essential contribution or implementation from a paper is what has been classified. Some systems, such as IBMLMS and SONET, are so comprehensive that they span several execution periodicity categories.

Single use Multiple use	Strategic solution	Tactical solution	Operational solution
Single-stage	ENGLAND (one-shot) KUWAIT (one-shot) METRON (one-shot) MONSANTO SANDF (one-shot) VILPAC	DRG HOMART TATA TINKERAFB YASUDA	
	AT&T BELLCORE DIGITAL PLANETS SYNTEX USARMY USPOSTAL	AT&TCLS BELLCOREPDSS GECAPITAL NYC SO TEXACO	
Static, directed acyclic graph	NYNEX	CAROLINA DELTA HASTUS LLBEAN LTVSTEEL NATIONAL SANFRAN SHUTTLE SPAIN	KEYCORP KODAK LTVSTEEL MOBIL REYNOLDS
	AT&TCAPS BETHLEHEM CHINA EPRI GM P&G SIPMODEL	EPRI PONTIS	NAVANLINES PRUDENTIAL
Decision-based directed graph	CITGO SADIA	CITGO DESERTSTORM MOSLS SADIA	AAASAS CITGO SADIA
	GTE IBMLMS SONET YELLOW	IBMLMS IBMOPT SANTOS SONET YELLOW	AACREW
Real-time directed graph			AAYIELD HANSHIN HARRIS ISRAEL
			IBMLMS

Table 2.1: Categorization of Edelman paper solutions

2.1.3 Discussion

The population of Table 2.1 exhibits a diagonal tendency; solutions are generally grouped from the top-left down and across to the bottom-right. This matches a reasonable mental model of applied solutions. As a solution's execution periodicity decreases (that is, its frequency of execution increases), solutions should become more complex, architecturally. The complications of working with real-time or non-aggregated data demand more sophisticated environments. Similarly, as the execution periodicity increases towards strategic solutions, more complex architectures become less suitable. There is no need to implement a system that supports real-time interruptions when the time horizon of the solution is on the order of months or years.

Forty-nine of the sixty-five categorized projects fall entirely within the three non-real-time architectures and the two longer-term execution periodicities. Of the remaining sixteen, parts of four are within the described region. Therefore, the region encompassing everything but real-time architectures and operational solutions—half of the table—accounts for more than three-quarters of the solutions. Of the fifty-three projects that have total or partial solutions in this region, forty-three have single-stage or static directed acyclic graph architectures. Thus, these two architectures and non-operational execution periodicities account for two-thirds of the solutions.

As an aside, it is important to note that Edelman papers typically represent significant efforts of development, applied operations research, and implementation. The more common, mundane tasks that this thesis targets are presumably simpler in their architectures, solution reuse, and periodicity. Hence, it is safe to say that the percentages identified above are likely lower bounds of the true proportions for the applied solutions this thesis targets.

The relationship between classification and implementation. How, then, does a solution's classification in the table affect implementation complexity, decisions, and contributions? Generally, more architecturally complex solutions require more complex implementations. Interestingly, as solutions become more complex, the implementation requirements of the operations research components form a smaller proportion of the total project. As systems must manage larger data sets, react to real-time events, or have sophisticated user interfaces, the operations research solvers are a smaller, albeit crucial, piece of the whole.

Given the high proportion of solutions in the upper-left of Table 2.1 and the larger role of operations research components in those solutions, this thesis will target specifically solutions with single-stage, static directed acyclic graph, and decision-based directed graph architectures with strategic or tactical execution periodicities. This will generate a maximum benefit for a minimal effort. This does not diminish the importance of real-time architectures or real-time execution periodicities. Instead, it indicates that this thesis represents a first step rather than the entire journey towards a solution that encompasses all solution archetypes.

2.2 Participants in the Solution Process

This section categorizes the various entities that participate in a solution implementation. This is a rough cut first step of object-oriented analysis, building on the results of the domain analysis. Despite the infinite variety and complexity of the space of all solutions, there are a relatively few broad groups of participants. These six that follow are the primary elements of most solutions.

Actors. The primary agents of change, actors are active objects[10]. The users and any programs acting on behalf of the user are actors. These objects change and set data, initiate or terminate execution, and acquire results. Activities generated by actors are randomly occurring; for example, the user could stop a solution at any time. Actors can exhibit spontaneous changes in behavior, without being operated on by another object. Programs that drive the solution process, either through macros or graphical displays, and whether or not the user is present, are actors from the viewpoint of the solution. That is, the solvers are indifferent to whether the activities were generated by the user or a program.

Problem data. Problem data are the specific data that form the inputs and outputs of a given instance of solution execution. These are the tables, matrices, vectors, sets, dimensions, costs, flows, etc., that are passed around among actors, data stores, and solvers (see the next two items). Problem data travel between entities on data flows (see below).

Stores. Stores are the persistent repositories of problem knowledge. There are two types of stores: data and model. Data stores manage collections of problem data. A single data store might contain the inputs and results of many solution executions, hence many sets of problem data. During the course of execution, a data store serves as an intermediate buffer to separate two entities. If two solvers are on different computers, run at different times of day, or understand different formats, for instance, then an intermediate data store removes any dependencies between the solvers. The upstream solver inserts its results into a data store, and the downstream solver queries its inputs out of the data store.

A model store is a data store whose data are model knowledge. For example, the expression of a math program, whether in AMPL or the Structured Modeling Language, is itself data, and is persisted in a model store. A familiar example of model stores is the spreadsheet. The location of cells and placement of functions and their references form a model, and the spreadsheet cell contents are the model store of that model.

Solvers. Solvers transform problem data. Any component that manipulates or transforms data is conceptually a solver. These are the primary focus of this thesis; data stores, model stores, and flows are well-researched and understood, but their interaction with solvers is less well defined. Examples of solvers include the traditional massive commercial systems CPLEX and OSL, smaller components such as a bin-packing engine or inventory calculator,

---

[10] According to Booch [7, p. 91], an active object is autonomous, encompassing its own thread of control and serving as the root of control.

and even domain-unrelated applications such as photo editors, search engines, and task schedulers.

Data flows. Data flows are resource flows that transport data. Resource flows are essentially the transfer of some shared resource between two objects. The resource can be anything used by the objects. In the case of data flows, the resource is typically a portion of the problem data or a piece of the model for a given solution execution. Resource flows in the framework will be explained and explored in the next chapter.

Control flows. Controls flows are another type of resource flow prevalent in applied solutions. Where data flows transport problem data or models, control flows transfer control of the system to another object. Calling a subroutine is an example of a control flow. Because of the variety of different environments, operating systems, and programming languages employed by solvers, the mechanisms for control flow are an important part of any solution.

A primary challenge of the framework is to standardize the interaction protocols among these participants. Current data store standards (SQL and ODBC) are sufficient for the framework's needs. Resource flows encompass a wide range of interactions, but the framework only needs to draw from a select few subsets of flow research. With the solver, much remains to be done.

Example: Two-stage directed acyclic graph solution

Consider the problem of loading rectangular pieces into flat processing beds, where each load of pieces requires a certain processing time, and the total available time is limited to a day. One possible approximation algorithm is to first develop a candidate list of possible loads from all of the available inventory, and then to select the best twenty-four hours of loads.

This is a simple, two-stage directed acyclic graph architecture (see Figure 2.9).

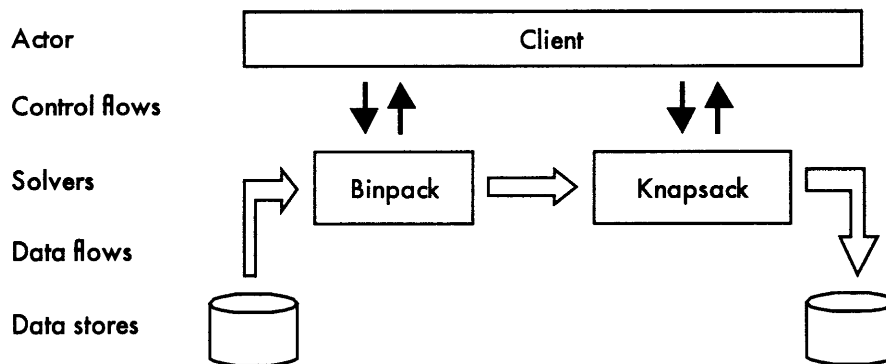


Figure 2.9: Sample two-stage solution demonstrating participants

1. First, a two-dimensional bin-packing is performed over a collection of pieces, generating a candidate list of possible loads. Pieces are partitioned by their thickness, so that pieces of different thickness cannot be mixed in the same load.

2. Second, from the collection of all available loads, a knapsack algorithm selects the best twenty-four hours of loads, where the value of a load is determined by the utilization of the bed space for that load, and its processing time is proportional to the thickness of pieces in that load.

The participants for this solution, as implemented, were:

- *Actors:* Microsoft Excel and a custom C++ client application.

- *Data stores:* A single Microsoft Access database with different tables.

- *Solvers:* Custom C++ bin-packing algorithms and a custom COM knapsack optimization component. The bin-packing algorithms were driven from the C++ client application, whereas the knapsack algorithm was driven by macros in Excel.

- *Data flows:* SQL query in C++ to retrieve data for bin-packing algorithms, and Microsoft Query for data needs in Excel.

- *Control flows:* Generic function calls in C++ for the bin-packing algorithms, and Visual Basic for Applications and COM in Excel for the knapsack algorithm.

2.3 Requirements of a Solver Implementation

This section describes the requirements and desirable features for any particular solver implementation. These are presented from the perspective of the solver as a stand-alone entity, applicable for single-stage solution architectures. Later sections present requirements for solvers that can be deployed in graph architectures.

The requirements of a solver implementation are as follows:

- *Executable.* The solver is an executable application; that is, it is object code that can be directly run by the user.

- *Invoking from different applications and environments.* The solver can be invoked from different applications and environments, such as programming languages and business productivity applications like spreadsheets.

- *Documentation and introspection.* The solver provides dynamic documentation and the ability to query its capabilities and intentions programmatically.

- *Progress updates.* The solver can update its client on its progress during execution.
- *Life cycle control.* The client can terminate or pause the solver gracefully during execution.
- *Dimension and type support.* The solver supports dimension and type manipulations.
- *Testing and validation.* The solver can easily be embedded within a testing and validation framework.
- *Computer-based training.* The solver is easy to use as an educational tool.

These are each discussed in detail in the following sections.

2.3.1 Executable

A solver or algorithm can exist in many forms. At one extreme, a recipe is an algorithm that provides no implementation or assistance. Instructions for combining hot water, crushed leaves, and oil of bergamot are an algorithm for creating a steaming flavored beverage. In operations research, a recipe of instructions might specify an algorithm for solving a transportation problem using the simplex method, including where and how to pivot. These "solvers" require extensive interaction with the client, be it a thirsty person or a quantitative analyst. The client must understand the algorithm and make it so.

At the other extreme, an automated, futuristic machine called a replicator exposes an algorithm for transforming molecular matter into any form. This machine requires no interaction with the client other than the specification of the object to create, such as "Tea, Earl Grey, hot." Literally more down to Earth, a software application might expose an algorithm for solving the transportation problem with only minimal input from the client, who must specify the suppliers, customers, demands, supplies, links, and the various costs. These "solvers" are self-contained, automated devices that embed algorithms and expose their functionality without requiring their clients to know how they work, or, just as important, how they were built.

In fact, often this latter issue, knowledge of how a solver is constructed, deters users of many solver implementations. Why should a quantitative analyst or applied operations researcher understand the command-line parameters and syntax of the MAKE utility and the Gnu C++ compiler? Yet far too often, they are asked to do just that in order to use an algorithm. The algorithm is provided as source code that builds reliably on a single system: the author's. For any other system, the user will likely have to recompile, possibly adjust the build settings, specify libraries for the target machine and operating system, and even rewrite portions of the code that depend on platform-specific functionality.

Understanding the development environment of the solver should not be a prerequisite of using that solver. Hence, the developer of the solver should provide the solver in a form that does not require this understanding.

For the majority of computer systems and operating systems, this means that a solver should be an *executable file*. An executable file is a file created from one or more source files and translated into machine code that can be run ("executed") by the operating system. The nature, structure, and format of an executable file (also called just an *executable*) depend on the target machine and operating system. For instance, for the Microsoft Windows family of operating systems, executables are either applications (EXEs) or dynamic-link libraries (DLLs). For a Java Virtual Machine, any class file is an executable[11].

The chief problem with executables is, of course, machine and operating system dependence. An application created for Intel processors with 32-bit Windows might not run on 16-bit Windows, will not run on a MIPS processor running Windows, and will not run on a PowerPC processor running MacOS (precluding emulation). There are two ways to tackle this problem. The first is to release multiple versions of the executable; this is prevalent with UNIX applications, where every version of every vendor's UNIX seems to require its own executable to run successfully[12]. The second is to minimize the operating system and processor differences by using a virtual machine such as that provided by Java.

In any case, the goal is to place the burden of implementation on the developer of the solver (a single entity) rather than on the users of the solvers (hopefully, many entities). Providing an executable instead of—or, better yet, along with—a bunch of source code can greatly reduce the burden on the users.

Customization and parameterization. An interesting side effect of this requirement is the change in how a user customizes or parameterizes an algorithm. Customizations might include changing control logic or the underlying data types used for calculations; parameterizations might include setting the maximum number of iterations or convergence tolerances. With the source code, a user has complete control over the entire algorithm, assuming he or she can comprehend both the algorithm logic and the expression of that logic in a programming language[13]. Thus, theoretically any customization or parameterization is possible with enough tweaking and fiddling.

With only an executable, the user is limited to those customizations and parameterizations exposed by the executable through initialization files, command-line options, and user-interface interactions. The advantage of this approach is that the user does not have to understand the programming language or even the algorithm logic entirely to parameterize an algorithm. The downside is that customization might be difficult if not impossible. For

---

[11] With Microsoft's Java Virtual Machine, any Java class file is also a Windows executable, because that JVM exposes Java classes as COM objects.

[12] This is the author's experience with the various UNIX workstations on MIT's Athena systems.

[13] User-friendly solvers might provide all parameterization constants in a separate source file. This would make it possible to change a parameter in one place and have it reflected throughout the algorithm, thus mitigating the need to understand the algorithm logic just to parameterize it.

instance, if the implementation uses 32-bit integers and the user wants to use 64-bit integers, there might be no recourse except to use a different solver. If the user had the source code, at least there is a chance she could rebuild the executable using the preferred integer type.

For the majority of users, the trade-offs in having only the executable versus only the source code give the advantage to having only the executable. These users are the untapped market of quantitative analysts, students, and modelers who do not know the software development process but who do want to use more sophisticated algorithms than they currently use.

When source code is important. In some circumstances, it is useful or even necessary to have access to the source code of a solver. When the client of a solver is written in the same programming language or environment as the solver, the solver could be integrated directly in the final application. If there are strict, real-time size or run-time requirements, then it might be necessary to be able to tweak the source code to satisfy these special constraints.

Some algorithms can be written as parameterized classes in specific programming languages (such as templates in C++). These classes greatly increase the reusability of the algorithm by making it easy to parameterize the solver at compile-time instead of run-time. Again, for someone who spends their days in a single programming environment, having the source code might be preferable to having the executable code.

Developers of solvers must weigh the benefits to their clients against the drawbacks for themselves of disclosing all of their implementation details. When an algorithm's logic is proprietary, it might not be possible to release the source code. When it provides competitive advantage, it might not be desirable to do so. Also, disclosing implementation details reduces the encapsulation of the solver. If clients develop their solutions based on known but undocumented implementation details that are not part of the public interface to the solver, then the author of the solver might have difficulty changing that undocumented behavior[14].

2.3.2 Invoking from different applications and environments

An improvement upon making the solver an executable application is to make the solver available to a variety of different applications and solution environments. Just as source code can lock a user into a particular programming language, implementation decisions can lock a user into a particular solution environment. For example, several useful tools in Microsoft Excel exist as Excel Add-ins. As such, they can only be accessed from Excel itself. Perhaps for Microsoft this type of vendor lock-in is good, but it is not beneficial for the user[15].

---

[14] Many parts of Microsoft Windows 95 exhibit this behavior. Microsoft specifically had to code behavior that might not be particularly desirable, but is at least consistent with older versions of Windows applications. For example, see Pietrek [83] for a discussion of the Win16Mutex.

[15] In fairness to Microsoft, in Office 95 and Office 97 they have developed applications that expose most of their functionality to other applications. Developers can now customize Office within their

The solver developer is typically interested in reaching as wide a market of potential users as possible. For small- to medium-sized solutions, several development environments are prevalent, and targeting these few environments captures a large segment of the potential market. Many solutions are developed in high-level programming languages such as C++, Ada, and increasingly, Java. Primarily visual tools, like Microsoft Visual Basic, Borland Delphi, and Sybase PowerBuilder, are also common, especially in business-oriented departments where analytical skills dominate programming skills. Even common business applications such as Excel, Microsoft Access, and Lotus 1-2-3 are host to numerous quantitative solutions.

How is the solver developer to target all of these environments?

Fortunately, as operating systems evolve into the next millenium, they are acquiring more competent component services. Most applications have transformed from using their own proprietary interconnection schemes to accepting the *de facto* component capabilities of the underlying operating system. Regardless of their dogmatism in the COM/CORBA holy wars, most companies developing for Windows embed support for COM in their applications, because COM is the underlying component technology in Windows and is supported by more applications. Pro-CORBA shops often support both, which is perhaps the best solution going into the future.

All of the environments and applications named above support COM (Java through Microsoft's Java Virtual Machine only). Given the prevalence of Windows systems in the small- to medium-sized solutions targeted by this thesis, COM is an ideal framework to support and build upon for the solver framework. Solvers built on COM are almost immediately available to Excel, Visual Basic, and Visual Studio, three of the most common development packages on the market.

2.3.3 Documentation and introspection

Documentation, especially printed documentation, for commercial, shrink-wrap software is in a deplorable state. Because of higher production and logistics costs and the opportunity for related book sales, vendors are minimizing printed documentation in favor of "online" help (extra CD space has zero marginal cost) and large, expensive, and separately available reference guides[16]. Significant software titles currently ship with heavy-stock installation quick-guides and registration cards to be returned to the vendor. Presuming simpler installations, even the former would disappear.

---

own applications and develop vertical, domain-specific applications that leverage the power of Office, which includes Excel.

[16] Witness the publishing divisions of Microsoft, Sun, and Oracle—Microsoft Press, SunSoft Press, and Oracle Press, respectively—which in earlier years might have been the bulk of these companies' documentation groups.

Documentation for solvers is deplorable as well, but for opposite reasons. Almost every existing solver relies only on printed documentation or readme files. These are often either incomprehensibly organized and written, or too sparse and general to help. A solver's documentation generally fits one of two types: RTFM or UTSL.

RTFM. Large or commercial solvers in the past have utilized the "Read the f***ing manual" approach of assisting users by forcing them to consult massive reference books. For instance, the CPLEX manual is 384 pages [17], while the OSL manual weighs in with 823 [47]. RTFM works in descriptive, expository, and tutorial situations, where the linear progression of a book coincides with the user's linear thought processes. These manuals work less well in reference form, where items are organized by topic, usually alphabetically, forcing the user to know where to look to find the desired topic. (This is the conundrum of using a dictionary to look up the spelling of a word.) Indices can help marginally, but books, naturally, are incapable of exhaustive word searches.

A rather simple and effective solution is to include an online, searchable version of a manual with the software. Microsoft's Developer Network Library (MSDN Library) is an enormous archive of articles, manuals, books, and samples of Windows programming and applications that is entirely searchable. While much of the MSDN Library is available in magazine articles or books, the presence of everything at the developer's fingertips online is invaluable. Ideally, commercial programs would include both printed and online versions of their documentation.

UTSL. Medium-sized or smaller, non-commercial, and free-ware solvers frequently employ the "Use the source, Luke" documentation strategy. As their entire official documentation, these solvers come with a very small readme file instructing the unsuspecting user to glean all relevant functions and parameters from the source code itself, which, at least, is included with the solver distribution. Of course, technically the source code is sufficient to learn everything about a solver. Nevertheless, time and budget constraints or domain inexperience limit a user's ability to effectively transform source code into meaningful documentation. These solvers can range from annoying and aggravating to entirely unusable.

One solution for the UTSL problem, besides perfectly clean, legible code, is to use an automatic documentation system, such as the javadoc utility that ships with the Java SDK. Applications such as javadoc parse specially tagged source code files and generate documentation from comments in the source code. With proper discipline, the source code can be entirely self-documenting.

To what extent, then, can the framework supplement and support the documentation process for future solvers? The next sections explore the uses for documentation in the context of solvers.

2.3.3.1 Documentation for selecting a solver

One of the first questions a modeler might ask of a solver is, "Is the solver suitable for my problem?" This question has two parts: whether the underlying algorithm is appropriate for the problem, and whether the specific implementation of that algorithm in the solver is appropriate for the needs of the modeler. Generally, the solver cannot be suitable if the implemented algorithm is not. If the algorithm is appropriate but its implementation is not, the modeler will need to find or develop another implementation of that same algorithm.

Suitability of the algorithm. After a very quick filter of determining whether a solver will even run on the modeler's system, the modeler will turn to assessing whether the underlying algorithm implemented by the solver can manipulate or solve the desired problem. Independent of any computer and implementation details, the algorithm must be able to solve the problem, or there must exist some simple transformation of the problem that the algorithm can solve. Satisfying this requirement means that the algorithm has a collection of inputs, outputs, and indices, with perhaps domain requirements over those collections, that match (or are a superset of) the requirements of the problem. For example, the knapsack problem has one index set, three inputs, and one output. The index is the set of possible items to place in the sack. The inputs are a scalar for the capacity of the sack, a vector over the index set for the value of each item, and a vector over the index set for the weight of each item. The output is an integer vector over the index set of the count of each loaded item. All values are non-negative. This algorithm can solve a problem that has only integral outputs, although there is no restriction on the input values.

There has been much research in the area of mapping problems to mathematical models [2, 3, 5, 26, 40, 45, 60, 74, 75, 81, 84, and 100]. This thesis presumes that the model is known, and the algorithm and solver remain to be selected. Integrating model selection into the framework is an extension of solver selection within the framework, with a different knowledge system.

Algorithm suitability also captures some expectations about inputs and outputs. For instance, whether the algorithm requires integral values or returns only integral values might be an important factor in some situations. Some network flow algorithms can guarantee run time bounds only with specific restrictions on input values, such as requiring integral or rational numbers. Other algorithms might output only integral values or ordinal relations (sorted lists).

For optimization algorithms, the type of solution generated is important. In mission-critical applications, the modeler might require that algorithms return feasible solutions *only*. It is important to be able to filter algorithms on whether they might generate infeasible solutions. Furthermore, it might be worthwhile to compare algorithms depending on whether they generate optimal or only near-optimal solutions.

Another important issue is run time. Various metrics of run time behavior can be measured of most algorithms. These metrics include expected and worst case run times, expected and worst case number of iterations, or perhaps expected gaps from optimality. For $\mathcal{NP}$-

complete problems, the modeler might be more satisfied with a quick approximation algorithm rather than a slow exact one.

Finally, the modeler might take some interest in the mechanics of the algorithm itself. This presumes access to the presentation of the algorithm, through a bibliographic citation, web site, white paper, or associated help file. Details in these documents might prove persuasive in algorithm selection.

Suitability of the solver. Assuming the algorithm is suitable, the modeler then assesses a specific implementation of that algorithm in the form of an executable solver. Aside from the concern of whether the solver implements the algorithm correctly, there are a number of factors to consider when selecting a solver. Most important is to verify that the solver satisfies the same requirements as the algorithm itself. That is, if the algorithm can manipulate non-integral values, then the implementation of that algorithm should support non-integral values. Assuming this—or, realizing that in the end the modeler works with an implementation of an algorithm and not the algorithm abstractly—it is sufficient to check that the solver supports the desired features, such as optimal or approximate solutions, feasible or infeasible solutions, etc., to account for the algorithm as well.

Beyond this, many implementation details arise that are mostly independent of the particular algorithm. The answers to these questions reflect the design decisions made during the development of the software. For instance, does the solver run as a background process? That is, can the user execute the solver and then actually work on other tasks on the same system, or will the solver consume most of the system resources? Can the solver run remotely, on a dedicated workstation? Can the solver be interrupted during processing, either to cancel or pause the algorithm? Once paused, can the solver save its internal states to storage, and subsequently load the states from storage, so that the solver can resume later? Does the solver maintain internal states in between runs, to optimize small perturbations and what-if analyses?

If the modeler is interested in the mechanics of the solver, then resources such as the source code (for free or public domain solvers), citations, white papers and technical reports, and web pages should be easily accessible.

Conclusion. Documentation for selecting a solver could be summarized in a list of features. Each item in the list is an ordered pair, where the first element is a key name (such as "citation" or "number of inputs") and the second element is the value for the key (such "*Operations Research* 41:3" or "2"). When automated, this list becomes part of a system-wide datastore of available solvers. The modeler can then search over this list to filter all solvers down to a set of solvers that satisfy desirable conditions. For instance, the modeler could query for all solvers that solve a shortest-path problem with negative arc lengths. It is important to standardize the method for adding to, iterating over, and removing from the datastore, or else the datastore becomes a marginal, customized, and non-extensible hard-coded knowledgebase.

2.3.3.2 Documentation for using the solver

After selecting a solver, the modeler will ask, "How do I use it?" That is, what are the protocols, subroutines, data structures, languages, and commands that the solver understands and expects? The types of documentation that contain the answer to this question are as diverse as the answers themselves. They include reference guides, tutorials, examples, step-by-step instructions, online help, web pages, readme files, comments in the source code, and the source code itself.

Regardless of its format, the documentation must contain the details for these activities:

- Creating and initializing the solver. This includes loading the solver's program code as well as preparing its initial data structures.
- Loading a problem into the solver.
- Setting solver parameters, such as tolerance levels and solution techniques.
- Executing the solver's algorithm on the loaded problem.
- Retrieving the output or solution, as well as associated execution statistics, from the solver.
- Destroying the solver. This involves unloading the problem, destroying any allocated memory, and potentially unloading the program code from memory (much of which might normally happen automatically).

Depending on the implementation of the solver, several of these procedures might be merged into a single step. For instance, non-interactive executable solvers might take all of the input information (problem data and parameters) as command line options and dump all of the outputs to the terminal window. These solvers have a single step: invoke the name of the solver with the name of the problem data file, and pipe the output to another file. Other solvers might be able to operate in interactive and non-interactive modes.

Reducing complexity and differentiation. In a user's ideal world, every solver works the same. Only the name of the solver, input data, and output data change from solver to solver, problem to problem. This way, the user has to learn a single mechanism for solving a problem, and this knowledge then applies to all solvers and problems. Human nature prefers this redundancy and familiarity.

The same principle applies to documentation. Even if the procedures for using two solvers are different, there is no reason the instructions for these procedures should not have the same structure. This way, at least, the user will know where to find the instructions.

If a solver follows an existing protocol such as the framework proposed in this thesis, for the documentation it is sufficient to state it does follow the protocol.

2.3.3.3 Introspection and automating the documentation

Even if the documentation for selecting and using a solver is superbly crafted, it is still currently just RTFM documentation. The final step is to automate the discovery process by embedding into the solver itself the ability to access solver capabilities, limitations, and protocols. That is, a client can ask the solver itself for its citation, worst case runtimes, feasibility of solutions, etc. This is a programmatic ability—a client can be instructed to use only solvers that generate feasible solutions, and the client can then query each solver to verify whether it generates any infeasible solutions.

This capability, for the client to programmatically access documentation inside the solver, is called *introspection*. The name is adopted from the introspection services in the JavaBeans component architecture [97]. The framework will include a specification for introspection, by which the solver can embed its documentation (or links to its documentation) within itself, and by which clients can access this documentation.

2.3.4 Progress updates

One of the most unsettling aspects of Boston's MBTA public transit system, the "T," is the uncertainty of the waiting time for a train. Walking onto an empty platform, a commuter has no idea when the next train will arrive. It could be two minutes or ten. She can only make a guess based on her past experience of waiting times under similar conditions. On some occasions, the arriving train might, with no other advance notice than a blast of its horn or a flash of its lights, speed right through the station, ignoring the hapless commuter completely—expectation established, resolution denied.

Many solvers provide no better guidance than the T. Once told to solve something, they tackle it with the single-minded focus of a cat stalking a bird, and utterly fail to tell the client (or the doomed bird) how much longer the wait will be. If the solver is not well-behaved, it can tie up the operating system, consuming CPU utilization and making it difficult to use the computer for other activities. The system might even appear to hang. How long should the user wait?

There is a simple solution to this problem. In this regard, the London commute easily tops the Boston commute. In Underground tube stations in London, every platform has a digital display of the expected arrival times for the next two or three trains, along with their destinations. While not entirely accurate in their predictions, the timings are close enough that when the sign reads ten minutes, it is a safe bet that the next train will not arrive in two. Even when the train takes longer than its given forecast, the psychological effect of having a forecast at all eases commuters. This is analogous to the floor indicator lights above elevator

doorways in a skyscraper—people waiting for an elevator can estimate (or gamble) which doorway will open first and when it will do so[17].

Similarly, a solver should be able to notify its client of its progress. This could transpire in a number of ways. Two common measures are *percentage complete* and *estimated time remaining.* Percentage complete captures what fraction of the processing has occurred. Typically this requires knowing precisely how much processing is required in the first place; this is not always possible. Estimated time remaining is a forecast of how much longer processing will continue, as a measure of wall-clock time (not task or CPU time). As an estimate, there is some fudging room for this forecast.

Percentage complete is useful when a process requires a known number of steps that can be reliably counted. An example is matrix multiplication. Every product and summation bring the solver one step closer to completion with certainty. Note that both being able to count steps and knowing the number of steps are necessary for calculating percentage complete. With the simplex method for solving linear programs, the iterations can be counted, but the total number of iterations is not necessarily known until the algorithm terminates.

The simplex method provides a good example for estimating completion. Assuming a minimization problem, during the simplex algorithm the current best cost is non-increasing, and the best lower bound on the cost is non-decreasing. Hence, the amount that the cost can decrease is itself non-increasing, and typically decreases some positive amount with each iteration. Therefore, some measure that takes into account the size of the gap between current best and lower bound can serve as the metric for the user. For instance, given the remaining gap, the (moving) average time per iteration, and the (moving) average decrease in the gap per iteration, an estimated time remaining might be simply:

EstTimeLeft = AvgTimePerIteration * (Gap / AvgGapPerIteration)

Client control. A progress notification capability should also allow the user to cancel the processing. At an Underground platform, if the posted waiting time is unacceptably long, a commuter can just walk away and find another means of transportation. Similarly, if an algorithm seemingly stalls or just is taking too long, the client should be able to *cancel* the execution and receive control of the system again. Additionally, it might be useful to allow the client to *pause* the solver execution, and then to *resume* it later from the same point. Ideally, the solver could save all of its internal states about where it is in the algorithm.

---

17 Extending the analogy even further: Elevator waiting areas often have mirrors, because research suggests that perceived waiting time is less when people have something to do, like preening, while waiting (Larson [59]). While waiting for subways, commuters can read poster and billboard ads for local attractions and movies. In software, modern installation applications display splash screens periodically during the installation process to give the anxious yet waiting user something to look at while files are copied. Perhaps solvers should display similar messages, such as, "Our barrier optimization uses the most modern techniques available to solve your problems."

This way, for instance, a client could schedule a solver to run only when the workstation is unattended, such as overnight. If the solver requires 24 hours, then it could run 8 hours a day over three days, with the exact same results and total CPU time as if it ran for 24 hours consecutively.

Presentation to the user. One way to present the progress notifications to the end user is through some simple GUI controls. One such control is the progress bar:

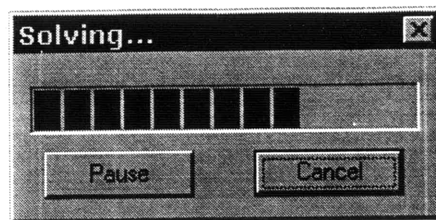


Figure 2.10: A progress bar window

Upon receiving a percentage complete notification from the solver, the client updates the progress bar to reflect the solver's current state. The progress bar might also have an associated set of cancel, pause, and resume buttons for interactive control of solver execution. Often, the progress bar represents *modal* activity—the application is suspended and cannot be otherwise used until the execution is complete. This makes progress bars useful for quick algorithms; typically those less than a few minutes.

Another way to present solver progress is through some indication on an application's status bar:

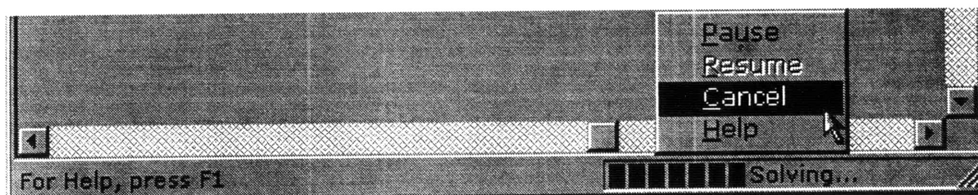


Figure 2.11: Progress notifications through the status bar

In this case, the progress notification is subtler. By clicking on the notification, a menu appears allowing the user to cancel, pause, or resume processing. This way of presenting progress notifications is *modeless*—the user can interact with the application while the solver is running (as long as the two can inter-operate without disturbing each other's data). So, for instance, the user could start one solver while preparing data or analyzing charts for another.

Modeless presentations to the user are suitable for longer-running solvers, from a few seconds to a few hours.

A third technique, suitable for the longest-running solvers, is to have another application, separate from the solver application, that could dynamically (and asynchronously) query the solver for its current progress. This introduces new functionality to the progress notification mechanism, but permits interactions where the solver runs in the background without any user interface component, and the user runs a command-line application to query the solver's state. An example interaction might look like this:

```
%mysolver &              ' runs the mysolver solver in the background
%myprogress              ' runs the notification application for mysolver
    myprogress: mysolver has 6h, 28m remaining.
%myprogress -cancel            ' cancels the mysolver solver
    myprogress: mysolver canceled.
%
```

2.3.5 Life cycle control

In the event that a solver will require more time than anticipated or desired by the user, terminating that solver should be a graceful action. Too frequently, users resort to killing processes the hard way, such as using "Ctrl-Alt-Delete" on PCs or "Ctrl-Command-Reset" on Macs. If the solver had acquired locks on system-wide resources, then this process homicide can have detrimental effects on the rest of the system. Any files opened by the solver might be left dangling, resulting in lost clusters on the system's drives. At the worst, it can lead to performance degradation and eventual system failure. If the solver is running on a network server, it might even be impossible to kill the process, because the user might need, but does not have access to, the machine itself.

A solver, especially one that might require long execution times, should provide a graceful means for terminating its execution. If a solver and client both support the progress notifications described in the previous section, then the client could terminate the solver as part of the notification procedure. This makes graceful termination a relatively easy feature to implement on top of progress notifications.

If the solver or client does not support progress notifications, then another, likely asynchronous messaging system might be required in order for the client to send a "terminate" message to the solver at any time.

Beyond termination, a solver should also provide for "pause" and "resume" capabilities, so that a client could pause processing at a system, in order, perhaps, to use the resources of the system for other activities. The solver should then be able to resume where it left off before. Altogether, these capabilities fall under the umbrella of *life cycle control.*

2.3.6 Dimension and type support

Dilbert: "I'm so lucky to be dating you, Liz. You're at least an eight."
Liz: "You're a ten."
Pause...
Dilbert: "Are we using the same scale?"
Liz: "Ten is the number of seconds it would take to replace you."
Scott Adams, *The Dilbert Future*

Clearly, it is important to use the same or consistent scale, type, and units when comparing or manipulating quantities. Scale, dimensions, and units refer to the quantitative, measurable properties of a number. Example scales are kilo and nano. Dimensions include length, area, weight, time, and cost. Units range from inches and feet to apples and cars. Type refers to the qualitative aspect of a number. Given two different numbers, "cost of production of a car" and "cost of shipping a car," production and shipping are the type of cost being measured. Bhargava, Kimbrough, and Krishnan [4] combine units and type into a single measure, *quiddity*.

Five times three equals fifteen if the numbers have no further meaning. Nevertheless, if five is the number of apples and three is the number of oranges, it makes little sense to multiply these quantities together. Instead of generating an answer of fifteen, an ideal system would report an error. Practically, in the absence of dimension and type information, systems do not fail when manipulating numbers. This is one of the most insidious software errors there is—data corruption. The software proceeds as normal, even reporting successful completion of its analysis. There is no way for the user to know that the data have been utterly corrupted and are not valid. System crashes and program crashes (caused, perhaps, by invalid pointers) are easily detected because they manifest obvious visual displays, such as the infamous Windows NT "Blue Screen of Death[18]." The user cannot help but notice that the system has failed. Data corruption, on the other hand, is nearly invisible, and hence should be especially warded.

Not everything is apples and oranges. In large-scale environments, especially those that leverage corporate databases or datamarts, the dimensions and types of input and output data are just as important as their quantities. Modelers rely on dimensions and types to verify the dimensional consistency and correctness of their models. There are algebraic laws for dimensions that identify valid manipulations and equivalencies [4]. Two of the consistency laws are that quantities may be added or subtracted only if their dimensions are equivalent, and that an equation is balanced if the dimensions of its left and right sides are equivalent.

---

[18] A complete operating system crash resulting in a dump of debugging information onto the screen in white text on a blue background. The only recourse is to reboot the system.

Modelers also rely on dimension information to convert quantities to equivalent dimensions. For companies with distributed, international databases, it might be necessary to convert all monetary values to a single currency, based on current exchanges rates. It might also be necessary to convert values from metric units in the International System (SI) to U.S. customary units (i.e., from meters to feet, kilograms to pounds). These conversions are often handled as part of the data-preparation and post-processing phases of an implementation. That is, the modeler assumes away all dimensional problems as "implementation details" and treats related values, such as all costs, equivalently (i.e., dimensionless). Not only does this not mitigate the problem (from a holistic viewpoint), but also it removes any opportunity for dimensional verification of the model.

At the price of such a lost opportunity, the modeler must have a good reason for assuming pre-converted quantities. There are several. Existing solvers generally have little or no support for input data with associated dimensional information. Adding dimensional support to custom solvers is non-trivial. Coding conversion tables by hand is lengthy, error-prone, and tedious. As far as most modelers are concerned, it is better just to do without.

The following sections detail features of solvers that can ease the modeler's burdens by incorporating dimensional information.

Awareness of dimensions. Solvers, particularly large-scale or industrial-strength solvers, should at least be aware of data with attached associated dimensional information. Namely, these solvers should be able to accept as input data that contain both numerical values and qualitative dimensional information. They should also be able to provide as output both numerical values and qualitative dimensional information. The output dimensions will likely be some transformation of the input dimensions.

At the client's request, the solver should verify the consistency of the input data's dimensional information. Just as the solver might check each input value for simple constraints, such as non-negativity and integrality, the solver can check that each constraint is dimensionally balanced based on the various dimensions attached to each input.

For example, consider the knapsack problem with inputs shown in Table 2.2. The constraint in the knapsack problem specifies that the sum of the per-element product of the Weight and the Count vectors must be less than the Capacity. An element of the Weight vector times an element of the Count vector yields pounds, so the sum of the per-element product is also pounds. Capacity is hours. Because there is no valid transformation between pounds and hours, the input data are invalid.

Input	Value	Dimensions
Capacity	24	hours
Value	[5 20 3 16]	dollars
Weight	[3 2 4 6]	pounds (lb.)
Count	decision vector	none

Table 2.2: Example knapsack problem with dimension information

Some of the literature (e.g. Bradley and Clemence [10]) propose folding this dimensional verification into the description of the model itself. In this manner, the client application is responsible for validating dimensions before loading data into the solver. This might have the deficiency of imposing a restrictive dimensional structure upon the modeler.

While it is important to include dimensional information whenever possible in the solver's documentation, it is equally valuable to allow the solver to verify the dimensional information itself, once it has received the input data. Once the solver has all of the input data, it can perform any numerical, feasibility, and dimensional verifications it needs in order to begin processing. Because the dimension conversion process is naturally a run-time procedure (at some point input data must be converted into the correct dimensions), it might be most efficient to perform dimensional verification at the time of conversion.

Manipulation of dimensions. Dimensions are a universal trait. They are not specific to any one solver or problem. Every data set has them, and every solver must manipulate them. Ideally, the expertise for converting and manipulating dimensions would exist in a single place. In current solvers that support dimensions, this is not the case. Most commercial solvers that support dimensions have their own internal databases of conversion factors, and they have their own procedures for adding new dimensions to those databases. To introduce a new dimension to a user's system, the user must add the dimension separately to each product. These current systems look like this (Figure 2.12):

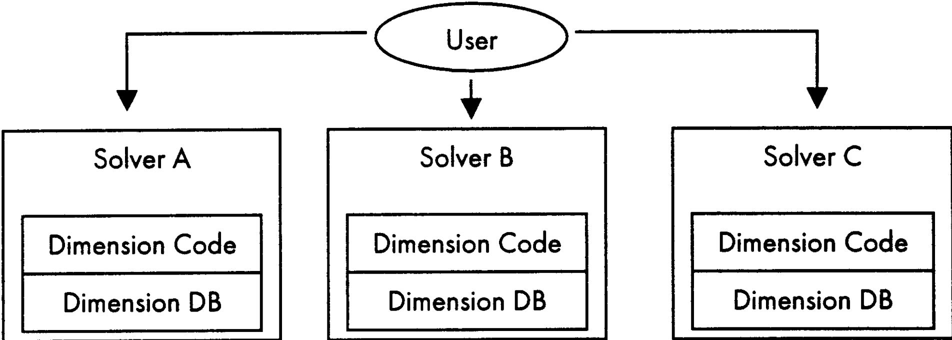


Figure 2.12: Traditional solver structure with decentralized dimension support

Forcing solvers to manipulate dimensions directly adds a significant burden to the solver developer. Just as solvers are encapsulated in components, dimension manipulation and conversion can be abstracted and encapsulated into components. In fact, components that manipulate dimensions are, in their own way, solvers.

Thus, instead of solvers relying on their own internal dimension databases or on a known system database of dimensions, solvers use other solvers for manipulating and converting dimensions. Therefore, a single solver for transforming dimensional information is sufficient to service all solvers. This solver is called a *dimension manipulator*. By adding a new dimension to this dimension manipulator, every solver is effectively updated with the new dimension. This is shown in Figure 2.13.

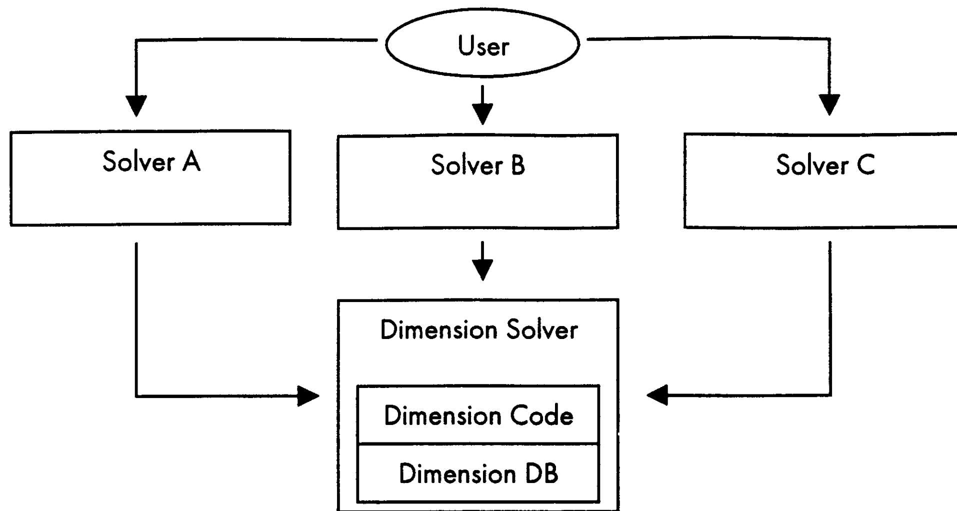


Figure 2.13: Solver structure with centralized dimension support

A better solution would be to provide mappings from different dimensions into different dimension manipulators. Thus, it is not necessary for a single manipulator to handle every dimension in the universe. Instead, manipulators could be optimized for certain types of transformations or dimensional standards.

Conclusion. The ability to handle data with associated dimensional information is important, but not necessary for small, simple solvers. As such, dimension and typing support is a desirable feature, but not a requirement, of solvers.

2.3.7 Testing and validation

Testing and validation is a crucial element of algorithm design and development. There are several reasons to test and validate a solver:

1. To validate that the algorithm as designed is correct. This is usually secondary evidence alongside a proof of correctness, but in some scenarios correctness through implementation is necessary.

2. To ensure that the implementation of the algorithm is correct. Although the designer of the algorithm might have got it right, the software developer might have introduced bugs into the solver.

3. To verify that the implementation satisfies any size and time restrictions specified during the requirements analysis. These requirements are especially prevalent in mission critical or real-time environments.

4. To compare the run time or closeness of optimality of the implementation to other implemented algorithms for the same problems.

The first and last reasons are primarily the algorithm designer's concerns, while the second and third reasons are primarily the developer's concerns. The first three are traditional software development quality and testing problems. Every project has algorithms of some sort that must be designed and implemented correctly. Usually these algorithms have associated time or space requirements. Peculiar to algorithm development research, however, is the need to compare an implemented algorithm with numerous other algorithms.

For many algorithm designers, this comparison process is the bane of their research existence, a necessary evil in demonstrating that their new algorithms are worthwhile and effective. First, the researcher must implement the new algorithm[19]. Second, the researcher must acquire (or implement) a collection of other implemented algorithms with which to compare the new one. Third, because of differences in how each implementation expects to handle inputs and outputs, mappings must be created from a default input and output data format to each implementation's requirements. Fourth, appropriate input data must be generated or acquired. These are frequently randomly generated or obtained from online problem databases. Fifth, the tests must each be run and timed. Finally, results are compared.

Looking at the third step above, it is apparent that the testing environment encompasses the primary problems this thesis is attempting to address, namely that solvers with different input and output data formats require different mappings to make them interchangeable. Hence, testing and validation should be easily captured by the resulting framework.

2.3.8 Computer-based training

A significant segment of users not captured by the domain analysis of applied solutions includes students and teachers. Although they might not work on massive implementations or large-scale problems, they do use the same tools found in many larger applied solutions. These tools include spreadsheets, high-level programming languages, and commercial solvers (academic versions).

Educational environments typically have looser requirements on the implementations of solvers. There is greater tolerance for poorer performance, lower numerical precision, smaller problem size constraints, lack of security, and inability to multi-process. The goal of the exercise is not the solution but the means to the solution; the goal of the solver is to illuminate the means while providing a solution.

Unfortunately, along with the greater tolerance for less well-developed solvers, students and teachers usually have to work with solvers that are difficult to use. This should not be the

[19] One of the great uses of graduate student labor.

case. On the contrary, because solvers are used as short-term, one-shot exercises, they should be as easy to use as possible. Teachers should not spend their time programming the glue to make solvers work on their problems, and students should not spend their time trying to figure out how to manipulate and program solvers themselves. Hence, to the extent that the framework increases reusability of solvers, it is ideally suited to educational environments.

Educational environments do have the distinction, however, of being particularly concerned with the execution of the solver as an end in itself. Consider the simplex method. In industry, the simplex method might be just a phrase that describes how one solver works and another does not. In education, the simplex method is a dissected process, a learning tool, and an endless source of qualifying examination questions. An industrial strength application might only worry about the current iteration number and what the estimated run time is for a simplex method solver. A student will need to know much more information at each iteration, such as the pivot point, the reduced costs, and the basis.

A solver can be specifically designed as an educational tool, if it can provide the right information at the right times during execution. A simplex method solver might provide more than just a progress notification at each iteration; it might allow the client to query all of the internal data structures. The client might then display these results interactively to the user as the simplex method pivots from basis to basis towards the optimal solution.

Another example is a network flow algorithm that uses residual networks. With a properly designed solver, the client could query the residual network, or changes to it, at each iteration. These changes could be displayed graphically in a picture of the residual network.

The mechanism for providing this educational information can be standardized as *computer-based training (CBT) hooks*. These hooks are generalizations of the progress notification capability, and provide more generic interaction between client and solver. The basic idea is that the solver notifies the client when certain events occur (such as, when an iteration ends or when a pivot begins). The framework should support and encourage CBT hooks. As will be seen in section 2.4.3, page 98, CBT hooks are a specialization of notifications.

2.4 Requirements of Networks of Solvers

Requirements for a stand-alone solver, as presented in the previous section, apply to all solvers. Solvers might work together in networks, each interacting to share data and results. However, to do so, solvers must be designed with this capability in mind. This section describes the requirements of a solver implementation that exists in a network of solvers. Requirements of networks of solvers apply to solvers designed to be incorporated into more complex solutions.

The requirements include:

- *Integrity of data.* The network maintains the integrity of input and output data, ensuring that no agent accesses invalid data.
- *Distribution and synchronization.* The network can exist across machines, processes, and time. Individual solvers can operate synchronously or asynchronously.
- *Notifications.* The network supports a variety of notifications of interesting events.
- *Global/local control.* The network supports either a global controller or autonomous control at each solver.

Each of the following sections discusses these in more detail.

2.4.1 Integrity of data

Imagine that in a network of solvers, each solver is its own autonomous agent. Each solver waits for its inputs to be set, executes when instructed to (or when all inputs have been set), and makes outputs available when it is finished.

A simplified, informal state diagram for a single solver appears in Figure 2.14. There are three top-level states, determined by whether the input data to the solver or the output data

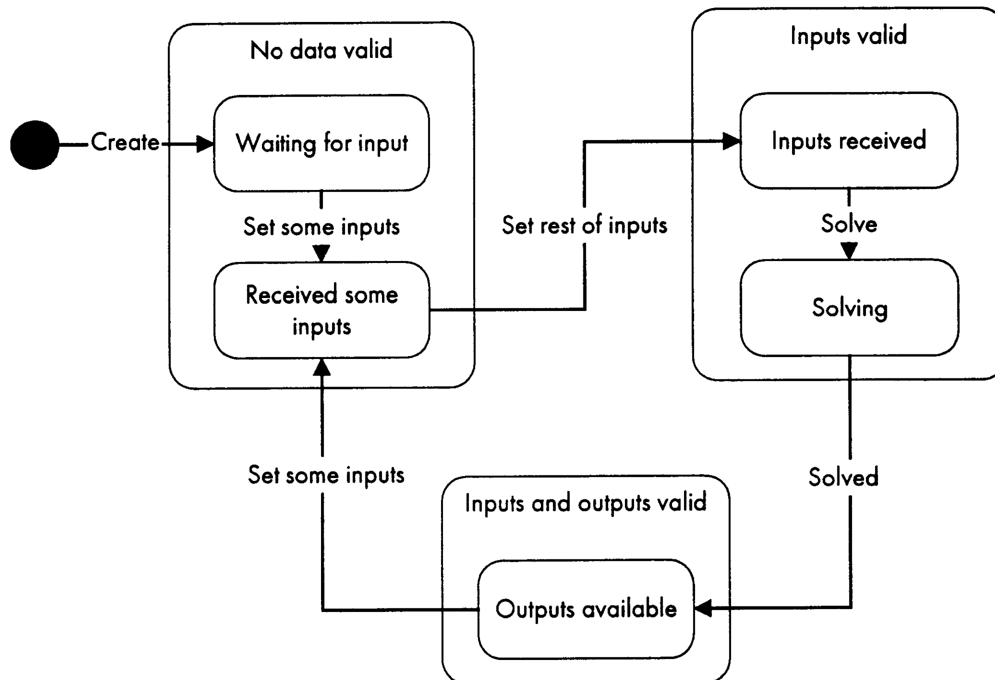


Figure 2.14: High-level solver state diagram

from the solver or both are valid or invalid. Input data are valid if all of the inputs for a single problem instance have been received by the solver. Output data are valid if the solver has completed its processing and no input data relating to the output data have been changed.

After creation, the solver begins in the *no data valid* state. No input data have been set, and no output data are available. Specifically, the solver is in the *waiting for input* sub-state. At this point, the solver has just been initialized and is idle. Once the client sets some of the inputs, the solver shifts to the *received some inputs* sub-state. Still, no data are valid, as defined above, because some inputs have been set, but not all. As the client sets the last input data in the solver, the solver transitions to the *inputs valid* state. Here, input data are valid but output data are not. The solver waits in the *inputs received* sub-state, ready to solve its problem.

When the client invokes the Solve command, the solver moves to the *solving* sub-state while the algorithm runs. While solving, the output data will be generated, and upon completion, the solver will transition to the *inputs and outputs valid* state. As long as the solver has outputs available, the client can retrieve output data from the solver.

At this point, either the solver can be destroyed (not shown), or a new problem instance can be fed into the solver. In the latter case, the client begins by setting some new inputs on the solver. This induces a transformation in the solver back to the *no data valid* state, because (a) some input data refer to one problem instance while other input data refer to another instance, and (b) the output data are no longer associated with the input data. The cycle continues until the solver is destroyed. Note that there is no *outputs valid* state (i.e., inputs invalid), because the outputs are only valid if the inputs are.

Other solvers, clients, or the user exercise the functionality of this solver during its lifetime. All of these clients might set inputs, retrieve inputs, retrieve outputs, and tell the solver to execute. The essence of data integrity is to guarantee that a client performs any of these actions only when it is valid to do so. For example, a client should not retrieve outputs if the solver is not in the *inputs and outputs valid* state.

For instance, suppose there are two clients, *Client A* and *Client B,* of a single solver, *Solver.* Client A is the primary client, providing the input data to the Solver as well as the command to solve. Client B is simply interested in retrieving the results once the Solver is finished processing. A normal sequence of activity is shown in the event trace diagram in Figure 2.15.

The steps in this event trace diagram are:

1. Client A sets all inputs in the Solver. The Solver transitions from the *waiting for input* sub-state to the *inputs received* sub-state.

2. Client A instructs the Solver to solve its problem. The Solver transitions to the *solving* sub-state.

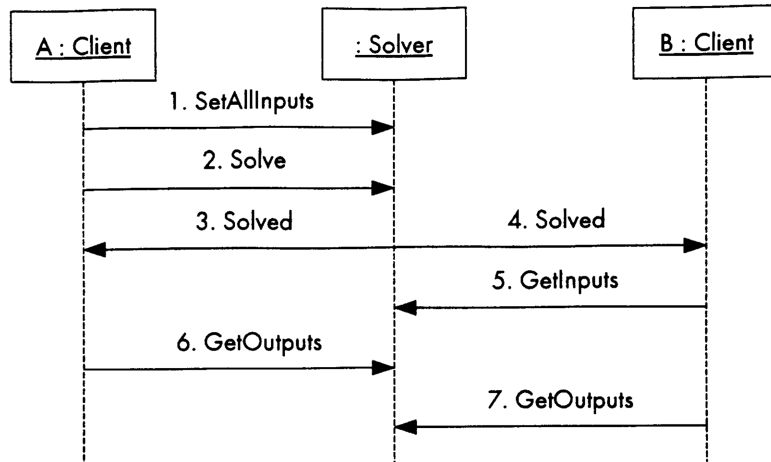


Figure 2.15: Event trace diagram for normal solver invocation

3. Upon completion, the Solver notifies both clients that it has solved the problem. The Solver moves to the *inputs and outputs valid* state. In this step, the solver notifies Client A.

4. In this step, the solver notifies Client B. The ordering of steps (3) and (4) is irrelevant.

5. Client B retrieves the input data of the problem from the solver. This is valid because the inputs are still valid.

6. Client A retrieves the output data from the solver.

7. Client B retrieves the output data from the solver. This step and the previous step are valid because the outputs of the solver are valid (because the solver is in the *inputs and outputs valid* state).

Note that steps (5) through (7) could come in any order, as long as the inputs are valid for step (5) and the outputs are valid for steps (6) and (7).

The following three scenarios demonstrate loss of data integrity. In each scenario, a client interacts with the solver in a manner that is invalid with the solver's state.

Scenario 1. In the first scenario, Client B attempts to retrieve the outputs before the solver has completed solving the problem. This event trace diagram appears in Figure 2.16.

1. As before, Client A sets all inputs into the Solver.

2. Then, Client A instructs the Solver to solve the problem.

3. At this point, the Solver is in the solving sub-state, which is inside the inputs valid state. Client B attempts to retrieve the outputs of the solver, but because the outputs are not yet valid, this should result in an error.

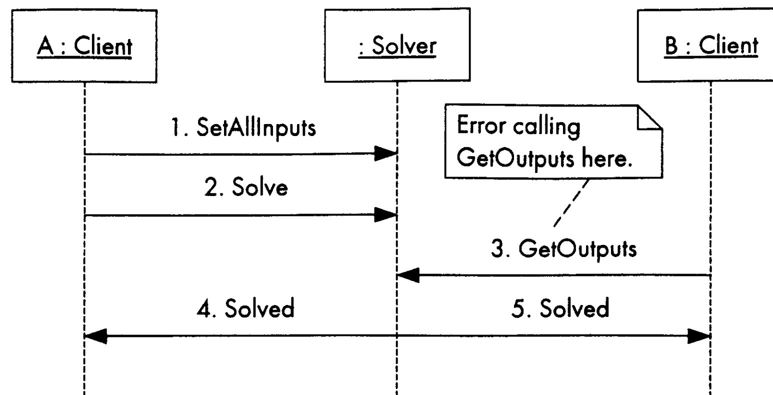


Figure 2.16: Event trace diagram of attempting to retrieve invalid outputs

4. Not until the Solver sends completion notifications to Client A and Client B are the output data valid. Step (4) is the notification to Client A.

5. Finally, the Solver sends the completion notification to Client B. Now, Client B could retrieve outputs from the solver. Again, the ordering of steps (4) and (5) is arbitrary.

The result is that Client B attempts to obtain output data before the solver has finished generating the output data.

Scenario 2. This time, Client B is trying to retrieve the input data, which it did not have access to *a priori*. Client B attempts to get the inputs at the same time the Client A is passing them into the solver. This event trace diagram appears in Figure 2.17.

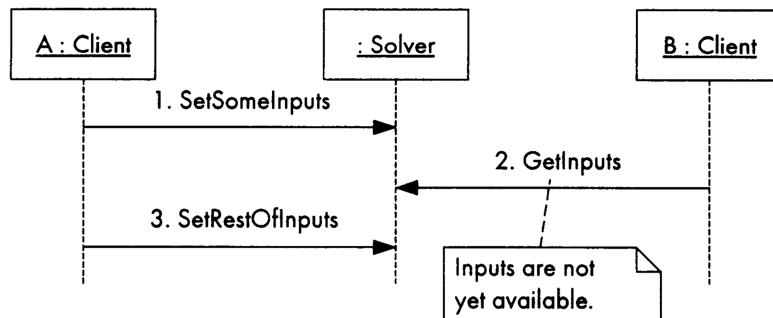


Figure 2.17: Event trace diagram of attempting to retrieve invalid inputs

1. This scenario begins with a twist. Instead of passing all of the inputs to the Solver at once, Client A passes only some of the inputs. This causes the Solver to transition only to the *received some inputs* sub-state, instead of the *inputs received* sub-state in the previous scenarios.

2. Client B then attempts to retrieve the inputs to the problem. Because the solver is still in the *no data valid* state, no inputs are available, and this is an error.

3. Client A completes passing all of the input data to the Solver, at which time the Solver transitions to the *inputs received* sub-state. Client B could at this time acquire the inputs.

To conclude, Client B has attempted to retrieve the input data before Client A has completed setting all of the input data in the solver.

Scenario 3. The first two scenarios present cases where Client B jumps the gun, trying to retrieve data before they have been generated. The final scenario presents a case where Client B is too late; the data have already been generated and then invalidated as Client A prepares a second problem instance. The event trace diagram for this scenario appears in Figure 2.18.

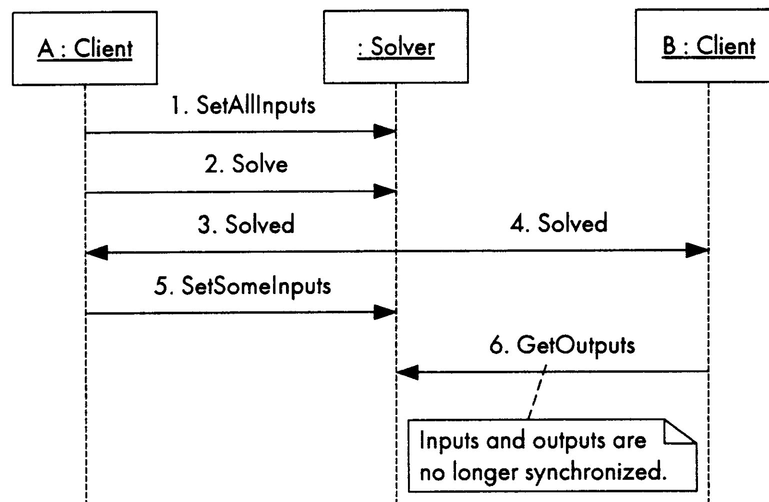


Figure 2.18: Event trace diagram of attempting to retrieve outputs too late

1. The first four steps are the same as normal interaction. Client A begins by setting the input data for the Solver.

2. Then, Client A instructs the Solver to solve.

3. The Solver notifies Client A that it has completed solving the problem.

4. The Solver notifies Client B that it has completed solving the problem. The ordering of steps (3) and (4) is arbitrary. The Solver is in the *inputs and outputs valid* state.

5. Now, Client A begins a new problem by setting some of the problem data. The forces the Solver to transition to the *received some inputs* sub-state, in which no data are valid. In

essence, the Solver has cleared the previous problem from its memory and is working on building a new problem.

6. Finally, Client B attempts to retrieve the output data. At this point, the output data are no longer valid, and the request must fail.

The problem here is that the Solver and Client A are unaware that Client B wants to retrieve the output data. Ideally, as soon as Client B receives the notification of solver completion, it can tell the Solver to lock its problem data, thus forbidding Client A from setting new input data. Then, Client B can retrieve the output data when it is ready, and then unlock the problem data, allowing Client A to begin a new problem. This transfers the error from Client B's attempt to retrieve the output data back to Client A's attempt to set new input data.

Conclusion. Data integrity must include the capability for a client to manage the lifetime of a solver's data sets, by providing locks on that data. As long as the data is locked, the solver cannot transition into a state that would invalidate or change the locked data. Furthermore, data integrity must ensure that clients do not access data when the data is not available.

2.4.2 Distribution and synchronization

Late on Monday night, June 16, 1997, an aging, 90MHz Intel Pentium-based personal computer cracked a message encrypted with a 56-bit key using the United States Data Encryption Standard (DES) [19]. This key length is the maximum allowable for software exported from the United States, and its breaking emphasized the corporate need for longer, more secure keys. The algorithm employed by the crackers? Enumeration. Of the approximately 72 quadrillion possible keys, 18 quadrillion were attempted before one yielded the encoded message, "Strong cryptography makes the world a safer place."

One 90MHz system processing a possible key every clock tick (clearly a liberal upper bound on processing rates) would require almost six years to analyze 18 quadrillion keys, yet this code was cracked in under six months. How? In a massive, centrally-concerted effort, thousands of people with similar off-the-shelf personal computers participated in testing possible keys. From a web site, volunteers could download software and a range of keys to try. Software at the web site managed the list of keys that had been attempted. Software on each client machine would iterate through a selected range of keys, trying each one against the encrypted message. The volunteers could run the client software at any time, such as overnight or as screensavers when they were not using their computers. This algorithm of enumeration exhibited both distribution and asynchronization.

Distribution. Distribution of activity enables a massive problem to be tackled by everyday, common machines. This code could not have been cracked by any single machine in five months. The scope of this problem required a coordinated effort among many machines.

There are a number of reasons to distribute a solution architecture across multiple computers. Sometimes, the various applications required to solve a problem exist only on

separate computers. There might be performance advantages to run subproblems in parallel. Some data preparation and manipulation might be most effectively located near the database server whereas number crunching analyses might be best located at a heavy-duty UNIX workstation removed from the database system. A Windows PC might then be most appropriate for viewing the results.

The framework for interconnecting solvers should not inhibit the ability to place the solvers in separate processes[20] or on separate computers. The framework should leverage the underlying component technology for inter-process communication (IPC), both across process boundaries (local procedure calls, or LPC) and across machines (remote procedure calls, or RPC).

Synchronization. Broadly, when a client requests some service of a supplier, the client has to decide whether to wait for the supplier to provide the response or whether to process something else. Synchronization captures this relationship between client and supplier. Typical function calls in modern programming languages are *synchronous*. The client calls another function, and the client does not do any further processing until that function returns; the client and its supplier, in the form of the function, are in synchronization. An *asynchronous* call is one in which the client goes on to its next processing step without waiting for the function to return. The client and supplier are out of synchronization, and some mechanism must exist in order for the supplier to inform the client when it has finished processing. Asynchronous calls are an important aspect of multithreaded programming, especially when machine boundaries must be crossed. The variability and randomness of network latency or utilization can significantly hinder performance when the client is always waiting for the supplier to return from a function call.

Within the framework, solvers should have the opportunity to work asynchronously if they can. That is, the protocol should not interfere with the synchronization of a solver and its client, and where possible, the protocol should support asynchronous activities.

Transparency. A user of the framework will want to be able to assign solvers to machines with utmost ease. If a system has two solvers on a single machine, and the user moves one of the solvers to another machine, there should be no additional tasks beyond specifying a new location for the solver. That is, the user should not have to add infrastructure or special code simply because the two solvers are now on separate machines. Fortunately, the underlying component framework, COM, is ideally suited for this requirement, because it effectively masks much of the remoting layer responsible for transporting function calls across processes or machines.

---

20 A *process* in this context is an instance of a running program (Richter [89]). Modern operating systems sandbox processes into their own memory spaces to minimize process interaction. Thus, inter-process communication is more complicated than just calling a subroutine. See Chapter 10 of Rogerson [90] for an explanation of crossing process boundaries with COM.

2.4.3 Notifications

The various entities in a graph solution architecture are not only submissive entities that do what others tell them. They often have very interesting things to say themselves. Progress updates (see section 2.3.4, page 81) are among the numerous statements a solver might make. These statements are called *notifications,* because the originating object notifies the destination of some event. The object that sends out notifications is called the notification source, the publisher, or the subject. Any number of objects can receive notifications; a receiving object is called a notification sink, a listener, an observer, or a client.

Each class of objects within the network might send out different kinds of notifications. For instance, solvers might notify listeners of progress updates, execution completion, discovery of infeasibility or dimensional inconsistencies, or of other generic failures. Data sources might notify listeners of changes to their data values and of availability of data for processing, for instance as a result of the completion of a specific query. For each class of objects, there might be both generic and specific notifications. All solvers can notify listeners of progress updates and solver completion. Only certain solvers will notify listeners when specific variables violate certain constraints or achieve certain values; for other solvers, such events might not be meaningful.

Consider a two-stage solution where the first stage is a mixed integer program and the second stage is a function evaluation over the feasible set of values in the integer program. The two stages are linked together so that they can run in parallel, with the second stage calculating values from the first stage whenever they are available. In this example, the second stage needs to know whenever a new feasible solution to the mixed integer program is calculated; it is not concerned with infeasible solutions. Hence, the second stage wants to receive a notification when a feasible solution is calculated. The first stage might provide such a notification if the mixed integer program solver has the built-in intelligence to notify clients of new feasible solutions.

As another example, the end user might be interested in a 95% solution—a feasible value that is verifiably within 5% of the optimal solution. Often, 95% solutions are quickly attainable, whereas achieving the remaining 5% can take disproportionately longer. The user might wish to find a solver that could send a notification when the 95% solution is obtained; such solvers could be hard to find. More likely, the user will find a solver that can send notifications after every iteration, during which it is possible to query the solver for the current solution, the best solution, and known lower and upper bounds or the optimality gap. Then, with some custom client-side code, the user can calculate the percentage of optimality during the notification using these queries. This will enable the user to employ many more solvers.

The notification mechanism

The notification mechanism is an example of the Observer pattern (see Gamma et al. [30]). It is actually quite simple. In essence, there are two objects. The *subject* has some notifications

of interest to broadcast, such as the completion of execution or a change of state. The *observer* desires to receive these notifications from the subject.

Before receiving any notifications, the observer must inform the subject that it wants to receive those notifications. This is frequently called an *advise*—the observer advises the subject. After the advise, the subject might post notifications to the observer. Upon receiving a notification, the observer might query the subject about its state. When the subject or observer terminates or when the observer is no longer interested in receiving notifications, the observer *unadvises* the subject, and the subject no longer sends any notifications to that observer. A single subject might send notifications to many observers, and with appropriate context a single observer might receive notifications from many subjects.

The framework should define or support a notification mechanism for notifications between solvers and clients and between data sources and solvers. To the extent that it is meaningful, specific notifications can be defined for each class of objects in various relationships. For instance, what notifications might a solver send to a client? What notifications might a data source send to a solver?

2.4.4 Global/local control

In single-stage solutions, it is fairly obvious that the client of the solver is the controlling entity. Once the client prepares the input data, it hands it off to the solver, instructs the solver to execute, and then retrieves the output data from the solver upon completion. In essence, the solver, the data stores, and the resource flows operate at the behest of the client. In this simplest scenario, there is not much for the client to do. In fact, the single-stage solution represents a minimal functionality requirement for the client. This is global control.

Moving to multi-stage and distributed solutions, it is possible to retain a single point of control. That is, a single client application could still direct numerous solvers and data stores spread across a network of computers, where no single solver or data store operates without a direct command from the client. Nevertheless, it is less certain that this is a wise or practical arrangement, primarily for two reasons.

First, a single point of control represents a single point of failure or congestion. If the client process or the network link to the client fails for whatever reason, then all of the solvers might become unstable or suspended, or they could even crash. If the step of taking the outputs from a solver and saving them to a database requires the client to initiate the database transaction, then this step depends upon the existence and availability of the client. If many solvers need to access many databases, then the client becomes a bottleneck as notifications from solvers queue at the client, awaiting processing. Second, maintaining the state knowledge and control at the client significantly increases the complexity of the client for typical networks of solvers. The client has to manage the interactions among solvers and data stores, maintain the integrity of data through the solution network, and create and

destroy solvers as appropriate. The arrangement with the client as the single source of control is similar to the Master-Slave pattern in Buschmann et al. [14].

To relieve the client of the duties of totalitarian control of the network, individual components within the network take on control responsibilities. Not only is the execution of the solution distributed among multiple solvers—and perhaps machines—but the control of that execution is similarly distributed among multiple solvers and machines. Each element, in effect, takes control of itself, responding to its environment based upon well-defined local control rules. Hence, this is local control.

For instance, consider a solver that aggregates, via average, the values in a database table. These values are themselves the output vector of another solver. In a global control scenario, the client must initiate the first solver, wait for it to complete, store the values into the database, initiate the second solver, wait for it to complete, and then read the final average from the second solver. This is demonstrated in Figure 2.19.

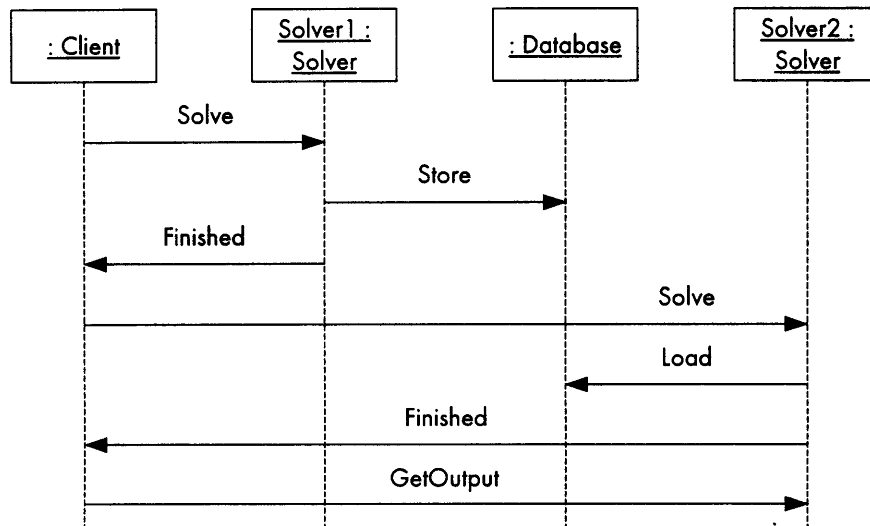


Figure 2.19: Example of global control in a 2-solver network.

With local control, the client initiates the first solver, then waits for the *second* solver to complete, and then reads the final average from the second solver. The first solver stores its outputs into the database, and through notifications tells the second solver it is finished. The second solver, after waiting for the first and receiving its completion notification, reads the table from the database and computes the average. This is shown in Figure 2.20.

In the local control scenario, each object knows how to react to its environment. The second solver knows to initiate its own execution when the first solver notifies it of completion. Compare this to the global control scenario, in which the second solver relies on the client to handle the completion notification and initiate the second solver.

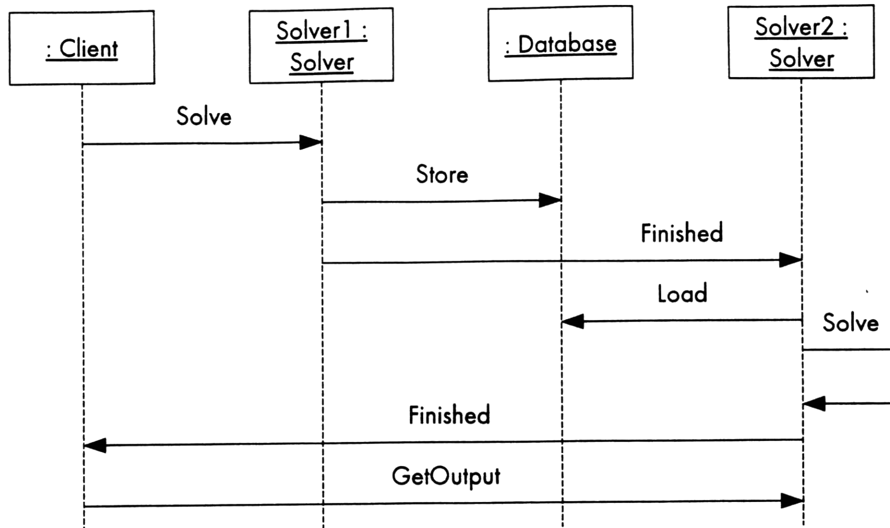


Figure 2.20: Example of local control in a 2-solver network.

The local control description might sound more complicated than global control, but in complex networks, local control lends itself nicely to reducing overall complexity, to encapsulating control and state information, and to managing resource flows in the network more effectively.

Because of the wide variety of possible architectures and control patterns, it would be most beneficial for the framework to support both global and local control. This can be accomplished through an infrastructure that provides the extra system support for local control while stepping out of the way in global control scenarios.

2.5 Requirements of the Core Services

Most GUI applications display dialog boxes from time to time such as this one:

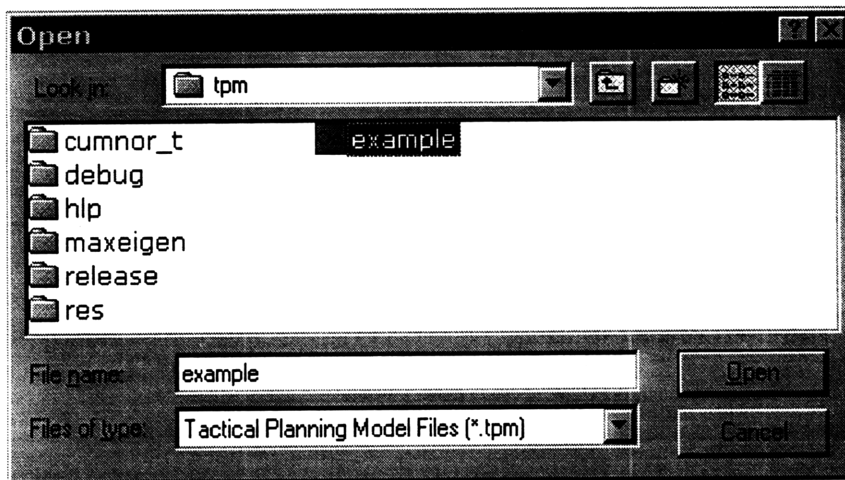


Figure 2.21: A common dialog

This particular dialog, the "Open" dialog, is one of the most prevalent in Windows applications, because it is used when the user wants to open a file for editing. Almost every application uses this exact dialog, and yet almost no applications actually contain the routines that implement the dialog and bring it to life. Why? Because Windows contains the implementation of the Open dialog so that no one else has to. It is part of the Windows common dialogs library (which includes the Save dialog, Print setup, and the color picker). In fact, much of Windows is not specifically operating-system functionality but is instead shared code—common functions and routines that both decrease software development costs and time and help standardize the look and feel of the operating system[21].

With solvers there are similar points of commonality, areas of functionality that might exist in every solver or client. These can justifiably be removed from all solvers and clients and placed into a central repository, akin to the shared libraries of Windows.

There is also functionality that is used by many applications but is not provincial to any one. Enumerating embeddable objects, such as charts and graphics, is one such activity. Most applications allow the operating system to enumerate these objects for them, leveraging system libraries and registries for these central tasks. Similarly with solvers, there exists functionality that is independent of any one solver or client, and belongs in a central repository.

The following sections describe two such areas of functionality:

- *Registry of available solvers.* This area parallels the enumeration of embeddable objects. When a client needs to select a solver, the core services can list the installed solvers on the machine, making it easy for any application to determine what solvers are available for processing.

- *Dimension and type support.* Centralized dimension support is partly a specialization of registration of available solvers. The core services can manage a list of dimension manipulators installed on a machine, as well as help traverse the network of dimension conversion paths.

Finally, during the course of development of the framework in the third chapter, various components will added to the core services as necessary. These include, primarily, helper objects that simplify the life of a developer but are conceptually contained within the specification of another part of the framework. One example is the *advise helper,* which implements the messy details of managing a list of connections in a notification relationship. Advise helpers are only useful within the context of notifications, so they will be discussed in the sections on notifications.

---

[21] Including the WebBrowser control that is the heart of Internet Explore 4.0.

2.5.1 Registry of available solvers

Returning to the example of inserting a chart into a document from the first chapter (see page 20), when the user asks to insert a generic object (Insert Object… in Microsoft Word), the application displays a list of objects that could be inserted into that application. How does the application generate this list of objects? If the user installs a new component or removes one, the list will be updated to include or remove that component the next time the application is run. All applications generate the same list of objects, too. How does this work?

Clearly, the maintenance of the list of insertable objects is a system-wide service. The operating system provides facilities for a component to register and unregister itself with the operating system as an insertable object. Specifically, when a component is installed, it will register with the operating system. The operating system then makes this component available to other applications as an insertable object. It's like registering to vote; after a new resident registers, she suddenly gets called upon by civic functions that use the voter list, notably jury duty. When a resident leaves town, she does not have to unregister her vote, but she registers a forwarding mail address. Components, on the other hand, must unregister themselves with the operating system when they are removed from the system. Thus, the operating system will no longer present that component to the other applications. (If the resident dies, then her postal service must be terminated, which is a form of unregistration.)

There are two essential services that the operating system provides:

- *Registration and unregistration.* The operating system facilitates the addition of new components into the system by exposing registration capabilities and the removal of components from the system through unregistration.

- *Enumeration.* The operating system provides a means for other applications to enumerate the registered components.

In the realm of solvers, the parallel task to selecting an object to insert into a document is selecting a solver to solve a particular problem. A fundamental prerequisite for solver selection is the ability to determine what solvers are installed on a system. This implies the existence of some mechanism for querying existing solvers on the system. In existing systems, client applications might maintain their own list of installed solvers. A solver might have to register with each particular client application in which it can work. Add a new client application, and it must either import another application's database of solvers, or else the solvers will have to be reconfigured to register with the new client.

For solvers of the future, the management of the solver database is best handled by a global service, so that a solver is available to all potential clients. Solvers do not add and remove themselves directly from the database (this incurs too much implementation dependency). Instead, solvers can register and unregister themselves with a solver registration service. This

service is the only component that needs to understand the actual structure of the solver database, minimizing the coupling between components.

Consider compound documents—a new document object component can register with the operating system. Every application that can insert these objects can then see that this new object is available. Solvers should work this way, too.

2.5.2 Dimension and type support

As discussed earlier (section 2.3.6, page 85), dimension and type manipulation and conversion is desirable and useful for every solver, but no solver should have to implement that functionality. Dimension conversion is orthogonal to the purpose of any solver. Instead, specialized dimension manipulators can handle the conversion process, and solvers can use these manipulators as if they were clients using other solvers.

So, just as a client should be able to enumerate the solvers installed on a system, it should be possible to enumerate the available dimension manipulators on a system. This suggests a specialized registry of dimension manipulators. Such a registry would have all of the features listed in the previous section, as well as some additional functionality.

There are essentially two requests a solver will have of a dimension manipulator. First, are two dimensional statements equivalent? For instance, is meters per second squared equivalent to miles per hour per second? (No without conversion; yes, given appropriate conversions.) Second, what is the factor to convert from one dimensional statement to an equivalent one? For example, to convert from meters per second squared to milers per hour per second, multiply by 2.2369. While a specific dimension manipulator can answer these questions, the core services must help the solver or client find that dimension manipulator among the many available. For instance, one dimension manipulator might be able to convert within the metric system, while another might be able to convert from the metric system to other systems. Only the latter is suitable in this example. Beyond registration and enumeration, therefore, the core services must provide a more intelligent query mechanism for quickly finding or assessing a dimension manipulator.

2.6 Conclusion

This chapter presented the requirements for a modeling framework of the future. Specifically, based upon a domain analysis and characterization of the roles of activities in observed industry solutions, it detailed a set of desirable features that solvers can implement to make them easier to use and more reusable in the context of applied operations research solutions. These features are partitioned as those that are necessary and useful for solvers that are used in single-stage architectures and those that are necessary and useful for solvers that are embedded in networks of solvers. Some features of the framework exist outside the scope of any individual solver and should reside in a centralized, core group of services provided by the operating system.

This chapter did not address requirements of the database, the source of input data for any solution. The scope, scale, complexity, and capabilities of existing database connectivity protocols, such as SQL, ODBC, and OLE DB are sufficiently powerful for the needs of the framework. The framework will present what it needs from data sources for specialized solver purposes, with the expectation that security, administration, table definition, and other capabilities are leveraged from one of the existing standards.

Likewise, this chapter did not address requirements of the client, except in the context of how the client might react to specific requirements of a solver. This is in order not to impose any limits upon the client. There is much existing research into model selection, model creation, solver selection, data manipulation, model presentation, and model life cycles that already addresses client-side issues. This chapter has instead focused on the interaction between the solver and its client. Chapter 3 will develop the framework following the structure presented here.

Left blank intentionally, this page.

CHAPTER THREE

FRAMEWORK

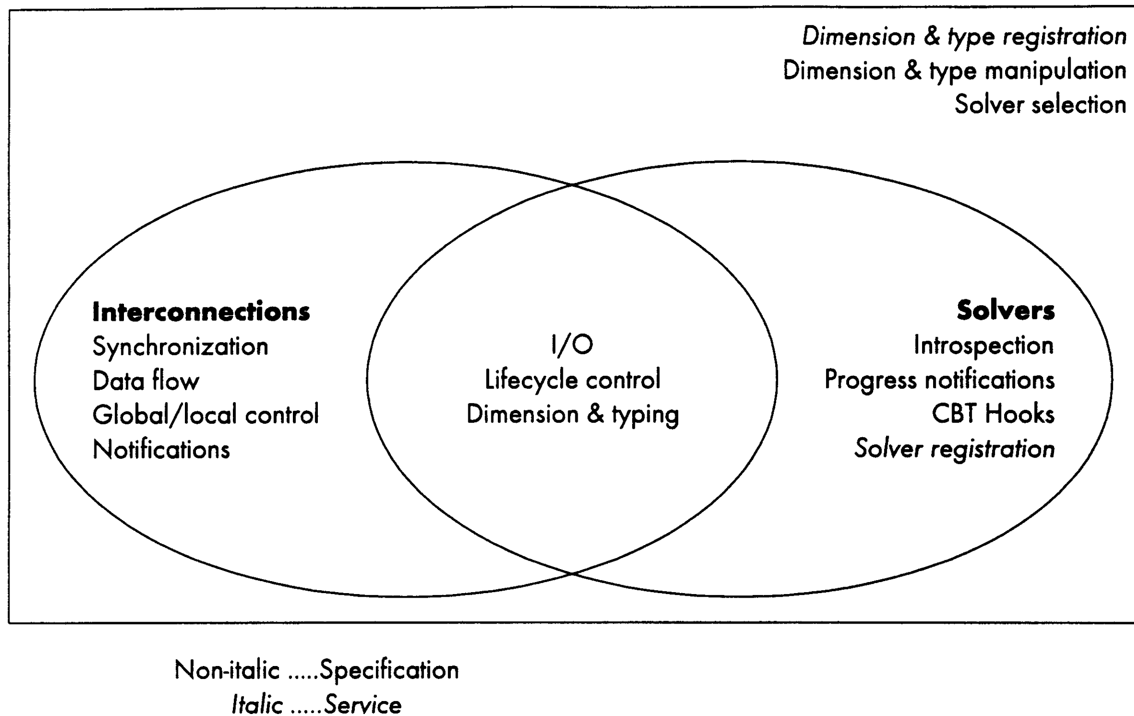
Chapter Two presented a guide of requirements for applied operations research solution implementations. This chapter presents one possible realization of a framework that satisfies many of those detailed requirements. According to Booch [7], a framework is a "collection of classes that provide a set of services for a particular domain." In the context of Microsoft COM, a framework is primarily a collection of interfaces, specification of behavior, and associated class implementations that provide a set of services for a particular domain.

The chapter begins by surveying the organization and scope of the framework. After a discussion of some implementation details, the role and behavior of data within the framework is described. Then, various issues relating to solvers as single entities are tackled, including the basic solver interfaces, introspection, progress updates, and life cycle control. Following this is the presentation of those aspects of the framework that specify the interaction of solvers in the more complicated solution architectures, including the basics of networking solvers, the components that enable networking solvers, and the resolution of global and local workflow control. The description of the core services rounds out the framework specification.

3.1 ORGANIZATION OF THE FRAMEWORK

The modeling framework will consist of a collection of numerous services and specifications. A *service* in this context is a library of code that provides system-level functionality. A *specification* is a protocol for behavior between one component and another. Specifications are only instructions for behavior, and do not include any code. Services, on the other hand, are implemented functions that conform to specifications.

The services and specifications support two different aspects of implementation: the solvers and the interconnections between solvers. Some services and specifications address both aspects, and some provide general capabilities. This diagram presents the classification of the services and specifications as a Venn diagram, in Figure 3.1.



Non-italic .....Specification
Italic .....Service

Figure 3.1: Overview of services and specifications of the framework

The following sections provide brief descriptions of each item.

3.1.1 General services and specifications

Dimension and type registration service. The dimension registration service registers (or removes) new dimensions and dimension manipulators with a system-wide database. Any solver or modeling environment that needs to convert numbers from one dimension system to another might use the dimension registration service to find a conversion component that can convert between the desired dimensions. Types express what a value is measuring. Their behavior is similar to that of dimensions, except that dimensions are typically universal quantities (1000m = 1km), whereas types are more flexible depending on the situation (1 car = 1800 pounds of steel, or perhaps 1 car = 1700 pounds of steel). Type registration behaves like dimension registration: a known type (car) might be associated with a type manipulation component that can convert that type to other types.

Dimension and type manipulation specification. The dimension manipulation specification defines a standard protocol for manipulating dimensions. Dimension manipulations include multiplication, comparison, and conversion. A component that implements this specification can manipulate a self-defined set of dimensions. Different components might manipulate different dimensions. Using the dimension registration service, a dimension (such as "hectares") might be associated with a dimension manipulation component that knows how to work with that dimension (such as converting to square meters). The type manipulation specification defines a standard protocol for manipulating types. Components that implement this specification are called "type manipulators" and can operate on a self-defined set of types. Using the type registration service, a type (such as "cost of car production") might be associated with a type manipulator that can work with that type.

Solver selection specification. The selection problem is to determine which model or solver is best suited to model or solve a particular problem definition. Algorithms for selecting solvers or model should function independently of the interface presented to the user and independently of the underlying database implementation that manages the known models and solvers. The solver selection specifications define protocols that separate the selection problem from the presentation of selection, via user interfaces, and from the database of solvers. The database—the solver registration service, described later—implements part of this specification for enumerating available solvers. Selection components, the components that encapsulate the logic, knowledge, or artificial intelligence of actually selecting solvers, implement the remainder of the specification.

3.1.2 Solver services and specifications

Introspection specification. The introspection specification, implemented by a solver component, allows the client of that component to examine the requirements, inputs, outputs, conditions, and source of the solver. For instance, a client can determine the author and citation of the algorithm, its expected or worst-case run times, and perhaps pre- and post-conditions. It can also query the capabilities of the solver, including what types of numbers if manipulates (integers, doubles, fixed-point), whether it can check input-validity, and how interactive it is.

Progress notification specification. The progress notification specification defines a protocol for enabling a solver to inform its client on its progress during its execution. This is a callback notification, where the solver *calls back* into the client periodically. The client can also instruct the solver to pause or cancel its processing. This specification makes it simple to create user interfaces with a progress bar that fills from 0% to 100% as the solver executes, and to allow the user to cancel the processing.

CBT hooks specification. CBT (Computer-based training) hooks provide a way for educational applications to tie into the inner-workings of a process. In the case of solvers, CBT hooks might be inserted at every iteration of an algorithm or whenever a residual network is updated. When used, the hooks would allow a program to track each step in the algorithm, making it possible to visually display each change as it happens. These hooks are a

specialization of a broader notification system, but are targeted specifically for educational purposes.

Solver registration service. The solver registration service enables a newly installed component to register itself with the system. Once registered, the solver is visible to client applications when they query the registration service for installed solvers. Also, the model/solver selection specification uses the solver registration service to enumerate available solvers.

3.1.3 Interconnection services and specifications

Synchronization specification. The synchronization specification defines the protocol for synchronizing activities between two components. When one solver uses another solver during its processing, it must understand the synchronization requirements of the embedded solver. A solver might require synchronous same-thread operation, which means that the calling solver is suspended until the embedded solver is finished. Or a solver might run asynchronously, meaning that the calling solver continues executing and that the embedded solver will notify the calling solver when it is finished. The two solvers essentially perform a "handshaking" at initialization to determine a compatible synchronization choice for their communications.

Data flow specification. Whereas synchronization manages the behavior when one solver passes control or sends events to another, data flow manages the requirements and availability of input and output data for solvers. In directed acyclic graph and more complex architectures, the availability of data is an essential concern. A solver must not execute before its inputs are ready. It must not change its outputs before all other solvers that use its outputs have had the chance to read the previous outputs. The integrity of data must be preserved. The data flow specification outlines a protocol for ensuring that the data flow requirements are satisfied among multiple solvers.

Global/local control specification. The workflow control specification specifies a protocol for managing the interconnections among various solvers in a complex architecture. While data flow manages the availability of inputs and the preservation of outputs, workflow control manages the decisions and flows of control in the network of solvers.

Notification specification. The notification specification provides a simple, standard way for almost any event of interest to be monitored and captured. In a traditional publish-and-subscribe framework, solvers publish a list of events that they monitor during their execution and for which they will provide hooks, and clients subscribe to events of interest. When these events occur, the solver notifies all subscribed clients that the event occurred, and the client can execute code that depends on that event. This is a generalization of the progress notification specification, which is a fixed notification mechanism.

3.1.4 Intersecting services and specifications

The specifications in the intersection of the Solvers and Interconnections domains of Figure 3.1 are applicable both to solvers as individual entities and to networks of interacting solvers.

I/O (solver interaction) specification. The input/output specification standardizes the method of providing inputs to and retrieving outputs from a solver.

Control specification. The control specification defines the basic control capabilities for a solver. These capabilities include pausing, stopping, restarting, and changing the scheduling priority of a solver's execution.

Dimensioning and typing specification. The dimensioning specification defines an optional protocol that a solver can implement to expose dimensioning capabilities. A component that implements this specification can work with inputs and outputs that have dimension information. The typing specification defines an optional protocol that a solver can implement to expose typing capabilities. A component that implements this specification can work with inputs and outputs that have type information.

3.1.5 Framework entities

Within the context of the domain analysis in section 2.1, page 52, and the identification of the participants in the solution process in section 2.2, page 70, the framework defines a collection of entities that form the basis of applied solutions within the framework. Each entity serves a purpose within one or more of the specifications identified in the previous sections. The entities and their roles within the protocols (specifications) are summarized in Table 3.1. Not all of the protocols are covered by these entities; this is because the framework in its current form does not address some of the issues, like dimension and typing and CBT hooks. These limitations are discussed in detail in section 5.2, page 272. Some of the entities are certainly intuitive, such as solver, set, and client. The others are each introduced, described, and detailed in the sections that define their related protocols.

Of all of these entities, only the Solver must be implemented by developers on a regular basis. The SolverRegistrar, DataAdviseHolder, SolverAdviseHolder, and DescriptionProp-Manager are each part of the core services (see section 3.9, page 214). While not technically part of the core services, the data flow entities and the networking entities have default, canonical implementations that are provided by the same libraries that implement the core services. So, objects like Data Element, Inbound Solver Site, and SolverInfo, all have a default, generic implementation that is suitable for most needs.

Protocol (section)	Entity	Purpose
Data flow (3.4)	Set	Manage set of items in the model
	Dimension	Manage dimensions of a vector or matrix
	Data Element	Manage collection of values: scalar, vector, matrix, etc.
	Set Factory	Create a Set
	Dimension Factory	Create a Dimension
	Data Element Factory	Create a Data Element
	Data Source	Provide and generate Data Elements
	DataAdviseHolder (part of core services)	Manage advise list to simplify Data Element implementation
	client of Data Source	Receive notifications from Data Source
Solver interaction (3.5)	Solver	Transform data and solve problems
Introspection (3.6)	Solver	Provide access to SolverInfo object
	SolverInfo	Provide introspection about structure and qualities of Solver
	DescriptionPropManager (part of core services)	Manage registry of description properties
Progress updates / Life cycle control (3.7)	Solver	Send notifications and provide status
	SolverAdviseHolder (part of core services)	Manage advise list to simplify Solver implementation
	client of Solver	Receive notifications from Solver
Networking interconnections (3.8)	Inbound Solver Site	Wrap inbound side of Solver, including inputs and suppliers of Solver
	Outbound Solver Site	Wrap outbound side of Solver, including outputs
	Mapping	Wrap a connection from one Solver to another, managing flow of data
Solver registration and selection (3.9.1)	SolverRegistrar (part of core services)	Register and unregister solvers and their SolverInfo objects

Table 3.1: Summary of framework entities

3.1.6 Framework interfaces

The framework is based on Microsoft COM. Entities and objects in COM are characterized predominately by their behavior as defined by the interfaces that they support. Hence, the majority of the framework specification is the definition and description of interfaces. In fact, it is not so much the entities that satisfy the framework requirements or define the various protocols and specifications as it is the various interfaces defined by the framework. Entities are in turn defined by the interfaces that they support. Any object can behave as any entity; when a particular interface on an object is used, that object is playing the role defined by that interface. For instance, the solver interaction protocol comprises six interfaces, including IRSolver, IRSolverInputs, IRSolverOutputs, and IRSolverParameters. Any COM object that implements at least these four interfaces is technically a solver within the

framework. Whenever a client uses the IRSolverInputs interface on an object, that object is playing the role of a solver. In fact, different classes of objects may support this interface, notably the Solver and the Inbound Solver Site entities.

Tables 3.2 through 3.7 summarize the interfaces defined by the framework, organized by specification/protocol and entity.

Entity	Interfaces	Purpose	Section (pg)
Set	IRSet	Access a set's values	3.4.2.1.a (121)
	IRSetModifier ‡	Modify a set	3.4.2.1.b (122)
	IRSetClone ‡	Clone a set	see reference
Dimension	IRDimension	Access a dimension	3.4.2.2.a (124)
	IRDimensionClone ‡	Clone a dimension	see reference
Data Element	IRDataElement	Access a data element's values and data source	3.4.2.3.a (125)
	IRDataElementClone ‡	Clone a data element	3.4.2.3.b (126)
	IRDataElementAccess ‡	Access a sub-element	3.4.2.3.c (126)
	IRDataElementScalar ‡	Optimize scalar access	3.4.2.4.a (128)
	IRDataElement1 ‡	Optimize vector access	3.4.2.4.b (128)
	IRDataElement2 ‡	Optimize matrix access	3.4.2.4.c (129)
	IRDataElementProperties ‡	Access properties	3.4.2.5.a (130)
	IRDataElementProperties-Change ‡	Modify properties	3.4.2.5.b (131)
Set Factory	IRSetCreator	Create a set	3.4.2.1.c (123)
Dimension Factory	IRDimensionCreator	Create a dimension	3.4.2.2.b (124)
Data Element Factory	IRDataElementCreator	Create a data element	3.4.2.3.d (128)
DataAdvise-Holder	IRDataAdviseHolder	Manage data advise list	3.4.3.3 (135)
Data Source	IRDataSource	Access a data source	3.4.3.1 (133)
client of Data Source	IRDataAdvise	Receive data change notifications	3.4.3.2 (135)

Table 3.2: Data element entities and interfaces (‡ = optional interface)

Entity	Interfaces	Purpose	Section (pg)
Solver	IRSolver	Control a solver	3.5.2 (139)
	IRSolverAsynchSolve ‡	Asynchronously control a solver	3.5.2.1 (140)
	IRSolverInputs	Access and set inputs	3.5.3 (140)
	IRSolverOutputs	Access outputs	3.5.4 (140)
	IRSolverParameters	Parameterize a solver	3.5.5 (141)
	IRDataSource	Access solver data source	3.4.3.1 (133)

Table 3.3: Solver interaction entities and interfaces (‡ = optional interface)

Entity	Interfaces	Purpose	Section (pg)
Solver	IRSolver	Establish connections	3.7.1.1 (170)
	IRSolverStatus ‡	Retrieve status and state	3.7.2.3 (176)
	IRSolverControl ‡	Asynchronously control a solver	3.7.3.2 (177)
SolverAdvise-Holder	IRSolverAdviseHolder	Manage update advise list	3.7.1.1.b (170)
client of Solver	IRSolverAdvise	Receive solver notifications	3.7.1.2 (173)

Table 3.4: Progress updates and life cycle control entities and interfaces (‡ = optional)

Entity	Interfaces	Purpose	Section (pg)
Solver	IRSolverProvideInfo ‡	Access SolverInfo	3.6.1.1.h (149)
	IRSolverParametersInterface ‡	Access parameter dispatch interface	3.6.1.4.b.2 (156)
SolverInfo objects	IRSolverInfo	Access solver structure	3.6.1.1.b (145)
	IRSolverInputInfo	Access input information	3.6.1.1.c (146)
	IRSolverOutputInfo	Access output information	3.6.1.1.d (148)
	IRSolverParamInfo	Access parameter information	3.6.1.1.f (148)
	IRSolverSetInfo	Access set information	3.6.1.1.e (148)
	IRSolverDimInfo	Access dimension information	3.6.1.1.g (149)
	IRSolverDescription	Access specific solver description properties	3.6.2.1.a (162)
	IRSolverDescriptionProperties	Enumerate all solver description properties	3.6.2.1.b (163)
DescriptionProp-Manager	IRDescriptionPropRegistration	Register description prop.	3.6.2.2.a (164)
	IRDescriptionPropEnumeration	Enumerate registered description properties	3.6.2.2.b (165)

Table 3.5: Introspection entities and interfaces (‡ = optional)

Entity	Interfaces	Purpose	Section (pg)
Solver Registrar	IRSolverRegistration	Register new solvers	3.9.1.1 (215)
	IRSolverEnumeration	Enumerate registered solvers	3.9.1.2 (215)

Table 3.6: Solver registration and selection entities and interfaces (‡ = optional)

Entity	Interfaces	Purpose	Section (pg)
Inbound Solver Site	IRSolverSiteIn	Access inbound solver site	3.8.4.1 (189)
	IRSolverSiteInMappings	Manage mappings	3.8.4.2 (190)
	IRSolverInputs	Wrap solver inputs	3.5.3 (140)
	IRMappingAdvise	Receive notifications from mappings	3.8.4.3 (190)
	IRSolverSiteInSolverFactory ‡	Manage solver creation	3.8.4.4 (191)
Outbound Solver Site	IRSolverSiteOut	Access outbound solver site	3.8.5.1 (193)
	IRSolverOutputs	Wrap solver outputs	3.5.4 (140)
	IRDataSource	Wrap solver data source	3.4.3.1 (133)
	IRSolverAdvise	Receive notifications from solver	3.7.1.2 (173)
Mapping	IRSolverMapping	Access mapping	3.8.6.1 (199)
	IRSolverMappingMechanism ‡	Specify mapping mechanism	3.8.6.3 (202)

Table 3.7: Networking and interconnections entities and interfaces (‡ = optional)

3.1.7 Framework reference

All of the interfaces, structures, enumerations, and constants are defined and described in a framework reference manual available as an MIT Operations Research Center working paper by Ruark [92]. A current version of the reference manual will always be available at the author's web site (see the section "About the Author," page 14).

3.2 The Solver Executable

The problem of making a solver an executable application (section 2.3.1, page 73) is easily solved. Within the framework, all solvers are COM objects. By their very nature, all COM objects are executable. Thus, all solvers are executable.

A primary issue when developing a solver is whether to create a stand-alone application (an EXE) or a dynamic link library (a DLL). In the context of COM, the question is whether to create a local server (EXE) or an in-process server (DLL). The primary advantage of the local server is robustness; if a client, and hence the client's in-process servers, crashes or parties on its memory space the solver is not brought down with it. Another advantage is that all instances of the solver can exist in a single memory space, which perhaps can be leveraged. The primary disadvantage of a local server is that every method call to the solver must cross process boundaries, which induces marshaling and can take time. The primary advantage of the in-process server is speed, and the primary disadvantages are lifetime dependency on the client application and lack of the robustness found in local servers. For a more complete exposition of choosing a server type, see Box [8].

3.3 Solver Interfaces

As was discussed in section 2.3.2, page 75, COM is a good choice for an object model foundation because of the popularity of the Windows family of operating systems in small- to medium-sized solution environments. By choosing COM, the framework can target a wide array of host and development environments that already support Microsoft's object model. Yet, even with the decision to use COM, its long and storied past does not make it trivial to define a single interface that is ideal for all COM environments.

Consider Visual Basic, arguably one of the most popular development environments in use today [71]. Through version 4.0, Visual Basic could only manipulate COM objects that supported automation, via IDispatch. IDispatch is a standard COM interface that defines a late-bound dispatching mechanism. The original automation framework, from type libraries to standard automation marshaling and dispatch support, derived from the Visual Basic interpretation and implementation of what was at the time known as OLE Automation. As a result, modern COM servers are still limited in what types of parameters their interfaces can support to still be labeled automation-compatible. Visual Basic 5.0 can deal with more types of interfaces[1], but still has a restriction on the data types it can access in COM objects because of an underlying restriction in the fundamental Visual Basic data types[2].

On the other hand, most of the standard COM interfaces are based on C parameters and implementations, especially parameters that are not accessible to Visual Basic. Any interface that uses an IID, CLSID, or DWORD is problematic for Visual Basic[3]. And such interfaces abound: IClassFactory, IPersist and its subclasses, IViewObject, and even IUnknown. These interfaces are easy enough to declare in C or C++, but nearly impossible to declare or implement in Visual Basic. Defining these interfaces to be automation compatible would have severely crippled any performance gains C and C++ could provide. If these had been dispatch-only interfaces, COM would never have become as successful as it has.

All of the interfaces mentioned in the previous paragraph are crucial for creating ActiveX controls. Starting with Visual Basic 5.0, ActiveX control creation is perhaps simpler in Visual Basic than in C++. How does a Visual Basic application implement IOleControl, IViewObject, and other interfaces that are not automation compatible? The Visual Basic code that the developer writes does not implement these interfaces at all. Instead, the Visual Basic development environment and run-time executable layer implement these interfaces with a standard implementation, and call into the user's Visual Basic code when necessary

---

[1] Namely, Visual Basic 5.0 can now use and implement interfaces that have automation-compatible types even if they do not derive from IDispatch. Furthermore, VB 5 can use but not implement interfaces that use user-defined structures of automation-compatible types.

[2] Visual Basic does not support unsigned integers; so while unsigned long is valid for COM interfaces, Visual Basic cannot use it.

[3] Hack workarounds are known; for example, see Brown [12].

for customized activity. That is, the Visual Basic run-time wraps the user's automation-compatible code with the necessary custom routines to implement the COM interfaces that define an ActiveX control. It looks something like this (Figure 3.2):

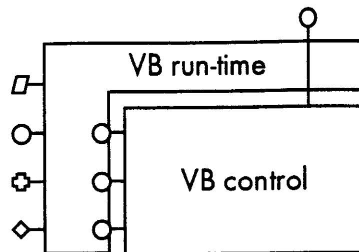


Figure 3.2: How the VB run-time wraps controls to expose complicated interfaces

The Visual Basic for Applications 5.0 environment, used by Microsoft Office 97, has similar capabilities to Visual Basic 5.0, although it cannot implement non-IDispatch-derived interfaces. VBA should be viewed as a powerful scripting engine rather than a hosting environment.

The future looks brighter, with the advent of COM+, the successor of COM, theoretically making interoperability with more interfaces a reality. However, in the meantime Microsoft has taken a step backward with its simpler scripting languages, VBScript and JScript. These two languages, designed to compete with Netscape's JavaScript for scripting web pages, and now a part of the Windows Scripting Host, are throw-backs to early Visual Basic generations. These interfaces only understand IDispatch interfaces in the most late-bound fashion. If an object does not expose functionality through IDispatch, then VBScript cannot access it. Because of a marshaling "feature" for IDispatch interfaces, an object that will be accessed from different apartments (i.e., must be marshaled) can only expose one IDispatch-derived interface reliably. The usual fix is to expose as many custom interfaces as is logical and necessary, and then to group all of those interfaces' features into a single, massive IDispatch-derived interface.

How does this jumble of languages and capabilities impact the design of the framework?

The framework takes an approach similar to Microsoft's. All of the main interfaces defined by the framework are custom interfaces (i.e., for C and C++), and most of them use parameters that might make Visual Basic uncomfortable. These interfaces are designed to be used by the solver implementation and the guts of client applications.

Components provided by the core services will be available to map from these custom interfaces to the more accessible automation compatible interfaces. So, a solver implementation might aggregate or defer to a core services component that maps IDispatch calls into the actual solver interfaces. These core services components serve the same

function as the parts of the Visual Basic run-time that map ActiveX control interfaces into suitable Visual Basic functions and subroutines.

The expectation is that these interfaces are the low-level guts of solvers, clients, and operational solutions, and that more sophisticated graphical or scripting clients that have specific vertical domain mappings will hide these interface details and present to the ultimate user a more accessible, even drag-and-drop interface.

Why do all the interfaces start with IR?

Every interface in the framework begins with the letters "IR." In fact, every named item in the framework follows some loose conventions. Because most everything in the Windows programming world exists in the global namespace, there is a problem ensuring uniqueness of names. (This is why GUIDs were designed.) The conventions in the framework are based on Microsoft's conventions for COM.

For instance, in COM, all standard interfaces begin with the letter "I," for interface. Hence, IOleControl is the interface for ActiveX controls (formerly OLE controls). C++ classes in most of Microsoft's samples, and even in the symbol set for much of the Windows NT core, begin with the letter "C" for class. Hence, COleControl might be a C++ implementation of the IOleControl interface.

The framework uses the letter "R" as an identifying mark[4]. The framework typically does the following. First, place the letter "R" in front of the name of an object. Then, following Microsoft's conventions for that type of object. Hence, a data element object is "RDataElement," and therefore has as its interface IRDataElement and might be implemented in a class CRDataElement. Constants are often all caps, and always have a preceding "R." Similarly, library functions also begin with "R."

The presence of the "R" in interfaces, library functions, classes, constants, serve to notify the reader that the particular name is part of the framework as opposed to part of COM proper. In situations where this is not the case, this will be mentioned.

Remember that in COM, interfaces are truly identified only by their IID (interface ID, the same as a GUID), and their textual names serve as convenient tools for humans.

3.4 Data Flow

All solvers have at least one data input or one data output. The flow of data between client and solver and within a network is an integral part of any applied solution. Naturally, data creation, manipulation, and observation is a complex topic. There are many existing

[4] From the author's name.

standards for accessing data, including SQL, ODBC, JDBC, DAO, and OLE DB, as well as a plethora of APIs for each particular database product, OLAP package, and DSS application. The multitude of problems in working with complex data sets, such as data security, concurrency and locking, transaction management, performance, reliability, and optimization, are not only beyond the scope of this thesis but also fortunately orthogonal to the use of the data within an applied solution and the framework[5].

One area where most modern standards are deficient, for the purposes of operations research solutions, is multidimensional analytic processing. That is, most database interactions are based on a rowset philosophy, where each row in a table is an individual entity, not necessarily related to other rows in the table. Rowsets are, by their nature, mappings from one dimension (the index of the row) into many dimensions (the value in each column of a row); that is, rowsets are useful for vector storage. Imposing a multidimensional interpretation upon rowsets usually requires specifying some columns in the table as dimensional columns and other columns as data columns. The dimensional columns define a domain lattice, and each row is a potentially non-zero element within that lattice. Fortunately, various organizations have realized the importance of multidimensional analysis in DSS, and a recent area of standardization is OLAP, on-line analytical processing (see McCright [65]). These efforts will eventually lead to standard interfaces and APIs for retrieving multidimensional data from a flat rowset schema.

Another area of deficiency is the simple manipulation of data structures. Granted, corporate database are complex creatures, and deserve correspondingly complex interfaces. However, in many cases an analyst wants simply to query a vector or a matrix from a database, and have access to that vector or matrix in the simplest of terms; that is, by row and column index. Databases are not geared towards such simple requests.

In general, solvers should not be aware whether they are working with data from a database, a file, a spreadsheet, or the screen. Solvers should simply work with a vector or a matrix and leave the implementation details of the vector or matrix to a specialized data component.

A solution is to define a standard set of interfaces that expose the functionality of a vector, matrix, or higher-dimensional lattice, and to develop components that know how to manipulate the various sources of data, such as databases, files, and spreadsheets, and that expose the standard interfaces. The solvers then use the standard interfaces, oblivious to where the data is located or how it is retrieved.

This technique is precisely that used by Microsoft's current data standard, OLE DB (see Dyck [25]). In OLE DB, any application or data source that can provide data to someone is a data provider. Special data provider components expose a standard set of data provider interfaces and know how to map their data sources into these provider interfaces. Applications that use data are data consumers, and expose their own set of consumer

---

[5] That is, with the simplifying assumption that the framework does not capture real-time events or real-time architectures.

callback interfaces. OLE DB is a COM standard, and so would nicely fit into the framework, except that OLE DB lacks multidimensional support. Microsoft is working on an OLAP standard that extends OLE DB, called OLE DB for OLAP [65, 72].

The framework therefore presents a data specification that is minimally sufficient for the needs of the framework. When necessary, components that use databases to access their data might leverage OLE DB or ODBC or another data standard. The actual manipulation of data elements, in the operations research sense, occurs through the interfaces presented here. The beauty of the interface discovery capability of QueryInterface is that as more complex, complete, and eventually useful data interfaces are created or adopted, clients and suppliers can discover this while always being able to fall back on the original interfaces when necessary.

3.4.1 Data structure in the framework

There are two primary pieces of the data puzzle in the framework: data elements and data sources. A *data element* is an object that contains actual data, be it a vector, matrix, scalar, or some other entity. Every vector, matrix, and scalar in a solution can have a corresponding data element. Data elements provide read and sometimes write access to the original source of the data values themselves.

A *data source* is an object that provides data elements. That is, a data source serves data elements to clients that request them. For instance, a database application is a data source, and the results from a query are a data element. A data source might be responsible for managing multiple data elements, where a data element is associated with a single data source. Almost any object can be a data source. In particular, the outputs of a solver are data elements; hence, the solver must be a data source.

In many cases, the data source and data element might be rolled into the same object. The following sections describe the interfaces and responsibilities of data elements and data sources.

3.4.2 Data elements

A data element is a single data entity in the solution model. It can be multidimensional, a vector, or a scalar. Its values can range from numbers to strings to potentially other objects. The basic assumptions of a data element are these:

1. The data element has a fixed, non-negative, integral dimensionality.

2. For each dimension of the data element there is a finite set that defines the possible values for that dimension. Example sets include 0...20, { Sunday, Monday, Tuesday, Wednesday, Thursday, Friday, Saturday }, { red, blue, green }, and { Somerville,

Mountain View, Muswell Hill }.The cardinality of the set determines the size of the dimension.

3. The sets must have a mapping from their actual values (as strings, colors, or cities) into the integers, and this mapping must not change during the course of a solution. That is, if New York maps to 12, it must always map to 12 even when other items are removed or added to the set while solving a problem. The next time the solution is executed, these mappings might change.

4. Every item in the data element is of the same type. That is, every item is a double, or every item is a long. The items do not need be stored contiguously in memory in any fashion, however.

In essence, a dimension is based on a set, and a data element is based on multiple dimensions. Their relationships are shown using UML below (Figure 3.3):

Figure 3.3: Relationships of data elements, dimensions, and sets

Before discussing data elements themselves, the interfaces for sets and dimensions must be defined.

3.4.2.1 Sets

Sets as defined by the framework are very simple entities. The primary purpose of a set in the framework is to map the name of some entity into an ordinal value that can be used for data lookup within a dimension and data element.

A set has two primary interfaces. One, IRSet, is for accessing set values and mapping values to ordinals. The other, IRSetModifier, is for modifying a set. An additional interface, IRSetCreator, is exposed by factory objects that create sets.

3.4.2.1.a Interface IRSet

IRSet is an accessor interface. The set object cannot be changed with this interface. Hence, for read-only sets it is sufficient to implement only IRSet. It contains the following methods:

GetName. Returns the name of the set.

GetCount. Returns the cardinality of the set.

GetItem. Given an index from zero to one less than that returned by GetCount, returns the value of that item in the set. Note that the index here is not the same as the ordinal value used for lookup in the dimension and the index. GetItem is used for enumerating the values in a set, one at a time.

IsMember. Given a set value, determines whether that value is in the set or not.

MapToOrdinal. Given a set value, returns the ordinal value for that set value. That is, the set value "New York" might map into ordinal value 12.

GetItemByOrdinal. Given an ordinal value, returns the set value that has that ordinal value.

GetSetBounds. Returns the lowest and highest ordinal values used by the set. The bounds can be used to create a vector for a dimension.

The interface also contains three methods for managing a lock on the set. These are **LockSet, UnlockSet,** and **SetLocked,** which lock the set, unlock the set, and return whether the set is locked or not, respectively. The set cannot be modified while it is locked.

The easiest mapping, of course, is to have the index of an item be the same as the ordinal value of the item. However, in cases where a set might change during the course of a single solution execution, this might not be possible. For instance, suppose before executing a solver, the ordered set is { NY, AZ, MA }, so that MapToOrdinal("NY") returns 0, MapToOrdinal("AZ") returns 1, etc. Similarly, GetItem(0) returns "NY" and GetItem(1) returns "AZ." Then, suppose after running the solver, "NY" is removed from the set, which now becomes { AZ, MA }. Then, MapToOrdinal("NY") returns an error, MapToOrdinal("AZ") still returns 1, and MapToOrdinal("MA") still returns 2. However, GetItem(0) now returns "AZ," and GetItem(1) now returns "MA."

3.4.2.1.b Interface IRSetModifier

Sets that support IRSetModifier permit modification of the set values. Usually, sets will be created, filled with values, and then not changed during the course of solution execution. IRSetCreator, below, handles the creation process, and IRSetModifier handles filling the set with values. It has the following methods:

Insert. Given a set value and an optional index, inserts the value into the set at the specified index. Note that when possible, the set might assign the ordinal value for the set value to be the same as the index, but this is not a requirement. Without the optional index, this is the normal function to call while building a set.

Set. Given an index, changes the set value at that index to a new value. The new set value takes the existing ordinal value of the old set value. Note that it is possible to maliciously destroy the set assumptions by swapping values in this manner.

Change. Given an old set value and a new set value, replaces the old with the new. Essentially, the set object searches for the old set value to find its index and then calls Set.

Clear. Empties the set.

Remove. Removes a set value from the set.

RemoveAt. Removes a set value, as specified by its index from zero to one less than that returned by IRSet::GetCount, from the set.

Generally, clients will create and change sets, and solvers will just pass them around. By design, sets will clear themselves whenever they are deleted; hence, it is not necessary to call Clear prior to releasing a set. Note also that there is no method for assigning the name of the set; this happens when the set is created, as described next.

3.4.2.1.c Interface IRSetCreator

The generic factory interfaces in COM, IClassFactory and IClassFactory2, create objects with a default state. There is no way to use these factory interfaces to initialize a new object with a custom state. Various schemes exist for creating objects with a custom state, including using any of the IPersist interfaces, using monikers, or using class objects. The solution adopted by the framework is to specify a custom factory interface, IRSetCreator, that enables the creation of a set with predetermined state. It has a single method:

Create. Given the name of a new set and the type of items stored in the set, creates a set with that name and those types of items, and returns it to the client.

A client can create a set by first creating a SetCreator object and acquiring its IRSetCreator interface. Then, the client creates the set itself by calling Create on the SetCreator object. Typically, the SetCreator and the resultant set will be implemented by the same executable, so that the process space, apartment, and other run-time characteristics of the SetCreator are adopted by the set when it is created.

The pattern of having a custom factory interface recurs with both dimensions and data elements.

3.4.2.2 Dimensions

Dimensions are similar to sets in their simplicity. The reason for having sets separate from dimensions is that a dimension is associated with a single set, whereas a single set might be associated with many dimensions. Once created, dimensions cannot be modified, although the underlying set might be. Dimensions therefore have only accessor and creator interfaces, IRDimension and IRDimensionCreator.

3.4.2.2.a Interface IRDimension

Dimensions do not have many attributes of their own.

GetName. Returns the name of the dimension. This can be different from the name of the set or any data element.

GetBounds. Returns the lowest and highest ordinal values used by the dimension. Often this delegates to IRSet::GetSetBounds.

GetSet. Returns the set associated with this dimension.

3.4.2.2.b Interface IRDimensionCreator

IRDimensionCreator, the factory interface for creating dimensions, has two creation methods. One creates a new dimension based upon an existing set object. The other creates a new dimension based upon a contiguous range of integers without requiring the pre-existence of a set object.

CreateFromSet. Given a string and a set object, creates a dimension with the name equal to the string and with the associated set determining the bounds and meaning of the dimension's values.

CreateFromRange. Given a string, a lower bound, and an upper bound, creates a dimension with the name equal to the string and with bounds from the lower bound to the upper bound. The dimension object must implement an implicit set object, which otherwise has no independent meaning.

3.4.2.3 Generic data element interfaces

To qualify as a data element within the framework, an object must support the IRDataElement interface. This interface provides the basic functionality of all data elements, such as accessing the name of the data element, accessing the element's dimensions, and generically accessing the values in the element. Most data elements will want to support the interface IRDataElementClone, which provides for making a copy of the data element. A less common interface, IRDataElementAccess, provides sub-element access. The factory interface for data elements is IRDataElementCreator; it parallels IRSetCreator and IRDimensionCreator, described above.

There are other data element interfaces that do not apply to every type of data element. For example, a vector implementation might support IRDataElement1, which provides vector access. IRDataElement1 would not be appropriate for a generic data element. These interfaces are presented in section 3.4.2.4, page 128.

3.4.2.3.a Interface IRDataElement

Interface **IRDataElement** is the fundamental data element interface. This is a required interface for data elements. Through this interface, clients can access the dimensions of the data element, its data source, and its name. A client can also get and set (if supported) values in the data element in a very general way. It has the following methods:

GetName. Returns the name of the data element.

GetDimensionCount. Returns the dimensionality of the data element; this is the number of dimension objects associated with the element.

GetDimension. Given an index between zero and one less than that returned by GetDimensionCount, returns the dimension object indicated by that index.

GetAt. Given an array of ordinal dimension values, returns the item at that position in the data element. For a three-dimensional data element, the following code would retrieve the item at (3, 6, 7):

```
long rgDim[3];
rgDim[0] = 3;
rgDim[1] = 6;
rgDim[2] = 7;
VARIANT data; VariantInit(&data);
pDataElement->GetAt(3, rgDim, &data);
```

If the one of the ordinal dimension values does not specify a valid ordinal, GetAt returns RSOLVE_E_INVALIDINDEX.

GetAt presents the data element as a mapping from the product space of the data element's dimensions into the space of whatever type the data element is. Due to the restriction that sets are finite, and because data element types currently map into COM's VARIANT, clearly it is a finite mapping into VARIANTs.

SetAt. Given an array of ordinal dimension values and a new data value, assigns the new data value to the indicated position in the data element. For a three-dimensional data element, the following code would set the item at (3, 6, 7):

```
long rgDim[3];
rgDim[0] = 3;
rgDim[1] = 6;
rgDim[2] = 7;
VARIANT data; VariantInit(&data);
data.vt = VT_R8;
```

```
data.dblVal = 42.0;
pDataElement->SetAt(3, rgDim, data);
VariantClear(&data);
```

If the one of the ordinal dimension values does not specify a valid ordinal, SetAt returns RSOLVE_E_INVALIDINDEX. Read-only data elements return RSOLVE_E_READONLY, while data elements that are locked return RSOLVE_E_DATAELEMENTLOCKED.

GetDataSource. Returns the data source that provides this data element.

3.4.2.3.b Interface IRDataElementClone

The interface IRDataElementClone enables a client to clone a data element. The client could then modify the clone or pass it on to another solver while retaining the original. Cloning is an important part of networked solutions. The single method, **Clone,** returns a new data element object that has the same dimensions, sets, and data values as the original.

There are no particular restrictions on how the clone is implemented. For optimizations, copy-on-write could be used, for instance. Similarly, there is no restriction on whether the same dimension and sets objects are used with both the original and the clone or whether those are cloned as well.

If a data element does not support IRDataElementClone, then the client will have to clone the element piecemeal[6]. Clearly, IRDataElementClone provides opportunity for significant optimizations as usually the implementation of a data element knows best how to optimize copying the data element.

3.4.2.3.c Interface IRDataElementAccess

Sub-element access is the process of extracting a lower-dimensional data element from an existing data element—for example, taking the first column as a vector from a matrix. Given the view that a data element is a mapping from dimension indices into a data value (see IRDataElement::GetAt, above), sub-element access is the creation of a new mapping by fixing some of the arguments in the original mapping and leaving the others free.

For example, suppose there is a three-dimensional data element, with dimensions of months, researchers { Steve, Penny, Thalia }, and customers { Tom, Liz, Chris }. The data element stores hours worked in a month by a researcher for a customer. This data element has 12*3*3 = 108 entries. Suppose a client wanted to access values specifically for the researcher

---

[6] There is certainly the opportunity here for a core services component which could clone any data element, and maybe even optimize the process by somehow using intimate knowledge of the data element's layout via private interfaces.

Penny. This is possible by using IRDataElement::GetAt and fixing the researcher dimension value to always equal Penny. Essentially, this creates a two-dimensional matrix with dimensions months and customers (Penny being the fixed researcher). However, this two-dimensional matrix exists in spirit only, as it would not be possible to pass the two-dimensional matrix to a solver without the client manually creating it by using IRDataElement::GetAt.

IRDataElementAccess enables the creation of the sub-element from within the data element, which due to internal implementation knowledge can be optimized just as with IRDataElementClone. IRDataElementAccess has a single method:

GetSubElementAt. Given the dimension values to be fixed, returns the sub-element of the data element that has those values fixed and all other values free. There are three inputs. First is the number of dimensions that will be fixed. Second is an array of integers that identify which dimensions are being fixed; these integers are the same used by IRDataElement::GetDimension (described above). The third input is an array of integers indicating the ordinal values of the fixed dimensions, as determined by IRSet::MapToOrdinal. The function creates a new data element containing the desired sub-element and returns it to the client.

Consider the example above, where the mappings are Steve = 1, Penny = 2, Thalia = 3, Tom = 1, Liz = 2, and Chris = 3, and the months range from 1 to 12. The dimensions are researchers = 0, customers = 1, and months = 2,. Suppose the client wants to determine the hours worked by Penny in the month of July for all the customers. The following shows the arrangement of the inputs to GetSubElementAt:

Number of dims.	Dimensions to fix		Values to fix at	
2	0	(researchers)	2	(Penny)
	2	(months)	7	(July)

The resultant sub-element would be a vector with one dimension—customers—and three values, one for each of the customers. Sample pseudo-code for this method would be:

```
IRDataElement* pNewDataElement = NULL;
long rgDim[2];
long rgValues[2];
rgDim[0] = 0; // researchers
rgValues[0] = 2; // Penny
rgDim[1] = 2; // months
rgValues[1] = 7; // July
pDataElementAccess->GetSubElementAt(2, rgDim, rgValues, RDA_COPY,
        &pNewDataElement);
```

The fourth parameter indicates whether the resultant sub-element should be a copy or a contained reference of the original data. If a reference (using RDA_CONTAINED), then changes to the sub-element will change the original data element, and changes to the original data element will be reflected in the sub-element. Note that this code is not exact due to the additional input parameter (not shown) of the interface identifier for the interface that should be returned.

3.4.2.3.d Interface IRDataElementCreator

IRDataElementCreator is the factory interface for creating stand-alone data elements. As with sets and dimensions, the implementation of a data element will also expose a factory object that can create data elements.

The single method, **Create,** takes as input an array of existing dimension objects and a type indicator for the type of values to store in the data element, and returns a new data element with those dimensions and suitable default values.

3.4.2.4 Specialized data element interfaces

Some interfaces are designed for optimizing access to common data element types. There is an interface each for scalars, vectors, and matrices. These interfaces can be exposed by a data element to allow the client to more efficiently and easily access these data element types.

3.4.2.4.a Interface IRDataElementScalar

IRDataElementScalar provides easy access for scalar data elements. If a data element exposes IRDataElementScalar, then the client can perhaps optimize its use of the data element with this interface; otherwise it must rely on other interfaces.

GetScalar. Returns the value of the scalar.

SetScalar. Sets the value of the scalar. Read-only data elements return RSOLVE_E_READ-ONLY, while data elements that are locked return RSOLVE_E_DATAELEMENTLOCKED.

3.4.2.4.b Interface IRDataElement1

IRDataElement1 provides easy access for vector data elements. If a data element exposes IRDataElement1, then the client can perhaps optimize its use of the data element with this interface; otherwise it must rely on other interfaces.

GetAt. Given an ordinal index, returns the value at that index in the vector. If the index does not specify a valid ordinal, GetAt returns RSOLVE_E_INVALIDINDEX.

SetAt. Given an ordinal index and a new value, sets the value at that index in the vector to the new value. If the index does not specify a valid ordinal, SetAt returns RSOLVE_E_INVALIDINDEX. Read-only data elements return RSOLVE_E_READONLY, while data elements that are locked return RSOLVE_E_DATAELEMENTLOCKED.

3.4.2.4.c Interface IRDataElement2

IRDataElement2 provides easy access for matrix data elements. If a data element exposes IRDataElement2, then the client can perhaps optimize its use of the data element with this interface; otherwise it must rely on other interfaces.

GetAt. Given ordinal row and column indices, returns the value of the matrix at that row and column. If the row or column does not specify a valid ordinal, GetAt returns RSOLVE_E_INVALIDROW or RSOLVE_E_INVALIDCOLUMN.

SetAt. Given ordinal row and column indices and a new value, sets the value at that row and column in the matrix to the new value. If the row or column does not specify a valid ordinal, SetAt returns RSOLVE_E_INVALIDROW or RSOLVE_E_INVALIDCOLUMN. Read-only data elements return RSOLVE_E_READONLY, while data elements that are locked return RSOLVE_E_DATAELEMENTLOCKED.

GetColumn. Given an ordinal column index, returns another data element that contains the specified column's values, as a vector. If the data element does not support sub-element access, it returns RSOLVE_E_NOTSUPPORTED. If the index does not specify a valid column, GetAt returns RSOLVE_E_INVALIDCOLUMN. Usually, implementations will delegate to IRDataElementAccess::GetSubElementAt.

GetRow. Given an ordinal row index, returns another data element that contains the specified row's values, as a vector. If the data element does not support sub-element access, it returns RSOLVE_E_NOTSUPPORTED. If the index does not specify a valid row, GetAt returns RSOLVE_E_INVALIDROW. Usually, implementations will delegate to IRDataElement-Access::GetSubElementAt.

3.4.2.5 Data element properties

An optional feature that a data source can provide is *data element properties*. For a given data element, there can be a set of named properties. Each property is essentially another data element that shares the same dimensions as the data element proper. However, semantically, the properties are too closely related to the data element to warrant being their own data elements, or else the properties are values that make no sense in array, vector, or matrix form.

A reasonable assumption for an applied solution is that data element properties coincide with ordered sets of dimensions. In this way, multiple data elements can share the same data

element property set, and when one of the properties changes, the change is reflected across both data elements.

For example, consider a data element that contains the production volumes on a series of five identical machines during 1997. One dimension is the set of five machines, while the other dimension is the set of twelve months of 1997. Suppose this data element is queried from a database table. Probably this table has sixty rows, one for each machine-month combination. This table has two columns identifying the particular machine and month a given row represents, and another column for production volume. Suppose this table also contains other columns, such as a row ID (an auto-number containing a unique ID for the row), forecast production values, and number of accidents at that machine per month.

Now, suppose a solver needs to calculate the error in the forecast for each machine for each month, and output the result to a new table. For easy lookup, the new table should columns for machine, month, forecast error, and row ID, where the same row ID is used in both tables to identify the same machine-month combination. In this case, the row ID is a property of the machine-month product space. For a given machine and a given month, there is a unique row ID that must be transferred from the source table to the destination table. This row ID might be a number without any numerical meaning in the model being solved. So, the data source that encapsulates this query can store the row ID as a data element property and propagate it forward to any other data elements that also use the same dimensions.

A data element object exposes the IRDataElementProperties interface if it supports the look-up of data element properties. It exposes the IRDataElementPropertiesChange interface if it also supports the modification of these properties.

3.4.2.5.a Interface IRDataElementProperties

A data element exposes IRDataElementProperties when it provides data element properties on its data. It has four methods:

GetItemPropertyCount. Returns the number of named properties.

GetItemPropertyName. Given an index from zero to one less than the value returned by GetItemPropertyCount, returns the name of the property identified by that index.

MapItemPropertyNameToId. Given the name of a property, returns the index of that named property if it is one of the named properties that exists on this data element.

GetItemProperty. Given the index of a named property, from zero to one less than the value returned by GetItemPropertyCount, and an array of ordinal dimension values, returns the property value corresponding to the given index and the provided position in the data element. See IRDataElement::GetAt for an example of similar usage of the ordinal dimension values.

3.4.2.5.b Interface IRDataElementPropertiesChange

If a data element supports data element properties and permits modifications of these properties, it does so through the implementation of IRDataElementPropertiesChange. This interface has a single modifier method:

SetItemProperty. Given the index of a named property, from zero to one less than the value returned by IRDataElementProperties::GetItemPropertyCount, an array of ordinal dimension values, and a new property value, sets the property value corresponding to the given index and the provided position in the data element. See IRDataElement::SetAt for an example of similar usage of the ordinal dimension values.

3.4.2.6 Creating and acquiring data elements

With an understanding of the capabilities of a data element, the question of how to acquire a pointer to a data element in the first place remains. There are two main ways to acquire a data element. The first is to create one from scratch. Assuming the sets and dimensions are known or available, it is a simple process to create a factory object for a data element and then call IRDataElementCreator::Create on the factory to create a new data element.

The other way to acquire a data element is from a solver or other component that provides a data element as its output. Internally, a solver might use a stand-alone data element component or it might roll its own implementation of one. Either way, the client does not personally create the data element; it only receives it from the solver.

In typically solution architectures, the client will have to create a data element only as part of initializing the original inputs to the problem. If these are queried from a database, then a specialized database component could even handle that, relieving the creation burden entirely from the client.

Nevertheless, for those times when a client must create a data element, the following pseudo-code illustrates how to create a vector:

```
// first, create the dimension factory object
IRDimensionCreator* pDimCreator = NULL;
CoCreateInstance(CLSID_RDimensionCreator, NULL, CLSCTX_SERVER,
        IID_IRDimensionCreator, (LPVOID*)&pDimCreator);

// create a vector that ranges from 0 to 10.
IRDimension* pDim = NULL;
pDimCreator->CreateFromRange(L"SampleDim", 0, 10, IID_IRDimension, (LPVOID*)&pDim);
pDimCreator->Release();
```

```
// create a vector data element factory object
// this uses a special vector implementation with CLSID CLSID_RVectorCreator
IRDataElementCreator* pVecCreator = NULL;
CoCreateInstance(CLSID_RVectorCreator, NULL, CLSCTX_SERVER, IID_IRDataElementCreator,
        (LPVOID*)&pVecCreator);

// create the vector
// 1 dimension, it will hold doubles (VT_R8 = double)
IRDataElement* pVec = NULL;
pVecCreator->Create(L"SampleVec", 1, &pDim, VT_R8, IID_IRDataElement, (LPVOID*)&pVec);
pVecCreator->Release();

// use the special data element interface to fill the vector
IRDataElement1 pVec1 = NULL;
pVec->QueryInterface(IID_IRDataElement1, (LPVOID*)&pVec1);
VARIANT var; VariantInit(&var);
var.vt = VT_R8;
for(int j=0;j<=10;++j) {
    var.dblVal = (j-5)*(j-5)+3.14;
    pVec1->SetAt(j, var);
}
VariantClear(&var);
```

3.4.3 Data sources

Data elements are the embodiment of the actual data values themselves. Data sources are the entities that provide these values to a client. Data elements are static; they contain data. Data sources are dynamic; they lock, update, and manage data lifetimes. A data source can provide more than one data element. For example, a database query might yield one data element per column in the resulting query set; the query itself might be the data source, and each column would be a data element provided by that source.

For stand-alone implementations of data elements (such as the one demonstrated in the example in section 3.4.2.6, above), the data source and the data element are usually the same object. That is, the same object exposes both the data element and the data source interfaces. In cases where the client acquires a data element from another object, such as a solver, then usually the provider object or solver will implement the data source interfaces.

At a high level, data sources have a fairly simple state space. A canonical data source state diagram is presented in Figure 3.4. There are two independent aspects of state. A recurring one is that of the lock count. In the upper half of the state diagram, there are two states indicating whether the data source is Locked or Unlocked. Values within data elements of this data source can be changed only when it is in the Unlocked state; and doing so causes the OnDataChange notification to be sent. The other aspect of state is the notification list, discussed in section 3.4.3.2.

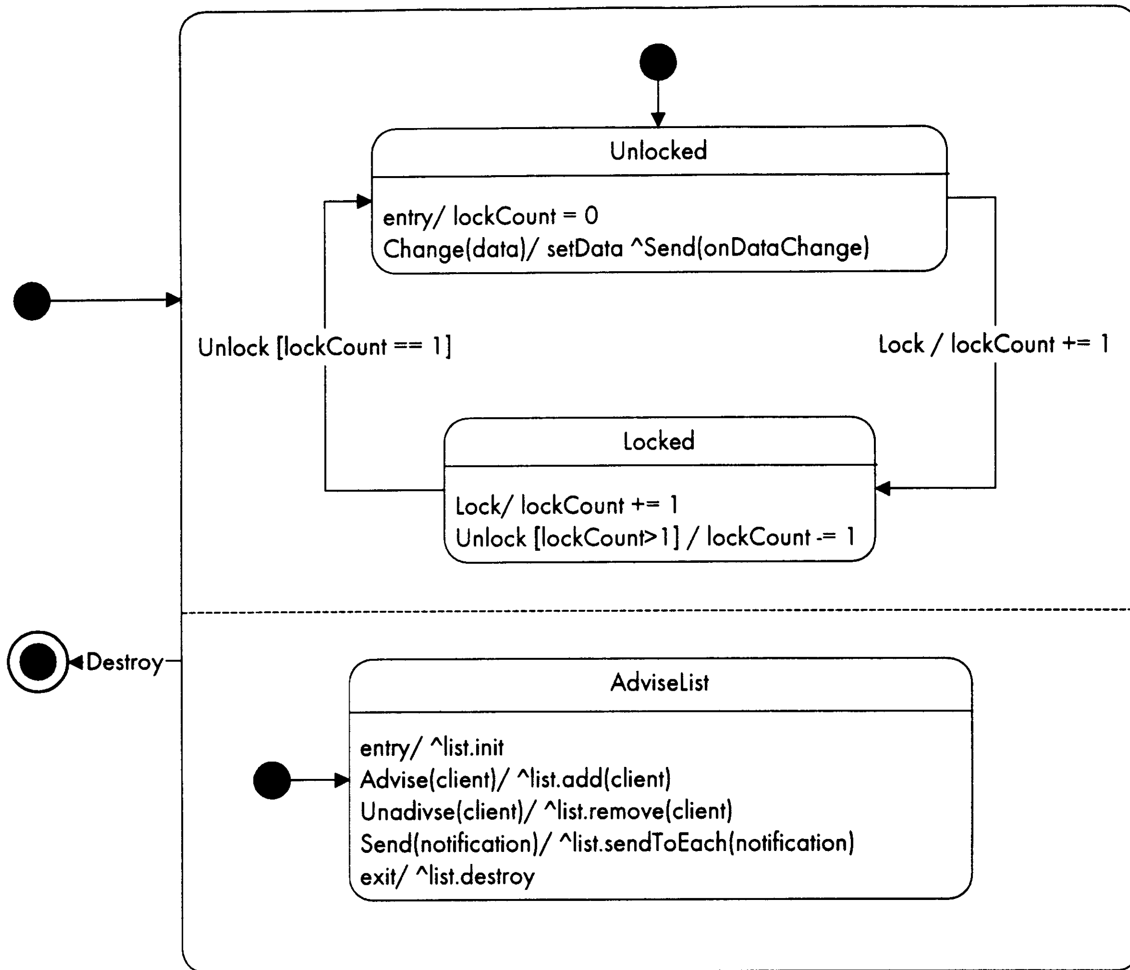


Figure 3.4: State diagram for a data source

The primary data source interface is IRDataSource. It provides for simple data notification advises, accessing its collection of data elements, and locking its data elements. The interface IRDataAdvise is implemented by clients that wish to receive data change notifications. The interface IRDataAdviseHolder is a helper interface that data sources can use to manage their collection of notification sinks.

3.4.3.1 Interface IRDataSource

Interface IRDataSource is the primary data source interface. It has eleven methods, of which nine are currently meaningful. The remaining two, CanMaintainSolverData and LockRequired, will be explained in more detail in section 3.8.6.2, page 200.

Advise. Establishes an advise connection for data change notifications. Takes as input an interface pointer to the notification sink interface, IRDataAdvise, and returns a magic cookie that identifies this particular connection.

Unadvise. Breaks an advise connection for data change notifications. Takes as input a magic cookie previously returned from Advise.

LockData. Increments the lock count on the data source and all of the data source's data elements. When the data source's lock count is non-zero, the data source is locked. Calls to any of the methods in the data elements that would change data will fail with RSOLVE_E_DATAELEMENTLOCKED.

UnlockData. Decrements the lock count on the data source and all of the data source's data elements (if they are above zero). While a data element is locked, calls to any of the methods in the data element that would change data will fail with RSOLVE_E_DATAELEMENT-LOCKED.

DataLocked. Returns a non-zero value if the data source is currently locked (as controlled by LockData and UnlockData).

GetDataElementCount. Returns the number of data elements provided by this data source.

GetDataElementName. Given an index between zero and one less than that returned by GetDataElementCount, returns the name of the data element for the given index.

GetDataElementAccessor. Given an index between zero and one less than that returned by GetDataElementCount, returns an accessor interface of the data element for the given index. Typically, this is IRDataElement.

GetDataElementModifier. Given an index between zero and one less than that returned by GetDataElementCount, returns a modifier interface of the data element for the given index. Typically, this is IRDataElement. The distinction between this method and GetDataElement-Accessor enables future implementations where accessor and modification functions are in different interfaces, as is the case with the significantly more capable OLE DB specification.

CanMaintainSolverData. Returns a non-zero value if the data source can maintain data element data even after it has been passed to a solver and during a solver's execution. This will be explained in more detail in the section on networking solvers (see section 3.8.6.2, page 200).

LockRequired. If the data source can maintain solver data, as determined by CanMaintainSolverData, this function returns non-zero if the solver must lock the data source and its data elements in order to use it as inputs for a solver. This will be explained in more detail in the section on networking solvers (see section 3.8.6.2, page 200).

3.4.3.2 Data change notifications and IRDataAdvise

Clients that wish to receive data change notifications must establish an advise connection with the data source by calling IRDataSource::Advise, described in the previous section. IRDataSource::Advise and IRDataSource::Unadvise are analogous to the standard COM methods IDataObject::DAdvise and IDataObject::DUnadvise. In particular, if the data source does not support notifications, it should return the standard error OLE_E_ADVISE-NOTSUPPORTED from Advise and Unadvise. If an invalid magic cookie is passed to Unadvise, it should return OLE_E_NOCONNECTION.

Data sources that do support data change notifications must maintain a list of the valid advise connections, usually in a map of interface pointers and magic cookies. This list must support insertion, removal, and enumeration of elements, and secondarily it must support lookup by magic cookie.

Interface IRDataAdvise is a simple data change notification interface that enables a client to be notified whenever a data element's data changes. Currently, it is broadcast by a data source, so that if only one data element changes in a data source, any clients using any data elements from that data source will receive this notification. Clearly this is a liberal notification scheme, but it suits the purposes of the framework. This notification is used primarily in networks of solvers to indicate when input data has been established and the solution process can begin. Because of the assumption that the framework does not address real-time architectures or operational solutions, it is safe to further assume that exogenous input data will not change randomly while a solution is in progress.

OnDataChange. This message is sent by the data source to all advised clients (connections established through IRDataSource::Advise) whenever any of the data source's data elements change.

3.4.3.3 Simplifying implementation with IRDataAdviseHolder

The previous section described the features of a map of interface pointers and magic cookies necessary for data change notifications. Because most data sources will implement such a list, the core services provide a special object called the DataAdviseHolder that implements the necessary functionality of this list, simplifying data source development. The DataAdviseHolder takes care of managing the map of pointers and of walking through the map to send updates whenever the data source wants to send a notification.

Generally, the data source will create an instance of a DataAdviseHolder for each instance of the data source. The DataAdviseHolder has CLSID CLSID_RDataAdviseHolder, and the data source can create it with the normal call to CoCreateInstance. The DataAdviseHolder supports the IRDataAdviseHolder interface, which has the following methods:

Advise. Takes as inputs a pointer to the data source's implementation of IRDataSource and the client's implementation of IRDataAdvise, adds the client's pointer to the DataAdviseHolder's internal map, and returns a magic cookie for the data source to pass back to the client.

Unadvise. Takes as input the magic cookie passed from the client to the data source and removes the interface pointer that corresponds to that connection from its internal map.

SendOnDataChange. Sends the OnDataChange notification to all connections in the internal map. See IRDataAdvise for more details.

The data source can incorporate the DataAdviseHolder object simply, by following these steps (examples are shown for C++).

1. Add an interface pointer to IRDataAdviseHolder to the class that implements the data source. In the header file for the data source class:

```
Class CRMyDataSource : public IRDataSource
{
    IRDataAdviseHolder* m_pDataAdviseHolder;
    // remainder of class definition
```

2. In the constructor of the data source class, make sure to initialize the pointer, and in the destructor make sure to release the pointer if it is non-zero:

```
CRMyDataSource::CRMyDataSource() : m_pDataAdviseHolder(0) {}
CRMyDataSource::~CRMyDataSource()
{
    if( m_pDataAdviseHolder )
        m_pDataAdviseHolder->Release();
}
```

3. Add implementations of Advise and Unadvise that look like this:

```
STDMETHODIMP CRMyDataSource::Advise(IRDataAdvise* pAdvise, DWORD* pdwCookie)
{
    if( pdwCookie )
    {
        if( !m_pDataAdviseHolder )
        {
         HRESULT hr = CoCreateInstance (CLSID_RDataAdviseHolder, NULL, CLSCTX_SERVER,
        IID_IRDataAdviseHolder, (LPVOID*) &m_pDataAdviseHolder);
```

```
        if( FAILED(hr) )
        return E_OUTOFMEMORY;
      }
      return m_pDataAdviseHolder->Advise(static_cast<IRDataSource*>(this), pAdvise,
      pdwCookie);
    }
    return E_POINTER;
}

STDMETHODIMP CRMyDataSource::Unadvise(DWORD dwCookie)
{
    if( !m_pDataAdviseHolder || !dwCookie )
       return OLE_E_NOCONNECTION;
    return m_pDataAdviseHolder->Unadvise(dwCookie);
}
```

4. Add at the point where the data source wants to update clients of a data change the appropriate call to SendOnDataChange:

```
// ... call notification
if( m_pDataAdviseHolder )
    m_pDataAdviseHolder->SendOnDataChange();
```

3.5 The Primary Solver Interfaces

To be a solver within the framework, a component must implement and expose the primary solver interfaces IRSolver, IRSolverInputs, IRSolverOutputs, and IRSolverParameters, as well as IRDataSource. These interfaces expose, naturally, the inputs, outputs, and parameters of a solver, as well as provide control over the solver's execution.

The basis of the solver interfaces stems from the framework's assumptions about the structure of solvers. These assumptions are discussed next, followed by a description of the primary solver interfaces.

3.5.1 Solver structure

Almost all solvers have a similar structure. They take inputs, generate outputs, and can be parameterized in some fashion. The following items clarify the assumptions the framework makes about solvers:

- The solver has a set of inputs, a set of outputs, a set of parameters, and a set of sets. The inputs are the actual problem data, the parameters are the configuration values for the algorithm, the outputs are the results of the analysis, and the sets are the collections of interesting entities in the problem.

- Each set is an enumeration of related entities. A set might be ordered or unordered, sorted or unsorted, or exhibit other properties. For instance, the union of all possible facility locations is a set for the facility location problem. Months of the year might form another set.

- Each input and each output is a single data element. This element can be a scalar value, a matrix, an unordered set, etc. Each input and each output has a set of dimensions, describing the domain of the input data element. Scalar input or output elements have no associated dimensions. See section 3.4.2, page 120, for more information on data elements.

- A dimension references one of the sets of the solver, so that the range of possible values for a given dimension is specified by the elements of one of the sets.

- A parameter is a single value that configures the solver.

In essence, a solver contains inputs, outputs, parameters, and sets. Each input contains dimensions. Each output contains dimensions. Each dimension references a set. This abstraction is represented in Figure 3.5.

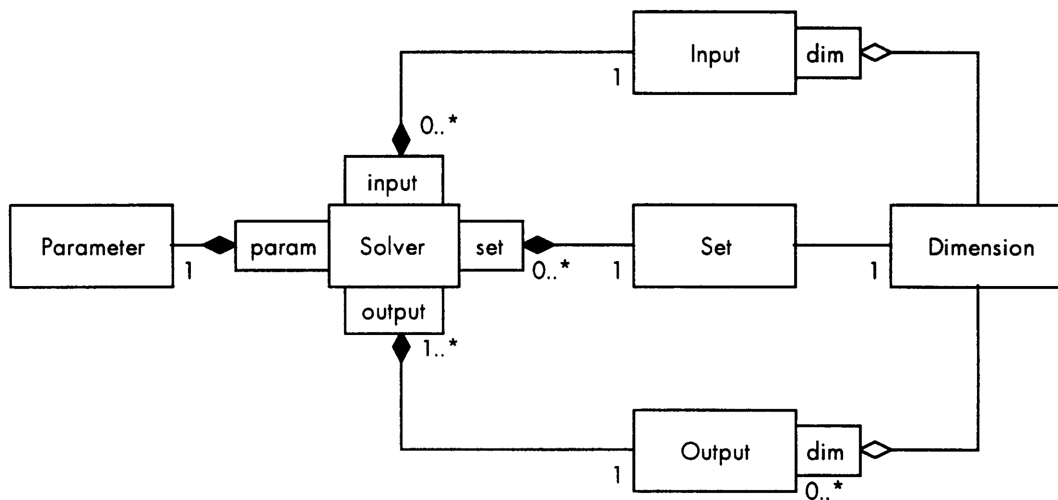


Figure 3.5: High-level class diagram of solver structure

Example—Knapsack problem. Consider the knapsack problem. This problem has four inputs, one output, one set, and no parameters, as shown in the object model in Figure 3.6. The set Items is the list of possible items to load into the sack. This can be simply the values from one to five if there are five pieces, or it could be the named values {apple, cheese, bread, water, beef jerky}. One input, Capacity, is a scalar. The other three inputs are each vectors, indexed by the dimension ItemDim which is enumerated by the set Items. The InitialSolution input is a first-guess solution for the problem, probably a zero vector by

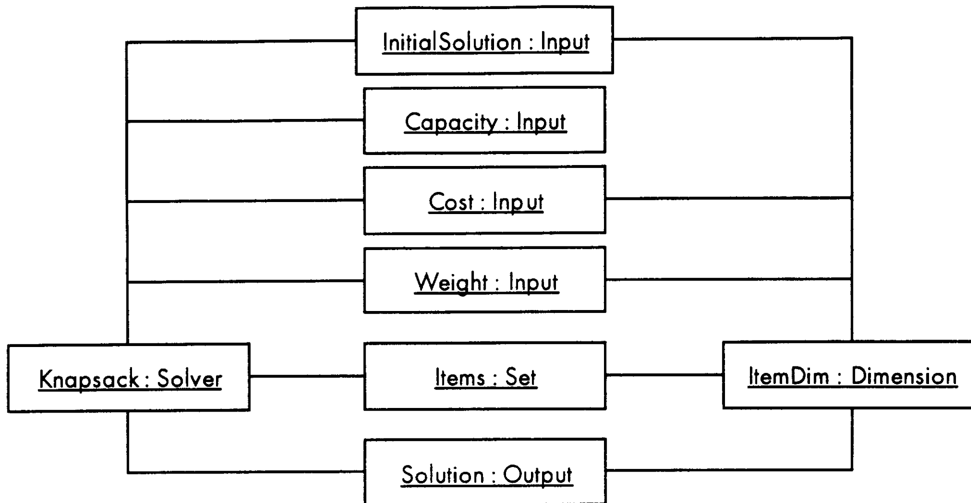


Figure 3.6: Object diagram of knapsack solver structure

default. The Cost and Weight vectors have their usual meanings for the knapsack problem. The output, Solution, is a vector indicating the number of each item to place in the knapsack.

It is not necessary for a solver's actual structure to correspond to the representation in Figure 3.5. However, for most solvers there are mappings from their structures to the one presented.

3.5.2 IRSolver

IRSolver is the fundamental solver interface. It is the cornerstone of a solver implementation. Its methods are partitioned into several areas of functionality.

First is generic state and life cycle control. The **Solve** method initiates the synchronous execution of the solver. The reason of existence of a solver is the Solve method. The associated methods **ClearInputs** and **ClearOutputs** help to reset the state of a solver if it will be used to solve multiple problem instances. See section 3.8.5.4, page 195, for more information on ClearInputs and ClearOutputs.

Two notification registration methods, **SolveAdvise** and **SolveUnadvise,** enable a client to establish an advise connection for progress updates. See section 3.7, page 169, for more information on solver progress notifications.

Finally, four other methods, **GetSolverSiteIn, SetSolverSiteIn, GetSolverSiteOut,** and **SetSolverSiteOut,** are used in a network of solvers. These functions are described in detail and in context in section 3.8.7.1, page 204.

3.5.2.1 IRSolverAsynchSolve

An optional interface, IRSolverAsynchSolve, enables a client to initiate asynchronous execution of a solver. Solvers that support such activation indicate so by implementing this interface. It has a single method, **AsynchSolve,** that has the same semantics as IRSolver::Solve, except that AsynchSolve should return to the caller as soon as possible, performing the actual solver execution on a separate thread or in a separate process.

3.5.3 IRSolverInputs: Setting the inputs

The interface IRSolverInputs describes how a solver exposes its inputs to clients, and how clients can pass inputs into the solver. Each input is a data element with an associated data source.

Three methods relate to setting and getting inputs:

GetInputCount. Returns the number of inputs to the solver.

GetInputData. Given an index from zero to one less than that returned by GetInputCount, returns the data source and the index of the data element within that data source that stores the values for the given indexed input. GetInputData can return a number of errors, including RSOLVE_E_INVALIDINDEX if the index parameter is invalid, RSOLVE_E_OUTOFORDER if the solver requires ordered inputs and the specified index is out of order, RSOLVE_E_SOLVING if the solver is currently solving, RSOLVE_E_INPUTSLOCKED if the inputs are currently locked and inaccessible, and RSOLVE_E_INPUTNOTSET if the input was never set in the first place.

SetInputData. Given an index from zero to one less than that returned by GetInputCount, a data source, and a data element index for that data source, assigns the specified data element to the indexed input of the solver. SetInputData returns errors similar to GetInputData (see above), including RSOLVE_E_INVALIDINDEX, RSOLVE_E_OUTOFORDER, RSOLVE_E_SOLVING, and RSOLVE_E_INPUTSLOCKED.

Three other methods manage the lock count state of the input side of the solver. **LockInputs** increments the input lock count, **UnlockInputs** decrements the input lock count, and **InputsLocked** returns whether the inputs are currently locked.

3.5.4 IRSolverOutputs: Getting the outputs

The interface IRSolverOutputs allows a client (or another solver in a network) to access the output data elements of a solver.

GetOutputCount. Returns the number of outputs exposed by the solver.

GetOutputData. Given an index from zero to one less than that returned by GetOutputCount, returns the data source and the index of the data element within that data source that stores the values for the given indexed output. GetOutputData can return a number of errors, including RSOLVE_E_INVALIDINDEX if the index parameter is invalid, RSOLVE_E_SOLVING if the solver is currently solving, RSOLVE_E_OUTPUTSLOCKED if the outputs are currently locked and inaccessible, and RSOLVE_E_OUTPUTNOTSET if the output has not yet been set by the solver.

Three other methods manage the lock count state of the output side of the solver. **LockOutputs** increments the output lock count, **UnlockOutputs** decrements the output lock count, and **OutputsLocked** returns whether the outputs are currently locked.

3.5.5 IRSolverParameters: Parameterization

IRSolverParameters is a simple interface that provide array access to the parameters of a solver. It has three methods:

GetParameterCount. Returns the number of parameters supported by the solver.

GetParameter. Given an index from zero to one less than that returned from GetParameterCount, returns the parameter identified by the index.

SetParameter. Given an index from zero to one less than that returned from GetParameterCount and a new parameter value, sets the parameter identified by the index.

3.6 Introspection

Section 2.3.3 outlined the need for active documentation that helps a user select and subsequently use a solver. As indicated in that section, the problem of documentation for using a solver is in large part resolved by virtue of having a standard interface and protocol for using all solvers within the framework. To some extent, once the user learns one, all are the same. However, solvers differ in their structure, in their inputs, outputs, and parameterization capabilities. Knowing thus how a solver operates, the user still needs to learn what the structure of a solver is in order to use it. To tackle this problem and the problem of solver selection, this section presents an *introspection* protocol that enables solvers to actively participate in the documentation process and for clients to discover at design-time and run-time the capabilities of a solver. Through introspection, a client can determine the structure and capabilities of a solver.

There are three aspects to the introspection model for solvers. The first, *SolverInfo*[7], characterizes the quantitative structure of the solver, such as its inputs, outputs, parameters,

[7] Named similar to the JavaBeans equivalent in the JavaBeans introspection API, the BeanInfo.

and dimensions. The second, *SolverDescription,* captures the qualitative details of an algorithm implementation, such as the author, developer, and citation. The third aspect is the discovery of capabilities based on supported interfaces, as determined by calls to IUnknown::QueryInterface.

3.6.1 SolverInfo: Paralleling the structure of the solver

The SolverInfo protocol enables a client to determine the nature of inputs, outputs, and parameters for a solver, in order to automate the application of the solver in a solution environment. If a solver does not support the protocol, then the client must use other techniques to ascertain the structure of the solver.

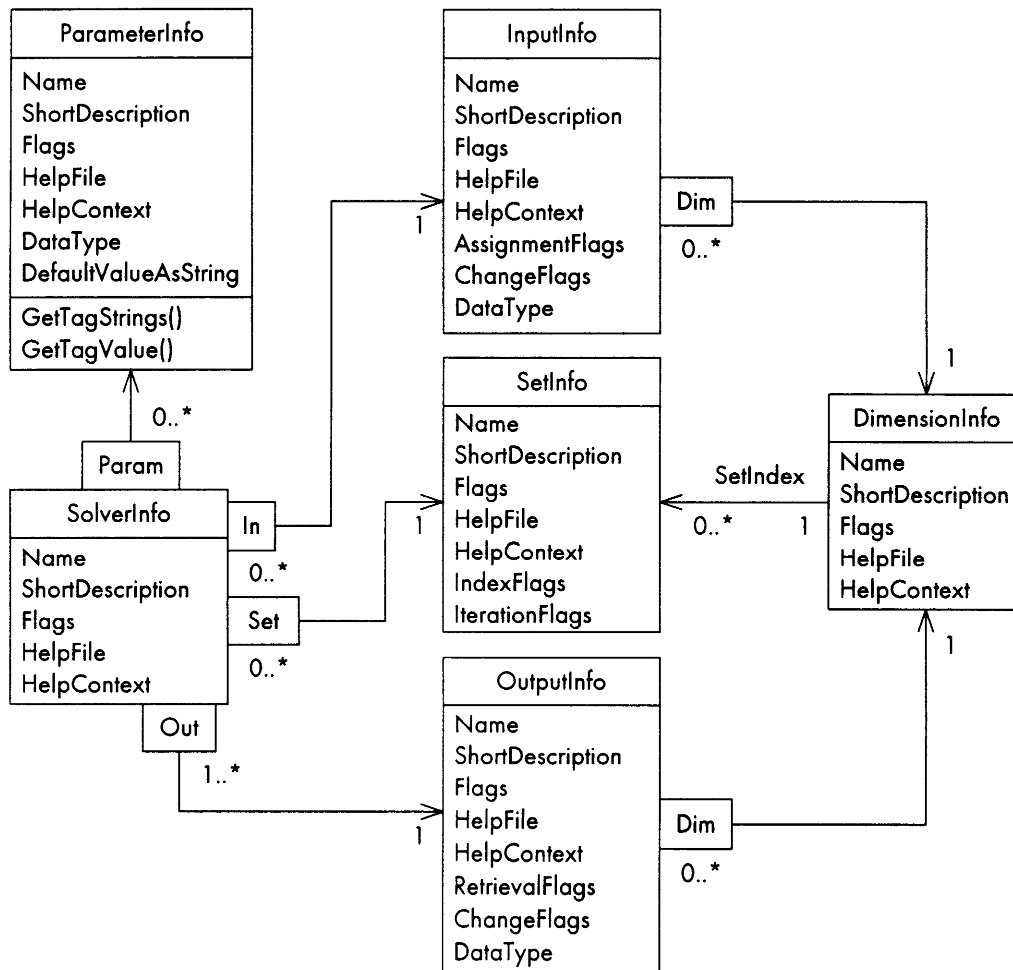


Figure 3.7: Class diagram of SolverInfo interfaces

SolverInfo structure. SolverInfo attempts to describe the overall outward appearance of the structure of a solver. To this end, the SolverInfo structure parallels that of a solver; see section 3.5.1, page 137, for a description and explanation of the presumed solver structure.

SolverInfo protocol. The SolverInfo protocol mirrors this structure in a set of interfaces that provide documentation for each piece of the solver structure. Specifically, the SolverInfo interface provides information about the solver itself, the InputInfo provides information about an input, etc. The relationship of these interfaces is displayed in Figure 3.7; the correspondence between the information interfaces and the solver structure in Figure 3.5 is self-evident. The attributes of each class will be discussed shortly.

A typical implementation of the SolverInfo protocol will create a separate object for each input, output, dimension, set, and parameter. Each object will support the interface that represents that object. Another object for the solver will provide access to all of these objects. For instance, a possible SolverInfo structure for the knapsack example described in section 3.5.1, page 138, is shown in Figure 3.8.

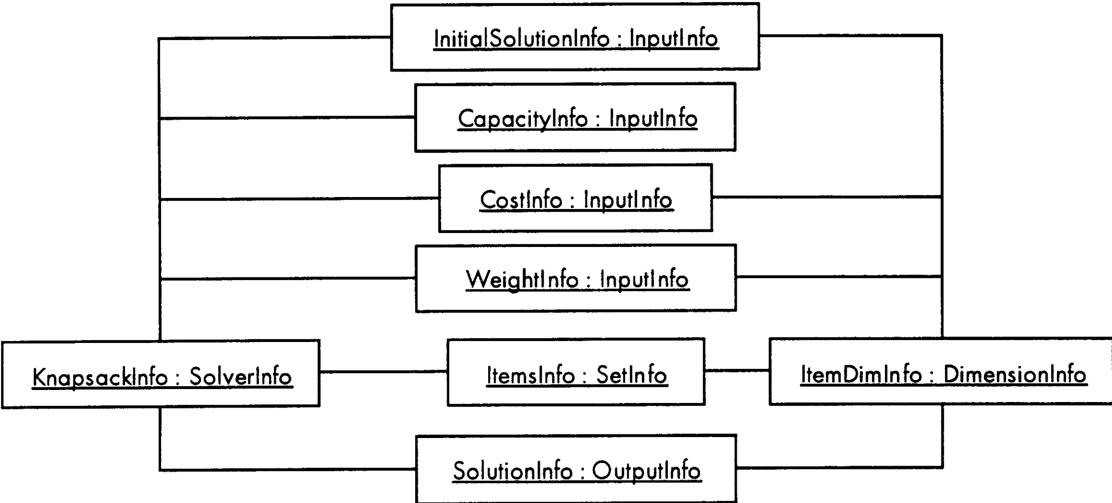


Figure 3.8: Object diagram of knapsack SolverInfo structure

The SolverInfo protocol is a collection of interfaces supported either by the solver or by an accessory object acting on behalf of the solver.

3.6.1.1 SolverInfo interfaces

This section describes the various interfaces of the SolverInfo protocol.

3.6.1.1.a Interface IRSolverBaseInfo

All of the solver information interfaces derive from a common base interface, IRSolverBaseInfo. This interface provides the method definitions of common routines for all of the information interfaces. It includes:

GetName. Returns a string containing the name of the object, whether it is a solver, dimension, input, output, or set. An example name for a solver might be "John's Knapsack." An input might be named "Weight" or "Cost," while a dimension might be "Items."

GetShortDescription. Returns a string containing a short, human-readable description of the object. For instance, the description for the input element "Cost" might be "A vector containing the cost of each item." The short description can be used in solver browsers to assist a user in understanding the elements of a solver (for solver selection, see section 2.3.3.1, page 78) and assigning mappings to inputs and outputs of solvers (for solver utilization, see section 2.3.3.2, page 80).

GetFlags. Returns a bitmask containing global and object-specific status information. Each bit in the returned value represents a true or false value for some property of the object. All SolverInfo objects can have the following flags:

RSI_EXPERT	The element is intended for expert users only. I.e., if a client has a "novice" setting, this element can be hidden and default values will be used. This is particularly useful for solvers with many parameters that can highly customize the operation of the algorithm.
RSI_HIDDEN	The element should not be displayed in a solver browser. I.e., the element is accessible programmatically for clients that know the element exists, but in a visual browser the element should be hidden from the user.

Table 3.8: Bitmask values for IRSolverBaseInfo::GetFlags

Each specific object will define its own set of flags that apply specifically to that type of object.

GetHelpFile. Returns a string that identifies a help file containing more extensive information on the particular object. By calling this and GetHelpContext, a client can load a help topic using the operating system's help functionality.

GetHelpContext. Returns a number that identifies a help context ID that references a help topic within the file identified by GetHelpFile (above). By calling this and GetHelpFile, a client can load a help topic using the operating system's help functionality.

3.6.1.1.b Interface IRSolverInfo

Interface IRSolverInfo is the primary interface for solver information. It provides details about the solver as a complete entity and accessors to the specific objects of the solver. As with all of the SolverInfo interfaces, it derives from IRSolverBaseInfo. It defines the flags in Table 3.9 for IRSolverBaseInfo::GetFlags. Typically, one of RSI_SYNCHRONOUS_EXECUTION or RSI_ASYNCHRONOUS_EXECUTION must be set.

RSI_ASYNCHRONOUS_INPUTS	If set, then the solver can accept inputs for a new problem instance at any time, for example while the solver is solving another problem. Otherwise, the solver can only accept inputs when it is doing nothing else. Most solvers would not set this bit. See page 91 for a canonical state diagram of solvers.
RSI_ORDERED_INPUTS	If set, then the solver can only accept inputs in the ordered specified by their indices (see below). If not set, then the solver can accept inputs in any order. The solver might need to have ordered inputs if the values of one input can modify the meaning or size of another input. For example, if one input is the number of elements to analyze and another input is the vector of elements, then the number of elements might need to precede the elements themselves if the solver needs to allocate internal memory to hold the elements.
RSI_ASYNCHRONOUS_EXECUTION	If set, then the solver supports asynchronous execution through the IRSolverAsynchSolve interface. The IRSolverAsynchSolve::AsynchSolve function of the solver will typically immediately return, and the client will then be notified via progress updates or notifications when the actual execution of the solver has completed. If not set, then the solver does not support asynchronous execution.
RSI_SYNCHRONOUS_EXECUTION	If set, then the solver supports synchronous execution. Calling the Solve function will invoke the solver, and it will not return until execution is complete. The client can receive progress updates and notifications nested within the Solve function. If not set, then the solver does not support synchronous execution.

Table 3.9: Bitmask values for IRSolverInfo::GetFlags

The remaining functions of IRSolverInfo provide accessors to the subordinate objects of the solver information:

GetInputInfoCount. Returns the number of input information objects. This is typically the same as the number of inputs to the solver.

GetInputInfo. Given an index from zero to one less than that returned by GetInputInfoCount, returns the input information corresponding to that index. See IRSolverInputInfo, below. If RSI_ORDERED_INPUTS is set in the value returned from GetFlags of the solver info object, then the ordering specified by GetInputInfo's index is the order in which inputs must be set. This assumes, therefore, that the ordering of inputs is invariant on the input values for solvers that required ordered inputs.

GetOutputInfoCount. Returns the number of output information objects. This is usually the same as the number of outputs from the solver.

GetOutputInfo. Given an index from zero to one less than that returned from GetOutputInfoCount, returns the output information corresponding to that index. See IRSolverOutputInfo, below.

GetSetInfoCount. Returns the number of set information objects. This is usually the same as the number of distinct sets used by the solver.

GetSetInfo. Given an index from zero to one less than that returned from GetSetInfoCount, returns the set information corresponding to that index. See IRSolverSetInfo, below.

GetParamInfoCount. Returns the number of parameter information objects. This is usually the same as the number of parameters that can configure the solver.

GetParamInfo. Given an index from zero to one less than that returned from GetParamInfoCount, returns the parameter information corresponding to that index. See IRSolverParamInfo, below.

3.6.1.1.c Interface IRSolverInputInfo

Interface IRSolverInputInfo provides information about a single input data element to the solver. As with the other interfaces, it derives from IRSolverBaseInfo, adding these methods:

GetAssignmentFlags. Returns a bitmask indicating properties of the input with respect to assigning new values to the input. These flags include the following (Table 3.10):

RSAF_CANUNLOCK-AFTERSETTING	This flag indicates that the solver copies the input data element internally, and does not use the data element passed to it in IRSolverInputs::SetInputData. Hence, the client or mapping can unlock or destroy the data element after calling SetInputData.
RSAF_OPTIONAL	The input element is optional. If this element is not set, the solver will use a default data element.
RSAF_CANMAINTAIN-DATA	The solver can maintain the input data element during execution; for more information on the solver maintaining data, see section 3.8.6.2, page 200.

Table 3.10: Bitmask values for IRSolverInputInfo::GetAssignmentFlags

GetChangeFlags. Returns a bitmask indicating properties of the input with respect to changing input values. These flags include the following (Table 3.11):

RSICF_READONLY	The input element is read-only. Once calling IRSolverInputs::SetInputData, the client cannot change values in the data element. If the solver specifies the flag RSAF_CANUNLOCKAFTERSETTING (see GetAssignmentFlags, above), then the client cannot modify the data element returned by a call to IRSolverInputs::GetInputData.
RSICF_CANGET	The input element is accessible to clients calling IRSolverInputs::GetInputData. Some solvers might copy the input data element into internal, optimized buffers that are not themselves data elements; in these cases, it is not possible to retrieve the input by calling GetInputData, and this flag would be cleared for those inputs.

Table 3.11: Bitmask values for IRSolverInputInfo::GetChangeFlags

GetDataType. Returns the data type of each element in the input. For example, an input could contain integers, doubles, dates, strings, etc. The data type currently maps into the same types available in the COM type VARIANT.

GetDimInfoCount. Returns the number of dimension information objects attached to this input. This is usually the same as the dimensionality of the input element for the solver itself. For instance, a vector input would have one dimension whereas a matrix input would have two.

GetDimInfo. Given an index from zero to one less than that returned from GetDimInfoCount, returns the dimension information corresponding to that index. See IRSolverDimInfo, below. The dimensions of the input are ordered by the ordering provided here. So, for a matrix $A(i,j)$, index zero corresponds to i, and index one corresponds to j.

3.6.1.1.d Interface IRSolverOutputInfo

Interface IRSolverOutputInfo provides information about a single output element of the solver. As with the other interfaces, it derives from IRSolverBaseInfo, adding these methods:

GetRetrievalFlags. Returns a bitmask indicating properties of the output with respect to retrieving values to the input. Currently there are no defined retrieval flags.

GetChangeFlags. Returns a bitmask indicating properties of the output with respect to changing output values. These flags include the following:

RSOCF_READONLY	The output element is read-only. The client cannot modify the data element retrieved from IRSolverOutputInfo::Get-OutputData.

Table 3.12: Bitmask values for IRSolverOutputInfo::GetChangeFlags

GetDataType. Returns the data type of each element in the output. For example, an output could contain integers, doubles, dates, strings, etc. The data type currently maps into the same types available in the COM type VARIANT.

GetDimInfoCount. Returns the number of dimension information objects attached to this output. This is usually the same as the dimensionality of the output element for the solver itself. For instance, a vector output would have one dimension whereas a matrix output would have two.

GetDimInfo. Given an index from zero to one less than that returned from GetDimInfoCount, returns the dimension information corresponding to that index. See IRSolverDimInfo, below. The dimensions of the output are ordered by the ordering provided here. So, for a matrix $A(i,j)$, index zero corresponds to i, and index one corresponds to j.

3.6.1.1.e Interface IRSolverSetInfo

Interface IRSolverSetInfo provides information about a distinct set of the solver. As with the other interfaces, it derives from IRSolverBaseInfo, adding one method:

GetSetFlags. Returns a bitmask indicating properties of the set that the solver requires. Currently there are no defined flags, so this function should return zero.

3.6.1.1.f Interface IRSolverParamInfo

Interface IRSolverParamInfo provides information about a single parameter that configures the solver. As with the other interfaces, it derives from IRSolverBaseInfo, adding these methods:

GetDataType. Returns the data type of the parameter. For example, a parameter could contain integers, doubles, dates, strings, etc. The data type currently maps into the same types available in the COM type VARIANT.

GetDefaultValueAsString. Returns the default value used by the solver, in a string representation. This is for browsers that wish to display the default value to the user.

GetTagStrings. Returns an array of strings that indicate possible values of the parameters, for parameters that have a finite set of possible values. Associated with each string is a magic cookie that can be used to retrieve the actual value of the parameter using GetTagValue.

GetTagValue. Given a magic cookie acquired from GetTagStrings, returns the value for the parameter associated with that cookie. The client can use the value to directly set the parameter in the solver.

3.6.1.1.g Interface IRSolverDimInfo

Interface IRSolverDimInfo provides information about a dimension associated with an input or an output. Whereas each input, output, parameter, and set is unique for a given solver, the same dimension can be used repeatedly for several inputs or outputs. Dimension information is only accessible through an input or an output.

Each dimension has an associated set that is one of the sets for the solver. That is, the values of a dimension are selected from one of the solver's sets. Dimensions are distinguished from sets, however, in that a set is unique to a solver while a dimension is not. Furthermore, many dimensions, of different names and uses, can be indexed over the same set.

As with the other interfaces, it derives from IRSolverBaseInfo, adding one method:

GetSetIndex. Returns an index into the array of sets, accessed by IRSolverInfo::GetSetInfo, that parameterizes this dimension.

3.6.1.1.h Interface IRSolverProvideInfo

An important issue is how a client initially retrieves the SolverInfo for a solver (see section 3.6.1.5, below). One solution is to have the solver directly provide its SolverInfo. This is an optimized scenario for when the client already has access to the solver. If a solver provides its SolverInfo, it does so by implementing the IRSolverProvideInfo interface, which has a single method. (Note this interface does not derive from IRSolverBaseInfo, as it is not one of the SolverInfo interfaces.)

GetSolverInfo. Returns an interface pointer to the SolverInfo for the solver.

3.6.1.2 Example using the interfaces

The following Visual Basic pseudo-code example shows how to retrieve the list of inputs and their associated dimensions from the IRSolverInfo interface:

```
Sub ShowInputInfo(Info As IRSolverInfo)

    ' Retrieve the number of inputs
    Inputs = Info.GetInputInfoCount

    For i = 0 To Inputs - 1
        ' Retrieve an input information object
        InputInfo = Info.GetInputInfo(i)

        ' Get the name
        Name = InputInfo.GetName
        Print "Input #" & i & ": " & Name & vbNewLine

        ' Get the number of dimensions for this input
        Dimensions = InputInfo.GetDimInfoCount

        For j = 0 To Dimensions - 1
         ' Retrieve a dimension information object
         DimInfo = InputInfo.GetDimInfo(j)

         ' Get its name
         Name = DimInfo.GetName
         Print Name & ", "

        Next j
        Print vbNewLine
    Next i
End Sub
```

Typical output from this subroutine would look like this:

```
Input #0: Capacity

Input #1: Cost
ItemNumber,
Input #2: Weight
ItemNumber,
```

3.6.1.3 Per-solver implementations of SolverInfo

For each solver, there can be at most one SolverInfo. When developing a new solver, a developer can also choose to implement the various interfaces that make up the SolverInfo protocol to describe the solver. By implementing IRSolverProvideInfo in the solver class, a client can access the SolverInfo implementation. (See the next section for another approach to implementing SolverInfo.) This is called a "per-solver implementation" because the SolverInfo is customized for each solver. That is, two knapsack solvers would have to use different code to implement their respective SolverInfo objects.

This technique has the advantage of providing the most flexibility. While a proper restriction is that the SolverInfo is invariant upon its creation, there is no reason the same solver cannot expose a different structure if it is created for different purposes. The primary disadvantage of this technique is that it requires coding the SolverInfo interfaces and the associated object relationships by hand. A more efficient but less flexible technique is to use a generic implementation of the SolverInfo object, described next.

3.6.1.4 A generic SolverInfo: SolverInfo Definition Language

This section presents an alternative to implementing the SolverInfo interfaces for each solver. The core services provides a generic implementation of a SolverInfo object that uses as its input a *SolverInfo Type Library* (SITL), based on COM type libraries and generated using the COM interface definition language (IDL). Almost all SolverInfo objects can be described using the variant of COM IDL presented here, named *SolverInfo Definition Language* (SIDL), so that a single, generic implementation is suitable for most cases. This transforms the SolverInfo implementation from a complex set of COM objects to the development of an IDL text file that describes the SolverInfo object. With the addition of pleasant-looking user interfaces for defining solver structures, the details of the creation and interpretation of a SITL can be entirely hidden from the solver developer.

COM IDL and type libraries. COM IDL is the Microsoft extension to the OSF DCE RPC Interface Definition Language (see Box [9]) that allows a developer to describe the interface of a class independently of any implementation, implementation language, operating system, or machine. Well-defined mappings from IDL to implementation languages like C++ or Java make it possible for programs in these languages to communicate with each other. Because this solver framework is a collection of COM interfaces, they are all described by COM IDL.

A special compiler named MIDL is used to parse COM IDL files and generate the mappings for C and C++[8]. Additional IDL code instructs MIDL to also create a mapping for other

---

[8] Specifically, MIDL generates header files that describe in C and C++ the declared interfaces as abstract base classes and source files for the remoting layer of those interfaces for local and remote procedure calls.

languages in the form of a type library. A type library is a collection of interface definitions (and definitions of other, less common elements) that can be accessed by a set of standard COM interfaces (for example, ITypeInfo and ITypeLib). Normally, type libraries are used by Visual Basic, Microsoft's Java Virtual Machine, and other development environments that cannot manipulate directly C header files.

The power of the custom() attribute. An especially powerful feature of the COM IDL specification is the capability to assign custom key-value attributes to almost every element in a library and interface. The custom attribute uses a GUID as the key and a string as the value. Any element that can have attributes in the IDL specification can also have a custom attribute. For example, an interface definition can look like this:

```
[
    uuid(808DEDFF-71C5-11D1-9101-00207810C741),
    custom(4D7BA5FF-61B3-11D1-90D7-00207810C741, "John Ruark"),
    custom(4D7BA5FF-61B4-11D1-90D7-00207810C741, "42")
]
interface SampleInterface
{
};
```

This interface has two custom attributes. The particular GUIDs used as keys have meaning only to those applications that are aware of them; hence, the introspection specification defines the necessary GUIDs for the framework. Usually, an included define file can be used to simplify the use of the GUID keys, so that the above interface might instead be coded as:

```
#include "RGuidDefs.h"
[
    uuid(808DEDFF-71C5-11D1-9101-00207810C741),
    custom(RGUID_SOLVERAUTHOR, "John Ruark"),
    custom(RGUID_ANSWERTOLIFETHEUNIVERSEANDEVERYTHING, "42")
]
interface SampleInterface
{
};
```

The included header file simply uses preprocessor defines to provide textual names to the GUID keys. This makes IDL files more understandable for the human viewers.

Because of the flexibility of the type library interfaces and the capability in type libraries to assign key-value attributes to almost every element in an interface, it is easy to map the capabilities of the SolverInfo object into the type library structure.

A SolverInfo definition language. The SolverInfo Definition Language (SIDL) is an extension to COM IDL that allows the creation of a type library that represents the structure of a solver. A special generalized SolverInfo object can then use the COM type library interfaces to provide a SolverInfo implementation for any solver with its associated SolverInfo type library. This section describes the extensions to COM IDL that define SIDL.

A solver may have an associated SolverInfo. This SolverInfo can be described by a SolverInfo Type Library (SITL). The SITL is a standard COM type library that also satisfies the following conditions:

1. The SITL has three interfaces, one each for the solver's inputs, outputs, and parameters. These interfaces are described below.

2. The SITL contains a named typedef for each of the solver's sets. The typedefs are described below.

3. The SITL's library block specifies a set of custom attributes that identify its three interfaces, as described below.

Hence, the structure of the SITL as described in COM IDL looks like this:

```
[
    ... standard TLB attributes, like uuid() and helpstring()
    ... SITL custom attributes
]
library ExampleSolverInfoLib
{
    ... library imports

    // typedefs for sets
    [
        ... typedef attributes
    ]
    typedef long ItemNumber;

    // Input interface
    [
        ... interface attributes, like uuid() and helpstring()
    ]
    interface Inputs
    {
        ... Input data elements
    };
```

```
    // Output interface
    [
        ... interface attributes, like uuid() and helpstring()
    ]
    interface Outputs
    {
        ... Output data elements
    };

    // Parameter interface
    [
        ... interface attributes, like uuid() and helpstring()
    ]
    dispinterface Parameters
    {
        properties:
        ... Solver parameters as dispinterface properties
        methods:
    };
};
```

Note that the SIDL is just one way to create a SITL. An application could use the COM type library creation interfaces (e.g., ICreateTypeLib and ICreateTypeInfo) to create a SITL just as MIDL and Visual Basic do to create normal type libraries.

3.6.1.4.a SITL set typedefs

Set typedefs are essentially placeholders for unique types. A set is a collection of like types, such as integers, strings, or reals. With the restriction that sets must contain only elements that can be mapped into COM type libraries, it is easy to define a set using a typedef. For example, this IDL typedef

```
[
    uuid(808DED01-71C5-11D1-9101-00207810C741),
    helpstring("The item number to potentially place in the knapsack.")
]
typedef long ItemNumber;
```

declares a set of elements of type long with the interpretation that each element in the set is an item number or index into a data element that will describe some feature of the knapsack problem.

The typedef must have its own uuid attribute to distinguish it from other types in the library. The helpstring attribute is optional.

3.6.1.4.b SITL interfaces

The SITL must contain three interfaces. Two of these interfaces describe the input and output data elements for the solver. The third interface describes the solver's parameters.

3.6.1.4.b.1 Input and output interfaces

The two input and output data element interfaces must follow these guidelines:

1. The interface should not derive from any other interface. The primary consequence of this is that the interface is not a COM interface, because it does not derive from IUnknown. Hence, the interface is used solely as a descriptive vehicle, and not as any kind of interface that would ever be exported or implemented.

2. The interface should have a unique interface identifier, specified by the uuid attribute.

3. The interface may support the optional helpstring and version attributes, for assisting interface browsers.

4. The interface can be named anything, although "Inputs" or "Outputs" is a reasonable choice. (Custom attributes in the library attribute list will identify the input and output interfaces by their interface identifiers, so the text names need not be unique.)

5. The interface should contain a method for each input or output of the solver. The name of the method is the name of the input or output. The parameters of the method, all of which should be [in] parameters, are the names of the sets, as typedefed in the type library, that define the dimensions of the data element. The data type of the return value of the method is the data type stored in each element of the array for that data element.

6. A method may support the optional helpstring attribute for assisting interface browsers.

Example. The input and output interfaces, along with the typedef of a set they use, for a knapsack algorithm might be:

```
// Set ItemNumber
[
    uuid(808DED01-71C5-11D1-9101-00207810C741),
    helpstring("The item number to potentially place in the knapsack.")
]
typedef long ItemNumber;

// Inputs interface
[
    uuid(808DED02-71C5-11D1-9101-00207810C741),
    helpstring("The inputs for my knapsack algorithm.")
```

```
]
interface Inputs
{
    [helpstring("Specifies the capacity of the knapsack.")]
        long Capacity();
    [helpstring("Specifies the cost of each item.")]
        long Cost([in] ItemNumber i);
    [helpstring("Specifies the weight of each item.")]
        long Weight([in] ItemNumber i);
};

// Outputs interface
[
    uuid(808DED03-71C5-11D1-9101-00207810C741),
    helpstring("The outputs for my knapsack algorithm.")
]
interface Outputs
{
    [helpstring("The total cost of the optimized knapsack.")]
        long Cost();
    [helpstring("Specifies whether each item is placed into the knapsack or not.")]
        VARIANT_BOOL Selected([in] ItemNumber i);
};
```

In this example, there are three inputs. The input Capacity has no dimensions (it is a scalar) of type long. The inputs Cost and Weight each have one dimension indexed by the set ItemNumber and contain elements of type long. Hence, Cost and Weight are vectors of integers. There are two outputs of this solver. The first, Cost is a scalar of type long. The second, Selected, is a vector indexed by the same set ItemNumber where each element is a boolean value, True or False, indicating whether or not the particular piece is included in the optimal knapsack loading.

3.6.1.4.b.2 The parameters interface and IRSolverParametersInterface

Parameters to a solver are not necessarily data elements in the sense that inputs and outputs are. A primary difference between parameters and inputs or outputs is that parameters are independent of the problem data and of any other solvers in the solution network. Because of the potential increased complexity of a parameter to a solver, the parameter interface in the SITL can be correspondingly more complex. Therefore, instead of using a restricted interface as described above for inputs and outputs, the parameter description is modeled on a standard dispatch interface (i.e., a dispinterface)[9].

---

[9] For more information on dispinterfaces and implementing them, consult Brockschmidt [11].

In fact, the solver can choose to implement the described dispinterface directly to permit the assignment of parameters to the solver, obviating the need to use the IRSolverParameters interface. This capability is necessary if the solver requires parameters more complex than those supported by the data element specification. Implementing the parameters dispinterface is perhaps preferable in situations where there are many parameters as well, as the developer can leverage existing code libraries to develop dispatch interfaces whereas the IRSolverParameters interfaces must be coded by hand.

This listing shows an example parameters interface description:

```
[
    uuid(2E578553-A14D-11D1-9170-00207810C741),
    custom(GUID_RSOLVERIMPLEMENTEDINTERFACE, "True")
]
dispinterface RDBLoadParameters
{
properties:
    [id(1), helpstring("The name of the database.")]
        BSTR DatabaseName;
    [id(2), helpstring("The user name for access to the database.")]
        BSTR UserName;
    [id(3), helpstring("The password for access to the database.")]
        BSTR Password;
    [id(4), helpstring("The SQL string to load the element from the database.")]
        BSTR Command;

    [id(5), helpstring("An array of dimensions used to load the element or an ordered array of
        dimension names used to load the element.")]
        VARIANT Dimensions;
    [id(6), helpstring("An array of column names used to generate per-item properties.")]
        VARIANT ItemPropertyNames;
    [id(7), helpstring("An array of column names used the generate the elements.")]
        VARIANT ElementNames;
methods:
};
```

Note that the id attributes are required for dispinterfaces. The helpstring, as always, is optional yet desirable. Typically, parameters do not have any associated dimensions, although this is not a requirement. Often, it is easiest just to wrap any vector or array into a VARIANT SafeArray, as the last three parameters in the example above describe.

The single custom attribute shown in the example can be used to indicate that the solver actually implements the dispinterface to permit assignment of its parameters. The GUID to

use is GUID_RSOLVERIMPLEMENTEDINTERFACE and the string should resolve to a VARIANT_BOOL (i.e., "True" or "–1" for true, "False" or "0" for false).

If the solver chooses to implement the parameters dispinterface, then it must implement the special interface IRSolverParametersInterface. This interface has a single method, **GetParametersInterface,** that returns the IDispatch pointer to the solver's implementation of the parameters dispinterface. Normally this dispinterface will not be implemented by the solver directly but by a sub-object of the solver. This is due to a bug in the current COM remoting layer that enables a remote client to access only a single dispatch interface on any given object identity.

3.6.1.4.b.3 Specifying flags on methods

As declared in the IRSolverBaseInfo interface, flags for data elements, sets, the solver, and parameters are usually bit-masked values. With SIDL, each possible flag value is mapped into a custom attribute whose string resolves into a VARIANT_BOOL ("True" or "False") or into some enumeration value as defined by the particular flag. The GUID of the custom attribute is specified by appending to the characters "GUID_" the name of the particular flag, as named in the previous sections. Therefore, the flag RSI_ORDERED_INPUTS would have attribute GUID GUID_RSI_ORDERED_INPUTS. All flags default to a "False" or zero value. An example is presented here:

```
interface Inputs
{
    [custom(GUID_RSICF_READONLY, "True")]
        long Capacity();
    [custom(GUID_RSICF_READONLY, "True")]
        long Cost([in] ItemNumber i);
    [custom(GUID_RSICF_READONLY, "True"),
     custom(GUID_RSAF_OPTIONAL, "True")]
        long Weight([in] ItemNumber i);
};
```

3.6.1.4.c SITL custom attributes

With the specifications of the set typedefs and the three interfaces complete, all that remains is to tie them together into a cohesive whole. This is accomplished by placing several custom attributes in the attribute list of the type library block. Each of these attributes is a string representation of a GUID; as such, they must resolve correctly through the COM API function CLSIDFromString into a proper GUID. These attributes are listed in Table 3.13.

Attribute GUID	Attribute interpretation
GUID_RSOLVERINPUTS	The interface identifier of the inputs interface.
GUID_RSOLVEROUTPUTS	The interface identifier of the outputs interface.
GUID_RSOLVERPARAMETERS	The interface identifier of the parameters disp-interface.
GUID_RSOLVERCLSID	The CLSID of the solver that uses the SolverInfo described in the type library. This attribute is optional.
GUID_RSOLVERTYPELIBID	The LIBID of the solver that uses the SolverInfo described in the type library. This attribute is optional.
GUID_RSOLVERDESCRIPTION	The GUID of the module that enumerates the SolverDescription properties (see section 3.6.2.3.a, page 166).

Table 3.13: Custom attributes for SolverInfo Type Library

An example library block header is shown here:

```
[
    uuid(2E578550-A14D-11D1-9170-00207810C741),
    version(1.0),
    helpstring("Ruark RDBLoadInfo 1.0 Solver Info Type Library"),
    custom(GUID_RSOLVERINPUTS, "{2E578551-A14D-11D1-9170-00207810C741}"),
    custom(GUID_RSOLVEROUTPUTS, "{2E578552-A14D-11D1-9170-00207810C741}"),
   custom(GUID_RSOLVERPARAMETERS,"{2E578553-A14D-11D1-9170-00207810C741}"),
    custom(GUID_RSOLVERCLSID, "{C42B0CC2-9E2F-11D1-9169-00207810C741}"),
    custom(GUID_RSOLVERTYPELIBID, "{EF665925-9997-11D1-915F-00207810C741}")
]
library RDBLOADLib
{
    ... remainder of SIDL file
```

In the above example, the library would contain an interface with the attribute uuid(2E578551-A14D-11D1-9170-00207810C741) that describes the inputs to the RDBLoad solver, and similarly for the output interface and parameter dispinterface.

With these custom attributes, the object that implements the generic SolverInfo object can acquire the correct interfaces for inputs, outputs, and parameters, regardless of their textual names, which might change depending on the locale of the user. Also, the SolverInfo object can find the proper solver implementation through its CLSID and determine more solver information with its type library LIBID, if these are included.

3.6.1.4.d Registry settings for solvers

Beyond the standard registry settings for type libraries and COM components, there are some registry settings which help to link a solver to its SolverInfo object. These entries are used by the core services implementation of the SolverRegistrar (see section 3.9.1.3, page 216).

> [HKCR\CLSID\{*solver's CLSID*}\SolverInfo]
> GUID=*string containing the LIBID of the SolverInfo type library*
> MajorVer=*DWORD containing the major version number of the SolverInfo type library*
> MinorVer=*DWORD containing the minor version number of the SolverInfo type library*

An example, with actual values is:

> [HKCR\CLSID\{B65B5871-97E7-11D1-915B-00207810C741}\SolverInfo]
> GUID="{2C6C8B60-984D-11D1-915D-00207810C741}"
> MajorVer=1
> MinorVer=0

3.6.1.4.e Adding SolverInfo to a solver

The framework core services define and implement several global functions that aid client and solver developers in working with the generic SolverInfo implementation. These functions include:

RLoadSolverInfo. Given a file name of a (SITL) type library or an executable containing a type library (according to the rules of the COM API function LoadTypeLib), loads and returns the general SolverInfo object described by the type library.

RLoadRegSolverInfo. Given a LIBID, version number, and locale identifier (according to the rules of the COM API function LoadRegTypeLib), loads and returns the general SolverInfo object described by the type library.

RLookupInfoGuid. Given the CLSID of a solver, uses special settings in the registry (see section 3.6.1.4.d, above) to lookup the LIBID and version number of the (SITL) type library that describes the SolverInfo of the solver.

RLoadSolverInfoClsid. Given the CLSID of a solver and a locale identifier, calls RLookupInfoGuid to determine the LIBID and version number of the solver's SolverInfo, and then calls RLoadRegSolverInfo to load the SolverInfo object.

Adding the generic SolverInfo implementation to a solver is straightforward. The developer only needs to take these steps:

1. Create and compile with MIDL the SIDL file. This generates a type library file.

2. Optionally add the type library to the solver executable as an embedded binary resource, with a line like "2 TYPELIB "RMinMaxInfo.tlb"" in the resource file.

3. Add the appropriate registry settings to the solver's registration code to hook up the solver and the SolverInfo type library, as described in section 3.6.1.4.d, above.

4. Add code to register and unregister the SolverInfo type library at the appropriate times, for example in DllRegisterServer/DllUnregisterServer for DLLs.

5. Implement the IRSolverProvideInfo on the solver, with an implementation for GetSolverInfo that looks like this:

```
STDMETHODIMP CRMySolver::GetSolverInfo(REFIID riid, LPVOID* ppVoid)
{
    if( ppVoid )
    {
        IRSolverInfo* pInfo = NULL;
        HRESULT hr = RLoadSolverInfoClsid(CLSID_RMinMax, 0, &pInfo);
        if( SUCCEEDED(hr) )
        {
         hr = pInfo->QueryInterface(riid, ppVoid);
         pInfo->Release();
        }
        return hr;
    }
    return E_POINTER;
}
```

3.6.1.5 Acquiring SolverInfo

A client can acquire a pointer to a solver's SolverInfo object in a number of ways.

- If the solver supports IRSolverProvideInfo, then the client can call GetProvideInfo to acquire the SolverInfo if the solver has already been instantiated. This is the only possible way to acquire the SolverInfo object if the solver implements it own SolverInfo object.

- Given the solver's CLSID, the client can call the framework function RLoadSolverInfoClsid to acquire the SolverInfo object.

- Given the SolverInfo type library LIBID and version number, the client can call the framework function RLoadRegSolverInfo to acquire the SolverInfo object.

- Given the file name of the SolverInfo type library, the client can call the framework function RLoadSolverInfo to acquire the SolverInfo object.

3.6.2 SolverDescription: The capabilities of the solver

The SolverInfo portion of the introspection protocol enables a client to discover the structure of a solver at run-time. If two solvers have precisely the same structure, though, there is no way for a client to distinguish between the two using the SolverInfo interfaces. The SolverDescription interfaces are designed to allow a client to determine the qualitative characteristics of a solver, thereby enabling the distinction between structurally similar solvers.

The nature of SolverDescription is based on the idea that the capabilities of a solver can be described broadly by a set of key-value ordered pairs. The keys are strings that identify qualitative or quantitative concepts such as "author" or "citation," and the value for a given key is the string representation of the key's concept, such as "John Ruark." SolverDescription defines a set of standard common keys and enables the definition of new, customized keys for future solvers.

SolverDescription is substantially simpler than SolverInfo, but the general layout of the protocol remains the same. Just as a solver may provide a SolverInfo object describing its structure, it may provide a SolverDescription object describing its capabilities. The solver might implement the SolverDescription object itself, or, by following an IDL-based language, leverage a system-provided implementation of a general SolverDescription object. This is all described in the following sections.

3.6.2.1 SolverDescription interfaces

There is only a single SolverDescription object for a solver. (Compare that to the hierarchical nature of the SolverInfo specification, which has an interface for the solver and each of its part types.) The SolverDescription object for a solver implements at least the IRSolverDescription interface. Another interface, along with a core services registration component, enables the creation of custom description tags. As with many items in the framework, a description property is identified by a GUID; in this case, these are typedefed to a new type, RDESCPROPID. All of the properties available in IRSolverDescription are also indexed by individual RDESCPROPIDs, and all property values are themselves VARIANTs. A solver also exposes the IRSolverDescriptionProperties interface if it supports description property lookup by RDESCPROPID.

3.6.2.1.a Interface IRSolverDescription

Interface IRSolverDescription provides quick access to a set of common, standard description properties. Most of these properties can be supported by all solvers. All of these

functions return a blank string or NULL pointer if the particular item does not apply to the solver.

GetAuthor. Returns the author or authors of the algorithm that the solver implements.

GetDeveloper. Returns the developer or developers of the solver software.

GetCitation. Returns a bibliographic citation for an article on the algorithm that the solver implements or for an article on the solver software itself.

GetSolutionFlags. Returns a bitmask providing some solver characteristics, including the following elements:

RSD_GENERATES_FEASIBLE_SOLUTIONS	The solver generates solutions that are feasible.
RSD_GENERATES_INFEASIBLE_SOLUTIONS	The solver generates solutions that are not feasible.
RSD_GENERATES_OPTIMAL_SOLUTIONS	The solver generates solutions that are optimal.
RSD_GENERATES_NONOPTIMAL_SOLUTIONS	The solver generates solutions that are not optimal.

Table 3.14: Bitmask values for IRSolverDescription::GetSolutionFlags

If both RSD_GENERATES_FEASIBLE_SOLUTIONS and RSD_GENERATES_INFEASIBLE_SOLUTIONS are set, then the solver might output either a feasible or an infeasible solution. The same holds for the optimality or non-optimality of a solution.

GetWorstRuntime. Returns the asymptotic worst case run time as a human-readable string.

GetWorstRuntimeAsTeX. Returns the asymptotic worst case run time as a TeX string.

GetURL. Returns a URL at which the client can find more information about the solver.

3.6.2.1.b Interface IRSolverDescriptionProperties

Interface IRSolverDescriptionProperties provides description property lookup by RDESCPROPID and enumeration of all description properties of a solver.

LookupProperty. Given an RDESCPROPID, returns a VARIANT containing the description property identified by the RDESCPROPID. If the solver does not have a corresponding property for that RDESCPROPID, the method returns RSOLVE_E_INVALIDINDEX.

EnumDescriptionProperties. Returns a COM enumerator object that provides enumeration of all the description properties of the solver. The enumerator interface is of type

IEnumRDESCRIPTIONPROPERTY, and it enumerates structures of type RDESCRIPTION-PROPERTY, which has the following typedef:

```
typedef struct tagRDESCRIPTIONPROPERTY
{
    RDESCPROPID  rdpidPropId;
    VARIANT      varProp;
} RDESCRIPTIONPROPERTY, *LPRDESCRIPTIONPROPERTY;
```

3.6.2.2 Description Property IDs

There is a pre-defined RDESCPROPID for each of the properties available in IRSolverDescription. These are shown in Table 3.15.

RDESCPROPID value	Maps to IRSolverDescription method
RDESCPROPID_Author	GetAuthor
RDESCPROPID_Developer	GetDeveloper
RDESCPROPID_Citation	GetCitation
RDESCPROPID_SolutionFlags	GetSolutionFlags
RDESCPROPID_WorstRuntime	GetWorstRuntime
RDESCPROPID_WorstRuntimeAsTeX	GetWorstRuntimeAsTeX
RDESCPROPID_URL	GetURL

Table 3.15: Pre-defined RDESCPROPIDs

A solver developer can add new RDESCPROPIDs to the system; these are then available for all other solvers and clients. Registration and unregistration occurs through the IRDescriptionPropRegistration interface. A client can enumerate all of the available RDESCPROPIDs as well as lookup a single RDESCPROPID by using the IRDescription-PropEnumeration interface. Both IRDescriptionPropRegistration and IRDescriptionProp-Enumeration are implemented by an object in the core services named the DescriptionProp-Manager, which has the CLSID CLSID_RDescriptionPropManager. Normally, neither solvers nor clients need to implement either of these interfaces.

3.6.2.2.a Interface IRDescriptionPropRegistration

IRDescriptionPropRegistration, implemented by the core services object DescriptionPropManager, has two methods. **RegisterRDESCPROPID** takes as input an RDESCPROPID, a locale identifier (an LCID), and a string identifying the name of the description property. The method registers the RDESCPROPID in the registry of

RDESCPROPIDs. The method **UnregisterRDESCPROPID** takes as input an RDESCPROPID and an LCID, and removes the corresponding RDESCPROPID from the registry. The locale identifiers are provided for localization of the name of the RDESCPROPID.

As an implementation note, the list of RDESCPROPIDs registered on a system are stored with the following structure in the registry:

```
[HKLM\SOFTWARE\RuarkSoft\Solvers\RDESCPROPID\{rdescpropid}]
lcid="description"
```

An example, with actual values, is:

```
[HKLM\SOFTWARE\RuarkSoft\Solvers\RDESCPROPID\{33BC1D01-700D-11D1-90F7-
        00207810C741}]
    409="Algorithm author"
[HKLM\SOFTWARE\RuarkSoft\Solvers\RDESCPROPID\{33BC1D02-700D-11D1-90F7-
        00207810C741}]
    409="Solver developer"
```

3.6.2.2.b Interface IRDescriptionPropEnumeration

A client uses the IRDescriptionPropEnumeration interface, implemented by the core services object DescriptionPropManager, to determine information about a description property.

The interface has two methods. The first, **EnumRDESCPROPIDINFO,** takes as input a locale identifier (an LCID) and returns as output a COM enumerator object that enumerates over a set of RDESCPROPIDINFO structures. The RDESCPROPIDINFO structure is defined as:

```
typedef struct tagRDESCPROPIDINFO {
    GUID        rdescpropid;
    LCID        lcid;
    OLECHAR   pwszDescription[108];
    } RDESCPROPIDINFO, *LPRDESCPROPIDINFO;
```

In this structure, rdescpropid is the RDESCPROPID, lcid is the locale identifier, and pwszDescription is the name of the description property. The locale identifier is repeated here in order to make this structure more versatile; another interface could return all registered description properties regardless of locale, for instance.

The second method, **LookupDescription,** takes as input an RDESCPROPID and an LCID and returns the name of the description property corresponding to the provided RDESCPROPID with the given locale. This method simplifies client implementations by saving the client the

trouble of enumerating the description properties itself when it knows the specific one it needs.

3.6.2.3 SolverDescription implementations

Implementations of SolverDescription objects parallel those of SolverInfo objects. In particular, a solver can choose to implement its own SolverDescription, using IRSolverProvideInfo to return the object to the client, or it can use the generic implementation. See section 3.6.2.3.b, page 167, for information on implementing a per-solver SolverDescription.

The generic implementation of SolverDescription is contained in the same implementation for SolverInfo. In the generic SolverInfo implementation, the root of the SolverInfo hierarchy, the object that implements IRSolverInfo also implements the SolverDescription interfaces IRSolverDescription and IRSolverDescriptionProperties. The specification of the solver's description properties is correspondingly folded into the SIDL and SITL specifications, as described next.

3.6.2.3.a Rolling into SolverInfo definition

Section 3.6.1.4, page 151, describes the SolverInfo Definition Language, based on COM IDL. The SolverDescription of a solver can be folded into the SIDL for a solver using the following steps:

1. Within the SIDL type library block, add a module block, with its own GUID attribute.
2. In the type library block for the SITL, add a custom attribute with attribute GUID GUID_RSOLVERDESCRIPTION and with value of the string version of the GUID of the newly created module block.
3. For each description property for the solver, create a constant value. The constant value should have these characteristics: Its helpstring attribute should be the string version of the RDESCPROPID for the property[10]. Its type should be the type of the description property, and should be an automation-compatible type (i.e., map into VARIANTs). Its name should be descriptive of the property type, such as "Author" or "Developer."

A sample module block with its library block header is shown here:

```
[
helpstring("RKnap 1.0 Solver Info Type Library"),
custom(GUID_RSOLVERDESCRIPTION, "{808DED05-71C5-11D1-9101-00207810C741}"),
```

[10] Constants in COM IDL modules cannot have uuid attributes, so helpstring is used instead.

```
//... other attributes, such as for GUID_RSOLVERINPUTS
]
library RKNAPINFOLib
{
    [
        uuid(808DED05-71C5-11D1-9101-00207810C741),
        dllname(""), helpstring("SolverDescription")
    ]
    module RKNAPSolverDescription
    {
    [helpstring("{33BC1D02-700D-11D1-90F7-00207810C741}")]
        BSTR const Developer = "John D. Ruark";
    [helpstring("{33BC1D03-700D-11D1-90F7-00207810C741}")]
        BSTR const Citation = "Nemhauser, G.L., and L.A. Wolsey, Integer and Combinatorial
        Optimization, John Wiley & Sons, New York, 1988, p. 433-434.";
    };
```

The generic implementation of the SolverInfo object will expose the SolverDescription interfaces if the description module is present, and it will enumerate the provided constants as the description properties of the solver.

3.6.2.3.b Adding SolverDescription to a solver

Adding a per-solver implementation of a SolverDescription is simple. Just create and implement an object that implements the SolverDescription interfaces, returning properties appropriate for the solver, and then provide a reference to this object in the implementation of IRSolverProvideInfo on the solver object. If the solver also provides its own SolverInfo through IRSolverProvideInfo, the SolverDescription interfaces should be implemented on the SolverInfo object that implements the IRSolverInfo interface.

Adding the generic implementation of a SolverDescription is equally simple, assuming that the solver already supports the SolverInfo object. The same core services functions, RLoadSolverInfo, RLoadRegSolverInfo, and RLoadSolverInfoClsid, that are used to load SolverInfo objects can be used to load SolverDescription objects, as the generic implementation of the SolverInfo object is the same as the SolverDescription. The client just needs to acquire the correct interface, as shown below:

```
IRSolverInfo* pInfo = NULL;
IRSolverDescription* pDesc = NULL;
HRESULT hr = RLoadSolverInfoClsid(CLSID_RMinMax, 0, &pInfo);
If( SUCCEEDED(hr) )
{
    hr = pInfo->QueryInterface(IID_IRSolverDescription, (LPVOID*)&pDesc);
    pInfo->Release();
```

```
}
//...use pDesc->
```

3.6.2.4 Acquiring SolverDescription

From the client's perspective, acquiring the SolverDescription is equivalent to acquiring the SolverInfo of a solver. If the client has an active pointer to the solver and the solver supports IRSolverProvideInfo, the client should use GetSolverInfo, asking for one of the SolverDescription interfaces, to acquire the SolverDescription. Otherwise, the client should first acquire the SolverInfo of the solver (see section 3.6.1.5, page 161), and then QueryInterface the SolverInfo interface pointer for one of the SolverDescription interfaces.

3.6.3 Discovering capabilities through QueryInterface

The previous two methods of determining capabilities or information about a solver rely on custom COM interfaces and their methods to specifically return that information. A third technique is to rely on the dynamic, run-time querying capabilities of the IUnknown::QueryInterface mechanism.

To recap briefly, the interface IUnknown, from which all custom COM interfaces derive, contains a method named QueryInterface. QueryInterface takes as input an identifier to a desired interface (an IID) and provides as output a pointer to the implementation of the requested interface if the object supports it or an error code if the object does not. Hence, from any COM interface of a COM object, it is possible to determine whether that object supports any specific COM interface.

By logically grouping related solver functionality into distinct, disjoint, and orthogonal interfaces, it is possible to determine if a solver supports a specific capability at run-time by simply querying the solver for the interface that defines that capability's protocol.

For instance, the interface IRSolverProvideInfo contains a single method, GetSolverInfo, that returns a pointer to the SolverInfo object for the solver. If a solver has the capability to provide its own SolverInfo object, then it declares that capability by implementing the IRSolverProvideInfo interface and handing a pointer to that interface to clients when they request that interface. If a client queries a solver for IRSolverProvideInfo and QueryInterface returns E_NOINTERFACE, then the solver does not have the capability to provide its own SolverInfo object, and the client will have to look elsewhere to acquire that object.

For more details on the QueryInterface mechanism, requirements, and philosophy, see Brockschmidt [11] and Box [9].

3.7 Progress Updates and Life Cycle Control

Section 2.3.4, page 81, identified the need for a client to be able to determine the progress of a solver, particularly for solvers that require excessive time to execute. There are two primary models for acquiring progress updates from a solver: the push model and the pull model. In the push model, clients and other interested entities register with (advise) the solver to receive notifications, and then during processing the solver pushes progress notifications to the registered clients. In the pull model, a client asynchronously queries the solver at any time during processing to pull the current progress from the solver. The framework defines interfaces for both types of progress updates, and a client or solver is free to choose the most appropriate methods.

Associated with the progress of a solver is the control of a solver. Section 2.3.5, page 84, discussed the need for the ability for a client to terminate or otherwise control the solver's execution. Some level of control is provided as part of the progress notification specification, but the framework also defines an interface, IRSolverControl, for direct, asynchronous control of a running solver.

3.7.1 Progress notifications (push)

The progress notification (push) method for progress updates uses the Observer (publish-subscribe) pattern (see Gamma et al. [30]). The main feature of this pattern is that the client and the solver switch roles—the client becomes a server, and the solver becomes a client. That is, the client implements the notification interface while the solver calls methods in that interface. This is typical of notification patterns and callbacks. The notification interface in this case is named IRSolverAdvise.

There are three main steps:

1. A client interested in receiving progress notifications registers with the solver prior to the solver's execution.

2. During processing, the solver sends progress update notifications to the registered clients through the IRSolverAdvise interface.

3. When the solver is destroyed or when clients are no longer interested in receiving progress update notifications, they unregister with the solver.

The following section discusses the registration and unregistration steps, and the subsequent section describes the actual notification protocol and IRSolverAdvise.

3.7.1.1 Registering for notifications

Two methods of the primary solver interface, IRSolver, allow a client to register and unregister with the solver to receive notifications.

SolveAdvise. The client calls IRSolver::SolveAdvise to establish a progress update advise connection with the solver. SolveAdvise takes as input a pointer to the client's implementation of the IRSolverAdvise interface (see the next section). It returns as output a magic cookie that identifies this particular connection. The cookie is unique in the context of all clients that have called SolveAdvise for that particular instantiation of the solver. Clients must use that magic cookie when referring to this connection.

SolveUnadvise. The client calls IRSolver::SolveUnadvise to disconnect a progress update advise connection with the solver. SolveUnadvise takes as input the magic cookie that the client had previously received through a call to SolveAdvise.

Interested clients must call SolveAdvise before the solver begins processing. They can call SolveUnadvise any time after calling SolveAdvise. During processing, any active connections might receive progress update notifications at any time.

3.7.1.1.a Implementing SolveAdvise and SolveUnadvise

SolveAdvise and SolveUnadvise are analogous to the standard COM interface methods IDataObject::DAdvise and IDataObject::DUnadvise. In particular, if the solver does not support progress update notifications, it should return the standard error OLE_E_ADVISENOTSUPPORTED from SolveAdvise and SolveUnadvise. If an invalid magic cookie is passed to SolveUnadvise, it should return the standard error OLE_E_NOCONNECTION.

Solvers that do support progress notifications must maintain a list of the connection pointers (IRSolverAdvise interface pointers) and their associated magic cookies. This list must support insertion, removal, and enumeration of elements and secondarily lookup by magic cookie. Maps are ideally suited for this purpose.

3.7.1.1.b Simplifying implementation with IRSolverAdviseHolder

The previous section described the features of a map of interface pointers and magic cookies necessary for progress update notifications. Because most solvers will implement such a list, the core services provide a special object called the SolverAdviseHolder that implements the necessary functionality of this list, simplifying solver development considerably. The SolverAdviseHolder takes care of managing the map of pointers and of walking through the map to send updates whenever the solver wants to send a notification.

Generally, the solver will create an instance of a SolverAdviseHolder for each instance of the solver. The SolverAdviseHolder has CLSID CLSID_RSolverAdviseHolder, and the solver can create it with the normal call to CoCreateInstance. The SolverAdviseHolder supports the IRSolverAdviseHolder interface, which has the following methods:

Advise. Takes as inputs pointers to the solver's implementation of IRSolver and the client's implementation of IRSolverAdvise, adds the client's pointer to the SolverAdviseHolder's internal map, and returns a magic cookie for the solver to pass back to the client.

Unadvise. Takes as input the magic cookie passed from the client to the solver and removes the interface pointer that corresponds to that connection from its internal map.

SendOnSolveComplete. Sends the OnSolveComplete notification to all connections in the internal map. See section 3.7.1.2.b, page 174, for more details.

SendOnSolveNotify. Sends the OnSolveNotify notification to all connections in the internal map. See section 3.7.1.2.a, page 173, for more details.

The solver can incorporate the SolverAdviseHolder object simply, by following these steps (examples are shown for C++).

1. Add an interface pointer to IRSolverAdviseHolder to the class that implements the solver. In the header file for the solver class:

```
Class CRMySolver : public IRSolver
{
    IRSolverAdviseHolder* m_pSolverAdviseHolder;
    // remainder of class definition
```

2. In the constructor of the solver class, make sure to initialize the pointer, and in the destructor make sure to release the pointer if it is non-zero:

```
CRMySolver::CRMySolver() : m_pSolverAdviseHolder(0)
{
}

CRMySolver::~CRMySolver()
{
    if( m_pSolverAdviseHolder )
        m_pSolverAdviseHolder->Release();
}
```

3. Add implementations of SolveAdvise and SolveUnadvise that look like this:

```
STDMETHODIMP CRMySolver::SolveAdvise(IRSolverAdvise* pAdvise, DWORD* pdwCookie)
{
    if( pdwCookie )
    {
        if( !m_pSolverAdviseHolder )
        {
         HRESULT hr = CoCreateInstance(CLSID_RSolverAdviseHolder, NULL,
        CLSCTX_SERVER, IID_IRSolverAdviseHolder, (LPVOID*) &m_pSolverAdviseHolder);
         if( FAILED(hr) ) return E_OUTOFMEMORY;
        }
        return m_pSolverAdviseHolder->Advise(static_cast<IRSolver*>(this), pAdvise,
        pdwCookie);
    }
    return E_POINTER;
}

STDMETHODIMP CRMySolver::SolveUnadvise(DWORD dwCookie)
{
    if( !m_pSolverAdviseHolder || !dwCookie )
        return OLE_E_NOCONNECTION;
    return m_pSolverAdviseHolder->Unadvise(dwCookie);
}
```

4. Add at the point where the solver wants to update clients of its progress the appropriate call to SendOnSolveNotify, and upon completion of processing, add a call to SendOnSolveComplete:

```
// ... begin processing
bool bProcessing = true;
while( bProcessing )
{
    ... do some work
    // now send notification
    if( m_pSolverAdviseHolder )
        m_pSolverAdviseHolder->SendOnSolveNotify(...parameters...);

    if( ...check for done... )
        bProcessing = false;
}
// complete, send complete notification
if( m_pSolverAdviseHolder )
    m_pSolverAdviseHolder->SendOnSolveComplete(...result...);
```

3.7.1.2 Sending/receiving notifications

In order to receive progress update notifications, the client must implement the IRSolverAdvise interface. To send notifications, the solver calls a method of the IRSolverAdvise interface for all active advise connections.

As described in section 2.3.4, page 81, there are two types of interesting updates. One is percentage complete; the other is estimated time remaining. Both of these are captured in a single method of the IRSolverAdvise interface, OnSolveNotify. Additionally, this interface defines another method, OnSolveComplete, that should be sent when the solver has finished all of its processing.

3.7.1.2.a IRSolverAdvise::OnSolveNotify

OnSolveNotify is the workhorse of the progress updates protocol. This function is designed to support percentage complete notifications, estimated time remaining notifications, or both at the same time. The function has three groups of inputs, as shown in Table 3.16:

dwFlag	A flag indicating what type of notification is being sent. See below.
dPercComplete	A double indicating the fraction complete if that type of notification is being sent. The value should be between 0 and 1. Multiple by 100% to determine the percentage complete.
lHoursLeft lMinutesLeft lMillisecondsLeft	Integers indicating how many hours, minutes, and milliseconds are left, respectively, if that type of notification is being sent. The total time left for the solver is the sum of all three values.

Table 3.16: Parameters of IRSolverAdvise::OnSolveNotify

The flag dwFlag is a bitmask that can take on the values in Table 3.17.

RSNF_PERCENTAGE	The parameter dPercComplete is valid.
RSNF_HOURS	The parameter lHoursLeft is valid.
RSNF_MINUTES	The parameter lMinutesLeft is valid.
RSNF_MILLISECONDS	The parameter lMillisecondsLeft is valid.
RSNF_TIME	All three time parameters are valid.

Table 3.17: Bitmask values for IRSolverAdvise::OnSolveNotify

The function has a single output that indicates the action the solver should take after notifying all observers. This output can take the following values (Table 3.18):

RSNA_CONTINUE	The solver should continue processing.
RSNA_CANCEL	The solver should stop (cancel) processing as soon as possible.

Table 3.18: Action codes for IRSolverAdvise::OnSolveNotify

Additionally, this action can have the bit RSNA_STOPNOTIFY set, which instructs the solver to not notify any other registered clients of the progress notifications. This is useful if there has been some catastrophic failure and the client is telling the solver to cancel at the earliest possible moment.

The flexibility of the time parameters allows the solver to provide estimated time left notifications from 1 millisecond to over 49 days with a granularity of one millisecond and up to almost half a million years with a granularity of one hour. That should be more than enough time for most solvers. The primary reason for this classification is to make conversion on the client side easy. If a solver can only estimate on the order of hours, it can set the RSNF_HOURS flag only and provide a best guess in the lHoursLeft parameter.

3.7.1.2.b IRSolverAdvise::OnSolveComplete

The second function of IRSolverAdvise is OnSolveComplete. This notification should be sent from the solver after it has finished its processing and is in its final state. The notification includes a single enumerated parameter that indicates the result of the solver's processing. It can take one of the values in Table 3.19.

RSCR_OK	The solver finished successfully.
RSCR_OPTIMIZED	The solver finished successfully and found an optimal solution.
RSCR_CANCELLED	The solver was cancelled during processing.
RSCR_OUTOFTIME	The solver ran out of time before completing (if, for instance, one of its parameters was a maximum time bound).
RSCR_MAXITERATIONS	The solver reached a maximum number of iterations before completing.
RSCR_APPROXIMATED	The solver finished successfully and found an approximate, not necessarily optimal or non-optimal, solution.

Table 3.19: Notification codes for IRSolverAdvise::OnSolveComplete

Some solvers do not optimize but only calculate; in these cases, RSCR_OK is preferred over RSCR_OPTIMIZED.

3.7.1.3 Assumptions about solver state during notification

The progress update notification is a simple arrangement. There are only a few assumptions about the state of the solver during notifications that must be observed. First, whenever the solver calls OnSolveNotify on its advised clients, its internal state must be consistent, because the clients might very well query other interfaces of the solver during the notification. This is true of most notifications; for example, at the end of an iteration a client might wish to determine per-iteration values such as an optimality gap. These values, when requested, must be consistent with the state of the solver at the time of the notification.

Second, the solver should not call OnSolveComplete until it has completely finished processing. Again, the internal state of the solver at completion must be consistent. It is usually best to wait to call OnSolveComplete until just before returning from the solver execution method.

3.7.2 Progress queries (pull)

The progress query (pull) mechanism gives a client running asynchronously with the solver the ability to query the solver for its current status and progress at any time. A solver supports progress queries by implementing and exporting the IRSolverStatus interface. If a client has any interface pointer to a solver, it can call QueryInterface to obtain the IRSolverStatus interface, and then it can asynchronously query the solver for its status.

Hence, there are two issues to resolve. First, how does the client get an interface pointer to the solver to begin with, and second, how shall it query the solver for its status? These are discussed in detail below.

3.7.2.1 Acquiring a running solver

The first issue is important in this scenario because the solver has no *a priori* knowledge of the client, as it does in the progress update notifications (push) mechanism. The usual scenario in the push case is that there is some global entity that is at the very least hooking all of the various clients, solvers and data elements to each other prior to initiating execution. It is reasonable, thus, that the various advises can be set up before everything gets started. In the pull case, however, an entirely random client might want to just step in, get the status of some running solver, and depart. For example, a separate utility application might try to attach to all running solvers on a system to determine each of their status in order to report to the user.

The usual way in COM for an application to acquire an already existing object is to use the *running object table,* and the framework will leverage that existing capability. Namely, if a solver

supports asynchronous progress queries from any client, it should register a moniker identifying itself in the running object table using IRunningObjectTable::Register. A good solution is to use the Composite Moniker comprising a Class Moniker to identify the solver class object and an Item Moniker to identify a particular running instance of the solver.

All the client then needs is the CLSID of the solver and a name for the particular running instance. The CLSID can be determined from the ProgID, for instance. With these two items, it can create a moniker, query the running object table using IRunningObject-Table::GetObject, and then call QueryInterface for IRSolverStatus if that was successful.

An obvious simplification of this process is to provide a core services object that wraps this functionality into a simpler interface. This *running solver table* is discussed in more detail in section 3.9.2, page 216.

3.7.2.2 Querying the solver

Because of the asynchronous nature of the query relationship between client and solver in the pull scenario, the client does not have the luxury of presuming that the solver is in any consistent internal state. As such, the client needs to make sure to only call methods on the IRSolverStatus interface, which defines the capability to retrieve both the state status and the progress of the solver.

3.7.2.3 Interface IRSolverStatus

The first method in IRSolverStatus enables a client to determine broadly in what state a solver is. **GetStatus** returns a bitmask that can contain any combination of the values in Table 3.20.

RSS_UNINITIALIZED	The solver has been created but has not been initialized with any inputs or outputs. Only default values might be set.
RSS_INPUTSVALID	All inputs of the solver have been set. The client could query the solver to receive the input data elements.
RSS_SOLVING	The solver is currently solving its problem.
RSS_PAUSED	The solver is currently paused in the process of solving its problem.
RSS_OUTPUTSVALID	The solver has solved its problem, and its outputs are valid.
RSS_DESTROYING	The solver is being destroyed.

Table 3.20: Status codes for IRSolverStatus::GetStatus

The second method, **GetProgress,** is the pull version of the IRSolverAdvise::OnSolverNotify notification method. It has no inputs and receives as outputs the same values that the solver would send with OnSolverNotify; see section 3.7.1.2.a, page 173, for more details.

3.7.3 Solver life cycle control

Section 2.3.5, page 84, discussed the need for the ability for a client to terminate the solver's processing. The progress update notification, described in section 3.7.1, page 169, provides one means for termination. A return parameter of the function IRSolverAdvise:: OnSolve-Notify allows the client to instruct the solver to cancel its processing.

One situation that cannot use this mechanism is analogous to the asynchronous progress query of section 3.7.2. Some client that has no advise connection or even any relation whatsoever with the solver might need to terminate the solver upon the user's request. To handle this, the framework defines the interface IRSolverControl. If the solver supports asynchronous termination then it implements and exports IRSolverControl. IRSolverControl is an optional interface, and a solver can choose to support synchronous control through IRSolverAdvise::OnSolveNotify but to not support asynchronous control via IRSolverControl.

3.7.3.1 Acquiring a running solver

The client that wishes to control the solver must first acquire a pointer to the solver. This can be accomplished in the same way as a client attempting to query the solver for its progress; see section 3.7.2.1, page 175, for details.

3.7.3.2 Interface IRSolverControl

IRSolverControl includes the following methods:

Abort. Instructs the solver to cancel its processing at the earliest possible moment. The solver should conceptually discard its internal state of the problem it is solving.

Pause. Instructs the solver to suspend its processing at the earliest possible moment. The solver should retain its internal state, on the assumption that the client will instruct it to resume its processing.

Resume. Instructs the solver to resume previously paused processing.

Wait. Instructs the solver to pause for a specified amount of time during its processing. This might be useful if the client needs access to the system resources for a known time amount.

3.8 Networking Solvers

Everything that has come before is applicable to solvers in single-stage architectures. With everything in the preceding sections and in the core services, to follow, many useful solutions can be built. Now it is time to turn to solving the problem of linking multiple solvers into a solution network.

Throughout, the development of the networking framework will refer to this example. Consider a four-stage solver network with the architecture shown in Figure 3.9.

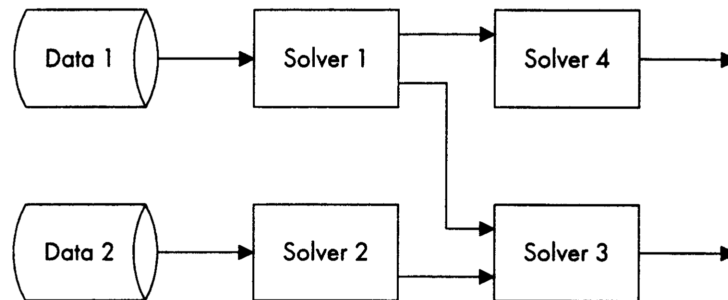


Figure 3.9: Sample solution network for framework development

This network has two data sources that each serve as an input to one of two solvers. A third solver uses the outputs of both of the first two solvers as its inputs, while a fourth solver uses only the outputs from the first solver as its inputs. The outputs from the third and fourth solvers are the end results.

The next section examines how traditional client applications might manage this solution network, which also happens to be a method for global control. Subsequent sections introduce aspects of the framework that relieve implementation work from clients and enable simplified local control.

3.8.1 Networks in traditional clients and global control

For a fixed, *a priori* known network such as the example network, the traditional implementation is also the most obvious. Not coincidentally, it is also the simplest. Independent of whatever the client does, in terms of user interface, data input, etc., the routines that solve the solution network can be purely procedural, in the grand tradition of flowchart-based design. The client's solution routine simply needs to:

1. Create or load the four solver objects or libraries.
2. Query or load Data 1.

3. Pass Data 1 to Solver 1, and run Solver 1.

4. Query or load Data 2.

5. Pass Data 2 to Solver 2, and run Solver 2.

6. Pass the outputs of Solver 1 and Solver 2 to Solver 3, and run Solver 3.

7. Pass the outputs of Solver 1 to Solver 4, and run Solver 4.

8. Retrieve the outputs from Solver 3 and Solver 4.

9. Destroy or unload the four solver objects or libraries.

In a client application where the solution network is embedded into the application and is inflexible, this nine-step procedure can be hard-coded into the application logic. When maximal control over performance, robustness, solver location, and efficiency is paramount, this is an ideal choice. The client application has complete, global control over the state of the entire network. Solvers do not act without instructions from the client.

Everything presented in the framework until this point can be leveraged by clients, for instance, to manage the passing of data, the initialization and parameterization of solvers, and solver selection. The framework does not interfere with the client's global control of a network of solvers.

The problem for the client arises when the underlying solution network is exposed directly to the user for manipulation. When the network structure can change, either through the addition or removal of solvers or of links between them, the client application can no longer hard-code the solution procedure. Instead, the client must dynamically create something akin to a finite state machine that ensures that the solvers execute in an order such that each solver has all of the inputs it requires when it executes. If the network is a directed acyclic graph, then a topological ordering of the network leads to the solver execution order. If there are cycles with decision points, then the client will need to be correspondingly more sophisticated.

In any event, the client is still a global controller. All of the intelligence of the network topology resides in a single place, at the client. Without the support of the framework, implementing local control—spreading topology knowledge across the network—would require much custom work on the parts of the client or solver developers. That is, solvers would require special code to deal with local control scenarios, which is functionality orthogonal to the purpose of solvers, and, hence, best avoided. The following sections develop the framework in order to mitigate that development burden.

3.8.2 Moving to local control

The goal of the networking framework is to minimize clients' and solvers' workloads in local control situations. Ideally, the behavior of the client is quite simple. Given the network topology as an input, the client should only have to create the solvers and the links, hook them all together, say "Go!" and then sit back, waiting for the answer to unfold. Obviously, there will be some more work than that, but the overall work should be minimized.

The need, then, is to move the various pieces of the control logic away from the client without forcing the solvers to shoulder any of the burden. This will require new components that sit within the network between the various solvers and data flows. It is these new components that the framework specifies.

In the context of the example network, of the original nine tasks, the following must be moved out of the client and into local control components:

1. Pass Data 1 to Solver 1, and run Solver 1.

2. Pass Data 2 to Solver 2, and run Solver 2.

3. Pass the outputs of Solver 1 and Solver 2 to Solver 3, and run Solver 3.

4. Pass the outputs of Solver 1 to Solver 4, and run Solver 4.

These four tasks can be categorized into two types of activities: passing data and running solvers. The client is still responsible for creating and destroying the solvers, establishing the initial data inputs prior to solver execution, and retrieving the final outputs after the final solver completes processing[11]. In essence, through creation and initial inputs, the client brings the network into an initialized, steady state, and through final outputs and destruction, the client takes the network out of steady state. During this "steady state," the local control components run the network.

Now consider the network from the perspective of one of the solvers. Sitting on top of Solver 3 in the global control scenario, what does an observer see? First, Solver 3 is created (giving the observer some place to sit). For a long time, nothing happens, as, unbeknownst to the observer, Solver 1 and Solver 2 execute. Then, Solver 3 suddenly receives the output of Solver 1 followed by the output of Solver 2, both coming from the client. Then the client instructs Solver 3 to execute, which it does. Upon completion, Solver 3 notifies the client it is finished, and then the client takes the output of Solver 3. Afterwards, the observer on Solver 3 is idle until Solver 3 is destroyed. This is shown (minus destruction, which works the same as creation in all examples) in Figure 3.10.

---

[11] The fourth chapter will describe solver components that manage data storage and retrieval, thereby elevating data access to the level of solvers and relieving data management duties from some clients.

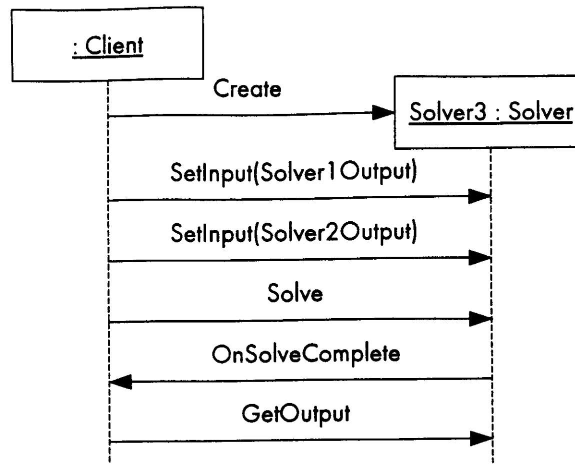


Figure 3.10: Perspective of Solver 3 under client global control

The desired behavior in a local control is different, as the solvers acquire some autonomy in their environment. Again, Solver 3 is created by the client. The client then tells Solver 3 that it will be using the outputs of Solver 1 and Solver 2, and that it should wait for their outputs to become available. So, the observer and Solver 3 wait as Solver 1 and Solver 2 execute. Thus, the solver has some context of its environment; it knows it is waiting for two solvers to finish before it can begin execution. Suppose Solver 1 finishes first; it notifies Solver 3 that it is done. Solver 3 takes the output from Solver 1 and sets its input. The waiting continues, now only for Solver 2. When Solver 2 finishes, it notifies Solver 3, which takes output from Solver 2 and sets its own input. With all of its inputs set, Solver 3 then begins executing. When complete, it notifies the client, which then retrieves its output. Afterwards, the observer is idle until Solver 3 is destroyed. This is presented in Figure 3.11, below.

The main differences are that the solver knows its whereabouts within the network, so to speak, it knows what it is waiting for to begin, and it has control over its own execution. However, adding this intelligence to every solver needlessly complicates solvers. Network knowledge and local control functionality are orthogonal to the purpose of a solver, which is to implement an algorithm. The solution within the framework is to add helper components into the network, at the point of each solver, to handle the needs of the network. A component sits in the network as a surrogate for the solver, wrapping the solver, handling the notifications and local control. The solver is shielded inside the component, and to the solver it appears as if there is global control. This component is called a wrapper, and each solver is wrapped by one. This is shown in Figure 3.12.

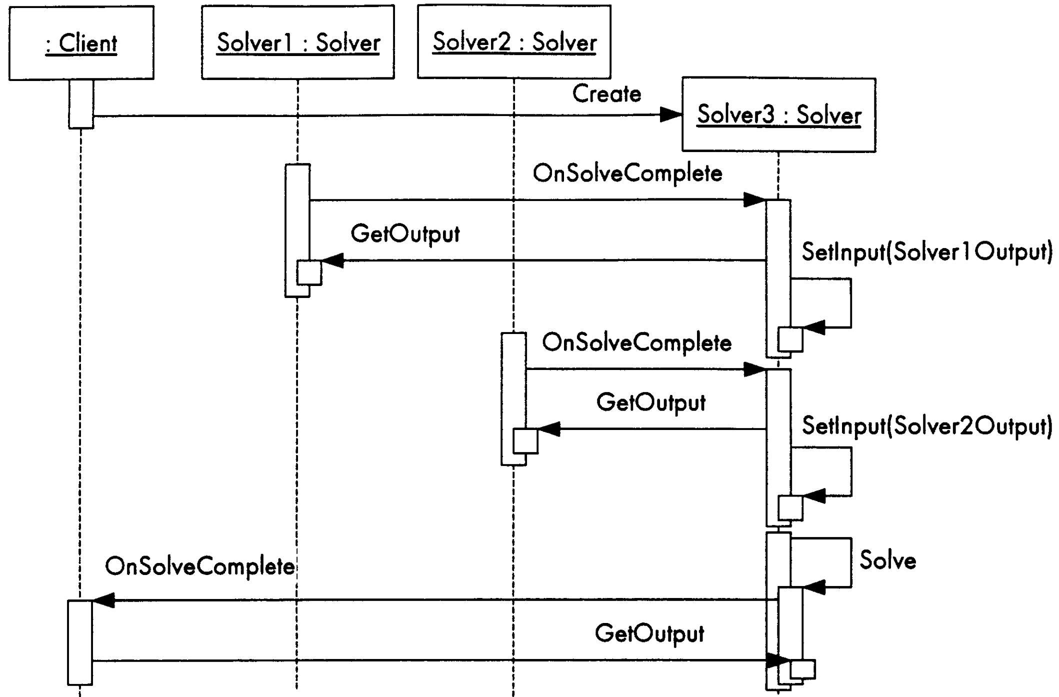


Figure 3.11: Perspective of Solver 3 under local control

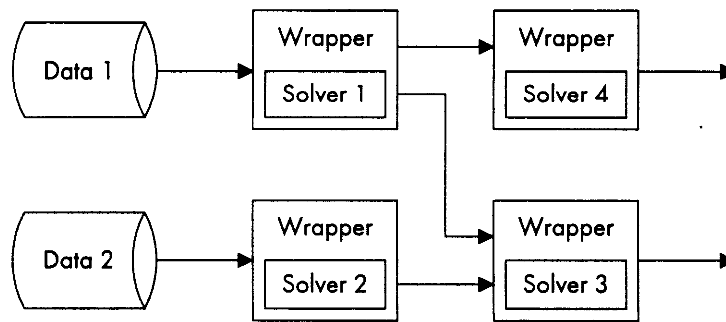


Figure 3.12: Sample solution network with wrapper components

If the observer were sitting on the wrapper of Solver 3, things would look a lot like the local control scenario just described. The client creates the wrapper, which in turn creates Solver 3. The client tells the wrapper that Solver 3 will be using the outputs of Solver 1 and Solver 2. So, the wrapper waits for Solver 1 and Solver 2 to finish. When Solver 1 finishes, the wrapper takes it output and perhaps stores it at an intermediate location. When Solver 2 finishes, the wrapper passes both Solver 1's output and Solver 2's output on to Solver 3, and instructs Solver 3 to execute. When Solver 3 is finished, it notifies the wrapper, which then notifies the client. The client retrieves the outputs from the wrapper, which retrieves them from Solver 3. When the client destroys the wrapper, it destroys Solver 3. This is presented in Figure 3.13, below. The wrapper is now a surrogate for Solver 3, and passes the important activities on to Solver 3 when necessary, such as retrieving outputs, creation, and destruction.

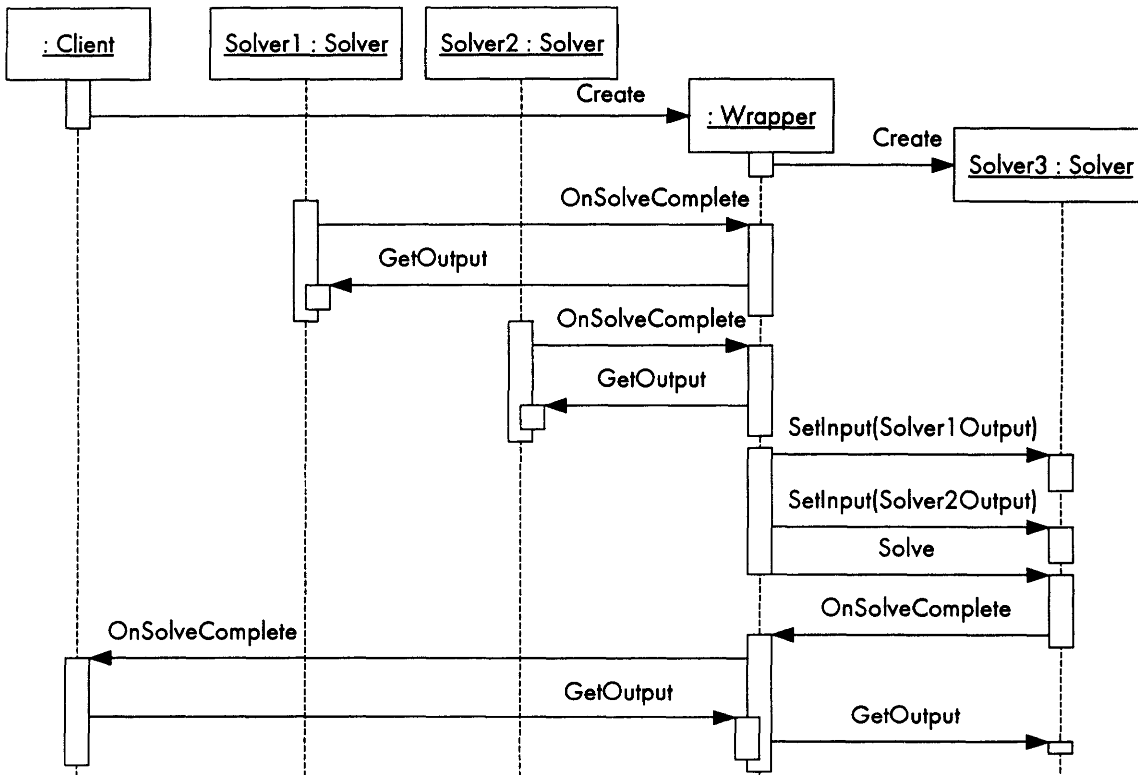


Figure 3.13: Perspective of a Solver 3 wrapper under local control

Place the observer back at Solver 3, and consider only those events from Figure 3.13 that Solver 3 receives or generates. As shown in Figure 3.14, below, the observer at Solver 3 will feel like it is back in the original, global control scenario, where the wrapper has taken the place of the client. Solver 3 is created, and then nothing happens as Solver 1 and Solver 2 execute. Then, Solver 3 suddenly receives the output of Solver 1, followed by the output of Solver 2, both coming from the wrapper. The wrapper instructs Solver 3 to execute, and

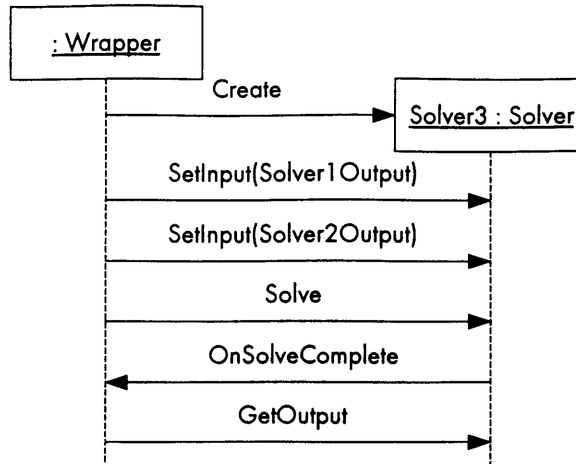


Figure 3.14: Perspective from Solver 3 under local control with wrapper

upon completion, Solver 3 notifies the wrapper that it is finished. The wrapper then takes the output of Solver 3. Eventually, Solver 3 is destroyed.

The addition of the wrapper, which can be a common component, has alleviated the need for any particular solver to worry about the state of all surrounding solvers, the availability of data, or the particular run-time configuration of the system. Figure 3.14 is identical to Figure 3.10 with the wrapper taking the place of the client. This shows that a wrapper can sufficiently hide network interactions in a local control scenario so that the solver cannot distinguish between global control and local control with a wrapper.

3.8.3 Solver sites and mappings

The wrapper described in the previous section is a container for solvers. The network that the client creates is no longer a network of solvers, but instead is a network of solver containers. What were once heterogeneous solvers, the "nodes" of the network are now homogenous containers. That these containers wrap the original heterogeneous solvers does not effect their behavior in the network. The "arcs" of the network will also be wrapped, by link containers. These two containers are the embodiment of nodes and arcs in a network of solvers in the framework.

3.8.3.1 Introducing solver sites

A solver container is called a *solver site object,* or *solver site* for short[12]. Much of the framework specification relates to the behavior, structure, and interface of solver sites, as these objects are the primary enabling technology of solution networks.

---

[12] The name derives from the interfaces exposed by ActiveX control containers in Microsoft's ActiveX specifications, namely IOleControlSite, IOleClientSite, and IOleInPlaceSite.

Now that the solver site is identified, it is easier to state its responsibilities. The solver site that contains a solver must:

1. Be aware of all solvers or external sources that provide inputs to the solver. These are the solver's *suppliers.*

2. Establish advise connections to receive notifications from the suppliers when they have data available for the solver.

3. Manage the flow of data from a supplier to the solver. This includes determining where the data itself should be stored and making sure it gets there.

4. Assign all of the input data elements to the solver when all input data are available.

5. Execute the solver.

6. Provide a locking mechanism to ensure data integrity. That is, the solver site must provide locks so that the solver does not trash output data with new data from another set of input data.

7. Expose the outbound side of the solver to any clients interested in receiving notifications or output data; these clients are the solver's *customers.* Customers will need to register with the solver site so that it can manage data integrity correctly.

8. Reset the solver by clearing its inputs and outputs and freeing any data elements held by the solver or the solver site.

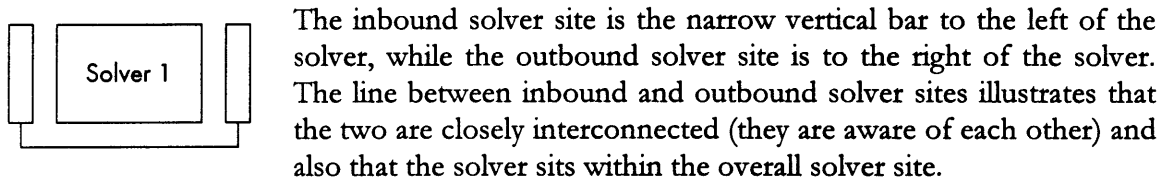
9. Possibly manage the creation and destruction of the solver itself, although alternatively this can be handled by the client.

Without loss of generality, any link in a solution network connects a supplier's solver site to a customer's solver site. Whether the supplier or customer is a client or database or some non-solver entity, it can still expose or be wrapped by the solver site object interfaces that will be presented shortly. Somewhere in that chain from the supplier's solver site through the link to the customer's solver site must lie the intelligence and the responsibility for managing the flow of data—in particular, acquiring the correct output data element from the supplier and passing it, possibly marshaling or converting it, to the correct input data element of the customer. Arbitrarily, the framework assigns this responsibility to the customer's solver site. Because a supplier might also be a customer, in fact every solver site has this responsibility. To be more precise, it is the inbound side of the solver site, called the *inbound solver site,* that has the responsibility, which can be succinctly put:

> *The inbound solver site is responsible for managing the mapping of data from the outputs of its suppliers to the inputs of its solver.*

The other end of the solver site, called the *outbound solver site,* has the easier duties. From the solver site's perspective, outputs are conceptually simple. Customers register for advise notifications, they lock whatever data they need access to, they acquire the data and take it away, and then they unlock it. Clearly the interesting activity takes place at the inbound solver site.

The functionality of a solver site is thus divided into its inbound functionality and its outbound functionality. In fact, in the framework the inbound and outbound sides can be separate objects entirely. Solvers could implement one side or the other of their own solver site, if they needed to add some specific, custom behavior. Solver sites will therefore be displayed as two bars that sit between inputs and outputs of a solver instead of a complete wrapper of the solver. (This makes more sense as well because the client will still communicate directly with the solver to set parameters, which a complete wrapper implies is not possible.) A solver site wrapping a solver appears thus:



The inbound solver site is the narrow vertical bar to the left of the solver, while the outbound solver site is to the right of the solver. The line between inbound and outbound solver sites illustrates that the two are closely interconnected (they are aware of each other) and also that the solver sits within the overall solver site.

The new image of the solver sites in the sample network is presented in Figure 3.15:

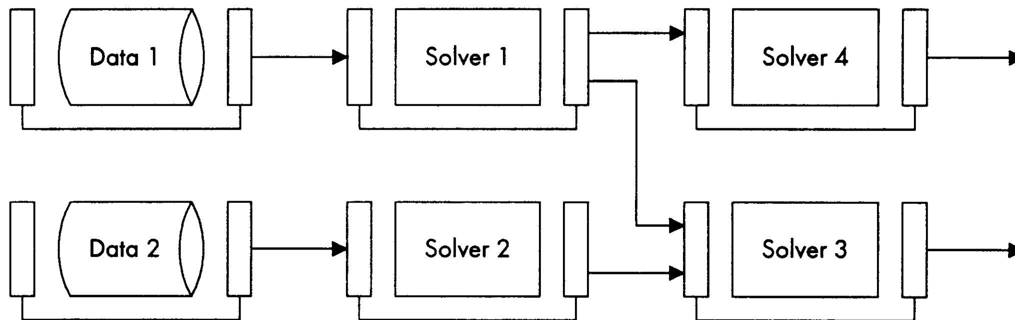


Figure 3.15: Sample solution network with solver site components

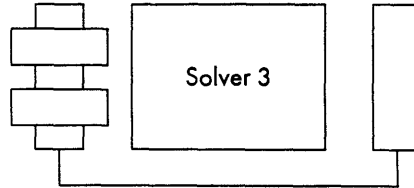
3.8.3.2 Introducing mappings

Solver sites wrap individual solvers. Without any further assistance, as a network becomes more dense and more links are added to the inbound side of a solver, the solver sites will have to manage more connections, more data, and more complexity. It will help to introduce another component associated with the links, subservient to the solver site but responsible for activity on any given connection from a supplier to a customer. This component is, in

essence, the embodiment of links in the framework. It is called a *mapping*, as it specifies the mapping from one object's outbound data elements to another object's inbound data elements.

To summarize, networks in the framework are captured by solver sites as the nodes and mappings as the links. Solver sites are containers for solvers, while (arbitrarily chosen) the inbound side of solver sites are the containers for mappings.

A mapping is displayed as a small horizontal box straddling the inbound solver site as shown here. There are two mappings on Solver 3's inbound solver site, one for each of Solver 3's inputs. The mappings overlap the inbound solver site to show that they are managed by the inbound solver site, but they extended beyond the inbound solver site to emphasize that they are the objects that retrieve suppliers' outputs and pass them to the solver.



When mappings are added to the sample solution network, the picture appears thus:

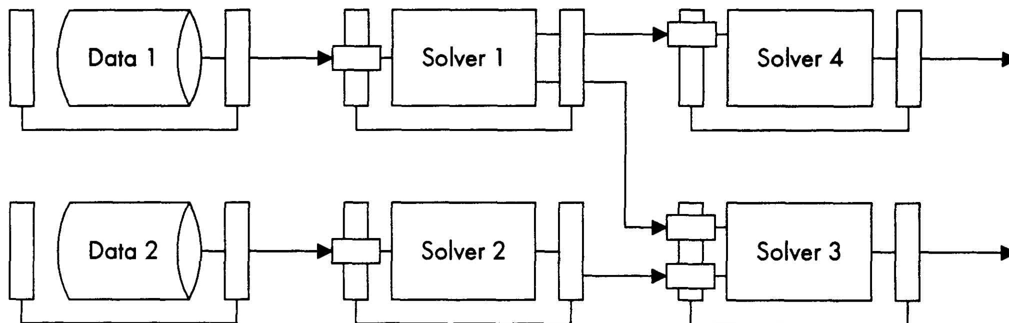


Figure 3.16: Sample solution network with mappings and internal solver connections

To complete the picture, the client, with its hypothetical implementation of the inbound and outbound solver sites, is added, where for convenience the client's inbound solver site is on the right of the client, as indicated by the presence of the mappings in Figure 3.17:

Note that in this figure, the client implementation abuts its solver site, to indicate that the site implementation is actually rolled into the client itself.

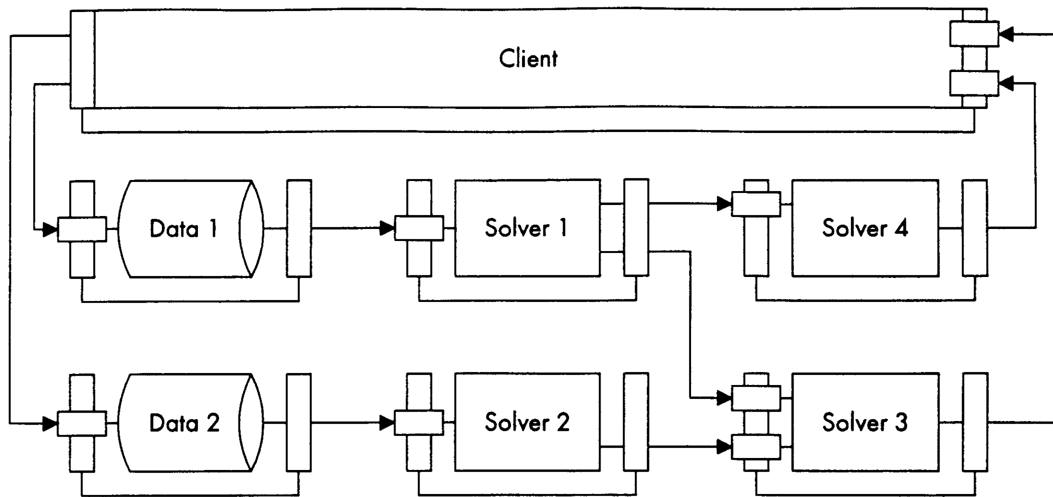


Figure 3.17: Sample solution network with mappings, solver sites, and the client

3.8.3.3 Partitioning responsibilities

The inbound solver site and its mappings and the outbound solver site work together to perform the nine duties described on page 185. These duties can now be reformulated and allocated to these three classes of objects, as follows.

The inbound solver site is responsible for:

1. Managing the collection of mappings for its solver's inputs.

2. Executing the solver.

3. Managing the creation and destruction of the solver.

The outbound solver is responsible for:

1. Providing a locking mechanism to ensure data integrity.

2. Exposing the outbound side of the solver to any customers.

3. Resetting the solver.

Each mapping is now responsible for the other activities, which are specifically concerned with a particular supplier-solver connection:

1. Being aware of the supplier on the connection.

2. Establishing an advise connection to receive notifications from the supplier when it has data available for the solver.

3. Managing the flow of data from the supplier to the solver on the connection.

4. Assigning the input data element managed by the mapping to the solver.

The following sections discuss in detail how the inbound solver site, the outbound solver site, and the mappings undertake their responsibilities.

3.8.4 Inbound solver sites

As discussed in the previous section, an inbound solver site, sitting between a solver and its suppliers, has three primary responsibilities: managing the collection of mappings, executing the solver, and optionally supporting the creation and destruction of the solver itself. As always, these activities are specified by interfaces, which include IRSolverSiteIn, IRSolverSiteInMappings, IRSolverSiteInSolverFactory, and IRSolverInputs. An object that supports at least IRSolverSiteIn and IRSolverInputs can function as an inbound solver site. These interfaces will be discussed in the context of the inbound solver site's responsibilities.

3.8.4.1 Inbound solver site basics: IRSolverSiteIn and IRSolverInputs

Two interfaces an inbound solver site must support are IRSolverSiteIn and IRSolverInputs. IRSolverSiteIn provides basic inbound site attributes. Most notably, it has two pairs of get/set methods for accessing and assigning the solver and the outbound solver site paired to the inbound solver site. These methods are, consequently, **GetSolver, SetSolver, GetSolverSiteOut,** and **SetSolverSiteOut.** While "wiring up" the network, the client calls SetSolverSiteOut on the inbound solver site and IRSolverSiteOut::SetSolverSiteIn on the outbound solver site to match them together[13]. IRSolverSiteIn defines an additional method, **IsSolverInputAvailable,** that takes as input an index from zero to one less than that returned by the SolverInfo's IRSolverInfo::GetInputInfoCount and returns a non-zero value if the input data element corresponding to that index is available in the solver or in the mapping for that input. If IsSolverInputAvailable is non-zero for every input, then the solver is ready to execute.

Inbound solver sites must also expose their own implementation of IRSolverInputs, the input interface for solvers (see section 3.5.3, page 140). For the most part, this implementation will delegate directly to the solver. This is necessary because the inbound solver site wraps the input side of the solver. A mapping will call IRSolverInputs::SetInputData on the inbound

---

[13] This induces a circular reference, so the client must be sure to clear the references by calling IRSolverSiteIn::SetSolverSiteOut and IRSolverSiteOut::SetSolverSiteIn with NULL pointers when destroying the network.

solver site in order to map its data into the solver. Exposing **IRSolverInputs** is also a feature because doing so (along with the outbound solver site's support of **IRSolverOutputs**, see page 193) enables the creation of solver sites that wrap custom solvers that do not conform to the framework solver specification. Furthermore, as the interfaces for solvers evolve, new solver sites can be created to ensure that old solution networks still work. Likewise, new solver sites can be developed to ensure that new solution networks that use new solver interfaces will still work with old solvers that use the old interfaces.

3.8.4.2 Managing the mappings: IRSolverSiteInMappings

The inbound solver site is a container for the mappings. For each input to the solver, there must be a mapping. The interface **IRSolverSiteInMappings** specifies the container activities that the inbound solver site must support. These activities include:

AddMapping. Given an existing mapping object, adds it to the collection and returns a magic cookie identifying the mapping within the container.

RemovingMapping. Given a cookie returned from AddMapping, removes the mapping from the container.

GetMappingCount. Returns the number of mappings currently in the container.

GetMapping. Given a cookie, returns the mapping corresponding to that cookie.

GetMappingByIndex. Given an index from zero to one less than that returned by GetMappingCount, returns the mapping corresponding to that index.

The mappings themselves maintain their own state data, such as the supplier and the particular solver input for that mapping. The inbound solver site will need to verify during processing that the correct number of mappings have been added to the container, and that all non-optional solver inputs have been mapped.

This flexibility in adding and removing mappings might seem like over-specification, especially under the assumption that there is one mapping per solver input. However, this specification permits enhancements such as a single mapping mapping into multiple solver inputs. The specification also enables different mapping implementations to coexist within the inbound solver site mapping container.

3.8.4.3 Executing the solver

In the local control scenario, it is the inbound solver site that controls the beginning of the solver's execution[14]. This is because the inbound solver site has the best knowledge of when

[14] Once running, the client can still pause or cancel the solver.

all of the suppliers' output data are ready. In particular, once the inbound solver site is running and all of the mappings have been established (via IRSolverSiteInMappings::AddMapping), then the inbound solver site waits for each mapping to notify it that its data element has arrived. Mappings establish data advise notifications with their suppliers (see section 3.8.6, page 199, below), and when a mapping receives the OnDataChange notification, it notifies the inbound solver site that it is ready with data (this processed is described below).

Once all of the mappings have notified the inbound solver site that they are ready with data, the inbound solver site instructs each mapping to map its data into the solver. How a mapping does this is discussed below, in section 3.8.6.2, page 200. After all of the inputs have been mapped, the inbound solver site begins the solver execution.

A mapping notifies the inbound solver site through the IRMappingAdvise interface. This interface has a single method, **OnMappingDataAvailable,** that takes as input the magic cookie that identifies the mapping sending the notification. The inbound solver site does not directly support this interface, but instead it creates a sub-object that does[15].

3.8.4.4 Creating and destroying the solver: IRSolverSiteInSolverFactory

Because the solver site is a container for the solver, and nearly completely wraps the solver, it is not necessary for the client to create the solver object itself in order to specify the solution network. If the solver object requires extensive resources, it is beneficial to delay the instantiation of the solver as long as possible. If the client does not require any interaction with the solver—if the solver has no parameters or the client is satisfied with the default parameter values—then the creation of the solver can be delayed until all of the input data are ready to be mapped and execution is ready to begin[16].

The inbound solver site specification enables this by allowing the client to specify the class of the solver object without creating it. In order to establish and verify the mappings, the inbound solver site requires only the SolverInfo of the solver, which is usually accessible from the CLSID of the solver (see sections 3.6.1, page 142, and 3.6.1.5, page 161).

---

15 This is a common pattern in COM when two objects need to reference each other without causing circular reference counts. The inbound solver site maintains a reference count on the mapping, but also needs to receive updates from the mapping. If the mapping held a reference count on the inbound solver site, there would be a circular reference. Instead, the mapping maintains a reference count on the inbound solver site's notification sub-object. This way, there is no circular reference count and the objects can be safely released.

16 An extension to the framework would be to cache parameters at the solver site, which would then set the parameters in the solver once it creates it. In this manner, solver creation could be delayed as late as possible.

An inbound solver site indicates support of this feature by implementing and exporting the IRSolverSiteInSolverFactory interface, which has four methods:

SpecifySolver. Provides the inbound solver site the CLSID of the solver, its class context flags, and flags for solver creation and destruction. The class context flags are from the CLSCTX enumeration (see the COM API function CoCreateInstance for details), while the other set of flags might include one value from each of Table 3.21 and Table 3.22.

Creation flag	Meaning
RSFC_NORMAL	The solver will be created in the normal fashion; by the client creating it independently and passing it to the inbound solver site via IRSolverSiteIn::SetSolver. This is the default.
RSFC_IMMEDIATE	The inbound solver site should create the solver during the call to SpecifySolver.
RSFC_LAZY	The inbound solver site should create the solver at its convenience.
RSFC_LASTPOSSIBLE	The inbound solver site should create the solver at the last possible moment. This is at the first call to IRSolverSiteIn::GetSolver or once all of the mappings have notified the solver site that their data is ready, whichever comes first.

Table 3.21: Creation flags for IRSolverSiteInSolverFactory::SpecifySolver

Destruction flag	Meaning
RSFD_NORMAL	The solver will be destroyed in the normal fashion. This is the default.
RSFD_ONSOLVECOMPLETE	The inbound solver site should destroy the solver once the solver has finished executing and all customers are finished acquiring the solver's outputs.
RSFD_ONSITEDESTROY	The inbound solver site should destroy the solver when it is itself destroyed.

Table 3.22: Destruction flags for IRSolverSiteInSolverFactory::SpecifySolver

Note that as always with COM, destruction of an object is usually at the object's discretion. That is, all of the clients can release their references to an object, indicating that the object is no longer needed, but rarely do the clients actually control the destruction of the object, the restoration of its resources, and its removal from memory.

SpecifySolverProgID. Similar to SpecifySolver, but takes a ProgID instead of a CLSID to identify the solver. A ProgID is a string that maps to a CLSID through the COM library function CLSIDFromProgID. This function might just call CLSIDFromProgID and then delegate to SpecifySolver.

CreateSolver. Instructs the inbound solver site to create the solver immediately. This method gives the client manual control over the creation of the solver. In most cases, the client will either create the solver directly and pass it to the inbound solver site using IRSolverSiteIn::SetSolver or else allow the inbound solver site to create it according to the creation flags in Table 3.21 rather than call CreateSolver.

DestroySolver. Instructs the inbound solver site to effectively destroy the solver immediately. (Note that in COM this means releasing any references to the solver.) This method gives the client manual control over the "destruction" of the solver. In most cases, the client will allow the solver to be destroyed naturally when the solution process is complete or allow the inbound solver site to destroy it according to the destruction flags in Table 3.22 rather than call DestroySolver.

3.8.5 Outbound solver sites

The outbound solver site has three tasks. When customers need the outputs of a solver, the outbound solver site must make sure those outputs remain valid; this is the data integrity problem, and it is solved through a locking mechanism on the solver. The outbound solver site also exposes the outputs to the customers. When all customers are finished with the outputs of the solver, the outbound solver site can reset the solver and release its locks, allowing another invocation of the solver if necessary.

An outbound solver site supports the interfaces IRSolverSiteOut and IRSolverOutputs. The tasks are discussed in detail below.

3.8.5.1 Outbound solver site basics: IRSolverSiteOut

Interface IRSolverSiteOut is the outbound solver site's version of IRSolverSiteIn for inbound solver sites. As with IRSolverSiteIn, it defines two pairs of accessor and assignment methods for the solver and the inbound solver site paired with this outbound solver site. These methods are, naturally, **GetSolver, SetSolver, GetSolverSiteIn,** and **SetSolverSiteIn.**

3.8.5.2 Locking the solver

The outbound solver site has responsibility for ensuring data integrity of the solver's outputs. This is managed by a simple locking mechanism on the solver and the inbound solver site. The locking mechanism is engineered so that the client can assume the following about the validity of outputs from a solver:

- When the client receives the OnSolveComplete notification (see section 3.7.1.2, page 173), the outputs are valid.

- If the client calls IRSolverOutputs::LockOutputs while the outputs are valid, the outputs will remain valid until at least the point at which the client calls IRSolverOutputs::UnlockOutputs.

Note that while the outputs might be valid outside of these bounds, the client cannot assume this.

The locking mechanism is pictured in Figure 3.18. The essence of the diagram is that the outbound solver site receives the OnSolveComplete notification from the solver and propagates this on to the clients in the form of OnDataChange notifications (indicating that data elements the clients are interested in have changed). The OnDataChange notifications are bracketed by calls to the solver's LockOutputs and UnlockOutputs methods. Hence, the outputs will be valid during the OnDataChange notification. If the client wants to do something with the output data, it calls GetOutputData and then LockData to lock the data for future use. The data source, implemented by the outbound solver site as a wrapper for the solver's output data source, internally notifies the outbound solver site of the LockData

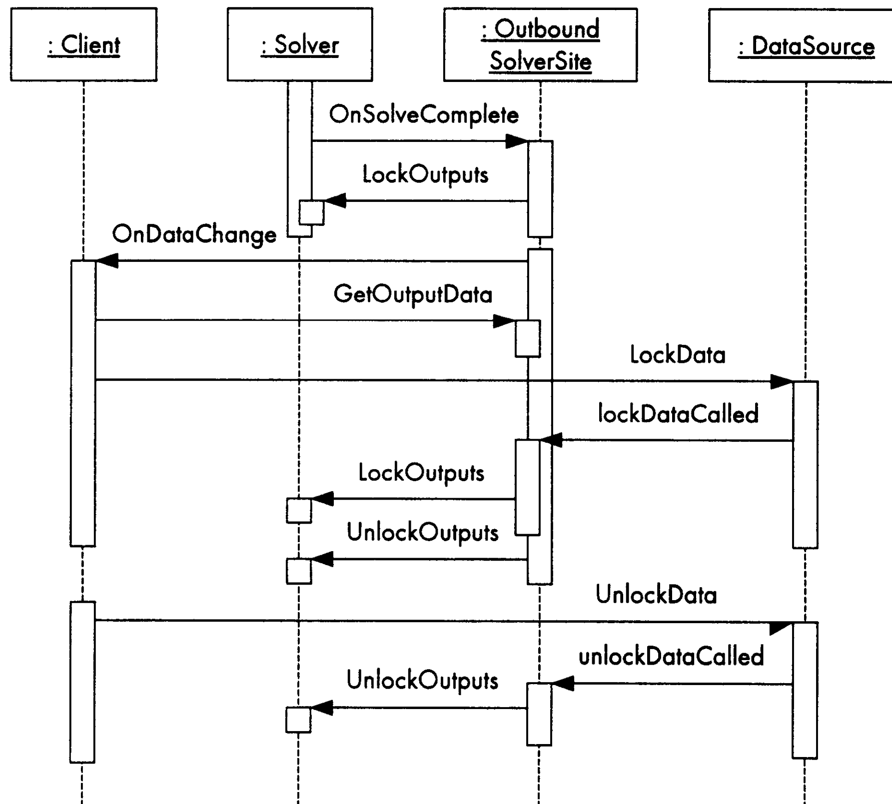


Figure 3.18: Interaction diagram of outbound solver site locking mechanism

call, so the outbound solver site locks the solver's outputs again. When the client is finished with the data, it calls the data source's UnlockData method, which eventually ends up as an UnlockOutputs call to the solver.

Not shown in this diagram is that the outbound solver site also calls the LockInputs method on the inbound solver site's implementation of IRSolverInputs. This instructs the inbound solver site not to allow any mappings to map data into the solver. When the outputs are finally unlocked, the outbound solver site calls UnlockInputs on the inbound solver site.

3.8.5.3 Exposing the outputs

Just as the inbound solver site exposes IRSolverInputs, often just delegating to the solver's implementation (see section 3.8.4.1, page 189), the outbound solver site completes the wrap of the solver by exposing IRSolverOutputs (see section 3.5.4, page 140). Customers retrieve the solver's outputs by calls to the outbound solver site's implementation of IRSolverOutputs. The outbound solver site can also usually just delegate to the solver's implementation of IRSolverOutputs.

3.8.5.4 Resetting the solver

When the final client has unlocked the solver's outputs, as specified above, the outbound solver site has the responsibility of resetting the solver if it will be used in another execution. It does this by calling IRSolver::ClearOutputs and IRSolver::ClearInputs. This happens after unlocking the outputs for the final time but before unlocking the inputs via the inbound solver site.

The solver uses ClearOutputs and ClearInputs to flush any internal data structures and to reset default parameters and inputs. During the final run of a network, the outbound solver site does not need to call ClearInputs and ClearOutputs, so upon destruction, a solver would clear any resources as necessary before unloading.

Another option for a solution network, of course, is to create a new instance of the solver during every iteration of the network. In this way, the solver will always be initialized to default values prior to setting new inputs.

3.8.5.5 Using outbound solver sites to make decisions

A potentially powerful aspect of wrapping the outbound side of the solver is that the outbound solver site can filter and direct the solver's completion notifications. That is, the outbound solver site is the primary recipient of the solver's completion notification, and it is responsible for notifying all of the mappings that require outputs from this solver that data they need are available. However, there is no restriction that all outbound solver sites must

notify all interested mappings that data are available. Instead, the outbound solver site could selectively filter mappings to receive data availability notifications.

This has the effect of inducing flow of control on a network. Because solvers do not execute until inbound solver sites tell them to, and because the inbound solver sites wait for all of their mappings to be ready, and because the mappings wait on the outbound solver sites for availability notifications, the outbound solver site can effectively manage the flow of control in a network based upon some criteria that are perhaps dependent on the solver's outputs.

An example should help to illustrate this power. First, imagine a solver that must process ten sets of input data, queried from a database using a simple SQL query parameterized by the iteration number, something like SELECT DEMAND.* FROM DEMAND WHERE ((DEMAND.SCENARIO)=[s]);[17] where [s] goes from 1 to 10. The solver might receive its input data element from a query engine solver and send its output to another query engine solver, so in each iteration the three solvers form a simple, directed acyclic (serial) graph architecture. The client could execute this directed-acyclic graph network ten separate times, changing the parameter [s] with each execution. This is illustrated in Figure 3.19 by the large cyclic flow between the solution network and the client.

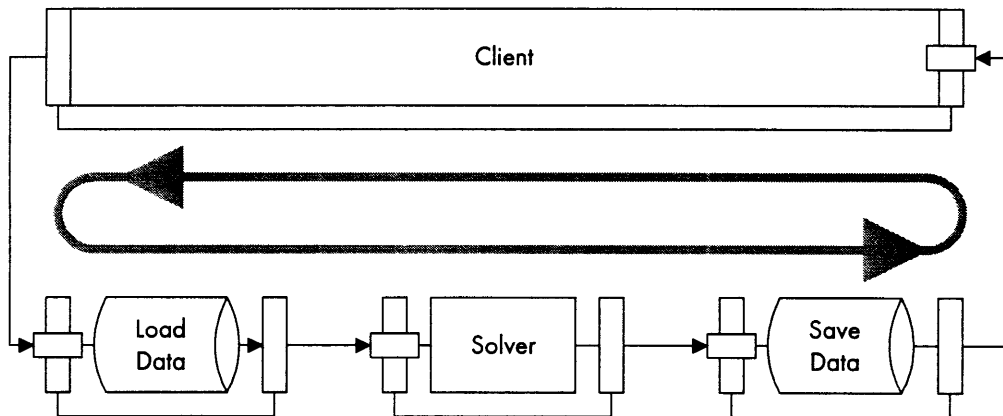


Figure 3.19: Iterating a solution network from the client

Another option would be to add intelligence to the outbound solver of the final solver (the "Save Data" solver in this case) so that during the first nine iteration is increments the parameter counter and notifies the "Load Data" mapping again, and during the tenth iteration it notifies the client. This is illustrated in Figure 3.20 by the cyclic flow within the solution network; this cycle is not perceived directly by the client. The special outbound solver site is shaded in this figure.

[17] This SQL statement was generated with Microsoft Access 95.

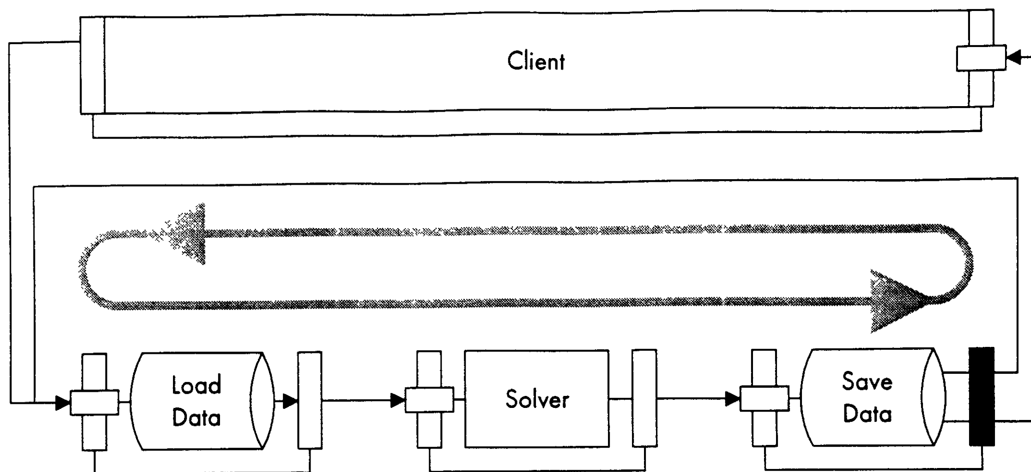


Figure 3.20: Adding intelligence to the outbound solver site to iterate within the network

In this example, there is no real input to the "Load Data" solver, as the iteration number is a parameter to the query engine rather than being a data element *per se.* So in the diagram where there are two links feeding into one mapping, it is understood that the mapping does not technically map data, and therefore has no real supplier. It is only necessary to make sure that first the client and then later the outbound solver site of the final solver can send the OnDataChange notification to the first solver's mapping.

3.8.5.6 Implementation: Outbound solver site state diagram

Figure 3.21 presents a possible state diagram for an outbound solver site. The object has a single global state, which represents the life of the object. Upon creation, it enters that global state, and upon destruction leaves it. The global state contains four concurrent sections. The first indicates whether the outbound solver site has been hooked to its inbound pair. The second concurrent section shows the OnSolveComplete notification and that the outbound solver site propagates this notification to its advise list. The third section shows the locking mechanism for the outbound solver site, and the fourth section shows the advise list for the site.

The advise list for the outbound solver site contains the mappings and other clients that hook to the solver's outgoing data source which is wrapped by the outbound solver site. When the outbound solver site sends the OnSolveComplete notification, it is sending it to these mappings and clients; this might trigger them to map data.

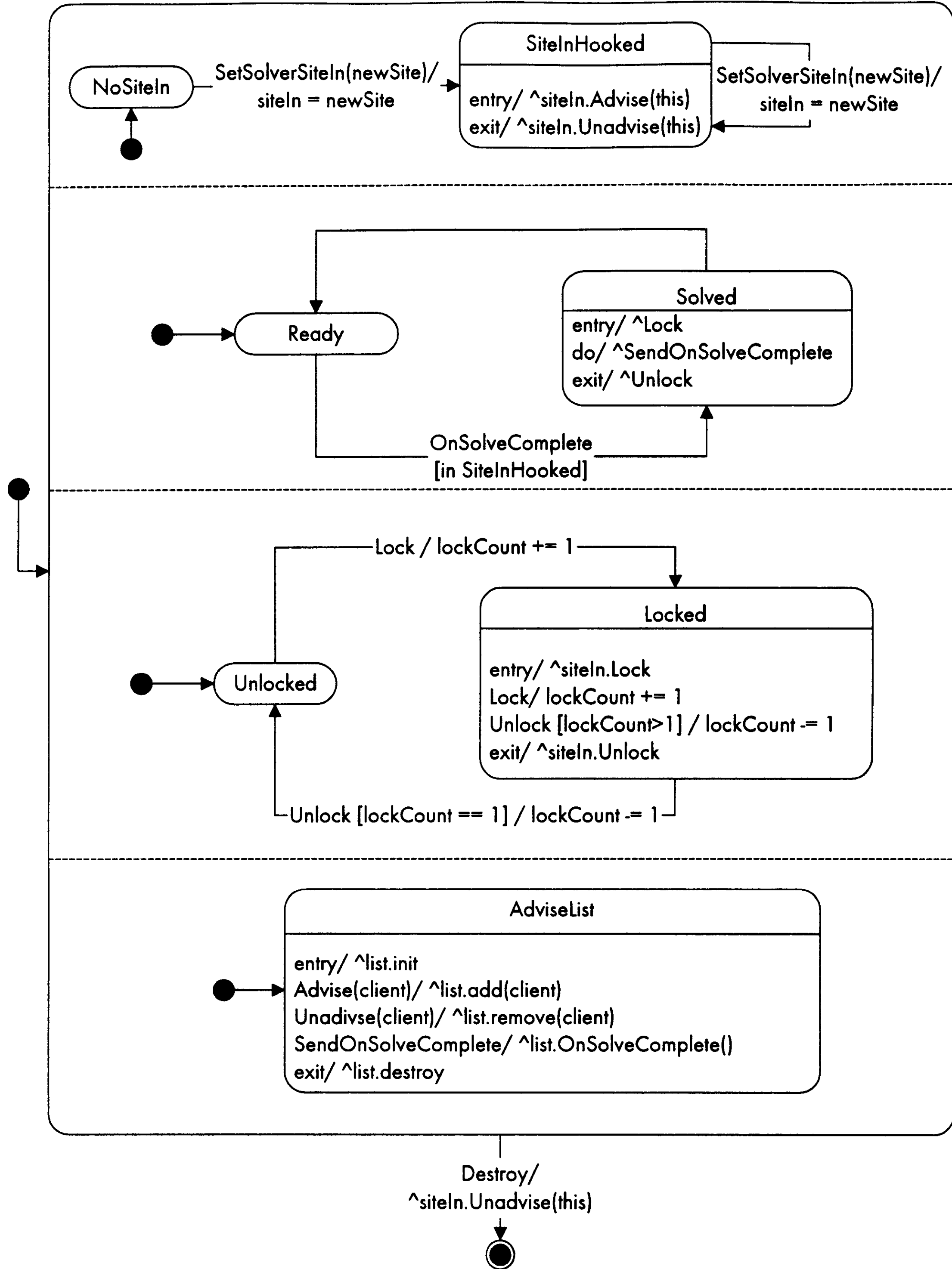


Figure 3.21: State diagram for outbound solver site object

3.8.6 Mappings

The bulk of the intelligence and capabilities in a solution network resides in the mappings. These components represent the links of data flows between solver components wrapped by solver sites. The mapping contains the knowledge of how to take an output or other data from one solver and turn it into an input for another solver.

For the most part, the framework specification of the primary solver interfaces, especially IRSolverInputs and IRSolverOutputs, makes the mapping's job significantly easier, as a single mapping component that works with those two interfaces is sufficient for most purposes in networks of solvers that support those interfaces. However, in cases where a solver has a custom interface for setting inputs or retrieving outputs, either a special mapping component can map into that solver, or a custom solver site can wrap the solver so that it can plug into a solution network.

A mapping has four primary responsibilities. First, it must know its supplier. That is, the mapping maintains knowledge of the source, or tail, of the link it represents. Second, it must receive notifications from its supplier when its particular data element is available (or changes). Upon receiving this notification, it must notify the inbound solver site, which will later instruct it to map its data element into the solver. Third, the mapping must know how to translate its supplier's data element into a form the solver requires. As mentioned above, most often the data element will be directly useable by the solver, so that this task is trivial. More complicated or special mappings might be able to perform pre-processing tasks on the data, as discussed below in section 3.8.6.4, page 203. Finally, the mapping must pass its data element to the solver. These final two tasks are sometimes referred to as "mapping the data."

The primary mapping functionality is specified and accessed by the interface IRSolverMapping, which all mapping components must implement. In particular, IRSolverMapping covers the first two tasks of the mapping. Because of the potential variety of mapping techniques for the third and fourth tasks, different mapping components might expose their settings in different ways. The standard interface for specifying precisely how mappings map their data is IRSolverMappingMechanism. This simple interface allows a client to specify the particular destination of the mapping's data element into the solver, based on the numbering of the inputs in the solver's SolverInfo. Mappings can be remarkably versatile, as they have complete control when mapping their data. Some possibilities will be discussed at the end of this section.

3.8.6.1 Mapping basics: Interface IRSolverMapping

Just like inbound and outbound solver sites, mappings have a standard interface that handles the basic chores. In this case, it is the interface IRSolverMapping. While somewhat large, the interface breaks down into five groups of methods.

First are three locking mechanism methods, **LockMapping, UnlockMapping,** and **MappingLocked.** These are like the similar methods in IRDataSource. Here, the inbound solver site uses LockMapping and UnlockMapping whenever its inputs are locked or unlocked with calls to IRSolverInputs::LockInputs or IRSolverInputs::UnlockInputs.

Second is an accessor and modifier pair for the inbound solver site that contains the mapping. The inbound solver site calls **SetSolverSiteIn** in its implementation of IRSolverSiteInMappings::AddMapping, while **GetSolverSiteIn** returns the inbound solver site.

Third is another accessor and modifier pair, this time for the data source that is the mapping's supplier. The client calls **SetDataSource** when creating the network to assign a data source as a supplier for this mapping. **GetDataSource** returns this data source, once assigned. In more complicated mappings, where the mapping might manage more than one data source, these functions might return the primary or first data source, or they might not be implemented.

Fourth is the method that actually maps the data, logically named **MapData.** This method is discussed in detail in the following sections.

The final group is another attribute accessor and modifier pair. The attribute is known as MaintainPreference, and it specifies the desired location of the actual data values during solver execution. The two functions are **SetMaintainPreference,** which sets the preference, and **GetMaintainPreference,** which returns the preference. The MaintainPreference attribute is discussed in the next section.

3.8.6.2 Transferring the data from the supplier to the solver

How does the mapping transfer data from the supplier to the solver? In COM, objects are handled by their interface pointers. A mapping or solver knows its data through the interface pointer to that data, such as IRDataElement*. Behind the interface pointer, however, is the actual data that the solver needs to execute. This distinction gives rise to three possible answers of how to map the data.

The mapping could pass the interface pointer to the solver, and the solver could use that directly. In this case, the solver uses the data managed by and stored with the supplier. The supplier would need to be forbidden from changing the data element while the solver is executing. This technique is called *data source maintains data.*

The mapping could clone the data, and pass an interface pointer of the clone to the solver. In this case, the solver uses the data managed by and stored with the mapping. Two copies of the data exist, but the supplier could change its data (such as by beginning another iteration of the solution). The solver's behavior remains unchanged. This technique is *mapping maintains data.*

The mapping could pass the interface pointer to the solver, which then immediately copies the data into its internal structures. In this case, the solver manages and stores the data. Two copies of the data exist, in possibly different forms entirely, but the supplier can change its data. In this case, the solver must specifically know it has to copy the data rather than just hold on to a lock on the original data. This scenario is possible only when the solver can take inputs as soon as the data is available from the mapping. That is, the solver can maintain the data only when it is idle and waiting for (unordered) inputs or when the solver can accept inputs asynchronously. This technique is *solver maintains data.*

The differences are summarized in the table here:

Who maintains data	Copies of data	Supplier lock required?	Special solver code required?
Data source	1	Yes	No
Mapping	2	No	No
Solver	2	No	Yes

Table 3.23: Characterization of who maintains data in mappings

The client can indicate a preference for who should maintain the data at each mapping, on a mapping by mapping basis. This preference is contained in the MaintainPreference attribute, accessed and modified by the GetMaintainPreference and SetMaintainPreference methods of the interface IRSolverMapping. The MaintainPreference attribute can take one of the values in Table 3.24.

MaintainPreference value	Meaning
RMP_UNKNOWN	The client has no preference.
RMP_DATASOURCE	The client prefers that the data source maintains the data.
RMP_MAPPING	The client prefers that the mapping clone the data and maintain it.
RMP_SOLVER	The client prefers that the solver copy the data and maintain it, if possible.

Table 3.24: Possible values for the MaintainPreference mapping attribute

The MaintainPreference attribute is just that: a preference. The mapping might not and does not have to use the location specified by the attribute when the time comes to actually store the data. When the mapping receives its data availability notification, it can dynamically determine where to maintain the data using the decision tree in Figure 3.22. The mapping can determine if the data source can maintain data by calling IRDataSource::CanMaintain-

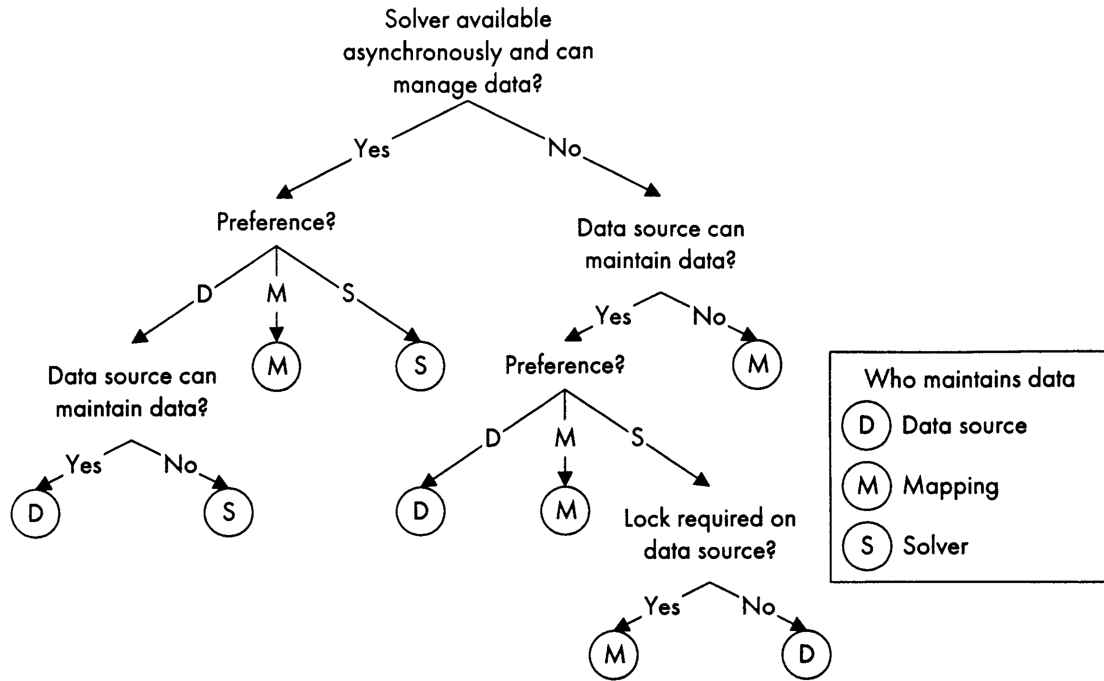


Figure 3.22: DetermineWhoMaintainsData: Decision tree

SolverData, and it can determine if a lock is required to do so by calling IRDataSource::LockRequired. The mapping can determine if the solver can maintain data by checking if the RSAF_CANMAINTAINDATA flag is set in the bitmask returned by the IRSolverInputInfo::GetAssignmentFlags method of the SolverInfo's input object for a particular input.

3.8.6.3 The basic mapping mechanism: IRSolverMappingMechanism

The mapping mechanism is the means by which a mapping takes its supplier's data element and passes it to the solver. The simplest mapping mechanism takes the data element from the supplier and passes it to one of the solver's inputs. Based upon the decision of where to maintain the data, the mapping might clone the data element before passing it to the solver.

To support this simple mechanism, the mapping might implement the IRSolverMappingMechanism interface. This interface allows the client to get or retrieve the SolverInputNum attribute through the **GetSolverInputNum** and **SetSolverInputNum** methods. The SolverInputNum is an index from zero to one less than the number of inputs in the solver (from IRSolverInputs::GetInputCount) that identifies the input to which the mapping will pass its data. In its implementation of IRSolverMapping::MapData, the mapping would call the inbound solver site's implementation of IRSolverInputs::SetInputData using the SolverInputNum as the destination input for the mapping.

3.8.6.4 Extending mapping mechanisms: The versatility of mappings

Mappings provide an opportunity for preprocessing of inputs. In essence, mappings can be miniature solvers themselves, manipulating data while in transit from a supplier to a customer.

An example is dimension aggregation. An aggregating mapping might take an *n*-dimensional data element and perform some simple aggregation over *k* of the dimensions to yield an *(n–k)*-dimensional data element. Summation is the most common example. Suppose a data element captures sales per day per employee. A solver might need sales per employee. The aggregating mapping might be designed to calculate the total sales or average sales per employee for the entire year. Rather than dedicating a special addition solver to the task, a special mapping could be used.

To support this custom functionality, the mapping needs to define and implement a custom interface, and the client needs to be aware of this interface. For these situations, a dispinterface or dual interface is often the best choice.

Another example of a special mapping is one that instead of applying fixed logic or mapping a data element directly to the solver allows the client to provide custom code to handle the mapping. Namely, the implementation of MapData in the mapping calls a client subroutine instead of providing its own functionality. One way to handle this is through a callback interface (again, custom-designed for the situation). Another, elegant technique is to engineer scripting into the mapping component. The client would provide a script to the mapping that the mapping executes in MapData. This script could be JavaScript, VBScript, or something similar. This technique is quite simple to implement in a COM environment given Microsoft's Active Scripting technology. For more information, see Microsoft's scripting technology web site [70].

The callback or scripting technique gives the client an amazing latitude for handling the mapping. The client could walk through the data element and only use the values of interest, or could set parameters on the solver based upon values in the data element, or manipulate other objects outside the scope of the solution network, or even display messages or windows to the user. In general, the mapping specification should be general enough to provide extensive freedom in mapping data into a solver's inputs. The primary assumptions about a mapping are that it relies on specific suppliers for inputs and that it initializes the input of a solver; how it does this is up to the mapping implementation.

3.8.7 Putting it together

With the pieces in hand, it is time to put them together and build the network. The solution network life cycle has three primary phases. First is the creation and wiring of the various components into the solution network. Second is the execution of the network. Tearing down and destroying the network is the final phase. The following sections describe these three phases in the context of the sample solution network.

3.8.7.1 Creating and wiring the network

The first step is to create the four solver sites and hook the inbound and outbound pairs together. An example, in C++, is shown here:

```
const nSolvers = 4;
IRSolverSiteIn* rgInboundSites[nSolvers];
IRSolverSiteOut* rgOutboundSites[nSolvers];

for(int i=0;i<nSolvers;++i)
{
    // Create Solver i's inbound and outbound solver sites
    CoCreateInstance(CLSID_RSolverSiteIn, NULL, CLSCTX_SERVER,
        IID_IRSolverSiteIn, (LPVOID*) (rgInboundSites+i));
    CoCreateInstance(CLSID_RSolverSiteOut, NULL, CLSCTX_SERVER,
        IID_IRSolverSiteOut, (LPVOID*) (rgOutboundSites+i));

    // Now wire them together
    rgInboundSites[i]->SetSolverSiteOut(rgOutboundSites[i]);
    rgOutboundSites[i]->SetSolverSiteIn(rgInboundSites[i]);
}
```

Wiring together the solver sites of Solver 3 is presented as the first two steps in the object interaction diagram of Figure 3.23. This wiring happens for all four pairs of solver sites.

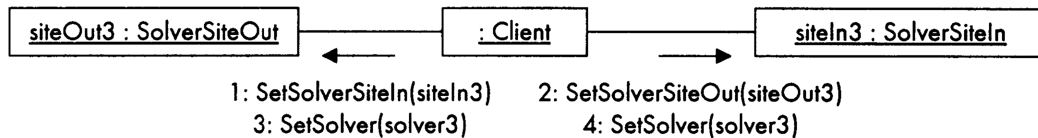


Figure 3.23: Wiring Solver 3's solver sites

Next, the client either instantiates the solvers and hooks them into the solver sites or specifies their creation attributes using IRSolverSiteInSolverFactory, if that is supported by the inbound solver site. Hooking Solver 3 to its solver sites are steps 3 and 4 in Figure 3.23. Example code, using fake CLSIDs for the solvers, is shown here:

```
IRSolver* rgSolvers[nSolvers];

// CLSIDs for Solvers 1-4 (these are sample CLSIDs only)
CLSID rgClsid[nSolvers] = {
    {0x33BC0F00,0x700E,0x11d1,{0x90,0xF7,0x00,0x20,0x78,0x10,0xC7,0x41}},
    {0x33BC0F01,0x700E,0x11d1,{0x90,0xF7,0x00,0x20,0x78,0x10,0xC7,0x41}},
    {0x33BC0F02,0x700E,0x11d1,{0x90,0xF7,0x00,0x20,0x78,0x10,0xC7,0x41}},
```

```
    {0x33BC0F03,0x700E,0x11d1,{0x90,0xF7,0x00,0x20,0x78,0x10,0xC7,0x41}}
    };

for(i=0;i<nSolvers;++i)
{
    // Create Solver i
    CoCreateInstance(rgClsid[i], NULL, CLSCTX_SERVER, IID_IRSolver,
        (LPVOID*)(rgSolvers+i));

    // Hook it up
    rgInboundSites[i]->SetSolver(rgSolvers[i]);
    rgOutboundSites[i]->SetSolver(rgSolvers[i]);
}
```

The next step is to create the mappings and place them into their respective inbound solver site containers. This is shown for Solver 3, again, in steps 1 to 4 of Figure 3.24, with example code below for the sample network:

```
// Mapping interface pointer array
const int nMappings = 5;
IRSolverMapping* rgMappings[nMappings];

// The mapping cookies for later removal
DWORD rgCookies[nMappings];
```

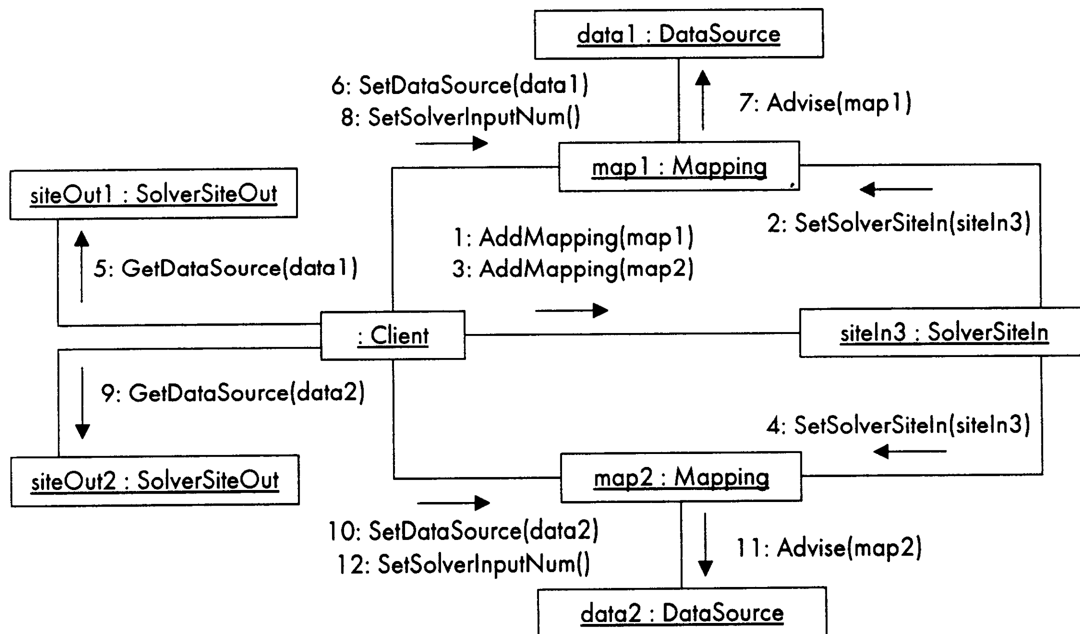


Figure 3.24: Interaction diagram showing wiring of Solver 3's site

```
// rgSitesForMappings[x] = y means that mapping x goes to inbound solver site y
int rgSitesForMappings[nMappings] = { 0, 1, 2, 2, 3 };

for(i=0;i< nMappings;++i)
{
    // Create Mapping i
    CoCreateInstance(CLSID_RMapping, NULL, CLSCTX_SERVER, IID_IRSolverMapping,
        (LPVOID*)(rgMappings+i));

    // Add mapping i to its specified inbound solver site
    IRSolverSiteInMappings* pMappings = NULL;
    rgInboundSites[rgSitesForMappings[i]]->QueryInterface(IID_IRSolverSiteInMappings,
        (LPVOID*)&pMappings);
    pMappings->AddMapping(rgMappings[i], &rgCookies[i]);
    pMappings->Release();
}
```

Finally, the client links the mappings to their specific solver outputs, data sources, and solver inputs. These are steps 5 through 12 in Figure 3.24 for Solver 3, with example code shown below for the entire four stage network:

```
// Hook in data sources to input side of each mapping
IRDataSource* pSource = NULL;
IRSolverOutputs* pOutputs = NULL;
UINT uElem = 0;

// First two mappings use pre-existing external data sources
rgMappings[0]->SetDataSource(pExternal1, 0);
rgMappings[1]->SetDataSource(pExternal2, 0);

// Third and fifth mappings use output from Solver 1
rgOutboundSites[0]->QueryInterface(IID_IRSolverOutputs, (LPVOID*)&pOutputs);
pOutputs->GetOutputData(0, IID_IRDataSource, (LPVOID*)&pSource, &uElem);
rgMappings[2]->SetDataSource(pSource, uElem);
rgMappings[4]->SetDataSource(pSource, uElem);
pSource->Release();
pOutputs->Release();

// Fourth mapping uses output from Solver 2
rgOutboundSites[1]->QueryInterface(IID_IRSolverOutputs, (LPVOID*)&pOutputs);
pOutputs->GetOutputData(0, IID_IRDataSource, (LPVOID*)&pSource, &uElem);
rgMappings[3]->SetDataSource(pSource, uElem);
pSource->Release();
pOutputs->Release();
```

```
// Now hook mappings to specific solver input ports
int rgInputsForMappings[nMappings] = { 0, 0, 0, 1, 0 };
for(i=0;i<nMappings;++i)
{
    IRSolverMappingMechanism* pMech = NULL;
    rgMappings[i]->QueryInterface(IID_IRSolverMappingMechanism, (LPVOID*)&pMech);
    pMech->SetSolverInputNum(rgInputsForMappings[i]);
    pMech->Release();
}
```

With that, the network is ready to go.

3.8.7.2 Component interactions

Once the network is created and wired together, the client is ready to begin solving the problem. Typically, this happens by triggering the **OnDataChange** notifications at all of the root data sources. These notifications will cause a propagation of solver execution through the network until the client receives **OnDataChange** notifications from all of the final data sources.

This section presents interaction diagrams for a solution network with one solver embedded in a solver site that has a single mapping. The diagrams are intended to show the life cycle of the solution process, from the point of triggering the initial **OnDataChange** until the network is returned to an idle state after the client has retrieved the final outputs. There are three diagrams, one for each of the possible objects that might maintain the input data.

Flow arrows (○——▶) are used to indicate the transferal of a large amount of data, either the input to or the output from the solver.

3.8.7.2.a Data source maintains the input data

In the first scenario, the data source maintains the data, as shown in Figure 3.25.

First, the client sets the input data into the data element (1); this causes the data element to send out its change notifications (2,3). The mapping locks the data source, which locks the data element (4,5). This is so that the data will not change until the solver is finished with it. The mapping then notifies the inbound solver site that it has data ready (6). At the appropriate time—there could be a delay if there were multiple mappings—the inbound solver site instructs the mapping to map its data into the solver (7). To do this, the mapping retrieves the data element from the data source (according to its mapping mechanism) and then passes it to the solver (8,9). After this is complete, the inbound solver site executes the solver (10). During processing, the solver retrieves the input data from the data element (11) to solve the problem. When it is finished, it sends the **OnSolveComplete** notification to the outbound solver site (12), which propagates the notification to the client in the form of an

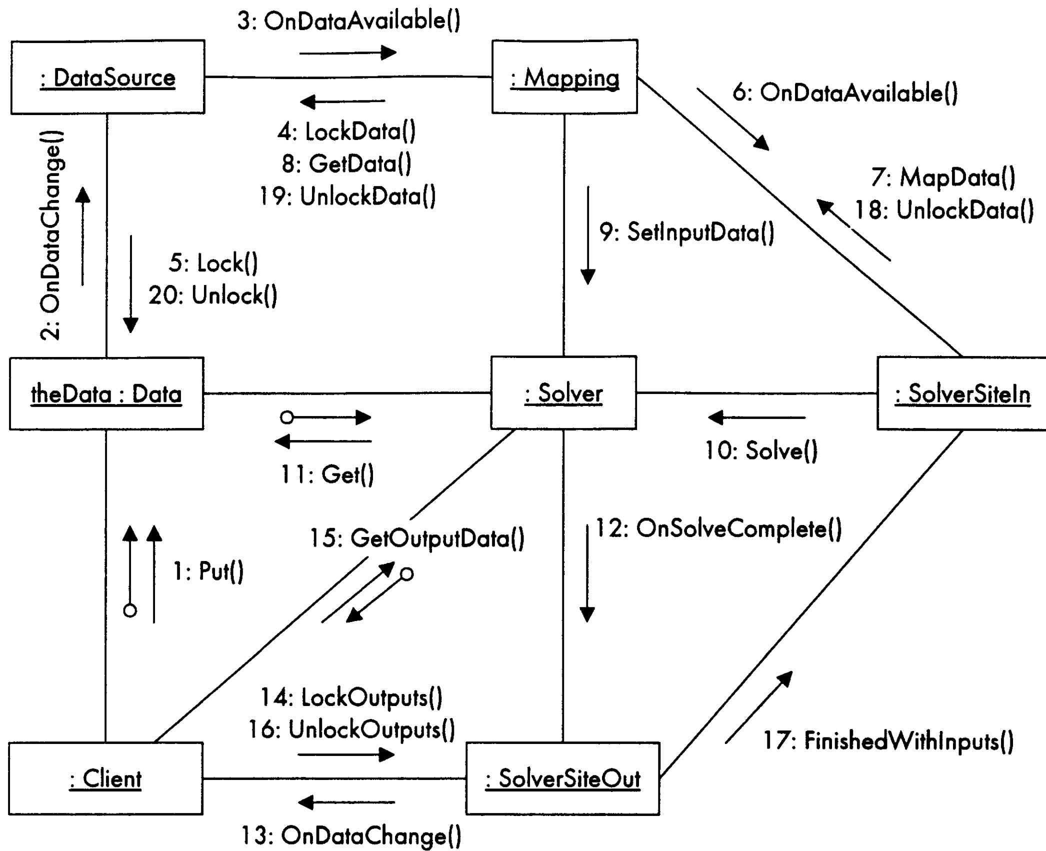


Figure 3.25: Interaction diagram when the data source maintains input data

OnDataChange notification (13). Upon receiving this, the client locks the outputs of the solver, retrieves the outputs, and then unlocks them (14-16). The lock is necessary so that the solver will not be run again before the client has successfully retrieved the outputs. Once the outputs are unlocked, the outbound solver site notifies the inbound solver site that it is finished with the inputs, whose locks are tied to those of the outputs (17). The inbound solver site instructs each of its mappings to unlock their data (18). Each mapping unlocks its supplier's data source, which unlocks the data element, allowing it to be modified or destroyed (19,20). At this point, the client could set new data, beginning the process over again.

3.8.7.2.b Mapping maintains input data

In the second scenario, the mapping maintains the data, as shown in Figure 3.26.

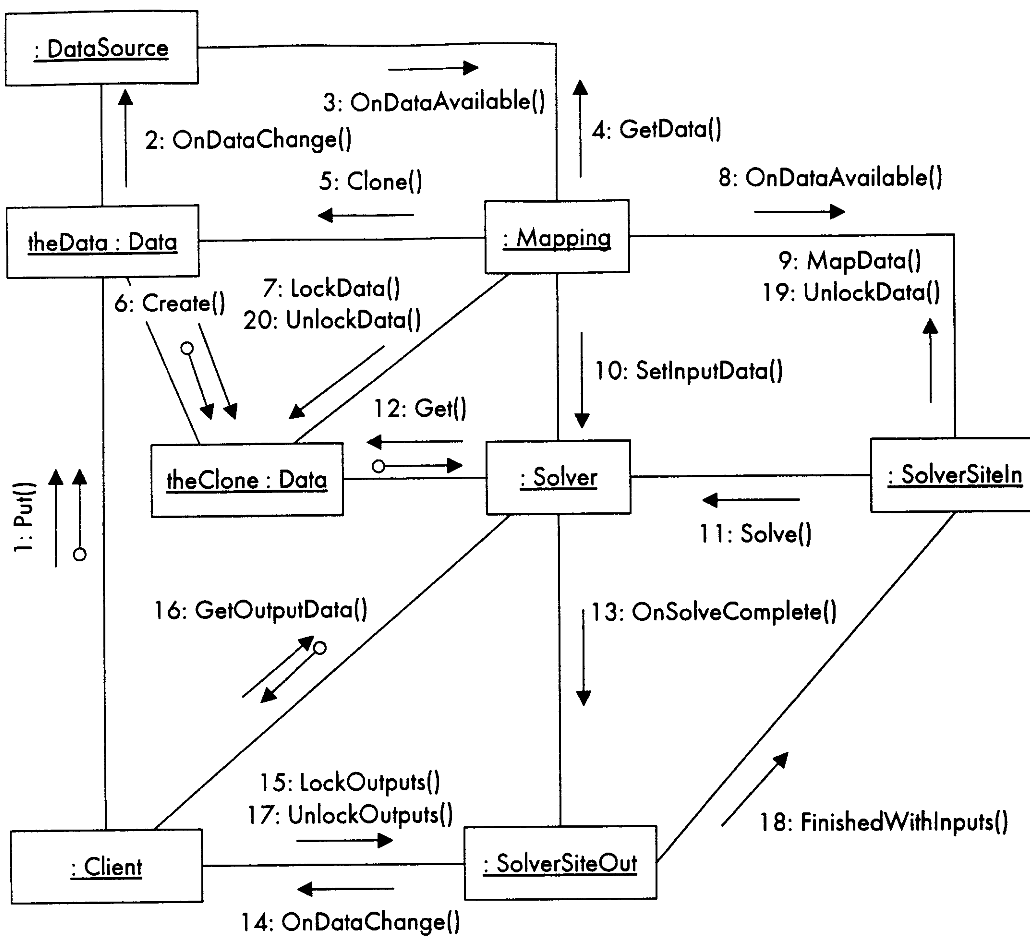


Figure 3.26: Interaction diagram when mapping maintains input data

First, the client sets the input data into the data element (1); this causes the data element to send out its change notifications (2,3). Unlike the previous scenario, where the mapping locked the data element because it would be used by the solver later, the mapping will now create a clone of the data element to use. So, the mapping retrieves the data element from its data source, and tells it to clone itself (4,5,6). The mapping locks the newly created clone (7), and then tells the inbound solver site that it has data ready (8). At the appropriate time—there could be a delay if there were multiple mappings—the inbound solver site instructs the mapping to map its data into the solver (9). To do this, the mapping passes the cloned data element to the solver (10). After this is complete, the inbound solver site executes the solver (11). During processing, the solver retrieves the input data from the data element (12) to solve the problem. When it is finished, it sends the OnSolveComplete notification to the outbound solver site (13), which propagates the notification to the client in the form of an OnDataChange notification (14). Upon receiving this, the client locks the outputs of the solver, retrieves the outputs, and then unlocks them (15-17). Once the outputs are unlocked,

the outbound solver site notifies the inbound solver site that it is finished with the inputs, whose locks are tied to those of the outputs (18). The inbound solver site instructs each of its mappings to unlock their data (19). The mapping unlocks its cloned data element (20), which is subsequently destroyed. At this point, the mapping is ready to receive a new data change notification.

3.8.7.2.c Solver maintains input data

In the third scenario, the solver maintains the data, as shown in Figure 3.27.

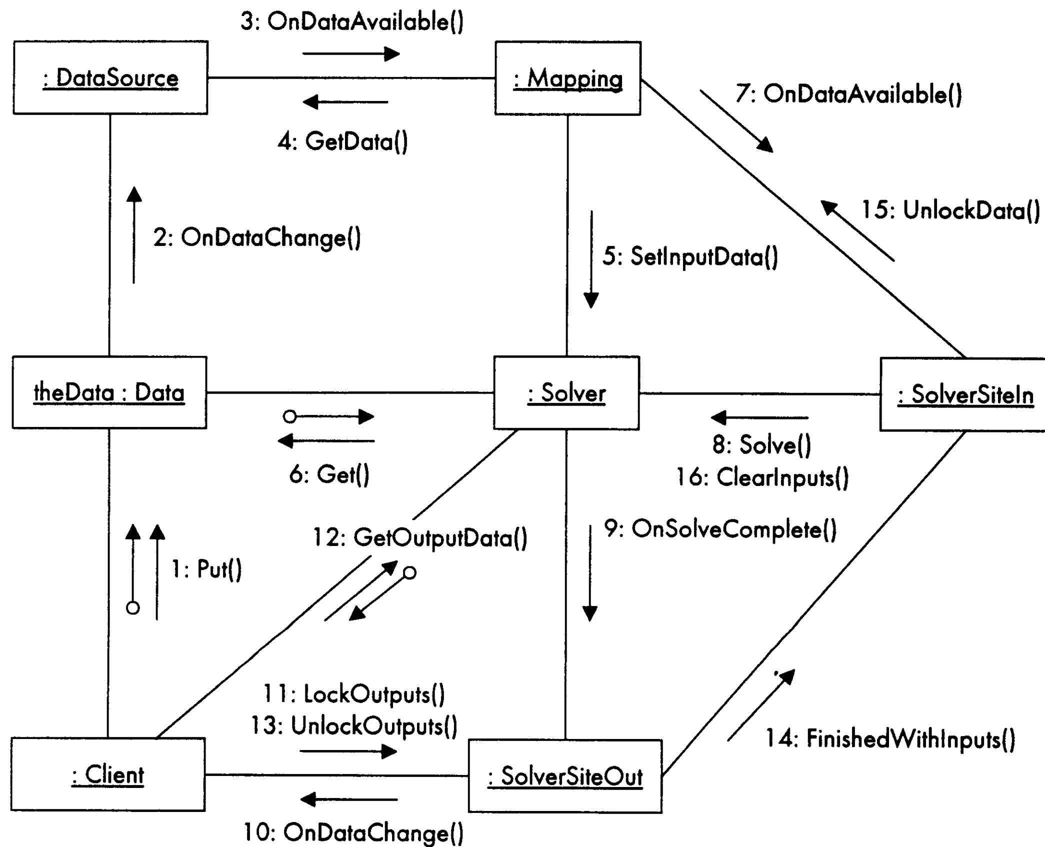


Figure 3.27: Interaction diagram when solver maintains input data

First, the client sets the input data into the data element (1); this causes the data element to send out its change notifications (2,3). Unlike the previous scenarios, the mapping will now map the data directly to the solver, as the solver needs to copy it internally. This is the only case in which the mapping calls MapData on itself without waiting for the inbound solver site. The mapping retrieves the input data element from the data source and passes it to the

solver (4,5). The solver retrieves the actual input from the data element (6) and does whatever is necessary internally to store the data. The mapping then tells the inbound solver site that it has data ready (7). The inbound solver site might instruct the mapping to map its data, but this is not shown because this is an empty operation, as the data was mapped when it was passed to the solver in steps 5 and 6. The inbound solver site executes the solver (8), which already has the complete input data internally. When it is finished, it sends the OnSolveComplete notification to the outbound solver site (9), which propagates the notification to the client in the form of an OnDataChange notification (10). Upon receiving this, the client locks the outputs of the solver, retrieves the outputs, and then unlocks them (11-13). Once the outputs are unlocked, the outbound solver site notifies the inbound solver site that it is finished with the inputs, whose locks are tied to those of the outputs (14). The inbound solver site instructs each of its mappings to unlock their data (15), and then instructs the solver to clear its inputs (16).

3.8.7.3 Destroying the network

Once all desired processing is complete, it is a simple matter to destroy the network. The client simply walks through the list of solvers, inbound and outbound solver sites, and mappings, and releases each of them. With the inbound solver sites, the client should call SetSolverSiteOut(NULL), while with the outbound solver sites, the client should call SetSolverSiteIn(NULL), and for both the client should call SetSolver(NULL), or DestroySolver on the inbound solver site if it supports IRSolverSiteInSolverFactory. Because all of the components in the network are COM objects, they are responsible for terminating their own lifetimes, so it is not necessary to actually delete or free any of the components.

3.8.8 How this fulfills the networking requirements

This section examines how the framework's representation and specification of the solution network attempts to satisfy the networking requirements identified in the second chapter.

3.8.8.1 Satisfying data integrity

Fortunately, by restricting the framework to non-real time activities, both in solution architecture and execution periodicity, the data integrity problem is not as great as it might have been. Nonetheless, there are some issues that the framework has had to resolve.

In a basic directed acyclic graph architecture, the data integrity problem is resolved by ensuring that no solver begins executing before all of its suppliers have finished. This is handled in the framework through the relationship between mappings and inbound solver sites. Before executing the solver, the inbound solver site waits for all of its contained mappings to notify it that their data are ready. The outbound solver sites of suppliers, or data sources themselves, notify mappings that data are ready only at the completion of solver execution. Where data are maintained (data source, mapping, or solver) is irrelevant to data integrity in these networks.

In a decision-based directed graph architecture, or in any architecture with relatively complex components, such as custom aggregating mappings or specialized outbound solver sites, ensuring data integrity is slightly more difficult. In global control scenarios, it will require more intelligence at the client, naturally, to manage data integrity in the presence of cycles. In local control scenarios, the most likely place for adding intelligence to the network is at the mappings. It is the mappings that receive the data change and data availability notifications, and it is the mappings that have control over where and how input data is stored or cloned.

For example, in a network with cyclic data flow, if for each mapping the data source maintains the data, then after the first iteration, every solver in the cycle might be locked, waiting for its customer to unlock it. This results in a deadlock situation, where each solver waits indefinitely. If one of the mappings clones its input data, then the cycle is effectively broken, as the act of cloning a data element enables the mapping to unlock its supplier, and the cycle is unlocked all the way around. This requires an awareness and appreciation of this cyclic complication on the part of the user or client. Either the user or developer must make sure to manually specify how to break the cycle, or the client must have appropriate analysis tools to find and break cycles.

Another example is when one solver iterates many more times, and more quickly, than its customer. The mapping receives data availability notifications from its speedy supplier while its solver is slowly working on the previous input. What should be the mapping's reaction? There are a number of possibilities. The mapping could aggregate the many inputs into a single one, via summation or average, for instance. It could queue each notification for future processing as individual inputs to the solver. Or, it could simply disregard any notifications received while its solver is busy. In this way, the mapping manages the dependencies of the resource flow from supplier to customer when they are not precisely in timing agreement. This begins to hint at the power of the mapping component as a manager of resource flow dependencies. For more information on these dependencies, see Dellarocas [20, 21].

3.8.8.2 Satisfying local/global control

The framework resolves the local/global control problem by supporting and encouraging local control while not hindering global control. Notably, a client can exert complete, global control on a network of solvers that are compliant with the specifications in the previous sections, leveraging introspection, progress notifications, and life cycle control, while completely ignoring any of the network specification. On the other hand, a client can create a network using the inbound and outbound solver site and mapping components and employ local control at each node, relieving it of most of the control responsibilities. The client can even create a network of mixed global and local control, by essentially wrapping the local control sub-networks.

3.8.8.3 Fulfilling the distribution requirement

As with the requirement that solvers be executable applications, the problem of distribution is largely solved by COM. One of the design philosophies of COM is that clients do not know anything about the servers they use except the definition of the interfaces through which they use those servers. Almost universally, the behavior of the server from the perspective of the client is independent of whether the server is in the same thread or process (or apartment in COM terminology) or on another machine. This is accomplished in COM through the use of proxy-stub objects that know how to marshal the parameters of interface methods across the process or machine boundaries; these proxy-stub objects are managed by COM and are transparent to the client. Hence, when a client interacts with an object in its same memory space[18], it calls methods directly on the server implementation. When the client is on a different machine, it calls methods on a proxy object that transfers the call to the other machine where the server resides. The process of transferring the method calls is called *marshaling*.

When developers define custom interfaces that they expect might be marshaled, they will develop the proxy-stub objects and ship those with the implementation of the interfaces themselves. In the framework, these proxy-stub objects are part of the core services, included with any implementation of the framework. Users of the framework do not need to worry about the presence of these proxy-stub objects for the standard framework interfaces.

Nevertheless, there might be instances where areas of the framework are concerned with distribution of components beyond the transparency provided by COM. In particular, many solution networks have the special property of needing to transport large data sets. In traditional programs, when two objects in the same memory space must share data, they usually share the data via a pointer to the data, which requires four bytes[19]. The data exist at a single location in the memory space, and both objects have access to it. In distributed systems, there is no guarantee that two objects exist in the same memory space, and when they do not but must share data, the data have to be copied from one memory space to the other. With the standard proxy-stub objects (which is *standard marshaling),* whenever marshaling occurs, data are copied[20].

Developers can take control of marshaling by using *custom marshaling* on their objects by implementing the IMarshal interface. Objects that implement this interface are responsible for marshaling their own data to other memory spaces or machines. This gives the developer complete freedom on how data are transported.

---

18 With the appropriate assumption that the client is in the same apartment as the server.

19 Four bytes on typical 32-bit operating systems; results might vary.

20 The COM infrastructure does a good job of optimizing marshaling by caching object references at the various client and server computers, so data are not necessarily always copied when an interface pointer is passed across apartment boundaries.

The result is that better data element objects could be designed for particular needs or instances of data flow. Knowing the use, source, and structure of a data element within a solution network, a developer could design a custom data element object that implements all of the data element interfaces as well as IMarshal. Because marshaling is transparent to clients, the data element can plug directly into any standard solution network without modifying other parts of the network.

3.8.8.4 Fulfilling the synchronization requirement

The synchronization requirement was fairly simple (see page 97): the framework should not interfere with the synchronization of a solver, and it should support asynchronous operations when solvers desire it. This requirement is easily solved by the specification of the network protocols. Because all solvers are contained within solver sites, the solver sites have complete control over whether the solver is run synchronously or asynchronously. Furthermore, the solver sites inherently act asynchronously; that is, they wait for notifications of input data availability from suppliers, then they initiate solver execution, and then they notify customers of output data availability. The inbound solver site, which is responsible for executing the solver, can work with either synchronous or asynchronous solvers. With synchronous solvers, the solver site simply calls the synchronous method and waits for its return. With asynchronous solvers, the solver site calls the asynchronous method and waits for the completion notification. Either way, to objects outside the solver site, the solver and its site behave the same. This alleviates any dependencies a client might have had on the synchronization of the solvers. This is another benefit of wrapping solvers within solver sites in the network.

3.9 Core Services

This section describes the functionality provided by the core services of the framework. As described in section 2.5, page 101, the core services provide centralized, global capabilities, such as the system-wide solver database, as well as recurring, common functionality, such as various helper components. Section 2.5 identified the need for both a global solver database and centralized support of dimension and type manipulation. The discussion of the solver database follows, while the discussion of dimension and type manipulation is deferred until section 5.2, page 272, because in its present form the framework does not address dimension and type information.

Also included in the core services, categorized as "miscellany," are the numerous components and functions that have been introduced throughout this chapter that act as supporting routines for the various implementations of solvers and clients. Examples include the various advise holder components and the generic SolverInfo implementation.

3.9.1 Solver database

As identified in section 2.5.1, page 103, there are two primary functional needs of a database of available solvers. One need is registration, so that solvers can alert the operating system when they are installed or removed from a computer. The other is enumeration, so that clients can determine what solvers are available on a machine.

There are many possible levels of registration and enumeration. The solver database could range from a simple, unordered, and unorganized list of solvers to a highly detailed, structural or hierarchical tree of related solvers, models, solvers that can solve models or their derivatives, and solvers that are substitutes for each other.

At its minimum, the framework provides an implementation of the former—a simple listing of the solvers installed on a machine without further thought to what these solvers are or how they might be used. Simple registration and unregistration of a solver is handled by the interface IRSolverRegistration. Enumeration is managed by the interface IRSolverEnumeration. Any component is free to manage a solver database of its own and expose these interfaces. The framework provides a standard implementation, the SolverRegistrar, that uses the Windows Registry and component categories to maintain the database.

3.9.1.1 Interface IRSolverRegistration

Because of the simple nature of the minimal database, registration is correspondingly simple. The method **RegisterSolver** takes as input the CLSID of a solver component and registers the solver in the solver database. The method **RegisterSolverWithInfo** takes as input the CLSID of a solver component and the GUID of that solver's associated SolverInfo object, and it registers the solver and its SolverInfo in the solver database. The method **UnregisterSolver** takes as input the CLSID of a registered solver component and removes it and its SolverInfo, if present, from the solver database.

3.9.1.2 Interface IRSolverEnumeration

Interface IRSolverEnumeration exposes the minimal enumeration functionality of the solver database. It has two methods.

EnumSolvers. Returns a COM enumerator object that enumerates the CLSIDs of all registered solvers. It returns an IEnumCLSID interface pointer. A client can use the CLSID to lookup the names of solvers or to instantiate solvers using CoCreateInstance.

EnumSolversWithInfo. Returns a COM enumerator object that enumerates pairs of CLSIDs and GUIDs of all registered solvers and their SolverInfo objects. This method returns a pointer to the custom interface IEnumRREGSOLVERINFO, which like IEnumCLSID is an enumerator interface. It enumerates structures of type RREGSOLVERINFO, which contains

two members, the CLSID of a solver and the GUID of its SolverInfo. If a solver does not have a registered SolverInfo, the GUID will be GUID_NULL.

3.9.1.3 The SolverRegistrar component

The SolverRegistrar component is an implementation of IRSolverRegistration and IRSolverEnumeration provided by the core services of the framework. The component has CLSID CLSID_RSolverRegistrar, and can be created using CoCreateInstance. To acquire either of the interfaces it implements, the client calls QueryInterface.

The SolverRegistrar works by using a moderately recent innovation in COM, *component categories,* which is a simple category association scheme. The SolverRegistrar defines a new component category CATID (the same as a GUID), CATID_RSolver. When a client calls IRSolverRegistration::RegisterSolver or IRSolverRegistration::RegisterSolverWithInfo, the SolverRegistrar uses the standard component category manager to register the solver as implementing the category CATID_RSolver. Additionally, with IRSolverRegistration::RegisterSolverWithInfo, the SolverRegistrar adds the SolverInfo key as described in section 3.6.1.4.d, page 160. For more information on component categories, see Chappell [15] for an overview and Box [9] for specifics.

3.9.2 Miscellany

Throughout the development of the framework, numerous helper components have been added to the core services. These components serve to aid development of solvers, clients, and data sources. In summary, they are:

AdviseHolder components. The AdviseHolder components are simple helpers that maintain a list of advise connections for an object, and provide broadcast of notifications to all of the advise connections with a single method call. The DataAdviseHolder, in section 3.4.3.3, page 135, helps data sources to notify clients of data changes. The SolverAdviseHolder, in section 3.7.1.1.b, page 170, helps solvers to notify clients of their solution progress and solution completion. These AdviseHolders are modeled after the COM DataAdviseHolder (which implements IDataAdviseHolder, without an "R").

SolverInfo components and functions. The generic SolverInfo implementation is provided by the core services. This implementation uses type libraries to characterize solvers' structures and descriptions, moving the customization of SolverInfo objects from code into source files. The SolverInfo implementation is described in sections 3.6.1.4, page 151, and 3.6.2.3, page 166. The functions that access and create SolverInfo objects are described in section 3.6.1.4.e, page 160.

Dispatch wrapper components. Dispatch wrapper components, mentioned briefly but not yet explained, are simple implementations of the COM dispatching interface, IDispatch, for specific components. Developers can use these wrappers to easily add dispatch support to

their objects with a minimum of overhead. The SolverDispatch wrapper, for instance, provides dispatch support for all of the functionality of a solver. There are also wrappers for inbound and outbound solver sites, mappings, data sources, data elements, and the various advise notification interfaces (in the form of COM connection points).

Dispatch wrappers work by implementing the dispatch functions as simple delegators to the various interfaces supported by an object. For instance, the SolverDispatch supports a property "InputCount" that simply calls IRSolverInputs::GetInputCount and returns the number of inputs.

Running solver table. Section 3.7.2.1, page 175, suggested simplifying the task of acquiring a running solver by implementing a *running solver table* that mimics the COM *running object table.* Actually, the running solver table would be a subset of the running object table. While not in the current core services, such an object could easily be defined. Its interface would support the registration and unregistration of running solvers, and a client could query for an enumeration of the running solvers or acquire an interface pointer of any running solver.

3.10 Conclusion

This chapter has presented a framework for useful, operation research-oriented solvers that can satisfy the requirements detailed in the second chapter. The framework is divided into three broad areas. First, the framework defines protocols, interfaces, and library functions useful for building and interacting with individual solvers. The features for individual solvers include basic solver interaction through the primary solver interfaces, introspection, progress updates, and life cycle control. Second, the framework defines protocols and interfaces for networks of solvers. The network portion of the framework introduces a number of new components, including solver sites and mappings. Finally, the framework specifies behavior of the core services, a centralized library implementation of global routines usable by all solvers and clients. To support solvers, the framework also provides a generic data element specification.

While some of the requirements have not been directly addressed by the framework, including dimension and typing support, notifications, and testing and validation, the framework is extensible and amenable to these activities. The absence of specific interfaces for these features should be viewed as an opportunity for future growth of the framework. These particular requirements are also discussed in the final chapter, in section 5.2, page 272.

Furthermore, because of the interface-based programming discipline imposed by the underlying choice of COM for the framework, any part of the framework could be reworked in the future without substantially affecting other parts. Each part can evolve to resolve new problems or better resolve the old ones, and existing solutions will still work within the framework.

Intentionally, this page left blank.

Chapter Four

Solvers and Applications

This chapter describes a collection of solvers and applications to demonstrate the use, effectiveness, and overhead of the framework. The goal of this chapter and the following chapter is to answer these questions about the framework:

- Does the framework work as described?
- What is the cost of developing a solver that complies with the framework?
- What is the cost of using a solver that complies with the framework in a client application?
- What are the benefits of developing and using a solver that complies with the framework?
- What are the barriers to entry to using the framework? That is, what does a solver or client application developer need to know in addition to the framework to develop for it? Also, what can be done to mitigate those barriers?

The first part of this chapter presents three solver modules that have to varying degrees been implemented both with and without the framework. The second part presents a number of applications that use the three solvers from the first part as well as some additional solvers. Each case demonstrates a different feature, characteristic, or benefit of the framework. The following chapter describes the benefits of and the issues regarding the framework, as well as future research areas.

4.1 SOLVERS

This section presents three solver modules in the framework. The goals are to establish the level of difficulty in creating a framework solver, that is, to determine the overhead imposed by the framework when creating a solver, and to show that this overhead is not an exacting price to pay, especially in light of the benefits that can be gained. The RandVar module contains several solvers that evaluate functions of random variables. The RNetOpt module contains three solvers of network flow optimization algorithms. The RLPWrapper solver wraps the CPLEX Callable Library, exposing the CPLEX linear programming function as a solver within the framework. In the subsequent Applications section, which discusses several applications around the framework, other solvers will be introduced as necessary to solve an application's problems. Before discussing the solver modules, though, it is instructive to examine the different techniques for creating a solver in the framework.

4.1.1 Packaging a solver

There are basically four ways, from an implementation perspective, to create a solver in the framework. These range from hand-coding all of the necessary COM and framework support in a low-level language such as C or C++ to implementing solvers in Visual Basic and Java to using a (future) solver development environment that hides all of the COM and framework details.

4.1.1.1 Raw packaging, C/C++

To hand-code a framework solver in C or C++, as all of the code in the thesis demonstrates, requires knowing COM sufficiently well to develop COM servers. The objects in the thesis were developed with a standard C++ library that is part of Microsoft's Visual C++ 5.0, the Active Template Library 2.1 (ATL), which handles all of the annoying COM details such as object creation, IUnknown implementation, etc.

This is the most complicated, but also the most flexible way, to build solvers. With complete control over the implementation of the COM aspects, the solver can be imbued with the broadest range of functionality. Additional interfaces that extend beyond the framework, such as IPersistStream or the ActiveX interfaces, can easily[1] be added to the solver. Nonetheless, the developer needs to remember all of the small details, such as managing correctly the reference counts and life cycles of COM objects and implementing the various framework interfaces. Much of the basic implementation of the solver interfaces are scattered throughout this thesis in code samples.

---

[1] Easily, that is, relative to the other packaging systems. Manually adding support for all of the interfaces required for an ActiveX control is about equivalent to solving an LP by hand.

4.1.1.2 Framework C++ class library

It would be fairly simple to create a C++ class library that handles most of the common tasks in creating solvers, such as dealing with the solver sites, advise holders, etc. This class library would, essentially, wrap the COM language of the framework into a nicely packaged C++ metaphor, with functions to override, implementation inheritance, and the like. This is precisely the technique used by ATL, mentioned above, for saving developers days of effort developing the COM infrastructure in their objects.

The downside of developing these class libraries is that they take a language-independent standard—the framework and COM—and add value in a language-dependent way. However, it more than pays for itself if sufficient numbers of solvers are developed using the class library.

With this class library, the C++ developer would not have to worry at all about COM, but would instead override pure virtual functions. The class library would take a call to IRSolver::Solve and map that into a call to a virtual function like "OnSolve" that the developer would then implement on the derived solver class. Similarly, the wrapper class would expose a method like "SendNotification" that internally calls IRSolverAdviseHolder::SendOnSolveNotify. Methods such as IRSolver::SolveAdvise would be hidden entirely from the developer.

An addition to the raw class library would be enhancements to the development environment for adding simple classes. Microsoft Visual C++ provides hooks for application wizards that can create a shell of a solver. ATL utilizes this feature to create the shell of a COM server, with basic implementations of the necessary server registration and execution functions, as well as to create the shell of new objects, including basic interface and class factory support.

4.1.1.3 Visual Basic/Java solver development

If part of the overall goal is to simplify the process of creating solvers, then it cannot be expected that all solvers will be created in languages like C and C++. Requiring knowledge of both C++ and COM is unacceptable for the wide potential audience of algorithm researchers and developers. For them, it will become imperative that more attractive environments like Visual Basic and Java support not just the application of solvers, but their creation as well.

To that end, there are two issues that need to be resolved. First is the support for the creation of COM objects of any kind. Visual Basic handles this with ease, because COM is the native language of the underlying Visual Basic implementations; all Visual Basic class modules are COM objects automatically. For Java, only Microsoft's implementation of the Java Virtual Machine can create COM components, but even so, it also can easily create COM objects.

The second issue is that most of the framework interfaces that an object in the framework needs to implement are not automation-compatible. Visual Basic and Java can only use and implement interfaces that are automation-compatible. The usability issue is circumvented by having objects that wish to be used in automation environments exposing IDispatch as well as the necessary framework interfaces. This is insufficient for implementation, however.

To enable implementation in Visual Basic and Java, as well as other automation-compatible environments, it will be necessary to create a collection of wrapper objects, one for each type of object in the framework—solver, data element, etc.—that would be implemented in the automation-compatible environment. These objects form essentially a class library similar to the one for C++ described in the previous section. Namely, one of these objects implements the non–automation-compatible interfaces demanded by the framework, and then calls automation-compatible methods on the object implemented in the automation-compatible environment. Whether these objects work through containment, COM aggregation, or some other idiom still remains to be determined. Most likely, a new automation interface will be defined that contains all of the methods that the Visual Basic object will have to implement. The wrapper object will call methods on this interface when handling the custom framework interfaces.

The end result will be that a framework-compliant solver can be constructed in Visual Basic or Java by implementing one or more automation-compatible interfaces, allowing the developer to know only the Visual Basic or Java environments and the particular wrapper specification, without being an expert in COM, C++, or the framework. With Visual Basic, the goal is to make implementing a solver as easy as implementing an ActiveX control, which is orders of magnitude easier in Visual Basic than in C++.

4.1.1.4 Solver development environment

As much as the framework and COM details could be hidden by a suitable Visual Basic and Java environment, the developer still has to know Visual Basic or Java. While this is not quite as stringent a requirement as knowing COM and C++, it might still be much to ask of researchers who create, rather than implement, algorithms. Beyond the previous techniques, it is inviting to imagine a future development environment that speaks the language of the researcher. Some environments, like AMPL or MPL, come close for *using* solvers, but no suitable commercial packages exist for *creating* them. A future environment might have the capability to implement algorithms as framework solvers by following algorithm instructions similar to what a researcher might publish. A statement like "for each item in a set" would be implemented as enumeration over a set object.

A few existing environments have fairly strong algorithm creation capabilities today. Two in particular are both popular and powerful. Matlab, with a focus on efficient matrix manipulation, has a programming capability that enables the creation of subroutines and functions. Mathematica, with a more analytical bent and a workbook metaphor, has a similar programming language. The most recent versions of these applications can also tie into mainstream applications like Microsoft Excel as well as databases. As solver implementation

environments, though, they have some problems. The most important is that a "solver" created in one of these applications is constrained to that application. Hence, one cannot be invoked from different applications. Another is that these applications have trouble using components from other applications for which specific customization is not included. These are problems typical of any modeling environment today.

If environments such as these include broader solver and application support, especially in the form of this framework, or if existing solver creation tools elevate their modeling capabilities to those of Matlab or Mathematica, then creating solvers of the future will be a significantly simpler task.

4.1.2 RandVar module

From 1995 to 1997, researchers at MIT developed a number of applications that required sampling from a normal distribution. One way to sample from a known continuous distribution is to generate a sample from the uniform distribution between zero and one, and then compute the inverse of the cumulative distribution function of the desired distribution at the generated sample point. The inverse is the sampled value. Sampling from a normal distribution thus requires the inverse normal CDF, which is not a closed formula. Numerous techniques exist to calculate it. Most take one of two approaches. One is to search over the cumulative distribution function, calculated by numerically integrating the probability density function, which *is* a closed formula. This can yield "exact" results at the cost of more computations. The other approach is to approximate the inverse CDF function by a high-order polynomial. This is a quick calculation, but has some approximation error, whose magnitude depends on the quality of the fit polynomial.

The applications were originally created in Pascal, but had to be converted to C++. The researchers had a library of object code for the InverseNormalCDF function for the Pascal application, but no such object code for the C++ application. In porting the Pascal version to C++, they had to find a suitable implementation of the InverseNormalCDF function as a replacement. The search for an adaptable source code implementation in C, eventually found among the numerous libraries of NetLib, took about one day. Adapting the C code, with its many separate source files, conflicting global namespace identifiers (the library had its own sqrt and floor functions, for instance), and its numerous machine configurations, required another day. In all, it took two days to implement a simple problem statement (calculate the inverse normal cumulative distribution function). To reuse the new code in other applications also turned out to be marginally complicated, because of the need to copy the correct source files around, namespace issues, and the like.

After another two days of effort, the InverseNormalCDF, the NormalCDF, the InverseNormalPDF, and the NormalPDF functions were all implemented within a single solver. This solver is the normal random variable, which follows the specification for random variables outlined in Appendix A.3.3, page 315. Now, calculating these parameters for a normal random variable is a simple matter of creating a normal random variable object

and asking for its InverseNormalCDF. It takes only five minutes to add normal random variable calculations to an application. Furthermore, because the normal random variable solver supports sampling, there is minimal additional effort[2] required to acquire samples from a normal distribution, which was the goal all along. This normal random variable, part of the RandVar module of random variable solvers, has been used in several applications since then, including FlexCap, described in section 4.2.2, page 239, and SIPModel, described in section 4.2.4, page 250.

Besides the normal distribution, the RandVar module contains solvers for exponential, Bernoulli, geometric, Poisson, and uniform distributions.

4.1.2.1 Packaging effort

A single C++ base class implements the common framework-related code, and a derived class for each distribution implements the distribution-specific calculations. The random variable base class is 270 lines of code. Each derived distribution is about 250 additional lines. This does not include the underlying calculations for each random variable, which technically are not part of the framework packaging.

The base class is an implementation detail, but demonstrates nicely the power of implementation inheritance as one COM implementation technique. By encapsulating the framework-specific, distribution-independent code in a base class, changes to the framework-specific code only need to be made in one place, which improves maintainability. Furthermore, in this case, the savings (about 270 lines) grow linearly with each additional distribution. This module is also an example of packaging related solvers into a single executable.

4.1.3 RNetOpt module

The RNetOpt module is an executable containing three network optimization solvers that demonstrates the effort required to convert an existing algorithm implementation into a framework solver. A publicly available network optimization library, available via ftp (now available via http at Goldberg [38]), provided C-based source code for various network optimization algorithms. Each algorithm was embedded within a stand-alone executable application. An algorithm was controlled by passing the problem in a moderately standard text format via the console input and by receiving the outputs in an entirely non-standard text format via the console output. Any particular algorithm file had primarily three parts: the algorithm itself, data structure management and initialization, and standard input/output and startup/shutdown code. The data structure initialization code was intricately mixed with the I/O code, which is a clear transgression of the separation of orthogonal functionality principle, described in section 1.4.3, page 27. If an end-user wanted to embed one of these

---

[2] The minimal effort is in creating a uniform random generator and passing it to the normal random variable sampling engine, which takes three lines of code.

algorithms into another application as a subroutine, it would require disentangling the data structure code from the I/O code and developing a life cycle and interface for interacting with the network code, because it operated in an entirely linear, flowchart-style fashion[3].

By wrapping one of these algorithms into a framework solver, all of these problems vanish. It is then easy to replicate the original application, with its console input and output, using the framework solver, but also it is substantially easier to use the solver in other applications. Three of these algorithms, for the shortest path problem, maximum flow problem, and minimum cost flow problem are packaged into a single executable, named RNetOpt.

4.1.3.1 Packaging effort

Each of the original implementations can be divided into a parsing step and a solution step. The parsing step includes all of the input/output code and error checking as well as the basic network data structure creation and pre-processing. The solution step contains the specific algorithm. The original three algorithms have the line counts and file sizes (in parentheses) shown in Table 4.1.

Algorithm	Parsing code	Algorithm code	Total
Maximum flow	456 (15k)	550 (12k)	1006 (27k)
Min-cost flow	475 (15k)	1606 (36k)	2081 (51k)
Shortest path	480 (15k)	224 (5k)	704 (20k)

Table 4.1: Network flow algorithm implementation sizes—original solvers

The solvers, as repackaged, have two parts. The first is the COM/framework layer, which is a simple transport mechanism to the underlying algorithm and data structures. The second part is the modeling, structure, and algorithm code. In going from the existing algorithms to the solvers, code that had been in the parsing section moves to the algorithm section, to reflect its true purpose. The results for the framework solvers are shown in Table 4.2.

Solver	Framework code	Algorithm code	Total
Maximum flow	393 (9k)	704 (14k)	1097 (23k)
Min-cost flow	356 (8k)	1990 (36k)	2346 (44k)
Shortest path	366 (8k)	534 (14k)	900 (22k)

Table 4.2: Network flow algorithm implementation sizes—framework solvers

Discussion

- In all three cases, the algorithm code size grew in line count and file size; this is because of the transferal of what is truly modeling code from the original parsing routines into the algorithm code proper.

---

[3] Another option, of course, is to use the algorithm as designed, as a stand-alone executable, and pass data to and receive data from the application by using pipes.

- In the case of the maximum flow and min-cost flow algorithms, the repackaged solvers have more lines but fewer total characters. Fewer characters represents a savings in re-packaging; more lines is due to differences in coding styles (such as placement of brackets).

- In the case of the shortest-path solver, the new algorithm code is substantially larger. This demonstrates a measure of the growth of a C application upon re-development into C++. Most of the algorithm code in the shortest path solver is divided between the mechanics of the shortest-path class itself[4], and what had been the parsing engine before. With the addition of the framework code, the resulting solver was larger than the original implementation. In fact, each of the solvers suffers from some of this natural C++ expansion, so the gain of moving from the parsing engine to the COM framework is underestimated in these tables.

4.1.4 RLPWrapper solver

This section discusses wrapping an LP optimization engine by a solver component. *Wrapping* in this context is the process of adding a simple shell that conforms to the framework around an existing solver that does not conform to the framework, without modifying the existing solver itself. This enables an existing solver to be used with little effort by clients and other solvers within the framework. The result in this case is a new solver, RLPWrapper, that wraps the optimization engine. This particular wrapper exposes only the basic optimization functionality. For simplicity it ignores the file management capabilities, output of bases, and performance characteristics of the optimization engine.

The solver is based loosely on the functionality of the CPLEX Callable Library version 4 [17]. The solver has two sets, the set of variables and the set of constraints. It specifies six inputs: the objective function, the right-hand side vector, the constraint matrix, a variable lower-bound vector, a variable upper-bound vector, and a constraint sense vector. It specifies five outputs: the final objective value (a scalar), the primal variables, the dual variables, the slack values, and the reduced costs. These data elements are summarized in Table 4.3 and Table 4.4.

The CPLEX Callable Library offers a staggering number of parameters. A select few have been included here, to demonstrate how parameters might be defined.

4.1.4.1 Packaging effort

The RLPWrapper solver requires about 1000 lines of C++ code. The details of completely wrapping the basic CPLEX Callable Library functionality are in Appendix A.1, page 289.

---

[4] Including some long descriptive names, which the original implementation noticeably lacked.

Input	Number of dimensions	Sets defining dimensions	Data type	Notes
ObjectiveFunction	1	Variables	double	
RightHandSide	1	Constraints	double	
ConstraintMatrix	2	Constraints, Variables	double	
LowerBound	1	Variables	double	Optional
UpperBound	1	Variables	double	Optional
ConstraintSense	1	Constraints	string	

Table 4.3: Inputs to the CPLEX wrapper solver

Output	Number of dimensions	Sets defining dimensions	Data type	Notes
ObjectiveValue	0	(none)	double	
PrimalVariables	1	Variables	double	
DualVariables	1	Constraints	double	
SlackValues	1	Constraints	double	
ReducedCosts	1	Variables	double	

Table 4.4: Outputs from the CPLEX wrapper solver

4.1.4.2 Results

This solver serves as an example of *how* to package an existing solver into the framework without modifying the original solver. Therefore, the additional code required for the wrapping bestows no savings on the development of the remainder of the solver, unlike in the first two cases. Wrapping a solver of this nature must be undertaken for the benefits that the clients will acquire. In this case, clients will have the ability to use a powerful linear programming optimization engine from any environment that complies with the framework, including Visual Basic, Microsoft's Java Virtual Machine, Excel, Word, etc. The wrapper serves as an enabling device, expanding the market opportunities for the solver and enhancing the capabilities of the clients that can use it.

4.1.5 Using a solver

Without the use of a modeling environment, there are basically two ways to use solvers in the framework. The first is to develop applications using the framework interfaces described in the third chapter, in C++ or another language suitable for calling methods on those interfaces. The second, if supported by a solver, is to use a scripting macro language like the macro language Visual Basic for Applications (VBA), found in Microsoft Office and other applications.

The first technique provides direct access to the actual interfaces that specify a solver's functionality. Developing a solution using this technique is akin to writing a C++ application; in fact, with the current specification and core services, C++ is one of the few ways that the first technique is even possible. The second technique provides access to a solver through a common scripting method known as late-bound dispatching, where the scripting engine and host application, such as Microsoft Excel, query the solver for its capabilities, method names, and signatures at run-time instead of design-time.

VBA provides an incredibly powerful programming capability, as it enables the complete automation of almost any task in its host application. VBA is available, through licensing from Microsoft, in a variety of applications today, including Microsoft Office, Autodesk AutoCAD, and Visio Corp. Visio. If a solver supports the COM mechanisms that enable this macro language to use it, then using the solver in an environment that contains VBA becomes a simple task of writing only a very few lines of VBA code. For instance, the code to use the MMQueue solver, described in section 4.2.3.5.a, page 247, is only ten lines (see Appendix A.4.2, page 323, for the code). The amount of code required will typically be some constant amount to create, execute, and destroy the solver object plus some linear multiple of the number of inputs and outputs.

Developers of solvers, knowing that environments like Excel might be popular platforms for their solvers, could choose to implement a custom Excel Add-in that hides or wraps the VBA code to execute the solver. Using the solver then becomes as easy as using any other Excel function. Eventually, using any solver could become as simple as using the statistical functions in Excel that provide a simple dialog box for parameterization and then generate a worksheet full of output data.

The first technique, using the raw interfaces, is the technique proscribed by the solver interfaces in section 3.5, page 137. This is the only technique that is guaranteed to work for all solver components. The client application uses the solver by setting inputs using the interface IRSolverInputs, executing the solver using IRSolver or IRSolverAsynchSolve, and retrieving the outputs using IRSolverOutputs.

Nonetheless, the flexibility of the interface-based nature of COM specifications enables any particular solver to support multiple methods of use. There are two ways for a solver to do this. One is through new custom interfaces. For example, a solver might define a new solver interface that has more advanced execution features than are available with IRSolver. The

new interface could be named IRSolver2. Or, a solver for a particular problem might define an interface with input, output, and control specific to that problem. For instance, a bin-packing solver could support an interface that allows for direct piece placement or bin retrieval without using the framework input and output interfaces. If enough solvers support the new custom interface, the interface becomes part of the "standard."

The second way is through the standard COM interface IDispatch. This interface is the very mechanism that enables the second technique described above, that allows the macro languages to use a solver. IDispatch provides methods that allow a client to access the type information of an interface, which provides the details of the properties and methods of an interface in much the same way as the SolverInfo interfaces provide the number and characteristics of inputs and outputs of a solver. With this type information, the client can determine the number and order of parameters to a function and dynamically construct the correct protocol for executing the function on the interface. The client then calls IDispatch::Invoke to execute the function; Invoke is the dispatching mechanism that gives the interface its name. Macro languages work by exploiting the type information an interface provides and the ability of an object to unpack a dynamically-created invocation request at run-time. For the user, the result is simple text-based coding with seemingly instantaneous execution.

4.2 Applications

This section presents several applications that have been or could be developed with the framework. Some of these cases demonstrate how much effort is required to include support for the framework, and in particular, for using solvers built for the framework, versus the effort required to include support for similar functionality without the framework. Other cases show the versatility of solvers created for the framework—that solvers can easily be used across separate applications and programming languages.

4.2.1 Monsanto

The first case study is an analysis of a project undertaken in 1995 by members of the MIT Integrated Supply Chain Management (ISCM) consortium. This project examined the yearly planning process for Monsanto's domestic Crop Protection products production and distribution for a single plant. See Graves et al. [42], Gutierrez [44], and Ruark [91] for more information on this project. The project resulted in a delivered software suite that optimized a linear program. This case examines how the project could be implemented within the context of the framework. As development progresses, several components that have general use beyond the Monsanto project are proposed. These components, together with the core services of the framework, result in a significant decrease in the size and complexity of the client code—code that is specific to the Monsanto project.

4.2.1.1 Project background and original implementation

There were several primary goals for this project. Chiefly among them were to quantitatively validate or challenge current production, distribution, and marketing policies, to understand the best positioning of inventory across the supply chain, and to ascertain the impact of structural changes to the supply chain. The model developed by the team to meet this goal incorporated multiple products, multiple time periods, uncertain demand, and multiple shared production and storage facilities. The model was formulated as a linear program with ten sets of variables and ten sets of constraints. The decision variables include production and packaging schedules for various product and product-package combinations over time, production levels at the various production and packaging resources, inventory levels of the many pieces that flow through the system between production and packaging resources and customers, and satisfied demand and lost sales for each packaged product over time. The constraints include the usual production capacity and inventory flow balance constraints as well as beginning and minimum ending inventory level and demand balance constraints. System costs comprise holding, distribution, production, and lost sales costs. For details of the LP, see Gutierrez [44] or Ruark [91].

In the fall of 1995, a team of MIT developers delivered a software package that optimized the model using the commercial LP optimization engine LINDO. Development time was split between creating the model generation code to interface with LINDO and creating the data generation and output generation code to interface with quantitative analysts. The model generation code was implemented as a custom executable application written in Pascal for the Macintosh. It has 4900 lines of Pascal code and required about one person-month's effort. The user interface code was implemented as a series of Microsoft Excel VBA macros, hosted in Excel 95. These macros translate between the worksheets in which the quantitative analysts enter input data and analyze output data and the text files required by the custom application. The macros required an additional two weeks of effort.

In 1996, Monsanto expanded the scope of the project and contracted an outside development firm to translate the software into AMPL calling CPLEX, with Microsoft Access as the user interface component. The new solutions presented for this application are comparable to the contracted solution.

4.2.1.2 Needed components

The Monsanto case requires three primary components. These are a linear program optimization engine for solving the model, a database access component for loading and storing the data and results, and a model generation component, for converting the various input data into the linear program formulation and generating the Monsanto model outputs from the linear program outputs.

LP optimization. The original implementation used LINDO, which provides a FORTRAN-based API to the LINDO run-time. A similar package from CPLEX, the CPLEX Callable Library, exposes a C-based API to the CPLEX run-time. The two solutions presented here

use the RLPWrapper solver that wraps the CPLEX Callable Library into the framework, described in section 4.1.4, page 226.

Database access. The original implementation had a set of custom functions for reading to and writing from comma-separated values (CSV) text files. The two solutions here develop two solver components, RDBLoad and RDBStore, for reading to and writing from OLE DB-compliant data sources.

Model building. Model building in this case is the process of generating the input matrices and interpreting the output vectors for the linear program. This involves translating the many different input matrices into the appropriate cost vector, constraint matrix, and right-hand side vector. It also includes translating the outputs back into meaningful vectors. The original implementation had extensive, hard-coded routines that generated the various rows and columns of the matrices. The two solutions presented here describe two solver components, the Monsanto solver and the RSML solver, that encapsulate the model building process.

4.2.1.3 Initial solution

The original implementation of the Monsanto software has three main phases. In the first phase, the user enters the various input values into sheets in a Microsoft Excel workbook. When ready to run the optimization, the user executes an Excel macro that saves all of the input data into a collection of temporary files that are used by the second phase. In the second phase, the user runs a custom application that reads the temporary input files, creates the corresponding linear program, drives the optimization engine, retrieves the outputs, and creates a collection of temporary output files. In the final phase, the user runs another Excel macro that opens the temporary output files and creates a new Excel workbook that contains all of the model outputs, some cost reports, and some utilization graphs.

In relation to the framework, the interesting activities of the original implementation are in the custom application. Namely, how the many input files are turned into a linear program, how the linear program is optimized, and how the linear program outputs are transformed into output files. The Excel macros, relatively simple by comparison, encapsulated much of the user-interface side of the client and can reasonably be expected to exist in any implementation that uses Excel as a database.

The custom application has four primary packages, as shown in the package diagram of Figure 4.1. The database code package contains the implementation for reading and writing matrices from or to the specified input or output files. The matrix code package manages the memory representation of the inputs and outputs. Constraint generation is responsible for creating the rows of the constraint matrix and interpreting the dual values and shadow prices, while variable generation is responsible for managing the columns of the constraint matrix and managing the variable list. The application, as originally implemented, uses the LINDO optimization library.

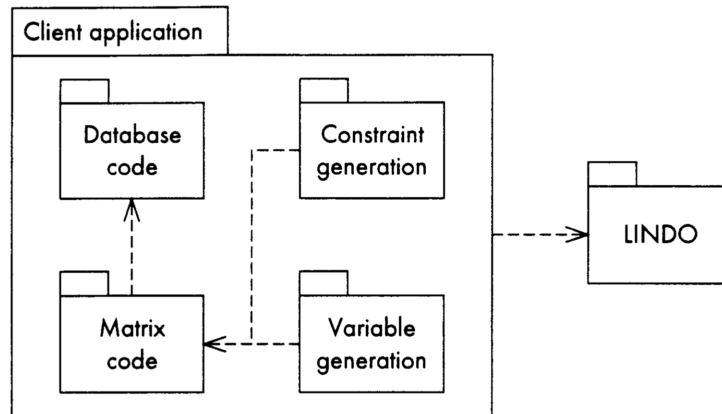


Figure 4.1: Package diagram for original Monsanto solution

4.2.1.4 Wrapping the optimization and the model building

The first change to the initial solution, from the perspective of the client, is to move the knowledge of the Monsanto LP model into its own solver. This involves two steps: first, creating a solver that wraps the LP optimization engine, and second, creating a solver that wraps the LP model itself. The RLPWrapper solver, discussed in section 4.1.4, page 226, serves as the optimization engine; this takes the place of LINDO in the original solution.

A new solver, the Monsanto solver, encapsulates the model building, the constraint and variable generation, of the Monsanto LP model. Thus, the client no longer has to know the LP model itself, just the inputs and outputs of the model. The Monsanto LP model is moderately complex. It has ten sets of variables, half of which have non-trivial bounds, and ten sets of constraints. The constraint matrix, right-hand side, objective function, and variable bounds are derived from thirty-one different matrices. The Monsanto solver must transform these thirty-one matrices into the six inputs required by the RLPWrapper solver. Upon completion of RLPWrapper's execution, the Monsanto solver must transform the primal variables, dual variables, reduced costs, and slack values into the ten variable sets' primal values and reduced costs and the ten constraint sets' shadow prices and slack values, totaling forty outputs. Appendix A.2, page 308, shows the SolverInfo file that specifies this solver.

4.2.1.5 First new solution

With the RLPWrapper and Monsanto solvers, a first solution for the Monsanto problem can be developed. The client can be greatly simplified over the original solution; it must only generate the thirty-one data elements that are inputs to the Monsanto model. These vectors and matrices can be read directly from the database. Generating these thirty-one data elements is significantly easier than generating the six data elements that go into the LP

engine itself, as required in the original implementation. The client is still responsible for the database interaction, as shown in the package diagram for this solution in Figure 4.2.

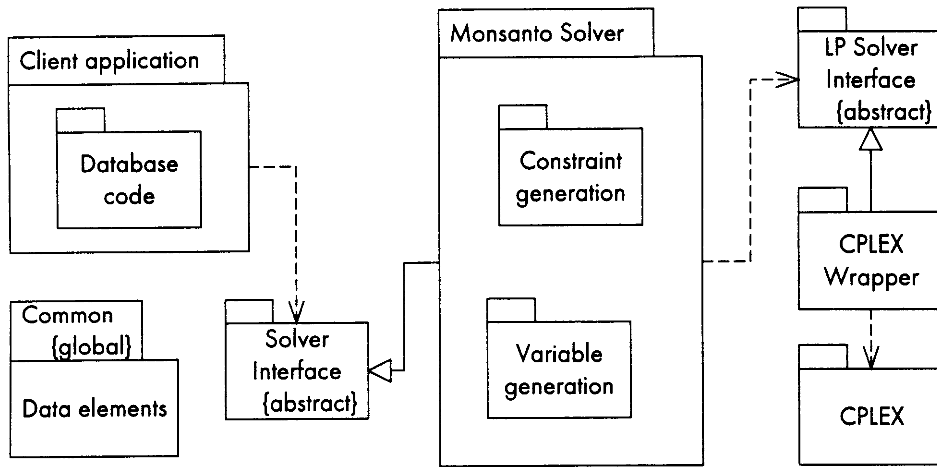


Figure 4.2: Package diagram for Monsanto solution with a special model solver

One of the primary benefits of this solution over the previous is that now the Monsanto model knowledge is independent of the client. Any number of clients can solve the model without having to embed the model knowledge, in the form of the variables and constraints, in each client. One client could be a GUI-based decision support tool that allows many interactive what-if scenarios, while another could be a backroom, command-line shell that drives the model for production needs[5].

Each client still is responsible for database access and data element generation. A further improvement will be to abstract the database interactions into their own solvers, which is explained next.

4.2.1.5.a Cost of using RLPWrapper and Monsanto solvers

Assuming that the data elements that serve as input to the Monsanto solver are already created, as discussed in the next section, using the solvers requires fewer than fifty lines of code. The code simply creates the solvers, iterates through the input data elements, assigning each one to an input on the Monsanto solver, executes the solver, and then retrieves the outputs. The creation of inputs and interpretation of outputs is handled by the database code.

---

[5] The Monsanto model does not quite have the execution periodicity to warrant such a production-environment implementation, so this is just an example.

4.2.1.6 Wrapping database access: RDBElements module

To build the input matrices in the original solution, the original application specifies as input two types of files. One type is a *dictionary file,* which specifies the members of one of the sets (such as Technical), one set member per line of the file. The other type is a *data file,* which contains the values for one of the matrices. The data file is a comma-separated value (CSV) text file, where each line is a non-zero value in the matrix. The first few columns (the number of columns depends on the file) specify the dimension indices for the non-zero, and the last column specifies the actual value.

These files are easily imported into a database that has a table for each of the original input files. In the previous solutions, the client is responsible for querying this database and building the matrices from the rowset format of the table. An obvious enhancement to the solution is to develop a new component that can generate the data element from the database, given enough parameters to access the specific table that references the input data. Similarly, another useful component would be one that can convert a data element into rowset form and append it to a table in a database.

These components, named RDBLoad and RDBStore, respectively, are designed as solvers themselves. Therefore, they can be embedded in a solution network and wrapped by solver sites, and they can respond to and generate the proper notifications. The RDBLoad takes no inputs but many parameters. These parameters specify the database, the SQL query string, and the meanings of the columns in the query string that define the dimensions and the data elements. Those parts of the SIDL file look like this:

```
// Excerpts from RDBLoadInfo.idl SIDL specification
[ uuid(2E578551-A14D-11D1-9170-00207810C741) ]
interface Inputs
{
};

[   uuid(2E578553-A14D-11D1-9170-00207810C741),
    custom(GUID_RSOLVERIMPLEMENTEDINTERFACE, "True") ]
dispinterface RDBLoadParameters
{
properties:
    [id(1), helpstring("The name of the database.")]
        BSTR DatabaseName;
    [id(2), helpstring("The user name for access to the database.")]
        BSTR UserName;
    [id(3), helpstring("The password for access to the database.")]
        BSTR Password;
    [id(4), helpstring("The command string to load the element from the database (SQL).")]
        BSTR Command;
```

```
    [id(5), helpstring("An ordered array of dimensions used to load the element (does not create
        ordinals for dimensions), or an ordered array of dimension names used to load the
        element (creates ordinals for dimensions).")]
        VARIANT Dimensions;
    [id(6), helpstring("An ordered array of column names used the generate per-item
        properties.")]
        VARIANT ItemPropertyNames;
    [id(7), helpstring("An ordered array of column names used the generate the elements.")]
        VARIANT ElementNames;
methods:
};
```

The RDBLoad solver partitions the column space of the table schema into four sets: dimensional columns, data columns, property columns, and unused columns. The first three columns are determined by the parameters Dimensions, ElementNames, and ItemPropertyNames, respectively. The dimensional columns specify the dimensions of the output data element. The client can specify a set of existing dimensions to use to create the data element, in which case the dimensional columns in the table are selected from the names of the existing dimensions, and the size of each dimension in the data element is the size of the existing dimension, regardless of whether every element in the dimension's set is found in the table. Or, the client can specify a set of column names to use to create the data element's dimensions, in which case each dimension is created from an enumeration of the distinct values in that dimension's column in the table.

Either way, each row in the table represents a potential non-zero value for each of the specified data elements. For a given row, the values of the dimensional columns provide the indices for the data element's value. The ElementNames and ItemPropertyNames are used to fill the various data element and data element properties.

The RDBLoad solver has a simple Solve implementation. It first opens the specified database and executes the given query, which returns the query rowset. It then creates the dimensions, if only dimensional column names were set in the Dimensions parameter, by iterating over the rowset and determining all of the distinct values in each of the dimensional columns. After the dimension objects are created (or just accessed from the parameter if that was the case), it creates a matrix large enough to hold the data[6] for each data element and data element property. Then it iterates over each row, determining where to store each non-zero value and storing the values for each data element and each data element property. Upon completion, it sends the OnSolveComplete notification and returns to the caller.

The RDBStore solver works in much the same manner, except that it outputs a table based on a data element, its dimensions, and its data element properties.

---

6 This could be a full matrix or a sparse matrix, depending on the desired characteristics of the RDBLoad solver itself.

4.2.1.6.a Baseline effort

The baseline effort for creating solvers of this complexity is quite large, but mainly it is database-specific implementation details to read rowsets, manage queries, enumerate unique values, and the like. The solvers described here are more versatile than the simple CSV-based database system of the original solution, so while more effort is required to build the database activity of these solvers, that is not due to the framework.

4.2.1.6.b Packaging effort

Packaging these solvers is a simple task, because they were designed and created from the beginning with the goal of making them solvers. The framework-specific code for each is around the minimal packaging amount for any solver with parameters, about 500 lines.

4.2.1.7 Wrapping the model: RSML solver

The Monsanto solver component discussed in the previous section contains much of the code that was in the original implementation's constraint and variable generation packages. This code simply builds the columns of the various input matrices by calculating the offset of each variable based on the thirty-one input matrices. It then builds all of the constraints of the matrix, remembering where each of the ten constraint sets is located within the single constraint matrix. All of this bookkeeping and iteration over so many dimensions and data elements bulks up the code. So rather than hard-code the variable and constraint logic, an improvement is to store the model knowledge in a declarative form and then develop the solver to work from that model description to dynamically generate the variables and constraints. This is exactly the purpose of many modeling languages, and the Structured Modeling Language is a good choice to use as an example.

This section discusses a pair of solver components that wrap a generic LP solver interface, with its six inputs and four outputs, with a dynamic modeling interface that is shaped by a model description in SML. One of the components takes an SML text as a parameter and dynamically enables the inputs required for the described model. The client sets these inputs, and the component "solves" by mapping these inputs into the six inputs required for the LP solver inputs. This is the SMLInput component. It can be viewed as the matrix generation module in many existing solution networks. These matrix generation modules typically combine many different inputs, queries, and matrices into a single set of LP inputs, represented in a standard format such as an MPS file. The SMLInput component is similar, except it uses standardized elements within the framework and outputs a set of data elements rather than an MPS file. It would be a simple step to then store these data elements into an MPS file if that is desired.

The other component, SMLOutput, takes as its parameters the same parameters as the SMLInput component and as its inputs the outputs from the LP solver component, and from the LP solver outputs it dynamically generates the output data elements as described by the SML text. This is the post-processing step that recovers all of the variables and

constraints from the massive, homogenous LP matrices. The ability to automatically process the LP outputs relieves the client and other components within the solution network from the burden of having to know precisely where in the LP matrices to find the variables and constraints of interest.

It would also be possible to implement the SMLInput and SMLOutput components as a single solver site (inbound and outbound together). This would ensure that the input and output sides are always wired together correctly.

Implementing the SML parser and modeling generation code might take several thousands lines; these are activities independent of the framework.

4.2.1.7.a Packaging effort

Packaging the SML solver is comparable to packaging the database solvers, on the order of several hundred lines of raw framework-related code.

4.2.1.8 Second new solution

The final solution examined here uses the LP wrapper component, the database solvers, and the SMLInput and SMLOutput solvers to solve the Monsanto problem. The SML solvers take the place of the custom Monsanto Solver from the first new solution, which once again reduces dramatically the coding requirements for this problem. The package diagram for this solution is shown in Figure 4.3.

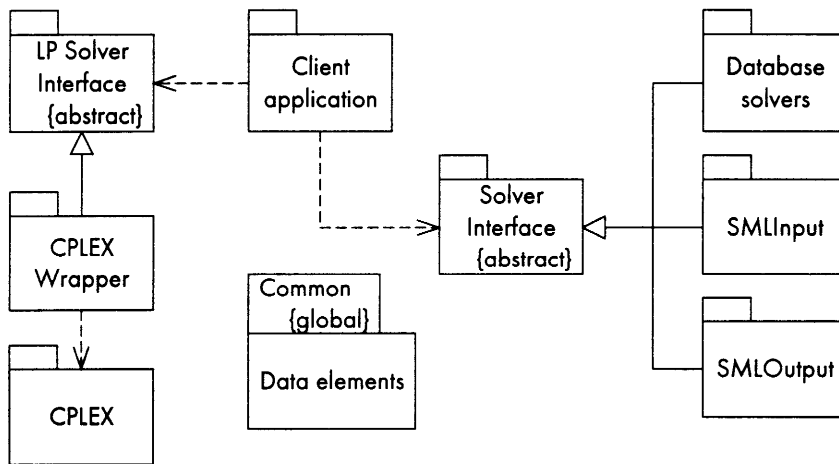


Figure 4.3: Package diagram of Monsanto solution with entire component-based solution

Now, the LP modeling knowledge is stored in the SML description of the Monsanto LP. Before, the modeling knowledge was hard-coded into the Monsanto Solver component, which would make modifying the Monsanto LP a laborious process involving a reiteration of

the compile-link-debug cycle. The ultimate end-users, the Monsanto quantitative analysts would have had to rely on the MIT developers to make even the slightest modification to the LP, via source code changes. Now, the quantitative analysts could change the LP simply by changing the text file that contains the SML description of the Monsanto LP, and appropriately re-wiring the data element inputs and outputs as necessary.

4.2.1.8.a Cost of using RDBElements and RSML modules

Using the database solver simplifies the database and matrix code in the client considerably. The client no longer needs to understand the input format, manage file operations, or deal with memory allocation for the input and output elements. Now, the client simply creates the thirty-one RDBLoad solvers, configuring each one with the proper query string and dimensions. The dimensions are created as independent objects, loaded from dictionary tables in the database. This insures that each data element is loaded with the proper dimension and set indices. Each solver is executed in turn to generate the data element that serves as an input to the SMLInput solver.

Using the SML solvers is a little more complicated than using the Monsanto solver itself. The client creates the solvers, sets the SML string to specify the Monsanto model, loops through the thirty-one inputs and assigns them, and then executes the SMLInput solver. The outputs of this solver are assigned to the inputs of the RLPWrapper solver, which is executed, and its outputs are passed into the SMLOutput solver, which then outputs the decision variables and constraint information.

4.2.1.9 Results

Table 4.5 shows the line count of the three proposed solutions of the Monsanto application. The rightmost column indicates the solvers used in each application. **Matrix** code includes routines for allocating and managing matrices. **Database** code includes the file operations, input parsing and output printing, and acquisition of the input values themselves. **LP** code includes the matrix generation in the original solution and the code to drive the solvers in all solutions. **Report** code includes the generation of special cost reports that are outputs of the application but not outputs of the model *per se*. **Utility** code includes various string, set, and file routines that are used by multiple components. **Total** code is the sum of all the parts.

Solution	Matrix	Database	LP	Report	Utility	Total	Solvers
Original	550	675	2450	500	750	4925	LINDO
Solution 1	550	675	50	500	50	1825	RLPWrapper, Monsanto
Solution 2	0	350	75	500	50	975	RLPWrapper, RDB, RSML

Table 4.5: Summary of Monsanto implementation techniques' code sizes

Discussion

This example demonstrates the significant client code size reduction that is possible if useful framework solvers are available. In particular, the supporting routines, such as the Matrix and Utility code, are highly reducible because their functionality is embedded within the appropriate solvers. The client no longer needs to specifically know how to read and write files, how to parse input structures, or how to read from databases. Moreover, the client does not need to be able to interact directly with any particular optimization engine in the proposed framework solutions.

4.2.2 FlexCap

Based on a paper by Jordan and Graves [52], FlexCap is an application that simulates a single-echelon production system to examine the impact of manufacturing process flexibility. This moderately simple model uses a network min-cost flow optimization to select production levels of multiple parts at multiple, capacitated production facilities based upon random demands at multiple demand points. Facilities have flexibility in that they can produce different products, and each facility can serve a set number of demand points. The goal is to calculate the expected shortfall in attempting to meet demand, as well as the utilization of each facility.

FlexCap has been in development at MIT, through three version, since 1995. The first incarnation was built for Macintosh systems using a custom framework developed at MIT (see McKay et al. [66]). The second and third versions were built with Microsoft Visual Basic; the third version improved upon the first two by adding extra modeling capabilities, such as non-deterministic production capacities and the ability to minimize cost or lost sales. A screenshot of the most recent version is displayed in Figure 4.4.

4.2.2.1 Needed components

FlexCap requires two interesting operations research components. One is a component to sample from a normal distribution. The second is to solve a minimum cost flow network optimization problem.

Normal sampling. The demands, and in the third version, the production capacities, are assumed to be independent and normally distributed. Because the application repetitively simulates demand cycles, it is necessary to sample from the normal distribution. A typical way to do this is to sample from the uniform distribution and then calculate the inverse of the normal cumulative distribution function of the random sample, which yields a sample from the normal distribution. So, FlexCap needs a way to either (a) sample from a uniform distribution and compute the inverse cumulative distribution function of the normal distribution, or (b) sample directly from the normal distribution.

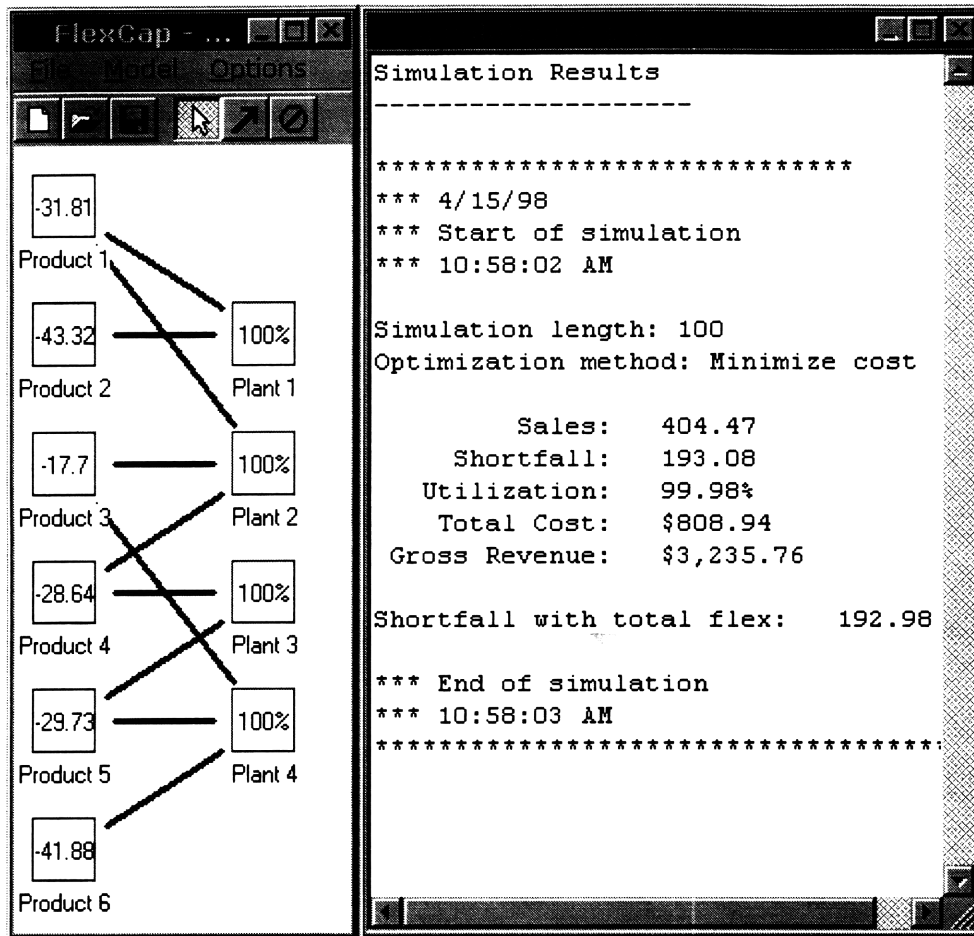


Figure 4.4: Screenshot of FlexCap

In the original solution, the underlying framework provided both a uniform sampling and an inverse CDF computation function. In the later versions, these routines had to be implemented directly, so they were packaged into the RandVar module, described in section 4.1.2, page 223.

Network optimization. The basic FlexCap model can be solved with a maximum flow network optimization algorithm. The first version used a custom implementation of a maximum flow algorithm, linked in as object code directly into the final executable. The second version used the maximum-flow code from the RNetOpt module, described in section 4.1.3, page 224. With the new capability in the third version of optimizing either lost sales or total cost, the problem was formulated as a minimum-cost flow network, so this version used the min-cost flow solver in the RNetOpt library.

4.2.2.2 Initial solution—OMAC

The initial solution was implemented in C for MacOS using OMAC, a custom framework built at MIT from 1993 to 1994 by McKay et al. [66]. OMAC is a Pascal and C framework that provides a skeleton environment for Macintosh applications and a set of many callbacks that allow for custom implementation of various event handlers, such as menu items and mouse actions. Furthermore, it provides a suitable selection of numerical algorithms, such as common statistical functions, including the inverse normal cumulative distribution function required by FlexCap.

OMAC generates entirely self-contained executables, so the solver is always part of the user-interface. It becomes difficult to separate the algorithm from its presentation—a solver cannot use the benefits of the OMAC libraries without being in an OMAC application. In fact, this difficulty helped spark the ideas of the framework presented in this thesis.

4.2.2.3 Framework solution, in Visual Basic

The two versions created in Visual Basic are roughly equivalent; the latest version, described here, adds some user interface features to account for the new optimization options, and the simulation uses the min-cost flow solver rather than the max-flow solver.

Visual Basic is a sophisticated integrated development environment (IDE). Much of the final code in the FlexCap application was added by the IDE. In particular, all of the property lets and gets and the collection capabilities in the class modules are handled automatically by Visual Basic[7]. Furthermore, class modules in Visual Basic are used by other components in the Visual Basic application and exposed to other applications beyond Visual Basic as COM objects. The FlexCap model was created as a set of class modules in Visual Basic, so the new version of the FlexCap application is usable from applications like Excel using VBA.

The FlexCap application is divided more soundly in the Visual Basic versions into modeling modules and user interface forms. To the user, it appears much as the original version did. Inside, it has become more object-oriented over time.

4.2.2.3.a Cost of using RandVar and RNetOpt

Using the RandVar module is extremely easy. It takes only one or two lines of code to create a normal random variable. Calculating the inverse cumulative distribution function is a simple matter of assigning the mean and variance and then asking for the inverse CDF at a

---

[7] Property lets and gets are the functions that access and modify object member variables. Essentially, Visual Basic adds the necessary skeleton code when it is requested; that is, when the developer wants to customize a particular routine. OMAC, on the other hand, presents all of the callbacks at once, and the developer has to fill in code to the correct functions as necessary.

point. In cases where the mean and variance do not change, these values need only be assigned the first time, saving some execution time.

Nonetheless, it is not quite the same as (or have the simplicity of) a simple function call, as provided in the base solution. A simple Visual Basic function could be provided by the developer of the RandVar library, though, that wraps the creation of the normal random variables solver and calls the inverse CDF function.

Using the RNetOpt module is comparable to using the RLPWrapper solver; that is, it takes around thirty lines of code to declare, create, fill, solve, and analyze the min-cost flow solver.

4.2.2.4 Framework solution, in Visual C++

To demonstrate the language-independent nature of the solvers, a version of FlexCap was also developed using Microsoft Visual C++ 5.0. This version is a straight port of the Visual Basic version described above. Besides yielding a significant performance improvement, the C++ version supports multiple threads. The simulation runs on a worker thread while a status dialog reports the progress of the simulation. The user can cancel the simulation at any time. The worker thread enables the user interface thread to remain responsive, even as the simulation length increases and as the per-iteration optimization time increases.

4.2.2.4.a Cost of using RandVar and RNetOpt

The use of smart COM pointers in Visual C++ 5.0 makes using the COM servers in C++ nearly as easy as using them in Visual Basic. About 90% of the Visual Basic client code that handles the simulation could be ported over to C++ line for line, with changes for syntax and variable names. Obviously this is not true of the user interface code, because of the different paradigms of developing interfaces between Visual Basic and Visual C++.

The RandVar and RNetOpt modules are the same files in both applications. These modules are indifferent to whether their clients are implemented in Visual Basic or C++ or some other language, as long as they follow the COM contracts of interfaces.

4.2.2.5 Results

Table 4.6 shows the approximate line counts of the various components in the FlexCap application. The rightmost column indicates which solvers or other frameworks are used in the development. **UI** code is the user-interface, non-modeling code of the FlexCap application. **Modeling** code includes the simulation and the plants, products, and links data structures and collections. **RandVar** code includes the random variable generation client code. **Network** code drives the max-flow or min-cost flow optimization algorithms. **Total** code is the sum of the four code groups.

Configuration	UI	Modeling	RandVar	Network	Total	Notes
Original	1000	250	30	80	1360	OMAC, MaxFlowLib
Visual Basic	1000	300	30	30	1360	RandVar, RNetOpt
Visual C++	1100	500	30	50	1680	RandVar, RNetOpt

Table 4.6: Comparison of FlexCap implementations

Discussion

- These numbers do not include code generated by the application development environment or provided as skeleton code by any framework or solver.

- The user interface in each application uses a pre-existing network display component, so the bulk of the user interface code deals with responding to events, validating data, and interacting with the modeling component.

- While the code sizes of the original and Visual Basic solutions are almost identical, the Visual Basic version was more quickly implemented, because of the integrated nature of the Visual Basic development environment and the wizards that place the developer in the right place at the right time.

- The Visual C++ modeling code size is larger than the other solutions for a number of reasons. First, the original solution's modeling component was only as complex as it needed to be and no more, it was highly-integrated into the other components, and it had no object-oriented aspects. Second, the Visual Basic environment generated automatically much of the structural code, such as collections and property lets and gets, that had to be implemented by hand in C++. Third, the C++ version is the most "object-oriented" of the solutions, and this incurs a natural mark-up of code size.

- Using the solvers is equally easy in all of the solutions, compared to the other programming challenges. Building the original application required knowing how to edit makefiles to include the max-flow object code. The Visual Basic environment required no knowledge on the part of the developer to include the COM solvers. The same is true of the Visual C++ environment, which has special C++ extensions that wrap COM pointers into relatively straightforward objects.

- The same RandVar and RNetOpt modules are used in both the Visual Basic and Visual C++ versions, without modification. Furthermore, these solvers are used in other applications, as well.

4.2.2.6 Postscript: JFlexCap, FlexCap in Java

Some effort was spent in creating a Java version of FlexCap. Using Microsoft Visual J++ 1.1 and Microsoft's Java Virtual Machine (JVM), it is a simple task to access COM objects on Windows platforms. Microsoft's JVM exposes all Java objects as automation objects, in the COM sense, and it imports COM interfaces and objects into the JVM as Java interfaces and objects. The necessary framework interfaces and RandVar and RNetOpt objects are imported as Java interfaces and objects, and using them is exactly the same as in Visual Basic. In fact, Microsoft's Java has precisely the same restrictions that Visual Basic has when using COM objects, in terms of implementing and using automation-compatible types.

Hence, Java can be used as another programming language that complies with the framework, under the restriction of using Microsoft's JVM. Extending the framework into Sun's version of Java requires, of course, mapping the framework interfaces into Java interfaces and using another communications mechanism between Java and other applications, such as CORBA or JNI.

4.2.3 M/M/k queueing model in Excel

Consider a familiar, common, and eminently understandable problem: calculating the expected waiting time for an M/M/k queue. Analysts at Scudder, Stevens, and Clark, Inc., faced a similar problem in 1995 when they were trying to determine what service rate would be necessary to satisfy a waiting time constraint in an M/M/k queue system that modeled their call centers. That is, how fast would their customer service representatives need to process calls in order to keep the waiting time at or under a given threshold? Usually, waiting time might be thought of as a function of service rate, rather than the other way around, but it is an invertible function, so service rate can also be a function of waiting time, all other parameters remaining constant.

So, first think about how to calculate the waiting time in an M/M/k queue given arrival rate λ, service rate μ, and number of servers k. One form of the equation is:

$$\text{ExpWaitTimeMMk}(\lambda,\mu,k)=\left(\frac{1}{k\mu-\lambda}\right)\frac{\left(\frac{(k\rho)^k}{k!}\right)\left(\frac{1}{1-\rho}\right)}{\left(\frac{(k\rho)^k}{k!}\right)\left(\frac{1}{1-\rho}\right)+\sum_{i=0}^{k-1}\frac{(k\rho)^i}{k!}} \text{ where } \rho=\frac{\lambda}{k\mu}.$$

This equation has two primary components: an iterative component $\sum_{i=0}^{k-1}\frac{(k\rho)^i}{k!}$ and an infinite series component that reduces to $\left(\frac{(k\rho)^k}{k!}\right)\left(\frac{1}{1-\rho}\right)$.

4.2.3.1 Needed components

This problem requires two components. One is some way of calculating the expected waiting time, and possibly other performance measures of interest in an M/M/k queue. The second is an algorithm for calculating the service time given a waiting time. The queueing component ranges from a spreadsheet model to a VBA function to a framework solver. The calculation of service time given the waiting time is expressed only as a VBA macro for exposition purposes. It is trivial to add it to the framework solver.

4.2.3.2 Initial solution

Almost every attempt by students and analysts to model the expected waiting time equation uses a spreadsheet to calculate the iterative part. Each row is one iteration through the loop. Often there are painstaking, laborious intermediate steps per row[8], but the optimized form of this spreadsheet appears in Figure 4.5.

Microsoft Excel - MMMQueue98

	A	B	C	D	E
1	lambda =	8	>=k part =	0.444	=(B4^B3)/(FACT(B3)*(1-B4/B3))
2	mu =	4	<k part =	7.000	=SUM(B7:B1000)
3	k =	5	p(queue) =	0.059701493	=D1/(D1+D2)
4	rho*k =	2	**Wait =**	**0.004975124**	=D3/(B4*B2-B1)
5					
6	n	(rho*k)^n/n!			Prob(n)
7	0	1	=1		0.134328358
8	1	2	=IF(A8>=$B$4,0,B7*$B$4/A8)		0.268656716
9	2	2	// repeats		0.268656716
10	3	1.333333333			0.179104478
11	4	0.666666667			0.089552239
12	5	0			0
13	6	0			0
14	7	0			0
15	8	0			0
16	9	0			0
17	10	0			0

Figure 4.5: Screenshot of spreadsheet modeling of M/M/k queue

The items within the box are the "inputs," while the "output," the expected wait, is shown in bold.

---

[8] Including, usually, separation of the calculation of rho^n and n!, which can lead to significant numerical inaccuracy.

This technique has a number of disadvantages, not the least of which is that in order to determine the desired end result, the service time given the waiting time, it is necessary to optimize by hand or use Excel's Goal Seek functionality. Goal Seek is a quite simple, if manual, feature that works by allowing the user to specify a target value for a target cell for which Excel will search by changing a source cell. Excel's Goal Seek appears in Figure 4.6.

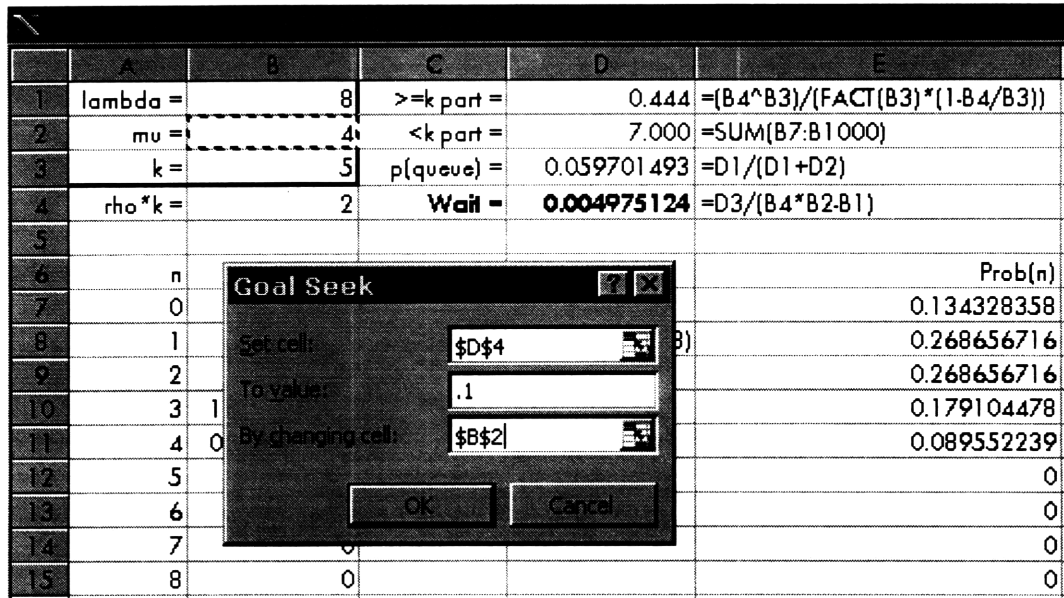


Figure 4.6: Screenshot of goal seeking in spreadsheet modeling

Another problem with the worksheet-based solution is that the maximum number of servers is limited to however many rows the matrix at the bottom is extended (in this case, 1000).

Most operations research practitioners have created this spreadsheet at least once in their career. Even after three of four times of modeling it, it can still take half-an-hour to an hour to create this spreadsheet from scratch, in its optimized and correct form.

With this as a baseline, the following three solutions present alternative techniques for calculating the expected waiting time and the service time given a waiting time.

4.2.3.3 Macro solution

Assuming that the developer starts with nothing other than the previous solution, the first solution is to develop a macro that performs the iteration as code, rather than hard-coding the solution into a worksheet. The macro is a new, user-defined workbook function, and reduces the entire worksheet above into a simple cell formula. In cell $D$4, the formula for the above calculation would be "=ExpWaitMMk(15,2,10)." It doesn't get any easier for the client than that. The problem is having to develop the function in the first place.

For many people, this in and of itself is a challenge. Usually it is much easier just to model into a worksheet rather than learn Visual Basic for Applications enough to write and debug a function. The two functions, which calculate the probability of queueing and the expected waiting time, took several hours to develop, debug, and optimize. The macros are provided in Appendix A.4.1, page 319.

For the user, this solution has a high coding requirement, but it is more flexible and substantially simpler to use than the traditional worksheet solution.

4.2.3.4 Add-in solution

Assume now that the developer has an Excel Add-in with an ExpWaitMMk function. This function might have been created using the VBA macro above or using a variant of the traditional worksheet solution, compiled into an Excel Add-in using Excel's Add-in creation capabilities.

The developer now has it as easy as possible. The worksheet formula for calculating the expected waiting time is the same as with the VBA macro: "=ExpWaitMMk(15,2,10)" and the formula for determining the service time from the waiting time is "=ServiceRateFromWaitTimeMMk(15,2,0.1)." But now the developer does not have to write the VBA macros, as the function already exists in the Add-in. This is component software for Excel. The solution is as flexible and as quick and useful as before, but the coding has been minimized to a simple formula in a worksheet cell.

What are the downsides? Add-ins only work in Excel (discounting some COM contortions). And, Add-ins run at the speed of the just-in-time compiled macro language in Excel, which is to say, not natively. The final solutions, using the framework, improves upon both of these aspects.

4.2.3.5 Framework solution

Now imagine that the developer has a framework-based solver that uses the random variable and queueing system extensions from Appendix A.4.2, page 323, for M/M/k queueing systems. To use this solver will require a macro, so it is not as easy as the Add-in solution's simple worksheet formula. However, it has the advantages over the Excel Add-in of being usable outside of Excel and of potentially running faster than the Add-in.

4.2.3.5.a MMQueue solver

The M/M/k queueing calculations were embedded into a framework solver, named MMQueue, that follows the specification in Appendix A.3.4, page 317, for queueing systems. The COM and framework overhead for this solver class is about 400 lines of C++. This handles the various queueing properties of IRQueueData. The random variable outputs, namely the random variables for waiting time, system time, queue size, and system size, are

created using the RandVar module, described in section 4.1.2, page 223. This provides an elegant solution, because the MMQueue solver must only create the appropriate kind of random variable and then assign the parameters. Whenever the output of a queue is a Poisson or geometric random variable, it takes two or three lines of code to create the output: one line to create the random variable, and one or two lines to set the parameters. When the output random variable is more interesting, and in particular is not implemented by the RandVar module, then the MMQueue solver has to implement its own version of a random variable. These can be read-only implementations, because they are outputs, and this simplifies the implementation of these objects considerably.

4.2.3.5.b Cost of using MMQueue

Using the framework solver in Excel requires some macros to create the solver, fill its values, and retrieve the results. A function macro for the expected waiting time in the M/M/k queue is shown in Appendix A.3.4, page 317. Similar macros would be required for each desired output. With this macro, using the framework solver is as simple as using an Add-in; it takes a single equation "=ExpWaitMMk(15,2,10)" to invoke the macro, which executes the solver. Therefore, using the framework solver from scratch requires knowing how to code a VBA macro. With that knowledge, it is not too much work to use the solver itself.

4.2.3.6 Framework with Add-in solution

Using the framework solution requires more effort for the analyst than an Add-in. The developer of the MMQueue solver (or any other solver) might recognize that Excel will be a popular target environment for the solver. The savvy developer can include an Excel Add-in that hides the necessary macros, thereby making the solver as easy to use now as in the Add-in case while also leaving the solver in a separate executable and thereby making it available to all clients that understand the framework.

4.2.3.7 Results

Table 4.7 summarizes the features of the four solution techniques.

The **Flexible** column indicates whether the solution technique is robust in its inputs. The spreadsheet solution is limited depending upon the number of rows filled in.

The **Code requirement** column identifies the effort required by the analyst to calculate the expected waiting time, from the initial assumptions.

The **Multiple calculations** column indicates whether the solution technique can be used in multiple cells in a worksheet in a recalculation. To recalculate the spreadsheet solution requires the analyst to manually change values.

	Flexible	Coding requirement	Multiple calculations	Usable outside Excel	Knowledge required	Best speed
Spreadsheet solution	No	Medium	No	No	Modeling, Spreadsheets	Excel
VBA macro	Yes	High	Yes	No	VBA, Modeling	Excel compiled
Add-in	Yes	Lowest	Yes	No		Excel compiled
Framework solver	Yes	Low	Yes	Yes	VBA, Framework	Native
Solver and Add-in	Yes	Lowest	Yes	Yes		Native

Table 4.7: Summary of M/M/k queue implementation characteristics

The **Usable outside Excel** column indicates whether the solution technique can be used in programs other than Excel.

The **Knowledge required** column indicates the areas in which the analyst must have working knowledge to develop a solution.

The **Best speed** column gives the fastest speed at which the technique can run.

Discussion

- The biggest gains, in productivity, reliability, and maintainability, come from having the algorithm knowledge encapsulated in a third-party solver, such as in the Add-in and the two framework solutions. The modeler simply uses the formula without even needing to know how the M/M/k expected waiting time is calculated.

- The framework solver has wider applicability than any of the other solutions. Because the queueing system is its own object, it is easier to incorporate many queues into a solution network of queues using the solver. Furthermore, the framework solver will work in other applications whereas the others generally will not[9].

- The framework solver by itself is not the easiest solution to use within Excel. Hence, if the solver developer knows that Excel will be a popular environment, the solver can be customized to work optimally within Excel. This yields the benefits of both the framework solver and the Add-in technique, because the solver is as easy as possible to use in Excel, like the Add-in, but works normally in other environments, like any solver.

[9] If Excel is installed, it is possible to run Excel macros from other applications, but it is a tough task.

4.2.4 SIPModel

Extending serial-line inventory analysis by Simpson, SIPModel is a graphically-intensive planning tool used by numerous companies for analyzing strategic inventory placement in supply chains (see Graves and Willems [43] for details). The SIPModel application has been in development at MIT since 1995, and has seen three major releases as of 1998: one for Macintosh and two for Windows. The application enables a client to graphically draw a supply-chain on a workspace, inspecting and modifying properties for each stage and link in the chain. Costs are dynamically, automatically calculated, and the user can optimize the supply chain at any time. The "algorithm" part of the application handles all of the model structures, as well as the cost calculations and the optimization. A screen shot of the workspace is show in Figure 4.7.

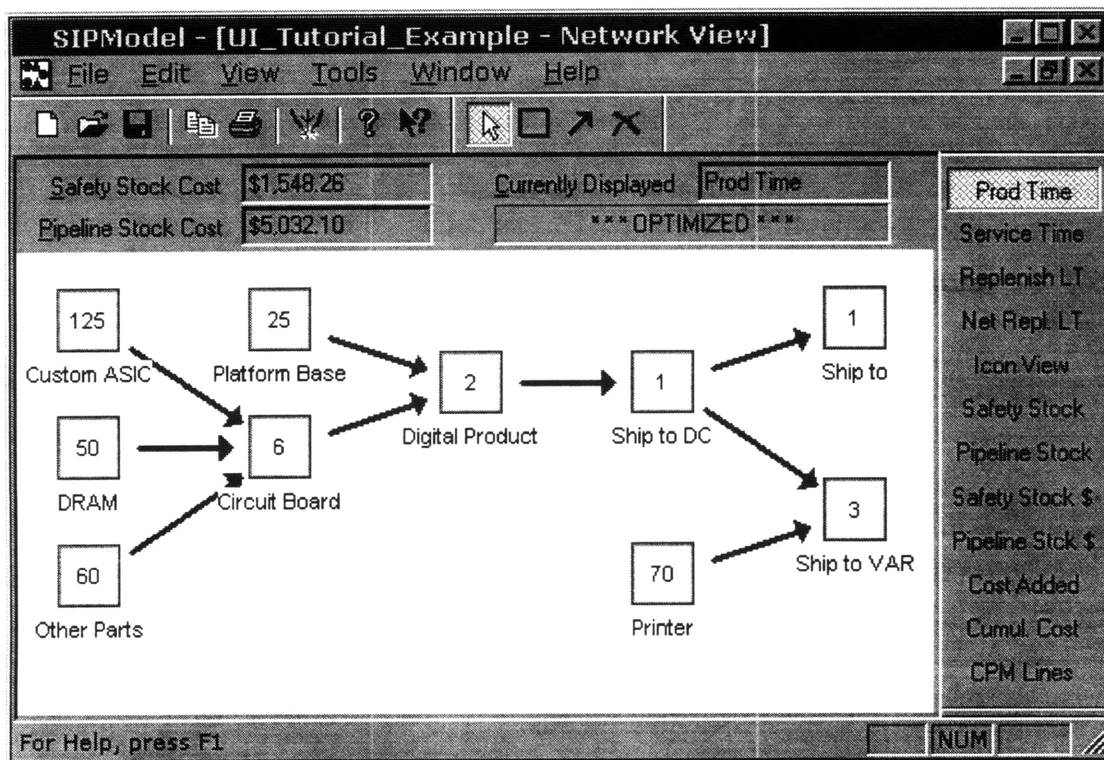


Figure 4.7: Screenshot of SIPModel 2.1 user interface

By double-clicking on one of the stages or links that connects stages, the user can view the many properties, costs, and performance measures associated with each element. An example display of the cost information for a stage appears in Figure 4.8.

SIPModel has evolved from handling serial lines only to handling networks that can handle pure-assembly into pure-distribution tree networks to handling general tree networks. As the algorithm grows more complex, the complexity of interaction with the user increases, the

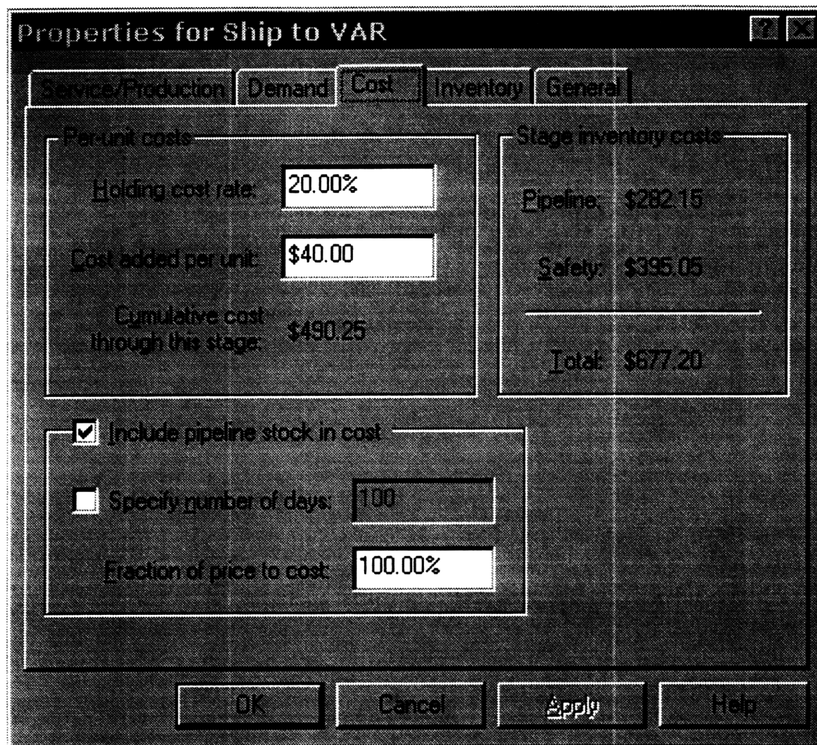


Figure 4.8: Sample property dialog from SIPModel 2.1

number of potential model inconsistencies grows, and the solver becomes more difficult to develop. The algorithms have always used dynamic programming to calculate the optimal service times at each stage; the ordering of the dynamic programming steps, the calculations of costs and the range of feasible service times, and the necessary data structures for backtracking through the dynamic program have been the source of complexity for this problem.

Nevertheless, the run times usually range from a few seconds to a minute for the most complex practical (i.e., modeling real life) networks. While the heart of this application is its cost calculations and optimization, the user spends far more time building, tweaking, and analyzing a network in the user interface than the algorithm spends finding an optimal solution. This imbalance of time will be reflected in the size of the various components. The user interface code is nearly twice as large as the modeling engine code.

4.2.4.1 Needed components

This application requires essentially a monolithic solver. That is, the application is built around a single algorithm, developed by Graves and Willems [43], and this algorithm encompasses the entire model knowledge. This algorithm component is named SIPEng (with a soft *g*). Thus, SIPModel refers usually to the application perceived by the user, but the implementation of SIPModel uses SIPEng. As with the main application, SIPEng has

had three versions. In the first two, it was interwoven into the SIPModel user interface fabric. In the third, SIPEng was implemented as its own COM server.

4.2.4.2 SIPEng solver

SIPEng is significantly more functional than a solver in the framework would need to be, and it pre-dates the framework; nonetheless, it follows the same philosophies promoted by the framework and was a source of inspiration for it. Going beyond the comparatively simple interfaces of a framework solver, the SIPEng library provides substantial COM services to automation controllers, like Visual Basic and Visual Basic for Applications. These services include collections, special data types for each stage and link in a network, and the ability to store a model to and load a model from persistent storage.

4.2.4.3 Cost of using SIPEng

Because SIPModel and SIPEng are custom applications, they are tightly integrated. Whenever the user changes any parameter or property in the graphical model, the SIPModel interface notifies SIPEng of the change, SIPEng recalculates new values as necessary, and the interface updates its display to reflect new cost values. In some cases, these evaluations are lazy—values are marked as "dirty" in the SIPEng structures, and are calculated only when requested by the interface or other parts of the optimization engine.

This integration imposes a slightly higher overhead on the SIPModel client application than would be found in a modeling environment that follows a more traditional build–pre-process–execute–post-process–analyze life cycle. That said, the COM interfaces themselves were very easy to work with, especially given the COM integration in Microsoft Visual C++ 5.0; in most cases, it only required one or two lines of code to retrieve or set the desired properties for any event. Having the solver details encapsulated behind a set of interfaces made it very easy to control.

4.2.4.4 Results

The SIPEng algorithm is 9,100 lines of C++ code. The COM routines are 6,000 lines of C++ code. Had the engine been written as a minimally sufficient, compliant solver within the framework (i.e., without all of the added dispatching capabilities), that six thousand could have been reduced to between two and three thousand lines. By comparison, the SIPModel user-interface that wraps the SIPEng library is nearly 25,000 lines of code.

Both the decision to separate the optimization into a separate module and the decision to wrap it with extensive COM support have proved justified. During development of the user interface, which lagged the development of the SIPEng optimization engine, the optimization engine was debugged with Microsoft Excel. Because they are separate modules, one can be updated independently of the other.

After deployment, a customer asked how much effort would be required to use the algorithm from Excel. They were delighted to learn that the solver already had that capability, and within a day they had developed the macros necessary to drive the solver repetitively with different scenarios queried from a corporate database. Because this capability had been built in at the beginning, the client had instant access and quickly developed a solution to their problem without having to wait for more development and debugging by the MIT developers.

4.2.5 ALCOA

A brief project undertaken by MIT researchers at ALCOA to examine loading heat treat units resulted in a software suite with embedded algorithms that provides an interesting case study for how the framework could have been applied. Within the plate mill of an aluminum production facility in Iowa, the delivered products are flat sheets of aluminum plate, whose gauges (thicknesses) range from an eighth of an inch to over sixteen inches, and whose lengths range from about ten feet to over one hundred feet. Products are distinguished not only by their dimensions but also by their alloy and flow paths that specify different processing steps, such as finished milling or treading. Even though there are potentially thousands of SKUs and hundreds of flow paths, every piece of metal going through the plate mill must be heat treated. In 1996, the plant had three heat treat units, massive horizontal ovens fifteen feet wide and over one hundred feet long. Each unit has a loading and unloading bed, each as long and wide as the oven itself. These units are huge capital investments.

Because of the long batch cycles and limited capacity of the heat treat units, they are the aggregate bottleneck operation for the plate mill[10]. For ALCOA, any increase in utilization at these units directly impacts the bottom-line. ALCOA charged the researchers with the task of examining the opportunity for improvements in utilization at the heat treat units, while taking into consideration the due dates of customer orders.

The problem examined is quite simply expressed. Customers order *pieces* of plate aluminum, specifying width, length, gauge, alloy, and final characteristics such as milling. Customer orders are organized into *lots,* where each lot comprises pieces of metal that all have exactly the same characteristics and flow path; this includes the width, length, gauge, alloy, flow path, and customer. A lot typically contains from one to sixteen pieces. Whenever the processing of a lot is split across multiple batches at a production step, a new lot number is assigned to each split part, in order to track exactly where and when each piece was processed. A batch process step at a heat treat unit is a *load.* Loads comprise pieces from one or more lots. Two pieces may run in the same load if they have the same or compatible alloys and are within a fixed range of gauges. Therefore, not every piece of every lot can be

---

[10] Other processes have less capacity or longer batch cycles, and in some flow paths another operation is the bottleneck for that path, but for the plate mill as a whole, the heat treats are the primary bottleneck.

loaded with every other piece from other lots. Because lots contain pieces of identical alloy and gauge, the lots are partitioned into sets of compatible lots; these sets are called *buckets*. Any two pieces from any lot in the same bucket can be loaded together.

Pieces are laid flat onto the loading bed, and they cannot be stacked. The production time of a load depends on the alloy and gauge of the pieces within the load; thicker metal requires more time to heat treat. Utilization is measured in terms of area, not volume, so lighter, thinner pieces with greater area contribute more to utilization than heavier, thicker pieces with less area. Given a bucket from which to choose pieces, the loading problem becomes a two-dimensional bin-packing problem. Each piece in a lot is assigned a value based upon the amount of bed space it consumes (its contribution to utilization) and the degree to which it is late or early relative to the due date.

Scheduling at the heat treat has traditionally been by hand. Every morning, a scheduler lists the lots that should be processed in the next forty-eight hours, and the production supervisor on the shop floor has freedom to mix and match those lots, depending upon layout constraints and metal availability[11], into loads at the heat treat. ALCOA has a thorough shop-floor database system that can identify the location of every lot in the mill. Each night, a snapshot of this database is stored on off-line servers. Each day in March, 1996, this snapshot was sent to MIT to provide the baseline data set of inventory.

The MIT researchers developed a simple, two-step system to develop a potential daily schedule. The system first walks through each bucket and creates hypothetical loads of all the lots in the bucket, using a bin-packing algorithm. Next, given all of the loads for all of the buckets, the system executes a knapsack algorithm to pick the best twenty-four hours worth of loads. This system was designed to be used as a planning assistance tool, helping the scheduler explore opportunities for deciding which lots should be processed over the next few days, and for how pieces might be loaded onto a bed.

4.2.5.1 Needed components

The ALCOA analysis requires three primary components. These are database access, a two-dimensional bin-packing algorithm, and a knapsack algorithm.

Database access. The data source for the ALCOA program is a single Microsoft Access database that contains many tables for different input factors. The primary input table is a list of all of the metal in inventory over the month of gathered data. The database component must be able to query the proper day and metal configuration in order to determine the list of available, mixable lots to combine in a load. The set of available pieces that can run together is referred to as a bucket of lots.

---

[11] That is, metal at the bottom of a large pile of inventory is not available without help from an expeditor.

Bin-packing algorithm. The first algorithm run against a bucket of lots is a two-dimensional bin-packing. Given the list of lots to load and the size of the bed in which to load them, the algorithm must return a set of loads and the mapping of loads to pieces to lots from which those pieces are selected. This algorithm is run for all possible buckets for any given day. In this manner, every piece in the inventory is mapped to some potential load that could be run during the course of the day.

Knapsack algorithm. The knapsack algorithm has the task of choosing at most twenty-four hours worth of loads from the list of all available loads output by the bin-packing algorithm. This is a normal, one-dimensional knapsack problem.

4.2.5.2 Initial solution

The initial solution involved a two-phase process, much like the Monsanto application in section 4.2.1, page 229. In the first phase, the user launches a custom, graphical client application that displays possible bin-packed loads graphically, and allows the user to specify some bin-packing configuration parameters, such as which bin-packing algorithm to use and the order in which pieces are sorted before being packed. This interface, with a set of possible loads for a certain alloy in the 2.000"–2.500" range, is shown in Figure 4.9.

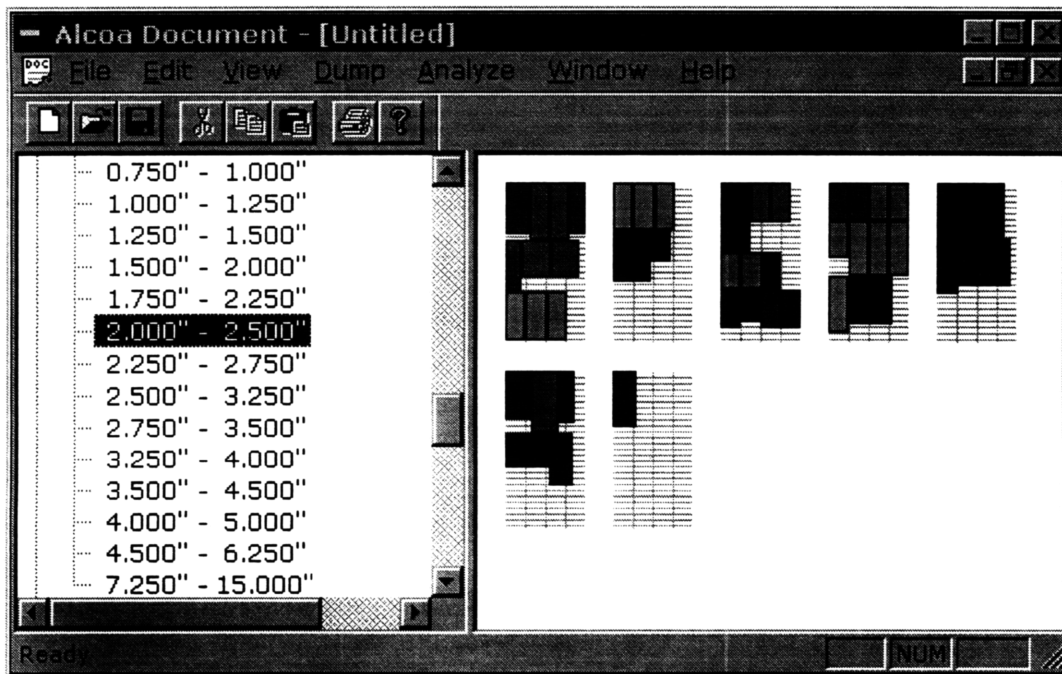


Figure 4.9: Screenshot of initial ALCOA client application

With the configuration parameters set, the user can initiate the first phase of the analysis. The client application steps through all of the possible buckets of lots for a given day, calculates the bin-packing of all of the lots within each bucket, and stores all of the possible

loads back into the original database, with a special time-stamp indicating which analysis run this is. All of the runs are stored in the same table in the database, so that past configuration settings can be examined.

In the second phase of the analysis, the user opens a special Excel workbook that contains the knapsack macros. Using Excel's data import facility, the user imports the list of possible loads output from the first phase, and then executes a knapsack algorithm macro on the imported data. The knapsack macro creates a knapsack solver object (a simple COM server at the time), fills it with the imported data, executes it, and stores the results back into the Excel workbook. The output of the knapsack algorithm is a list of which loads should actually be run in the next twenty-four hours to maximize the value of the day. The value of a load, determined during the first phase, is a measure of the load's utilization as well as the lateness of each piece on the load.

This tool made it relatively easy to examine all thirty days' worth of data that had been acquired from ALCOA using two separate bin-packing algorithms (finite first fit and finite next fit). Further analysis using data of what loads actually ran indicated that the simple on-line bin-packing algorithms, in ideal conditions, performed as well or better than the loads that were actually run about 75% of the time.

Apart from the two-phase, multi-application, manually-oriented nature of the process, the initial solution suffered from numerous design flaws. For instance, it was quite cumbersome to add new bin-packing algorithms to the client application. The two bin-packing algorithms available were C++ classes coded directly into the application executable. The user interface code interacted directly with the classes, and not through an abstract base class, so in order to add new algorithms, the entire client application would need to be recompiled. Furthermore, some custom and arbitrary structures were passed frequently among the objects.

Also, parts of the knapsack phase of the process were hard-coded into the Excel spreadsheets. This included, for instance, the maximum size of the data sets that might be imported into Excel from the outputs of the first phase. Similar constraints were found in the Excel modeling of the M/M/k queueing system in section 4.2.3.2, page 245.

To alleviate these problems, the three required components could be framework components. The RDBElements module, from section 4.2.1.6, page 234, serves as the database access, while two new modules implement the bin-packing and knapsack algorithms.

4.2.5.3 RBinPack module

The RBinPack module comprises two two-dimensional bin-packing algorithms, though it certainly could contain more. The first is finite next fit, a simple algorithm with linear time and constant memory requirements. The second is finite first fit, a better-performing algorithm with quadratic time and linear memory requirements.

Each algorithm is implemented in a solver. Each solver has the same structure, and hence, the same SolverInfo description. In fact, this is an instance where a single SolverInfo description could service every bin-packing algorithm that the ALCOA client can use. In situations such as this, the SolverInfo takes on a role beyond describing any particular solver, but instead becomes a model description—something like a ModelInfo object.

As with the queueing and random variable extensions to the framework, the bin-packing solvers define custom interfaces for more direct access to the solver functionality that is especially designed for the two-dimensional bin-packing problem. So, the bin-packing solvers are both framework compliant but also have direct access interfaces for clients that know they are there. The client has to query each solver for this interface; if the solver supports it, the client can optimize its access to the solver. Otherwise, the client needs to use introspection to discover the structure of the solver.

In implementation, the two bin-packing solvers use a common base-class to handle the COM and framework details. It is therefore simple to add new bin-packing algorithms by simply deriving a new class and overriding the necessary virtual functions that customize the implementation.

4.2.5.4 RKnapAlg solver

The knapsack algorithm, packaged into the RKnapAlg solver, is one of the simplest solvers there is. Its implementation is on par with the shortest-path solver in the RNetOpt module, described in section 4.1.3, page 224. This particular knapsack algorithm solves a 0-1 knapsack problem, so that the output is a vector of True or False values for each item, indicating whether or not the item is placed in the knapsack or not.

4.2.5.5 Framework solution

The framework solution, though not implemented, would be a single application that handles both phases from the original solution. This simplifies the task of the user, who before had to exit the custom application after the first phase in order to execute the second phase from Excel.

Internally, the primary difference would be the use of the framework interfaces to separate the bin-packing algorithms from the client code. That is, the client would use the framework interfaces to interact with the bin-packing algorithms, thus removing the compile and link dependencies that existed between the two in the original application. This will make it easier to incorporate new bin-packing algorithms, an important capability given the simplicity of the original two heuristics.

In fact, the real task of the bin-packing algorithm is to generate a set of loads for all the pieces in a given bucket. A solver could be developed that provides an interactive load generation capability, allowing the user to specify a set of loads from pieces in a bucket. These manually-created loads would then be used in the knapsack algorithm. The solver

would follow the framework interfaces and would also provide its own user interface. While the framework specifies no user-interface interfaces, a number of existing options are likely suitable, including ActiveX controls or a separate application environment. Another solver could be one that wraps the entire problem into a mathematical program, and uses the RLPWrapper or RSML solvers to generate a good feasible solution.

4.2.5.6 Results

Without having implemented the framework version of this application, it is difficult to quantify the productivity and code size improvements that would be possible using the framework. Certainly the database components will simplify development of the client considerably. Nonetheless, the main benefits in this example are qualitative.

Using the framework will make it substantially easier to add new bin-packing algorithms to the system. In the original solution, a new bin-packing algorithm would have to be added to the executable, requiring a new compile-link-debug-ship cycle. Using the framework, the system would be designed to select at run-time among the available, installed bin-packing algorithms[12]. Adding a new bin-packing algorithm would be as simple as installing the executable and registering it. The client code would not need to change. Furthermore, the application could be designed to potentially incorporate other techniques for solving the entire problem, rather than hard-coding the two-phase bin-packing, knapsack algorithm. Without too much effort, using the tools available in the final Monsanto solution, described in section 4.2.1.8, page 237, a math program of the problem could be included as one solution, allowing for comparisons between the optimization and approximation methods.

4.3 Conclusion

This chapter presented a number of solvers and applications to demonstrate the cost and benefits of using the framework to build solvers and solve problems. These are summarized in Table 4.8. While the framework does impose an overhead in packaging solvers, in some cases this overhead is about the same as the overhead might be in developing another form of interface for the solver (see, for instance, the RNetOpt module, section 4.1.3, page 224), and usually the overhead is small compared to the size of the algorithm code itself.

In traditional solution applications, like the Monsanto project (section 4.2.1, page 229), the framework and solvers built for it can provide significant client code reductions. In others, such as FlexCap (section 4.2.2, page 239), solvers built for the framework can be easily reused by different applications in different programming environments.

The next chapter describes the benefits for the core target audiences and some primary issues regarding the framework, using these solvers and applications as examples, and concludes the thesis.

---

[12] Using COM component categories, for instance.

Solver/Application	Section (Page)	Example demonstrates...
ALCOA	4.2.5 (253)	Potential application of the framework to daily scheduling problems.
FlexCap	4.2.2 (239)	Versatility of framework solvers in different programming environments, like VB and C++.
M/M/k queueing model	4.2.3 (244)	The perspective of an analyst using the framework in Excel.
MMQueue solver	4.2.3.5.a (247)	Packaging a queueing engine using the extensions in Appendix A.3, page 311.
Monsanto	4.2.1 (229)	Potential reduction in client-side code from using components and the framework.
RandVar module	4.1.2 (223)	Cost of packaging algorithms into a reusable module. RandVar is used in several applications.
RBinPack module	4.2.5.3 (256)	Packaging multiple solvers with the same structure.
RDBElements module	4.2.1.6 (234)	Way to wrap database access into a solver, unifying solvers and database actions.
RKnapAlg solver	4.2.5.4 (257)	Packaging a simple knapsack algorithm solver.
RLPWrapper solver	4.1.4 (226)	How to wrap an existing LP optimization engine.
RNetOpt module	4.1.3 (224)	Cost of repackaging existing algorithms. Similar or smaller code sizes and wider applicability.
RSML solver	4.2.1.7 (236)	Packaging a modeling language as a solver in the framework.
SIPModel	4.2.4 (250)	Savings in net development time provided by adding support for COM and the framework.

Table 4.8: Summary of solvers and applications from Chapter 4

Intentionally blank, this left page.

Chapter Five
Conclusion

The previous three chapters in turn presented the need, a solution, and its application. The need is for more reusable solver components, at the level of easy to use, executable applications. The solution presented here is a framework for implementing and using solvers and data, built on top of the binary standard COM. Its application demonstrated that the framework can improve reusability and reduce client coding requirements while not overly imposing upon solver developers.

This chapter attempts to explain the *why:* Why should analysts, solver developers and researchers, and modeling environment developers be interested in and adopt the framework for using solvers and modeling environments, implementing solvers, and implementing modeling environments? The chapter begins with a discussion of the benefits generated by the framework for each of these three target audiences. It then describes some of the issues that might limit the framework's applicability or hinder its adoption. This is followed by a listing of potential research areas relatıng to the framework, both to address the issues raised in the preceding section as well as to propose new topics that the framework could tackle. The chapter ends with an overall conclusion of the thesis.

5.1 Benefits of the Framework

Section 1.1, "The Target Audience," page 19, identified three groups of users who should benefit from a standardized, component-based framework. End users—clients, analysts, and applications developers—use the solvers, components, and integrated modeling environments to solve real world problems and build modeling applications. Solver developers and researchers build the individual components that implement specific algorithms or models. Tool developers build the integrated modeling environments, class

libraries, and component services that tie together the individual labors of the solver developers and the integrative needs of the end users.

The framework presented in Chapter Three brings desired benefits to each of these groups, depending on their needs. These benefits are described in the following sections.

5.1.1 For clients, analysts, and application developers

The clients of applied operation research solutions, the analysts who model, create, and validate those solutions, and the applications developers who implement them are perhaps the most important target audience of all. Without them, the solver and tools developers have no reason to produce their components. (Without solver and tools developers, the life of the analyst is just more difficult.) The primary benefits for this audience of a standardized, component-based framework, and of this one in particular, are threefold. First, with the framework it is easier for an end-user to transform a problem model into an implemented application. Second, the framework encourages and increases reuse of personal and third-party components, which decreases development time, costs, and bugs. Third, the framework reduces dependencies on specific modeling environments or solvers, which increases freedom and choice. These are discussed in further detail below.

5.1.1.1 It is easier to transform a model into an application

Whether it is to reduce implementation time in order to concentrate more fully on modeling the problem itself or to mitigate the learning curve for implementation expertise, one of the primary goals of the end user is to simplify the task of implementing solutions. The framework accomplishes this in three ways.

First, the framework reduces the size of the code, and hence development effort, required to implement the custom features of a particular solution. In part, this is a natural result of reusable component technology. The framework adds to this by standardizing the interaction mechanisms for solvers and their data, thereby simplifying the process of adding, changing, and wiring solvers in a solution network. With the addition of modeling environments and tools based on the framework, much of the work is already complete, and the end user only has to specify and create the solvers, wire them up, and provide the data.

Example. The Monsanto application, described in section 4.2.1, page 229, demonstrates the reduction in custom, client code as progressively more and more generic components are added to the problem. ☙

Second, the framework reduces the effort required to use *someone else's* solver. Because solvers in the framework are COM executables, they are already compiled and ready for execution. The end user does not have to learn or understand the third-party solver's programming language, or comprehend its build environment, just to have a usable solver. The executable requirement induces usability onto solvers and other components. Furthermore, because the interactions of solvers and data are standardized, the end user only has to understand and be

able to use a single pattern of interaction—that defined by the framework. So rather than having to deal with separate command-line interfaces, obscure initialization files, batch files, environment variables, a clunky user interface, or arbitrary APIs, the user interacts through a single mechanism.

Example. The network optimization algorithms converted into the RNetOpt module, described in section 4.1.3, page 224, are examples of the simplification of using someone else's solver provided by the framework. Each of the original algorithms was embedded within a stand-alone executable application. An algorithm was controlled by passing the problem in a moderately standard text format via the console input and by receiving the outputs in an entirely non-standard text format via the console output. Any particular algorithm file had primarily three parts: the algorithm itself, data structure management and initialization, and standard input/output and startup/shutdown code. The data structure initialization code was intricately mixed with the I/O code. If an end-user wanted to embed one of these algorithms into another application as a subroutine, it would be necessary to disentangle the data structure code from the I/O code, and to develop an API for interacting with the network code, because it operated in an entirely linear, flowchart-style fashion[1]. By wrapping one of these algorithms into a framework solver, all of these problems vanish. ∞

Finally, the framework enables simple manipulation of complex solution networks. The framework's specification for networks of solvers, perhaps coupled with a suitable modeling environment, make it an easy task to piece together interesting networks with a minimum of effort. The ability to create a network with local control can simplify the client code, because the client does not have to implement a global controller. The standardization of data elements and solver protocols relieves the client of the burden of significant data transformations between solvers.

To conclude, by decreasing required custom client code size, by removing the hassle of non-executable third-party solvers, and by simplifying the process of creating solution networks, the framework reduces the overall effort, in code size and development time, and the overall knowledge level required to build effective operations research solutions.

5.1.1.2 The framework encourages and increases reuse

Reusability of software components can provide many benefits for clients, analysts, and application developers. Aside from the obvious gains in productivity in reduced development times, there are benefits through increased reliability, better consistency, and improved maintainability. Meyer [69] devotes an entire chapter of his tome to studying the benefits as well as the techniques of building reusable software. Two aspects of the framework in particular encourage and increase the potential for reusing solvers.

---

[1] Another option, of course, is to use the algorithm as designed, as a stand-alone executable, and pass data to and receive data from the application by using pipes.

First, because of the ease of using solvers, it is now no longer more cost-effective to implement the same algorithm again and again for different applications. Implementing an algorithm, especially validating it, can be complicated work. Yet until now in many cases re-implementing an algorithm has been a simpler task than trying to reuse someone else's existing implementation. As discussed in the network example in the previous section, current implementations are often too closely integrated into and dependent upon platform-specific operations. Extricating the algorithm implementation from the other source code functionality proves to be more difficult than building the entire solver from scratch to conform to the desired characteristics. In the context of the framework, however, because solvers are executables with well-defined interfaces, using an existing solver is substantially simpler than building the same algorithm in a new solver—even one that does not conform to the framework.

Reuse in this manner does not just improve productivity. Assuming that the solver is a correct implementation, it is more reliable to use the existing solver than to attempt to implement a new one. Quick implementations of algorithms, suitable for the rapid needs of application development, are apt to contain numerical inaccuracies, off-by-one bugs[2], and similar problems. Furthermore, finding a suitable algorithm for a common problem can even be tedious, and wastes the analyst's time.

Example. The RandVar module, described in section 4.1.2, page 223, is an example of a heavily reused set of random variable objects. Various functionality from that module has been used, without modification in any way to the module itself, in the various implementations of the FlexCap application (section 4.2.2, page 239) and in the MMQueue solver and the M/M/k analysis in Excel (section 4.2.3, page 244). ☙

The second aspect of the framework that encourages reusability is the standardization of interfaces and protocols for the interaction of solvers, clients, and data elements. As described in section 2.3.3, page 76, when many components follow a single standard, overall complexity is reduced, and it becomes easier for a developer to use those components. All else being equal, a developer is more likely to choose a component with a familiar interface rather than one with an unfamiliar interface. Also, because two components that solve the same problem with different algorithms are now implementing the same interfaces, it is a simple task to interchange them. This allows the developer to explore alternative solution alternatives with the least possible effort.

5.1.1.3 The framework reduces dependencies

Seemingly simple decisions made early in a solution or development process and often in an ad hoc manner can have long-term effects that outlast any single project and continue to haunt well after everyone has forgotten the circumstances of the original decision. The most common decision of this type is what programming language, modeling environment,

---

[2] Usually, writing loops from 1 to *n* when it should have been 0 to *n* or 1 to *n*-1.

processor architecture, operating system, etc., should be used or targeted for project and application development. The choice can lock in the developer or client for years—the more time and value added to a particular machine or programming language, developing libraries, interfaces, and a knowledge base, the more difficult and expensive it is to change to a different system. In many cases, being forced to switch, because of customer demand or obsolescent technology, is tantamount to starting over.

While particularly true, and most harshly felt, for developers who have built a foundation of code and knowledge in a specific programming language, and then must switch (say, from Pascal to C to Java), this is also true for clients and analysts who have built an equally firm foundation in a modeling environment like GAMS but then have to switch, because their modeling environment does not support the solvers, data capabilities, or modeling interfaces required by their problems or their customers. The analyst has, through acquiring experience and models in a particular modeling environment, established a dependency on that environment. The more extensive the experience, code, and models, the more difficult it is to switch to another environment, especially if much of the code and models have to be rebuilt in the new environment. Just as Pascal functions will not work in C without modifications, many expression of models in modeling environments will not work in other environments without substantial revisions. This can be alleviated to the extent that modeling environments have import and export filters (and programming tools have language converters), but that shifts the dependency to one of import-export relationships.

The framework breaks these dependencies by establishing standards that all solvers, modeling environments, and clients follow. Interoperability is no longer an issue. Modeling environments are separated into graphical tools for building models and solver tools for executing them. The value and differentiation of modeling environments comes not from managing the solvers and solution networks, as that is captured within the framework, but from assisting the modeler in building models and managing data more effectively.

One area of the framework that is particularly promising is the introspection protocol. Coupled with the core services registration facility, the framework standardizes the processes of discovery, enumeration, introspection, and documentation. Any modeling environment or client can enumerate the installed solvers on a machine, query them for their fundamental qualitative and quantitative characteristics, select an appropriate one for application, and access further documentation, via help files or web sites, as necessary. Because the database of solvers resides outside any single modeling environment, all environments have access to all solvers.

Example. Solvers no longer need to be designed to be run within a single environment only—although they could still be optimized for a single environment while still running in many. The M/M/k queueing model from section 4.2.3, page 244, is an example of this. In its final form, the queueing model follows the random variable and queueing system extensions to the framework, so it can operate within any compliant client. On top of this solver component, specialized code is added to simplify the task of computing common values, such as expected waiting time, within Excel, via Excel Add-ins. Using the M/M/k solver from Excel is as simple as using a native function because of the Add-in. The key distinction

between previous systems and the framework is that the Add-in is no longer a requirement for simplified operation, but is now an optimization. Without the Add-in, the solver is still accessible. From a different perspective, if a solver was developed as an Add-in for Excel, it is practically inaccessible from other applications. However, if the solver was developed as a solver in the framework, then it is equally accessible to both Excel and other applications. The developer has the further choice to optimize its use in Excel with an Add-in. ☙

Two further benefits of breaking the dependence upon modeling environments and specific solvers are that clients and analysts can more often use their preferred tools and environments and that it becomes easier to interchange elements within an environment. Because solvers are no longer dependent on import and export filters or similar custom APIs to work in a modeling environment, the analyst's preferred modeling environment can now handle more solvers than it could before. Furthermore, new solvers are automatically accessible from the same modeling environment, without updates to the environment. This extends the life of the modeling environment and of the analyst's knowledge base. Whereas many current modeling environments support only the largest, monolithic math programming optimization engines, with the framework they will support all solvers, whether monolithic or specialized for the smallest of problems.

5.1.2 For solver developers

The developers who implement algorithms, which in the context of this thesis are specifically operations research algorithms, are the primary target of much of the technical detail of the framework. Solver developers make many crucial decisions when implementing an algorithm, such as which machine, operating system, and programming language to target and use. Often, these decisions limit the potential customer base for an implementation. The first benefit of the framework for solver developers is that it expands the customer base instead of shrinking it. Another decision solver developers might have to make is for which modeling environments to customize their implementations. Each environment might require a non-trivial amount of work to implement necessary interfaces, expose the correct functionality, and create a proper installation. The developer then depends on the success of these environments for the success of the solver. The framework breaks this dependency by making the solver equally accessible to all environments that support the framework. Finally, while developing a solver to comply with the framework requires more code in the solver itself, the net code size, integrated over the life of the solver for all of the solver's customers, application developers, and future maintenance of the solver itself, is smaller for solvers in the framework. These three benefits are discussed in more detail below.

5.1.2.1 The framework enlarges the customer base

Choices a solver developer makes during the early design phases of the development cycle, especially the system design choices such as hardware, operating system, and programming languages for implementation, inevitably serve to diminish the customer base. Initially, anyone can use an algorithm. Discounting emulation, only owners of Intel-based systems

can use an executable compiled for Intel processors. Only users of Windows can use Windows applications. Only C++ programmers can use C++ class libraries. Every decision divides the potential customer base into the *can*s and *cannot*s.

Given the initial limiting decisions to use an Intel-Windows combination, using the framework enlarges the potential customer base compared to typical class libraries or source code implementations. COM, the underlying component technology on which the framework is based, is ubiquitous in Windows applications. Since Microsoft began licensing the Visual Basic for Applications environment, and because it freely distributes the VBScript and JScript engines, many Windows applications now embed automation and simple if not complex macro capabilities. VBA and the scripting hosts VBScript and JScript can control COM objects, and therefore they can control framework solvers. Because solvers are COM executables and not source code distributions, any application that can contain COM objects is a potential modeling environment. Solvers are no longer limited to users who know FORTRAN or C, but now can reach analysts and modelers who live and breathe Excel and Visual Basic.

What about Java? While the "Intel-Windows combination" is a minor limitation given the massive installed base of these systems, it is nevertheless still a limitation. It is tempting to think that reworking the entire framework in Java would extend the customer base and applicability to even more analysts. This is partly true for users who crunch their algorithms on UNIX systems. But Java is not yet the elixir operations researchers have been seeking, for several reasons. First, as analyst Peter Coffee [16] notes, Java is not so much "write once, run everywhere" as it is "write once, run anywhere that has JVM and a full set of API classes." Others label it a "write once, test everywhere" capability. The productivity in actually writing Java code compared to C or C++ code is offset by the need to test against different implementations of the virtual machine. Second, Java is still a slow, cumbersome environment in which to solve sufficiently complicated algorithms. Just-in-time compilation helps, but for best results it is necessary to compile to native code, and that removes any multi-platform benefits of Java class files. Finally, Java is not yet sufficiently supported by the most common modeling environments for the small- to medium-sized problems targeted by the framework, Microsoft Excel, Visual Basic, and the like. Given Microsoft's strategy that Java is a programming language and not a platform, it is unlikely that those environments will fully support Java applications unless they have been tweaked to use COM through Microsoft's virtual machine. Therefore, Java just becomes another programming language, like C++, for building COM components.

In the end, the choice of COM does limit solutions to Windows environments (discounting the existing implementations of COM on UNIX), but most of the problems and environments targeted by the framework already have that limitation. For these environments, and for solution developers working within them, the framework does expand the customer base, providing more potential users, and therefore more potential revenue.

5.1.2.2 The framework reduces dependencies

Section 5.1.1.3, above, described some of the dependencies created by clients, analysts, and application developers when they choose to use a particular modeling environment, tool, or programming language, and how the framework removes those dependencies. Solver developers have similar dependencies, with a slightly different emphasis. Whereas clients are dependent upon their choices because of invested time, available capabilities, and a high switching cost, solver developers are dependent upon their implementation choices and targeted modeling environments for their very success.

Developing a solver specifically for a modeling environment requires extra work beyond the function of the solver. This includes adding the necessary interfaces, callbacks, and data structures, and using the appropriate APIs for the modeling environment. This extra code must be developed for each modeling environment with which the solver will operate. The success of the solver depends on the success of the modeling environments it targets. If the developer chooses to target what is essentially a sinking ship, then the solver will go down with it. The developer has to balance the costs of developing for many environments against the mitigation of risk that this brings.

Some environments are even more infuriating: they do not have any plug-in API or any other suitable means for extending the environment with new components or solvers. In these cases the solver developer must rely on the environment developer to specifically add support for the solver into the environment. This requires either sufficient market presence, money, or diplomacy to convince the environment developer to invest the time and development effort to extend the environment in such a manner.

Example. Consider a parallel in word-processors, the import-export file capability. Many word-processors understand several common standards, such a ASCII text files, rich text format (RTF) files, and more recently, HTML files. But the native, often optimal, storage format for word-processors is usually not a standard format. Microsoft Word alone has several different formats for its versions over the years. When the old applications no longer exist but their documents remain (such as when Word 97 replaces the Word 95 application and its document format but not the old files), the documents run the risk of becoming inaccessible. If no current word-processor can read the old application's file formats, the documents are effectively destroyed[3]. Modern word-processors read old word-processors' formats using components known as "import filters." These components know how to translate other file formats into the native format for an application. At the other end, "export filters" can transform the native format into other formats. The Open and Save As dialogs in applications show the existence of these filters, depending on what types of files can be opened or saved.

---

[3] Thus, for long term backup, it is important to backup not just the documents but the applications that can read those documents, and the operating systems that can run those applications, and the hardware that can read the backup medium, and so forth *ad infinitum.*

Developers of applications that have their own file formats and want them to be usable by Microsoft Word have three choices. First, they could license Microsoft's document format, and create documents directly in that format. This is equivalent to using the API or software development kit provided by a modeling environment to develop a solver that conforms to the environment's specifications. Second, they could convince program managers at Microsoft that the new file format is sufficiently important to warrant Microsoft developing a new import filter for the format. This is equivalent to a solver developer convincing a modeling environment developer to build in support for a solver. The cases in which Microsoft or the environment developer would capitulate are comparable too: the file format or solver developer needs to have sufficient market share or money. Finally, they can use a standard format, such as RTF or ASCII text, for communicating with Word. Besides being compatible with Word, the document format would then be compatible with all word-processors and other applications that understand the standard. In current modeling environments, the closest equivalent is a text-based model format like MPS, but MPS is insufficient for solution needs that extend beyond math programming. ☙

The framework introduces a standard of communication, removing the dependencies of custom modeling environment specifications, APIs, and interfaces, and the reliance upon modeling environments for equivalent import and export filters.

5.1.2.3 The framework yields a net savings in development

It is tempting, as a solver developer, to survey the framework, with its many requirements on solvers, and decide that developing a solver for the framework is too complicated and demands too many resources, and that it is better to implement a solver "the old-fashioned way." Indisputably, developing a solver for the framework requires more development time, code, and discipline than not developing for the framework, all else being equal. However, assuming that the solver receives even moderate acceptance by customers, the initial extra investment yields an overall net savings in development time, productivity, and reuse.

It is a maxim that over-engineering early in a process saves time and effort down the road. The questions are how much effort must be spent early, and how much is saved when the rewards are reaped?

The overhead of wrapping an algorithm from scratch, without the assistance of any helper libraries, can range from 350 to 2000 lines of code, depending on the complexity of the algorithm (number of inputs, outputs, parameters, etc.). The use of skeleton base classes and helper components significantly reduces this overhead. Certainly, for calculating averages or other simple functions, 350 lines of code can be a substantial overhead. On the other hand, for sophisticated algorithms this additional code is minimal compared to the development of the core of the algorithm. Furthermore, there are several ways to package algorithms into framework solvers, some of which provide significant improvements of the overhead required and hide the framework details from the solver developer. See section 4.1.1, page 220, for more details.

Example. The LP Wrapper solver, described in section 4.1.4, page 226, demonstrates that around one thousand lines of code are sufficient to wrap a core subset of the CPLEX Callable Library. These one thousand lines enabled many other applications, such as Excel, Visual Basic, Active Server Pages (and hence, web sites), to use the CPLEX optimization engine with very little effort of their own. ☙

Example. The RandVar module, described in section 4.1.2, page 223, and the RNetOpt module, described in section 4.1.3, page 224, are two examples of libraries that repackaged existing algorithm implementations within the context of the framework. Wrapping the RandVar library required several hundred lines of framework-specific code, while the RNetOpt library required the same amount of new code but significantly more modification of the existing code. Those modifications were due to the design differences between the original implementation, which had a strict program flow with console input and output, and the framework-compliant version. Both of these libraries have been used by applications developers and clients in several applications that require random variable calculations and sampling or network optimization. These applications, built in Excel, Visual Basic, and C++ have easily incorporated the libraries with only a few lines of code. ☙

Example. The SIPModel application, described in section 4.2.4, page 250, contains one particularly noteworthy example of a net development savings. In the most recent release, the optimization engine, known as SIPEng, was separated into a module distinct from the user interface application. Going beyond the comparatively simple interfaces of a framework solver, the SIPEng library provides substantial COM services to automation controllers, like Visual Basic and Visual Basic for Applications. These services include collections, special data types for each stage and link in a network, and the ability to persist a model to storage. Both the decision to separate the optimization into a separate module and the decision to wrap it with extensive COM support have proved justified. During development of the user interface, which lagged the development of the SIPEng optimization engine, the optimization engine was debugged with Microsoft Excel. After deployment, a customer asked how much effort would be required to use the algorithm from Excel. They were delighted to learn that the solver already had that capability, and within a day they had developed the macros necessary to drive the solver repetitively with different scenarios queried from a corporate database. Because this capability had been built in at the beginning, the client had instant access and quickly developed a solution to their problem without having to wait for more development and debugging by the MIT developers. ☙

5.1.3 For modeling environment developers

Developers of modeling environments add value to solutions by simplifying the modeling process, by providing a unifying, integrated interface to data, models, and solvers, and by hiding the details of solver invocation. The framework benefits developers of modeling environments by standardizing the implementation language of solvers, making it easier to develop a modeling environment that can use many solvers. The framework simplifies through standardization the invocation process, and makes building networks of solvers easier. The time saved by the framework on the back-end of the modeling environment can

be devoted to the front-end, providing a richer and more marketable experience for the analyst. These benefits are discussed in more detail below.

5.1.3.1 It is easier to develop an environment that uses more solvers

Modeling environments bring an analyst and a solver together into a workspace, mapping the user's needs onto the solver's capabilities. Without the analyst or solver, the environment is worthless. For a modeling environment to be successful, it has to support the solvers that analysts want to use. When solvers support different invocation patterns, adding support for solvers is a cumulative effort without economies of scale. Each solver requires its own supporting code. If the solver interface or semantics change during an upgrade, the modeling environment needs to be re-developed and re-deployed with support for the new version[4]. By using the framework, the modeling environment developer adds support for a single invocation specification, and the environment automatically supports every solver that complies with the framework. Furthermore, by using the framework instead of a custom plug-in specification for the environment, it is much easier for solver developers to build solvers usable within the modeling environment. That is, if a solver developer has to choose between adding code to comply with a framework supported by several modeling environments as well as existing business productivity applications or adding code to comply with a single modeling environment's plug-in specification, the first choice is clearly preferable to the second. The modeling environment gains by supporting more solvers, because the analyst can then spend more time within the environment to solve broader problems, rather than needing to use other environments or applications to utilize particular solvers[5].

Modeling environments do not only invoke solvers. They also assist the analyst in selecting models and solvers, hooking them together, and assigning data inputs and storing data outputs. A modeling environment that uses the framework can reduce its development effort by leveraging the core services to provide common capabilities while supporting the framework. For instance, the environment can use the core services' solver registration and enumeration facilities to determine which solvers are installed on a system. With the list of solvers, the environment can use the introspection protocol to show the analyst the qualitative capabilities and structure of each solver in a solver browser. Because all framework solvers support introspection, the environment can provide a unified view of every solver in one location. Users appreciate such uniformity, because it reduces their learning curve. The networking specification, discussed in the next section, simplifies the creation of solution networks and of assigning and retrieving inputs and outputs.

---

[4] This is true, for instance, with AMPL and CPLEX; a particular version of AMPL works with CPLEX 4.0 but not CPLEX 5.0, so it is necessary to retain a copy of CPLEX 4.0 even after upgrading to CPLEX 5.0 in order to use CPLEX with AMPL.

[5] This is similar to the goal of many web sites: add sufficient content and capabilities so that the web surfer never has to leave the site. The site becomes the anchor point, acquiring substantial visibility and name recognition, as well as increased potential for ad revenues because of long-term viewers.

5.1.3.2 It is easier to build distributed solution networks

One aspect of building a modeling environment that can be troublesome, and for which the framework provides a solution, is the construction of solution networks, especially distributed networks where solvers reside on different systems. The modeling environment can use the networking protocol in section 3.8, page 178, to create networks whose stages embody local control. This can simplify the creation of the environment. Furthermore, the distributed capabilities of COM (i.e., DCOM) make it easy, almost transparent, to distribute solvers across a network of computers. The local control network protocol thereby makes it easy to parallelize multi-stage solution networks across multiple machines, where each solver will begin executing once its inputs are available. The modeling environment might even be designed to create the network, start the solution process, and then shut down, leaving a minimal implementation of the progress notification interface viable to receive the final solver completion notification. Upon receipt of this notification, the environment could reawaken or send an email to the user.

5.2 Issues

This section addresses thirteen of the primary issues concerning the framework. Some issues concern areas where the framework does not fulfill the requirements of Chapter 2. Others relate to the implementation of solvers, environments, and applications that use the framework. For some, potential future research opportunities are discussed.

1. The framework is limited to platforms that support COM.

The framework builds a specification on top of COM, and it relies on the COM run-time services for object creation, marshaling, component categories, memory allocation, and more. The COM specification itself is platform-independent. The platform-dependency arises because of the few systems that support COM. Some version of COM is present in all of Microsoft's 32-bit operating systems[6]. DCOM, which enables machine-to-machine communications, is available only on Windows NT 4.0, Windows 95 with Internet Explorer 4.0, and Windows 98. The core of DCOM is also available on a number of UNIX systems, as Microsoft has been developing ports of important subsets of COM and DCOM in an effort to make COM more widespread.

Three factors mitigate the COM restriction imposed by the framework.

First, as argued in sections 2.3.2, page 75, and 3.3, page 116, limiting a framework to Windows and COM is reasonable because of the popularity of 32-bit Windows operating systems in solving the small- to medium-sized problems targeted by this thesis. That is, most of the applications targeted by the framework will be running on Windows regardless, and

[6] Windows 95, Windows 98, Windows NT 3.51, Windows NT 4.0, and Windows 3.1x with Win32s.

environments like Excel and Visual Basic are built around COM, so it turns out not to be such a great limitation.

Second, COM objects on Windows systems are becoming more and more accessible to other, non-Windows systems through the growing support for COM-CORBA bridges and mappings. Almost all systems that do not support Windows support CORBA, a "competing" object model. Some companies that produce CORBA ORBs, like IONA Technologies, Inc., and Digital Equipment Corporation, have either licensed or pledged support for COM in their products in the form of COM-CORBA bridges [39, 22]. These bridges translate calls from CORBA-compliant objects into calls to COM objects and vice versa. Microsoft and the OMG, the overseer of CORBA, have been working on a COM-CORBA mapping, as well, which standardizes the communications between CORBA and COM objects. The ideal end result, which is quite probable, is that COM objects and CORBA objects will be able to communicate seamlessly without being aware of each other's object model. Developers will then be able to work in whichever object model they prefer. In particular, the framework can still be based on COM yet be accessible to both COM and CORBA clients.

Third, in the absence of any COM-CORBA bridge or mappings, the framework could be specified as an all-CORBA solution. This would create two separate frameworks, one in the COM world and one in the CORBA world, but this would greatly ease a transition in the future when the two are interoperable. Certainly, none of the aspects of the framework transgress any CORBA principles or capabilities, so mapping the framework into CORBA is largely a technical exercise.

Basing the framework on COM is truly a value-added proposition, as it enables the framework to leverage COM's extensive object services. Such activities are horizontal to the vertical framework effort, and hence it is a great savings to not re-specify such services when they are already available in the form of COM. The limitation that this imposes is a small price to pay given the popularity of Windows systems for the targeted application set and the almost assured future interoperability between COM and CORBA.

2. **Solvers must be compiled into binaries for each intended processor and operating system.**

Section 2.3.1, page 73, specified that framework solvers should be executable objects, rather than simply source code files. This implies that solvers must be binary executables, which have to be compiled for each particular processor and operating system. Such a restriction is mitigated by emulators for other processors and operating systems, such as FX!32 for DEC Alpha processors and Insignia Solutions' SoftWindows 95 for MacOS.

Java is one potential tool for circumventing the multiple-executable syndrome. With Java, a single class file is sufficient for all systems that have a suitable Java Virtual Machine that can interpret that class file. As Java adopts more CORBA capabilities, or as Microsoft enhances its Windows and Macintosh implementations of the Java VM to incorporate COM, Java will become a viable language and platform for creating and using solvers and driving other

applications. For now, Java is a suitable self-contained environment, when requirements do not dictate the need for other applications that do not support or understand Java classes.

3. **Because the interfaces use types that are not automation-compatible, most of the interfaces cannot be used directly from automation environments.**

Most of the interfaces defined by the framework are designed for C++ semantics, for several reasons. First and foremost, the majority of standard COM interfaces follow the same philosophy. Second, modeling environments and most solvers will typically be implemented in C++ at this time. Nevertheless, Visual Basic and the Visual Basic for Application clients like Microsoft Excel and Word, and Autodesk AutoCAD are common solution development tools, and it is desirable that COM solutions work in these environments. Unfortunately, as mentioned before (see section 3.3, page 116), Visual Basic and its derivatives understand only a subset of the types that C++ understands; these types are the automation-compatible types. Some of the types used by the framework, such as UINT and REFIID, are not automation-compatible, and therefore interfaces that use those types are not available to Visual Basic or VBA without appropriate kludges.

Each object in the framework should support IDispatch, the primary automation interface. A single implementation of IDispatch should contain suitable methods for all of the interfaces implemented by the object; it is, essentially, a "garbage can" for everything the object can do. Because different objects of the same type (two different solvers, for instance) might support different sets of required and optional interfaces, a single dispinterface (IDispatch plus specification of properties and methods) is not suitable for all solvers. Therefore, each object must implement its own IDispatch, and the clients that must use IDispatch for automation-compatible types or because they do not understand custom interfaces must discover at run-time the exact calling mechanism for those objects. While not a travesty, this is not an ideal condition.

One remedy is to define a new set of framework interfaces that are all automation-compatible. In some cases, this will incur a performance penalty when customized types must be wrapped into VARIANTs, SAFEARRAYs, or their own custom objects. A set of automation-compatible interfaces, even if they do not derive from IDispatch, would be entirely usable from Visual Basic without any extra helper objects. VBScript and other "dumb" hosting environments still require a single IDispatch implementation on each object, however.

A second remedy is to add significant support, in the form of helper objects, development wizards, and frameworks, for using the framework from automation-compatible environments. Even though Visual Basic might not be able to implement all of the framework interfaces, helper objects can simplify the creation of solvers within Visual Basic by implementing the non-compatible interfaces and delegating to Visual Basic when necessary. This is discussed further in section 4.1.1.3, page 221.

The final remedy, less appealing than the others, is to wait for Microsoft to release the specifications and eventual implementation of COM+, the successor to COM, that is scheduled to ship with Windows NT 5.0, and to perform all development and analysis with NT 5.0 and COM+. At this time, the ambiguous feature set and unlikely availability of COM+ on other Microsoft platforms makes this a questionable strategy.

4. **The framework does not support dimension and typing information.**

   Section 2.3.6, page 85, described the need for dimension and typing support, especially in enterprise systems. While none of the interfaces or protocols presented in the framework specifically address dimensions or typing, adding a dimension and typing protocol to the framework would not be too difficult. Because COM development is interface-based programming, new interfaces could be defined to provide dimension and typing support on each of the primary objects in the framework. Data elements, solvers, and the network components would each implement additional interfaces that provide access to dimension manipulation, conversion, analysis, and validation.

   For example, a data element might support an interface to retrieve the dimensionality and type of its contained data. Another interface might provide the capability to change the dimensionality of the data element, under appropriate conversion factors. A solver might support an interface that has methods for validating the dimensional consistency of its inputs. The input and output interfaces could be extended to support dimensionality and typing, allowing clients to query the dimension or type of outputs, for instance. The core services would include a new set of registration and enumeration services, for maintaining a database of dimension manipulators[7].

   The real challenge in adding dimension and typing support to the framework is in specifying the relationships among dimension systems and dimensions and types. A dimension manipulator converts or validates dimensions, but how do two manipulators work together to convert dimensions between each other? Converting from one dimension to another is essentially a network traversal process, where each node is a dimension and each arc is the conversion function from one dimension to another equivalent dimension. If a path exists from one node to another in the network, then the two dimensions represented by the nodes are equivalent, and the conversion function is the composition of the conversion functions along the path. The collection of nodes is theoretically unbounded, so patterns must be identified to simplify constructs. These patterns include dimension multiplication, division, and exponentiation. The collection of available arcs represents the knowledge of dimension conversion. If a developer knows it is possible to convert from Fahrenheit to Celsius but does not know the equation, the arc from Fahrenheit to Celsius is effectively non-existent. A dimension manipulator defines a set of arcs in the network; for example, a dimension manipulator that converts temperatures might connect all known temperature dimensions in a complete sub-network. The union of all of the arcs from all dimension manipulators on a computer represents the complete dimensional knowledge available for the computer.

---

[7] See page 85 for more information on dimension manipulators.

A core services function could have the task "convert the value x from dimension d_1 to dimension d_2." The challenge of this function is to determine the existence of a path from d_1 to d_2 in the network of dimensions. Another function might be to "reduce a dimension expression," such as reducing (miles per hour) * (seconds) into just miles. This requires more than just calculating the conversion factor from hours to seconds; it requires discovering that such a conversion exists at all. This can be discovered and calculated over the network of dimensions nodes and conversion arcs, but how best to optimize it or implement it remains unclear.

The challenge of any framework extension in this area, then, is to develop a sufficiently powerful conversion mechanism that can handle complicated dimension and type conversions, reductions, and manipulations quickly and reliably, while making it flexible enough to handle new dimensions, types, and manipulators.

5. **The framework does not specifically support testing and validation.**

Section 2.3.7, page 88, introduced the need for coherent and simple testing and validation capabilities. In particular, an ideal testing and validation framework would make it easy to interchange different implemented algorithms (solvers) in a testing environment. The framework does not included specific support for testing and validation.

With the most optimistic assumption that all algorithms to be compared or tested have been developed as solvers within this framework, it is trivial to develop a testing client that repeatedly solves the same problem with the different solvers. In this manner, the testing and validation problem is solved easily. Less optimistically, it might be necessary to develop special solver sites or mappings that can interface with proprietary or custom input and output interfaces of solvers that do not conform.

An extension to the framework that might benefit certain testing operations would be the introduction of a performance-related interface for solvers. The parameterization interface, IRSolverParameters, is intended primarily as a way to configure the behavior of the solver. However, it could also be used to retrieve outputs from the solver, such as the active basis, the number of cuts or iterations, or the elapsed CPU time. It might be better, however, to add another interface, something like IRSolverStatistics, that provides performance outputs. Clients could then query this interface for the various testing results.

6. **The framework does not provide specialized support for notifications and CBT hooks.**

Notifications are one way that a server can talk back to a client, telling the client about its progress or state. Section 3.7.1, page 169, described progress update notifications in the framework, which is a specific, well-defined notification mechanism. The need for a more generic notification mechanism was introduced in section 2.4.3 on page 98.

Generic notifications are distinguished by two important factors: First, different solvers (or data sources, solver sites, etc.) might generate different notifications. While every solver has

some concept of progress, not every solver has iterations, optimality gaps, or feasible solutions. Therefore, solvers need to be able to send only those notifications that are relevant for their behavior. Second, clients need a general way to register with an object to receive notifications. Rather than requiring every client to understand how to register with every object, which introduces too many interface dependencies, a single set of interfaces should suffice to establish notification advise connections between server and client. The IRSolver::SolveAdvise and IRSolver::SolveUnadvise methods are an example of this—a standard method of registering to receive progress and completion notifications.

Fortunately, the COM specification already contains a generic notification mechanism in the form of connection points[8]. Connection points are simple yet powerful. Suppose a client wants to receive notifications of a certain kind from a server. These notifications are defined by some interface, often a dispatch interface, say IMyNotificationsDisp. This is an interface that is implemented by a sub-object on the client. The client first acquires the IConnectionPointContainer interface of the server, via a QueryInterface from another pointer. With this interface, the client queries the server if it supports notifications through the IMyNotificationsDisp interface. If it does, the connection point container on the server establishes an advise connection, and whenever the server sends out notifications of the requested type, it will send them to the client through the IMyNotificationsDisp interface.

How does the client know that the server will support the IMyNotificationsDisp interface for notifications? Usually, the server will indicate this in its coclass type library definition, by declaring IMyNotificationsDisp as a [source] interface. The server might also implement IProvideClassInfo2, a standard COM interface that could provide this information as well.

ActiveX controls use connection points, as does Visual Basic (the "with events" keywords use connection points for implementation), so they make an excellent mechanism for generic notifications within the COM-based framework.

As for computer-based training (CBT) hooks, discussed in section 2.3.8 on page 89, the connection point mechanism for notifications is also satisfactory for CBT hooks. Some distinction might be made in the specification of notification and CBT hook interfaces between regular notifications and CBT hooks, as many clients will not be interested in the latter at all.

7. The framework provides simplified solver selection capabilities, but no model selection or intelligent solver selection capabilities.

In providing a minimalist solver registration and enumeration capability, through the SolverRegistrar component and its IRSolverRegistration and IRSolverEnumeration interfaces, the framework ignores an entire field of research in knowledge-based model selection and solver selection. Much of this work proposes various knowledge-based database schemes,

---

[8] For details on connection points and connection point containers, see Brockschmidt [11].

hierarchical model and solver relationships, or selection criteria to help a user to select a model to match a problem and a solver to solve a model. The framework's approach is to ignore the notions of problems and models, and focus on solvers. Furthermore, all solvers exist on a flat namespace; that is, the solvers are all members of a single set of available solvers. This relationship is expressed in the IRSolverEnumeration interface, which simply enumerates all installed solvers on a system without regard to their purpose or the models they solve. The philosophy of the framework is that more a sophisticated structure for solvers and models can be imposed upon the flat solver namespace.

Is this a sufficient solution? Most likely, too much has been left unspecified. In characterizing the dimension and typing problem, the thesis describes dimension manipulator objects, which are small components whose only purpose is to convert dimensions and types for other objects. In a similar vein, the solver selection problem might specify *model selectors* and *solver selectors,* components that manage model and solver selection systems. Each selector might be based on a particular system developed by a team of researchers. The client could then choose a selection system, and the environment would execute functionality on the selector for that system in order to select a model or solver.

The key issues in establishing such a system for selection are standardizing the model selector and solver selector interfaces and protocols, and specifying how the installation of a new solver is propagated to all of the model and solver selectors on a system, so that they can appropriately incorporate the new solver into their particular selection scheme.

Standardizing the selector interfaces would not be too difficult, as the selector has a simple directive: return a solver identifier (a GUID or CLSID) for the solver selector, or a model identifier (some GUID, likely) for the model selector. Specifying an installation notification system is a challenging task, as the solver, at installation, needs to be able to announce what models and problems it can solve, and traversing the networks of solvable models, sub-problems, and related models is at the core of the research question.

8. The framework provides a limited data flow specification.

The data flow specification presented in section 3.4, page 118, is truly prototypical. It is a minimally sufficient specification for the needs of the remainder of the framework. In intensive database environments, or in environments that require more attention to the dynamic, non-deterministic, multi-user nature of most data sources, the existing data flow specification is inadequate. This is clearly an area where database expertise is required to establish a suitable, practical standard for handling multi-dimensional objects in the framework. One solution might be to use the OLE DB for OLAP specification [72], which itself is built on top of COM.

One of the benefits of COM's use of GUIDs for uniquely identifying interfaces is that anyone can define an interface, give it a GUID, and declare it to be a standard. Whether anyone actually uses that interface is another matter entirely. While all of the interfaces in the framework form just such a "standard," the data element interfaces in particular are worthy

of note. These interfaces are clearly a crutch, a solution to a complex problem simplified for the sake of exposition and to give the solver specification something on which to work. Fortunately, none of the solver interfaces specifically use the IRDataElement interface, thereby making it easy in the future to adopt a new data specification without redesigning the solver interfaces.

9. **The framework does not specify whether or how real-time directed graphs and operational solutions work within the networking protocols.**

The discussion of solution archetypes in section 2.1.3, page 69, limited the focus of the framework to solutions that do not have a real-time directed graph architecture or an operational or real-time execution periodicity. This limitation allowed significant simplifications in the data flow specification (section 3.4, page 118) and in the networking specification (section 3.8, page 178). In particular, because no objects need to respond to real-time, non-deterministic events, their behaviors can be well-characterized.

The domain analysis of section 2.1 argued that even with this limitation, the framework still could apply to a significant set of applied operations research solutions, especially those of the small- to medium-size that run on Windows and are specifically targeted. Nevertheless, a future research area is extending the framework to account for real-time activity, and to enable the use of solvers in operational environments.

10. **In solution networks, the framework only manages data and simple control flows. By default, flows are "lockstep flows;" details for managing other flow types are not specified.**

In the taxonomy of flow dependencies characterized by Dellarocas [20], a *lockstep flow* is described as one "where there must exist tight synchronization between producers and users of resources. All resources produced must be used and no resource can be produced until all previous resources have been used by all designated users." This is precisely the model adopted by data flows in section 3.4, page 118, within the framework.

It is just one type of flow, however, and Dellarocas describes several others, all of which have a place in the more complex solution architectures. These include *persistent flows,* where for example a solver is executed once and its output is then stored persistently and used many times by another solver, and *transient flows,* where an upstream solver might be executed many times while a downstream solver is executed much less frequently, and the downstream solver uses only the most recently available upstream output. Another example might be a case where an upstream solver produces outputs constantly, while a downstream stage aggregates those outputs into a single input.

It is probable that most of these flow types can be captured by mappings, introduced in section 3.8.6, page 199. Mappings are the essence of links on which data flows in the solution network, and are therefore the likely recipient of per-flow functionality. Section 3.8.6.4, page 203, hinted at the versatility of mappings, describing aggregating or scripting mappings. Other mappings could be designed to specifically handle the other types of

resource flows, such as persistent flows and transient flows. An area for further research is in identifying what mappings are necessary to capture the most interesting solution architectures, and how those mappings interact in complicated networks.

11. There is a non-trivial implementation overhead to add framework support to a solver.

In implementing an algorithm, adding support for any new feature requires more code. As demonstrated by the solvers in section 4.1, starting on page 220, the overhead imposed by the framework can range from a few hundred to a few thousand lines, depending upon the level of support and complexity of the inputs, outputs, and parameters of the algorithm.

When beginning implementation of an algorithm, a developer must ask, is it worth it to develop this solver for the framework? Without a class library or solver development environment, the overhead to do so might prove too great. Hopefully, however, the benefits outlined in section 5.1.2, page 266, as well as elsewhere in this thesis, are compelling enough to convince the developer that the overhead is a small price to pay for the enlarged potential customer market and usability within mainstream development environments.

Clearly an area for future work is in minimizing this overhead, through class libraries, new environments, and the like, as described in section 4.1.1, page 220. Besides minimizing the overhead, the existence of popular modeling environments that support the framework will increase the return on coding investment.

12. Developing for the framework requires knowing COM and the framework interfaces.

In its current form, the framework relies directly on raw aspects of COM. Throughout there is mention of IUnknown, reference counting, CoCreateInstance, IDispatch and automation, and other COM concepts and terminology. Obviously, discussions that include these COM-isms assume that they are at least recognizable if not utterly familiar; such is the nature of the technical details of vertical extensions to COM or any other object model.

Therefore, developing a solver that complies with the framework or a client that leverages the framework using current tools (that is, without solver development environments or suitable modeling environments) requires that the programmer understand COM enough to create and use COM interfaces and components. In terms of complexity and learning curves, such a requirement is similar to requiring that the developer know C++. Object models are complicated entities; understandably so, given the problems they address.

Don Box, a leading guru on COM, describes his learning experience with COM:

> *It took me roughly six months before I felt I understood anything about COM. During this initial six-month period of working with COM, I could successfully write COM programs and almost explain why they worked. However, I had no organic understanding of why the COM programming model was the way it was. Fortunately, one day...I had an intense epiphany and at once COM seemed obvious to me...I understood the primary motivating factors behind*

COM. From this, it became clear how to apply the programming model to everyday programming problems. Many other developers have related similar experiences to me. (Box [9], page xix)

At the heart of COM is a programming discipline, a programming way-of-life. Understanding the technology, the function calls, and the interfaces comes later, on a just-in-time basis. The book quoted above, Box [9], provides an excellent introduction to the technical aspects of COM, while Chappell [15] provides an equally excellent introduction to the philosophical aspects. Box's book is geared towards C++ programmers, while Chappell's is accessible to anyone who uses Windows.

As more class libraries, development environments, modeling environments, helper objects, and other tools support the framework, the requirements on any one developer or quantitative analyst will diminish. For example, quantitative analysts who live in Excel do not need to understand reference counting, as that is hidden by the Visual Basic layer. In an integrated modeling environment, an analyst will not even need to know what COM is to use and network solvers. In a full-featured solver development environment, the same might eventually be true for developing solvers.

13. The framework requires a critical mass of buy-in and support to achieve true value.

Indisputably, the return on investment for a developer to add framework support to a solver that is never used is zero. The ROI for adding COM support is certainly not zero, as it is COM that enables solvers to be used from existing, mainstream applications like Excel and Visual Basic. The framework adds on top of COM a set of custom interfaces focused on applied operations research solutions. Primarily, it is this specialization that must be justified, and that is the topic of section 5.1.

To achieve true value, the framework—any framework—must attain a critical mass level of support, from the solver community, the main modeling environments, and the many clients. Ideally, a popular solver or modeling environment would support the framework or another open standard, providing a "killer application" for the framework or standard. Until that happens, support must be acquired solver by solver, application by application.

This framework has sufficient issues, such as an inadequate data flow specification, to keep it from becoming widely adopted. Nevertheless, it provides a foundation upon which a truly useful and utilized framework can be built.

5.3 For Future Research

The previous section indicated the need for further research in some areas, such as a more robust data specification, specifications for dimension and typing, testing and validation, and CBT hooks, and an analysis of more complicated solution types. There are certainly many additions and enhancements to the framework that are possible, and some of these are discussed here.

First and foremost would be the creation of a graphical modeling environment that uses the framework to hook solvers together. This environment would allow an analyst to describe a network of solvers, data sources, and their relationships as a set of visual nodes and arcs, as pictured in Figure 5.1. This environment would demonstrate the feasibility of using the framework to catalog, browse, network, and control various solvers installed on a computer.

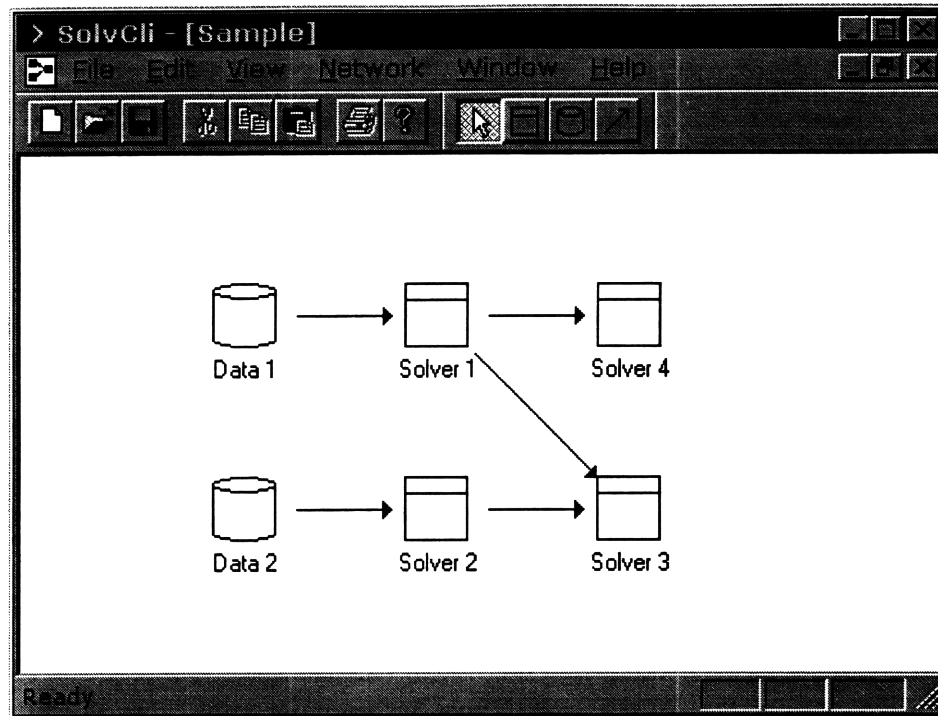


Figure 5.1: Screenshot of modeling environment solution network

Another important area of future work is the evangelism of the framework, or at least the primary ideas of the framework. As stated in the previous section, to be truly successful, the framework would require significant support from third-party vendors, ideally the creators of the most popular solvers. The framework should likely be re-formulated (or extended) under supervision of an experienced standards or organizational committee with input from the major solver and environment developers and users. Such an effort would elevate the framework or its progeny to true standards status.

Some other areas of potential research include:

- *Allowing sets to be inputs to and outputs from a solver.* Currently, sets are used only in determining mappings from set values into indices for dimensions and data elements. Some problems might need to manipulate sets directly as inputs and outputs. This can be approximated by using a vector of boolean values to simulate membership in a set, but another solution is to simply output a set object containing the items. The facility

location problem is one example where the output might be more meaningfully rendered as a set.

- *Adding solver performance statistics.* A useful addition would be a specific interface or group of interfaces that solvers could implement to provide performance characteristics, such as elapsed CPU time, number of solutions tried, number of iterations executed, optimality gap, etc. The progress update interfaces presented in section 3.7, page 169, provide only for remaining time or percentage, but do not provide the exact numbers that might be necessary for performance analysis and tuning.

- *Adding a dimension and typing specification.* The need for interfaces for dimension and typing is discussed in the previous section, item number 4, page 275.

- *Developing automation-compatible interfaces and helper objects for Visual Basic and scripting environments.* The need for low-level supporting COM objects to make it easier to use the framework from Visual Basic, VBScript, and other automation-based environments is discussed in the previous section, item number 3, page 274.

- *Adding solver and model selection interfaces.* The need and some possible directions for adding specific solver and model selection interfaces and component helpers is discussed in the previous section, item number 7, page 277.

- *Developing a robust data flow specification.* The need for a more substantial, useful data flow specification is discussed in the previous section, item number 8, page 278. This is an area that clearly requires significant data expertise from industry partners.

- *Standardizing solution network persistence.* Currently, it is up to each client individually to be able to reconstruct a solution network. One possible feature would be a standardized persistence mechanism for storing the layout and configuration of solvers, data sources, and their links in a solution network. This would enable different environments to share networks, and would make it easy to pause and store an entire network execution.

- *Supporting more complex solution networks.* The need to be able to handle more complex solution network archetypes and flow types more complex or different than simple resource and control flows is discussed in the previous section, item numbers 9 and 10, page 279.

- *Examining ways to reduce implementation overhead.* Implementing a solver to support the framework requires a non-trivial overhead, as described in the previous section, item number 11, page 280. Different ways of packaging a solver are discussed in section 4.1.1, page 220, and an area of research and development is creating ways to simplify solver creation by reducing the overhead.

5.4 Conclusion

Are the benefits of the framework and its potential for further research sufficient to make up for the issues previously raised? This is the proverbial $64k question. In its current form, the framework requires more work, particularly with the data flow specification, and it requires support from vendors of solvers and environments to achieve true success. The framework should be viewed as a first step toward a standard for interaction, rather than the final word. Nevertheless, even with its current shortcomings, there are some potent ideas that perhaps represent the true contribution.

First is the nature of interface-based specifications. The framework is a specification of interfaces between objects. One object implements and exposes many interfaces, and clients know objects only through those interfaces. Interface-based programming is the philosophy of most modern object models, like COM and CORBA, so clearly this is nothing new. Furthermore, there has been some research to propose interfaces for solvers at the level of modeling languages (see Ramirez, Ching, and St. Louis [86] and Eck, Philippakis, and Ramirez [26]). However, the framework presented in Chapter Three might be the first proposal of an interface-based specification for the actual implementation of solvers and environments.

Interface-based programming provides a near-optimal encapsulation of objects. But three additional features of the COM standard are perhaps more important, especially early in the evolutionary cycle of the framework. First is the ability of an object to support multiple interfaces. As the standard evolves, new interfaces are defined that support more features or resolve newly discovered problems. Subsequent versions of solvers and environments support the new interfaces as well as the old ones. Everything remains backwards-compatible for those situations where older components use the older interfaces. The widespread acceptance and use of COM is a testament to multiple interfaces. Over the years, as functionality has been added to Windows, interfaces to use that functionality have been added to COM.

The second feature is that interfaces are immutable. That is, once defined, an interface cannot change. New features in the form of new properties or methods require a new interface. Immutable interfaces allow a client or server to guarantee behavior of an interface regardless of when the objects were implemented or what version number they have. So, as the framework evolves, new interfaces will be defined, but the old ones remain, ensuring that older solvers and environments continue to work in the framework well into the future.

The third feature is the use of globally unique identifiers (GUIDs) for identifying almost every unique item in the framework. All of the interfaces, objects, property identifiers, solvers, etc., have their own statistically unique identifier, a 128-bit GUID. Because of potential name violations from different developers or because of language translation issues, elements are not identified by names. Instead, each element is identified uniquely by its GUID. Textual labels attached to interfaces, solvers, and the like, are contextual; the GUID is universal. The benefit for the framework here is that anyone can add new

interfaces to the framework, just as with COM, by defining an interface and assigning it a new GUID. Suddenly, it is part of the standard, to the extent that anyone uses the interface. The random variable and queueing system interfaces defined in Appendix A.3, page 311, are examples of this capability. The framework is a fluid entity.

The three features described above are features of COM that the framework has co-opted. They do not represent new contributions to the field of software. However, their application to the particular domain of applied operations research solutions is new. Interface-based specifications represent a different and, for this field, new way of thinking about how to implement solvers and modeling environments, at the low level of component interaction.

Some parts of the framework present interesting ways to tackle some well-known problems. For instance, the introspection protocol transforms the documentation process from a textual representation, which is fraught with potential language translation problems and namespace collisions, into an binary, interface-based representation where each element is uniquely identified by a GUID or is identified within the context of a GUID. With GUIDs, the homonym problem, where the same name refers to two distinct concepts, simply disappears, and all that remains is the synonym problem, that of determining when two items with different names refer to the same concept. Furthermore, the introspection protocol is an *implementation* protocol, not a modeling protocol, so that existing documentation schemes could use the introspection protocol in their implementation. One example of a modeling protocol is the SolverInfo Definition Language, presented in section 3.6.1.4, page 151. Others schemes could easily be used instead, and as long as they compile down to the SolverInfo Type Library specification, which is the proposed binary standard that leverages existing COM type libraries, the documentation scheme would be a valid introspection method for the framework.

The networking components, discussed throughout section 3.8, starting on page 178, are also noteworthy. The goals of the networking protocol are twofold. First is to make it simple for a client or modeling environment to create a network of solvers that exhibit local control, where each node in the network executes of its own accord once its inputs are available. In a distributed environment, this might be more efficient or more suitable than using a global controlling process. Second is to separate the functionality of managing the networking details and of executing algorithms. Solvers embody algorithms; they should not have to manage the various interconnections, mechanisms, and notifications that make a network function. The network details are encapsulated into the network components: the solver sites and the mappings. These components wrap each solver, presenting to each one the façade of a single client, as if the solver existed as a single stage.

The potential versatility of mappings, which manage the flow of data from one solver, data source, or client, to another, is also an important feature. Mappings to handle all manner of resource flows, data types, specialized solver interfaces, or flow requirements could be developed and incorporated into the existing framework network protocol. Mappings are the connectors, the translators, between two components in a network, and as specified are generic yet highly flexible objects.

Finally, the core services, a centralized repository of installed solvers and their documentation objects, and eventually a repository of dimension manipulators and model and solver selectors, is an important part of the framework. A common registry enables a solver to register itself with the core services once, and then to have that solver available to all clients and modeling environments that use the core services. Modelbases and solver databases for individual environments or particular methodologies have been proposed before, but the core services represent a database that imposes no model-solver relationships or structure, and is open to all environments and methodologies. While the lack of structure or selection capabilities inherent in the framework might be viewed as a deficiency, it can also be viewed as an opportunity. The core services is a foundation upon which methodologies, modelbase structure, and selection knowledge can be built; it represents a greatest common factor for all selection needs, and hence is always a viable technique for selecting solvers.

This comes full circle back to the nature of interface-based programming. The framework describes a viable standard for building applied operations research solutions. It might not have all of the best features, or approach everything with precisely the correct interfaces or patterns or protocols, but it can always be a technique of last resort when two components cannot find any other way to communicate. As the framework evolves, and as implementations of solvers and modeling environments evolve, and new interfaces and standards replace entirely the functionality of interfaces in the current framework, the old interfaces can be slowly phased out. New versions of solvers will eventually implement only the new interfaces, decreasing the development effort by not supporting the old. In the interim, there will be solvers that support both old and new; it is an evolutionary process.

CODA

In a survey article on visualization, optimization, and software, Jones [51] predicts the future of optimization and modeling software. Two points in particular are germane. First, "Spreadsheets will therefore become the delivery vehicle of optimization, at the expense of algebraic modeling languages." Second, "More and more optimization systems will be composed of modular building blocks linked together by facilities provided by the computer operating system." This thesis addresses the issue of modularity at the level of the operating system. The framework is an implementation standard that enables modularity, encapsulation, and independence of components. It essentially forms part of the operating system, and it exists independently of any one solver, modeling environment, or client. By using the COM standard as its foundation, it is accessible to the most common spreadsheet packages in use today.

Jones also predicts that a single algebraic modeling standard will emerge, and by inference, will dominate and propel the optimization market. This could very well be the "killer application" that leads to an explosion of solvers, modeling tools, applications, solutions, and exposure. The framework is not this sought-after killer application, certainly. It is,

however, a motivating force, an opportunity, and a starting point. The framework is not the grail, but leads on the path towards it.

In Wolfram von Eschenbach's *Parzival,* the early thirteenth century Germanic retelling of the French Arthurian romance *Conte du Graal* by Chrétien de Troyes, the protagonist knight Parzival commits in his youth three sins that impel the story and for which he must atone. One of the sins occurs at the castle Munsalvæsche, home of Anfortas the Angler, Parzival's uncle, known otherwise as the Fisher King, keeper of the Gral. (In Wolfram's version, the "Grail" is called the Gral, and is a stone rather than the vessel popularized by Malory.) At Munsalvæsche, Parzival sees the procession of the Gral and the lance that drips blood from its tip. Parzival learns that Anfortas is afflicted with severe injury, so that he cannot stand and walk. But as a simpleton instructed never to question a host, Parzival remains silent throughout the evening among these wonders, though questions ring in his head. In the morning, when he has finally resolved to ask his questions, the castle stands empty. Parzival arms himself and leaves the castle, to be confronted quickly by a page and a maiden who both admonish him for not asking the Question, the most important of the questions Parzival had desired to ask. For the very act of asking the Question would have cured and freed Anfortas and brought happiness to everyone in the castle. His sin was in not asking the Question in the presence of Anfortas and the Gral, while he had the chance. The adventures that follow form Parzival's quest to find the Gral again, to confront Anfortas, and to ask his Question, as well as to atone for his other two sins. When finally he returns to Munsalvæsche and meets Anfortas again, he does not fail:

> *Parzival wept. "Tell me where the Gral is," he said. "If the goodness of God triumphs in me, this Company here shall witness it!" Thrice did he genuflect in its direction to the glory of the Trinity, praying that the affliction of this man of sorrows be taken from him. Then, rising to his full height, he said, "Dear Uncle, what ails you?"* [104, p. 394-5]

With the Question asked, Anfortas was cured, and Parzival became the new Gral King. The Question is part of the mechanics of the story; by not asking it during his first visit to Munsalvæsche, Parzival must return at the end of the story to complete the quest. But it also symbolizes his maturation from an inexperienced youth, when during his first visit he did not express the sympathy he felt for Anfortas by asking what ails him, to a knight worthy of being the Gral King, when his first act upon meeting Anfortas the second time is to ask the Question.

Hatto, the translator of the version quoted above, interprets Wolfram to be saying that if Parzival, who fell so low as to even reject God at one point, can rise to be the highest knight in the world, then others can rise high as well. "For if a pot-hunting, crack-jouster could win the Gral, Wolfram was suggesting to [his audience], why not they too? Their Grals were waiting [104, p. 415]." The story of this unique individual serves as inspiration for all.

To bring the metaphor to the task at hand: The framework might not be the Gral, but it is, perhaps the Question. The act of asking it, of using and adopting it or its descendants, might lead to the cure of the ailment, the current disparity between modern software and modern operations research algorithms.

Howzat!

APPENDIX A
SAMPLE CODE AND EXTENSIONS

This appendix provides code listings referenced in the fourth chapter as well as some possible extensions to the framework.

A.1 DEVELOPING THE RLPWRAPPER SOLVER

The following sections detail the development of the RLPWrapper solver component that wraps the CPLEX Callable Library [17], described in section 4.1.4, page 226. There are six main phases to the solver implementation: generating the SolverInfo, implementing the skeleton functionality of the solver, implementing the inputs, implementing the solve action, implementing the outputs, and implementing the parameters.

The solver uses the CPLEX function CPXloadlp to load the LP into the CPLEX engine. This function takes the entire linear program as its input. These inputs have special format requirements. Because of this and because the solver itself can take inputs in any order, the solver creates an internal buffer to store the inputs whenever IRSolverInputs::SetInputData is called. The same holds for the outputs. The CPLEX function CPXsolution returns all of the outputs at once, in arrays of doubles. The solver must copy these values to its own data elements to expose them as outputs. The solver uses CPXoptimize to solve the LP.

A.1.1 Generating the SolverInfo

The first step is to create the SolverInfo source file. Based on the specification in section 4.1.4 for inputs and outputs, the resulting SolverInfo SIDL file looks something like this:

```
// File RLPWrapperInfo.idl
//
[
    uuid(3494F400-BD4B-11D1-9194-00207810C741), version(1.0),
    helpstring("Ruark RLPWrapperInfo 1.0 Solver Info Type Library"),
    custom(GUID_RSOLVERINPUTS,      "{3494F401-BD4B-11D1-9194-00207810C741}"),
    custom(GUID_RSOLVEROUTPUTS,     "{3494F402-BD4B-11D1-9194-00207810C741}"),
    custom(GUID_RSOLVERPARAMETERS,"{3494F403-BD4B-11D1-9194-00207810C741}"),
    custom(GUID_RSOLVERCLSID,       "{3494F481-BD4B-11D1-9194-00207810C741}"),
    custom(GUID_RSOLVERTYPELIBID,   "{3494F480-BD4B-11D1-9194-00207810C741}")
]
library RLPWrapperInfoLib
{
    importlib("stdole32.tlb");
    importlib("stdole2.tlb");

    [ uuid(3494F404-BD4B-11D1-9194-00207810C741) ]
    typedef long Variable;

    [ uuid(3494F405-BD4B-11D1-9194-00207810C741) ]
    typedef long Constraint;

    [ uuid(3494F401-BD4B-11D1-9194-00207810C741) ]
    interface Inputs
    {
        double ObjectiveFunction([in] Variable v);
        double RightHandSide([in] Constraint c);
        double ConstraintMatrix([in] Constraint c, [in] Variable v);
        BSTR ConstraintSense([in] Constraint c);
        [custom(GUID_RSAF_OPTIONAL, "True")]
        double LowerBound([in] Variable v);
        [custom(GUID_RSAF_OPTIONAL, "True")]
        double UpperBound([in] Variable v);
    };

    [ uuid(3494F402-BD4B-11D1-9194-00207810C741) ]
    interface Outputs
    {
        double ObjectiveValue();
        double PrimalVariables(Variable v);
        double DualVariables(Constraint c);
        double SlackValues(Constraint c);
        double ReducedCosts(Variable v);
    };

    [
```

```
        uuid(3494F403-BD4B-11D1-9194-00207810C741),
        custom(GUID_RSOLVERIMPLEMENTEDINTERFACE, "True")
    ]
    dispinterface RLPWrapperParameters
    {
    properties:
        [id(1), helpstring("Global time limit."), custom(GUID_RDEFAULTVALUE, "1e+75")]
        double TimeLimit;

        [id(2), helpstring("Use CPU time (True) or wall clock time(False)."),
         custom(GUID_RDEFAULTVALUE, "True")]
        VARIANT_BOOL CPUTime;

        [id(3), helpstring("Dual pricing algorithm."), custom(GUID_RDEFAULTVALUE, "0")]
        long DualPricingAlgorithm;

        [id(4), helpstring("Maximum iteration limit."), custom(GUID_RDEFAULTVALUE, "Varies")]
        long IterationLimit;

        [id(5), helpstring("Lower object value limit."), custom(GUID_RDEFAULTVALUE, "-
        1e+75")]
        double ObjectiveLowerLimit;

        [id(6), helpstring("Upper object value limit."), custom(GUID_RDEFAULTVALUE,
        "1e+75")]
        double ObjectiveUpperLimit;

        [id(7), helpstring("Feasibility tolerance."), custom(GUID_RDEFAULTVALUE, "1e-06")]
        double FeasibilityTolerance;
    methods:
    };
};
//EOF.
```

The MIDL compiler generates the type library RLPWrapperInfo.tlb, which is embedded into the final executable.

A.1.2 The solver skeleton

Development begins with the solver skeleton. The class that implements the wrapper is named CRLPSolver. The implementation shown here uses the Active Template Library 2.1, part of Microsoft Visual C++ 5.0. The generic C++ solver class header file for a solver that implements typical solver interfaces and uses the SolverAdviseHolder component is:

```
// RLPSolver.h : Declaration of the CRLPSolver

#ifndef __RLPSOLVER_H_
#define __RLPSOLVER_H_

#include "resource.h"     // main symbols
#include "RSolve.h"

/////////////////////////////////////////////////////////////////////////////
// CRLPSolver
class ATL_NO_VTABLE CRLPSolver :
    public CComObjectRootEx<CComSingleThreadModel>,
    public CComCoClass<CRLPSolver, &CLSID_RLPWrapper>,
    public IRSolver,
    public IRSolverInputs,
    public IRSolverOutputs,
    public IRSolverProvideInfo,
    public IRSolverParameters
{
    CComQIPtr<IRSolverSiteIn, &IID_IRSolverSiteIn> m_siteIn;
    CComQIPtr<IRSolverSiteOut, &IID_IRSolverSiteOut> m_siteOut;
    CComQIPtr<IRSolverAdviseHolder, &IID_IRSolverAdviseHolder> m_solverAdviseHolder;
public:
    CRLPSolver() {}

DECLARE_REGISTRY_RESOURCEID(IDR_RLPSOLVER)

BEGIN_COM_MAP(CRLPSolver)
    COM_INTERFACE_ENTRY(IRSolver)
    COM_INTERFACE_ENTRY(IRSolverInputs)
    COM_INTERFACE_ENTRY(IRSolverOutputs)
    COM_INTERFACE_ENTRY(IRSolverProvideInfo)
    COM_INTERFACE_ENTRY(IRSolverParameters)
END_COM_MAP()

// IRSolver
    STDMETHOD(GetSolverSiteIn)(REFIID riid, LPVOID* ppVoid);
    STDMETHOD(SetSolverSiteIn)(LPUNKNOWN pUnk);
    STDMETHOD(GetSolverSiteOut)(REFIID riid, LPVOID* ppVoid);
    STDMETHOD(SetSolverSiteOut)(LPUNKNOWN pUnk);
    STDMETHOD(Solve)();
    STDMETHOD(ClearInputs)();
    STDMETHOD(ClearOutputs)();
    STDMETHOD(SolveAdvise)(IRSolverAdvise* pAdvise, DWORD* pdwCookie);
    STDMETHOD(SolveUnadvise)(DWORD dwCookie);
```

```
// IRSolverInputs
    STDMETHOD(LockInputs)();
    STDMETHOD(UnlockInputs)();
    STDMETHOD(InputsLocked)(DWORD* pfLocked);
    STDMETHOD(GetInputCount)(UINT* pcInputs);
    STDMETHOD(GetInputData)(UINT uInput, REFIID riid, LPVOID* ppVoid, UINT*
        puElement);
    STDMETHOD(SetInputData)(UINT uInput, LPRDATASOURCE pSrc, UINT uElement);

// IRSolverOutputs
    STDMETHOD(LockOutputs)();
    STDMETHOD(UnlockOutputs)();
    STDMETHOD(OutputsLocked)(DWORD* pfLocked);
    STDMETHOD(GetOutputCount)(UINT* pcOutputs);
    STDMETHOD(GetOutputData)(UINT uOutput, REFIID riid, LPVOID* ppVoid, UINT*
        puElement);

// IRSolverProvideInfo
    STDMETHOD(GetSolverInfo)(REFIID riid, LPVOID* ppVoid);

// IRSolverParameters
    STDMETHOD(GetParameterCount)(UINT* puCount);
    STDMETHOD(GetParameter)(UINT uIndex, VARIANT* pVar);
    STDMETHOD(SetParameter)(UINT uIndex, VARIANT Parameter);
};
#endif //__RLPSOLVER_H_
//EOF.
```

The skeleton implementation provides only some of the solver interface functions, namely those supporting the advise list and the solver sites:

```
// IRSolver
STDMETHODIMP CRLPSolver::GetSolverSiteIn(REFIID riid, LPVOID* ppVoid)
{
    if( !ppVoid ) return E_POINTER;
    *ppVoid = NULL;
    if( !m_siteIn ) return NOERROR;
    return m_siteIn->QueryInterface(riid, ppVoid);
}

STDMETHODIMP CRLPSolver::SetSolverSiteIn(LPUNKNOWN pUnk)
{
    m_siteIn = pUnk;
    return NOERROR;
}
```

```
STDMETHODIMP CRLPSolver::GetSolverSiteOut(REFIID riid, LPVOID* ppVoid)
{
    if( !ppVoid ) return E_POINTER;
    *ppVoid = NULL;
    if( !m_siteOut ) return NOERROR;
    return m_siteOut->QueryInterface(riid, ppVoid);
}

STDMETHODIMP CRLPSolver::SetSolverSiteOut(LPUNKNOWN pUnk)
{
    m_siteOut = pUnk;
    return NOERROR;
}

STDMETHODIMP CRLPSolver::SolveAdvise(IRSolverAdvise* pAdvise, DWORD* pdwCookie)
{
    if( !pdwCookie ) return E_POINTER;
    if( !m_solverAdviseHolder )
    {
        HRESULT hr = CoCreateInstance(CLSID_RSolverAdviseHolder, NULL, CLSCTX_SERVER,
        IID_IRSolverAdviseHolder, (LPVOID*) &m_solverAdviseHolder);
        if( FAILED(hr) ) return E_OUTOFMEMORY;
    }
    return m_solverAdviseHolder->Advise(static_cast<LPRSOLVER>(this), pAdvise, pdwCookie);
}

STDMETHODIMP CRLPSolver::SolveUnadvise(DWORD dwCookie)
{
    if( !m_solverAdviseHolder || !dwCookie ) return OLE_E_NOCONNECTION;
    return m_solverAdviseHolder->Unadvise(dwCookie);
}

// IRSolverProvideInfo
STDMETHODIMP CRLPSolver::GetSolverInfo(REFIID riid, LPVOID* ppVoid)
{
    if( !ppVoid ) return E_POINTER;
    LPRSOLVERINFO pInfo = NULL;
    HRESULT hr = RLoadSolverInfoClsid(CLSID_RLPWrapper, 0, &pInfo);
    if( SUCCEEDED(hr) )
    {
        hr = pInfo->QueryInterface(riid, ppVoid);
        pInfo->Release();
    }
    return hr;
}
```

Obviously, this skeleton code, along with other skeleton code for other features of different solvers, could easily be consolidated into a C++ class library that simplifies the creation of solvers and clients.

A.1.3 Implementing solver inputs

Implementing the input side of the wrapper comes next. There are six inputs, which are stored internally in the form that CPLEX uses. As such, the inputs are write-only. To manage inputs, the solver uses a simple switch statement to dispatch the data element to a function that will set the particular input within the solver.

```
STDMETHODIMP CRLPSolver::GetInputCount(UINT* pcInputs)
{
    if( !pcInputs ) return E_POINTER;
    *pcInputs = 6;
    return NOERROR;
}

STDMETHODIMP CRLPSolver::GetInputData(UINT, REFIID, LPVOID*, UINT*)
{ return RSOLVE_E_CANNOTGETINPUTDATA; }

STDMETHODIMP CRLPSolver::SetInputData(UINT uInput, IRDataSource* pSrc, UINT uElement)
{
    // check for errors
    if( !pSrc )           return E_INVALIDARG;
    if( uInput > 5 )      return RSOLVE_E_INVALIDINDEX;
    if( inputsLocked() )  return RSOLVE_E_INPUTSLOCKED;
    if( solving() )       return RSOLVE_E_SOLVING;

    // get the data element
    LPRDATAELEMENT pElement = NULL;
    HRESULT hr = pSrc->GetDataElementAccessor(uElement, IID_IRDataElement,
        (LPVOID*)&pElement);
    if( FAILED(hr) ) return hr;

    // based on the input, the that particular input in the solver
    switch( uInput )
    {
    case 0: hr = setObjFunc(pElement);     break; // Objective function
    case 1: hr = setRHS(pElement);         break; // RHS
    case 2: hr = setConstraint(pElement);  break; // Constraint matrix
    case 3: hr = setSense(pElement);       break; // Constraint sense
    case 4: hr = setLower(pElement);       break; // Lower bound
    case 5: hr = setUpper(pElement);       break; // Upper bound
    }
```

```
        pElement->Release();
        return hr;
    }
```

The CRLPSolver class defines a number of member variables for storing these inputs.

```
    class CRLPSolver :   // ...base classes here
    {
    // ...
    protected:
        int          m_nCols;          // number of variables
        int          m_nRows;          // number of constraints
        double*      m_rgdObjFunc; // array of objective function values
        double*      m_rgdRHS;         // array of right hand sides
        char*        m_rgchSense;     // character array of constraint senses
        int*         m_rgnMatBegin;// Next four are the constraint matrix arrays
        int*         m_rgnMatCount;
        int*         m_rgnMatIndex;
        double*      m_rgdMatValue;
        double*      m_rgdLowerBound;        // array of lower bounds on variables
        double*      m_rgdUpperBound;        // array of upper bounds on variables
        int          m_nColSpace;    // Size of the variable space (same as m_nCols)
        int          m_nRowSpace; // Size of constraint space (same as m_nRows)
        int          m_nNZSpace;     // Size of the non-zero space
    // ...remainder of class here
```

All six of the setXxx functions called in SetInputData, listed above, are member functions of the CRLPSolver class. For the five of these inputs that are vectors, the functions to set those inputs look very much alike, so only one is shown here:

```
    HRESULT CRLPSolver::setObjFunc(LPRDATAELEMENT pElement)
    {    // check the dimension against any existing variable vectors
        HRESULT hr = checkVectorDimension(pElement, &m_nCols);
        if( FAILED(hr) ) return hr;

        m_nColSpace = m_nCols;
        m_varDim.Release();
        _cloneOrCopyDimension(pElement, 0, &m_varDim);

        delete [] m_rgdObjFunc; m_rgdObjFunc = NULL;
        m_rgdObjFunc = new double[m_nColSpace];
        if( !m_rgdObjFunc ) return E_OUTOFMEMORY;
        return fillDoubleArray(pElement, m_rgdObjFunc);
    }
```

First, setObjFunc needs to make sure the data element being passed to it is a vector. This is handled by the CRLPSolver member function checkVectorDimension. Furthermore, checkVectorDimension has the responsibility for validating that as the inputs arrive they agree in terms of the number of variables or constraints:

```
HRESULT CRLPSolver::checkVectorDimension(LPRDATAELEMENT pElement, int* pnSize)
{
    // Data element must be one dimensional
    UINT uDim = 0;
    HRESULT hr = pElement->GetDimensionCount(&uDim);
    if( FAILED(hr) || uDim != 1) return RSOLVE_E_WRONGNUMBEROFDIMENSIONS;

    // now, if the size is non-zero, they must match
    long nLower, nUpper;
    LPRDIMENSION pDim = NULL;
    hr = pElement->GetDimension(0, &pDim);
    if( FAILED(hr) || !pElement ) return E_FAIL;

    pDim->GetBounds(&nLower, &nUpper);
    pDim->Release();
    long nSize = nUpper-nLower+1;
    if( *pnSize && (nSize != *pnSize) ) return RSOLVE_E_WRONGDIMENSIONSIZE;

    if( !*pnSize )
        *pnSize = nSize;
    return NOERROR;
}
```

To see if the vector does not match already existing inputs, checkVectorDimension takes as its second parameter a pointer to the member variable indicating the number of variables or constraints, depending on which input is being set. If no inputs have been set, these values are zero, and checkVectorDimension will set one or the other using the size of the vector. Hence, the first input passed to the solver determines the number of variables or constraints, or both, to which all other similar inputs are compared.

The method setObjFunc then clones the dimension associated with the vector using the helper _cloneOrCopyDimension and stores it in a class member variable. The variable and constraint dimensions are used to generate the output vectors, so that they use the same domain and sets as the input dimensions. Then, setObjFunc takes the data element vector and copies its values into a C array of doubles. This copy is handled by the member function fillDoubleArray, which illustrates iterating over the values of a data element:

```
// Given a vector data element (assumed), copy its values into a pre-allocated array of doubles
HRESULT CRLPSolver::fillDoubleArray(LPRDATAELEMENT pElement, double* rgd)
{
    long nLower, nUpper;
    LPRDIMENSION pDim = NULL;
    pElement->GetDimension(0, &pDim);
    pDim->GetBounds(&nLower, &nUpper);
    pDim->Release();

    VARIANT var; VariantInit(&var);
    for(long i = nLower ; i <= nUpper ; ++i)
    {
        pElement->GetAt(1, &i, &var);
        VariantChangeType(&var, &var, 0, VT_R8);
        *rgd++ = var.dblVal;
        VariantClear(&var);
    }
    return NOERROR;
}
```

For the ConstraintSense input, which is a vector of characters, there is a similar method named fillCharArray.

To set the matrix requires a little bit more work because CPLEX uses a sparse matrix format that stores only the non-zero values of the constraint matrix in a single array. Nonetheless, the general structure is the same as setObjFunc, checkVectorDimension, and fillDoubleArray.

A.1.4 Solving the linear program

To solve the LP, the wrapper's implementation of IRSolver::Solve first checks whether all of the inputs have been set, whether the solver is already solving, and whether CPLEX has been initialized correctly:

```
STDMETHODIMP CRLPSolver::Solve()
{
    if( solving() ) return RSOLVE_E_SOLVING;
    if( !gotAllInputs() ) return RSOLVE_E_NOTALLINPUTS;
    if( !cplex_env() ) return E_FAIL;
```

The method solving returns true if the solver is already in the Solve method. This is an important check because there is technically nothing to prevent the client from calling Solve again during any progress notifications. The method gotAllInputs ensures that all of the non-

optional inputs have been set by checking that the corresponding member variables are non-NULL.

```
// ...Solve() continued
    LockInputs();
    ClearOutputs();
    m_bSolving = true;   // indicate that solving is in progress
```

Next the method locks the inputs, so that other solvers will not try to set new inputs. It also clears the outputs, in preparation for a new solution. This is a least-optimal scenario, as the solver is effectively isolated during the entire solution process. A more intelligent solution, requiring more code, might be to clear the outputs as late as possible, allowing clients to access outputs from a previous run even as the solver is working on the next.

```
// ...Solve() continued
    // build the lower and upper bounds if need be
    buildDefaultBounds();
```

The lower and upper bound inputs are optional, so the buildDefaultBounds method checks to see if the member variables for lower and upper bounds have been set, and if not, it creates them and initializes them to default values, namely a lower bound of zero and an upper bound of infinity.

The next step is to load the problem into CPLEX's system and solve it. This is normal CPLEX Callable Library code.

```
// ...Solve() continued
    // Load the LP
    CPXLPptr pLP = CPXloadlp(cplex_env(), "Problem", m_nCols, m_nRows, CPX_MIN,
        m_rgdObjFunc, m_rgdRHS, m_rgchSense, m_rgnMatBegin, m_rgnMatCount,
        m_rgnMatIndex, m_rgdMatValue, m_rgdLowerBound, m_rgdUpperBound,
        NULL, m_nColSpace, m_nRowSpace, m_nNZSpace);
    if( !pLP )
    {
        m_bSolving = false;
        UnlockInputs();
        return E_OUTOFMEMORY;
    }

    // Solve it
    int nResult = CPXoptimize(cplex_env(), pLP);
```

After CPLEX has determined a solution or returned an error, the next task is to retrieve the outputs. The solver uses scalar and vector components provided by the framework library, created from the dimensions cloned while setting input data. The member variables **m_objValue**, **m_primal**, **m_dual**, **m_slack**, and **m_reduced** hold the five outputs.

```
// ...Solve() continued
    // Create the output vectors
    LPRDATAELEMENTCREATOR pCreator = NULL;
    CoCreateInstance(CLSID_RScalarCreator, NULL, CLSCTX_INPROC_SERVER,
        IID_IRDataElementCreator, (LPVOID*)&pCreator);
    pCreator->Create(L"ObjectiveValue", 0, NULL, VT_R8, IID_IRDataElementScalar,
        (LPVOID*)&m_objValue);
    pCreator->Release();

    CoCreateInstance(CLSID_RDoubleVectorCreator, NULL, CLSCTX_INPROC_SERVER,
        IID_IRDataElementCreator, (LPVOID*)&pCreator);
    pCreator->Create(L"PrimalVariables", 1, &m_varDim, VT_R8, IID_IRDataElement1,
        (LPVOID*)&m_primal);
    pCreator->Create(L"DualVariables", 1, &m_conDim, VT_R8, IID_IRDataElement1,
        (LPVOID*)&m_dual);
    pCreator->Create(L"SlackValues", 1, &m_conDim, VT_R8, IID_IRDataElement1,
        (LPVOID*)&m_slack);
    pCreator->Create(L"ReducedCosts", 1, &m_varDim, VT_R8, IID_IRDataElement1,
        (LPVOID*)&m_reduced);
    pCreator->Release();
```

It is a simple matter to retrieve the solution from CPLEX using **CPXsolution**, to fill all the values into the data elements, and delete the solution retrieved from CPLEX:

```
// ...Solve() continued
    // Now retrieve the solutions from CPLEX
    double dObjVal;
    double* rgdX = new double[m_nColSpace];
    double* rgdPi = new double[m_nRowSpace];
    double* rgdSlack = new double[m_nRowSpace];
    double* rgdRed = new double[m_nColSpace];
    CPXsolution(cplex_env(), pLP, NULL, &dObjVal, rgdX, rgdPi, rgdSlack, rgdRed);

    // set the vectors and scalar
    m_objValue->SetScalar(CComVariant(dObjVal));
    fillOutputVector(m_primal, rgdX);
    fillOutputVector(m_dual, rgdPi);
    fillOutputVector(m_slack, rgdSlack);
    fillOutputVector(m_reduced, rgdRed);
```

```
    delete [] rgdX;
    delete [] rgdPi;
    delete [] rgdSlack;
    delete [] rgdRed;
```

The helper function fillOutputVector transfer a C array of doubles into a vector data element, in much the same way that fillDoubleArray does the opposite. Finally, to clean up, Solve unloads the problem, turns off the solving flag, and unlocks the inputs, so that a new problem can be loaded:

```
// ...Solve() continued
    if( pLP )
        CPXunloadprob(cplex_env(), &pLP);
    m_bSolving = false;
    UnlockInputs();
    return NOERROR;    //REVIEW: OUTPUT STATUS
}   // End of Solve()
```

A.1.5 Implementing the output methods

With the outputs loaded, it is a simple matter to pass them out to clients in calls to GetOutputData:

```
STDMETHODIMP CRLPSolver::GetOutputCount(UINT* pcOutputs)
{
    if( !pcOutputs ) return E_POINTER;
    *pcOutputs = 5;
    return NOERROR;
}

STDMETHODIMP CRLPSolver::GetOutputData(UINT uOutput, REFIID riid, LPVOID* ppVoid,
        UINT* puElement)
{
    if( !ppVoid || !puElement ) return E_POINTER;
    if( uOutput > 4 ) return RSOLVE_E_INVALIDINDEX;
    if( solving() ) return RSOLVE_E_SOLVING;
    if( !haveOutputs() ) return RSOLVE_E_OUTPUTNOTSET;

    CComQIPtr<IRDataElement, &IID_IRDataElement> element;
    switch( uOutput )
    {
        case 0: element = m_objValue; break;
        case 1: element = m_primal; break;
        case 2: element = m_dual; break;
```

```
            case 3: element = m_slack; break;
            case 4: element = m_reduced; break;
        }
        return element->GetDataSource(riid, ppVoid, puElement);
}
```

A.1.6 Implementing the parameters

The final step in wrapping the solver is to implement the parameters. The IRSolverParameters interface is relatively straightforward. First, the solver must return the number of parameters, seven in this reduced example, via GetParameterCount:

```
STDMETHODIMP CRLPSolver::GetParameterCount(UINT* puCount)
{
    if( !puCount ) return E_POINTER;
    *puCount = 7;
    return NOERROR;
}
```

Next, retrieving a parameter is a simple matter of calling the appropriate CPLEX parameter function and coercing the result into the generic VARIANT structure returned to the client:

```
STDMETHODIMP CRLPSolver::GetParameter(UINT uIndex, VARIANT* pVar)
{
    if(uIndex>6) return E_INVALIDARG;
    if( !pVar ) return E_POINTER;

    switch( uIndex )
    {
    case 0: // TimeLimit
        pVar->vt = VT_R8;
        CPXgetdblparam(cplex_env(), CPX_PARAM_TILIM, &(pVar->dblVal));
        break;
    case 1: // CPUTime
        int i;
        pVar->vt = VT_BOOL;
        CPXgetintparam(cplex_env(), CPX_PARAM_CLOCKTYPE, &i);
        pVar->boolVal = (i==1) ? VARIANT_TRUE : VARIANT_FALSE;
        break;
    case 2: // DualPricing Algorithm
        pVar->vt = VT_I4;
        CPXgetintparam(cplex_env(), CPX_PARAM_DPRIIND, (int*)&(pVar->lVal));
        break;
    case 3: // IterationLimit
```

```
        pVar->vt = VT_I4;
        CPXgetintparam(cplex_env(), CPX_PARAM_ITLIM, (int*)&(pVar->lVal));
        break;
    case 4: // ObjectiveLowerLimit
        pVar->vt = VT_R8;
        CPXgetdblparam(cplex_env(), CPX_PARAM_OBJLLIM, &(pVar->dblVal));
        break;
    case 5: // ObjectiveUpperLimit
        pVar->vt = VT_R8;
        CPXgetdblparam(cplex_env(), CPX_PARAM_OBJULIM, &(pVar->dblVal));
        break;
    case 6: // Feasibility Tolerance
        pVar->vt = VT_R8;
        CPXgetdblparam(cplex_env(), CPX_PARAM_EPRHS, &(pVar->dblVal));
        break;
    };
    return NOERROR;
}
```

Setting parameters is equally easy, with the coercion this time from the VARIANT to the specific type required by CPLEX:

```
STDMETHODIMP CRLPSolver::SetParameter(UINT uIndex, VARIANT Parameter)
{
    if(uIndex>6) return E_INVALIDARG;
    switch( uIndex )
    {
    case 0:  // TimeLimit
        setDblParam(CPX_PARAM_TILIM, Parameter);
        break;
    case 1: // CPUTime
        VARIANT v; VariantInit(&v);
        VariantChangeType(&v, &Parameter, 0, VT_BOOL);
        CPXsetintparam(cplex_env(), CPX_PARAM_CLOCKTYPE, v.boolVal ? 1 : 2);
        VariantClear(&v);
        break;
    case 2: // DualPricing Algorithm
        setIntParam(CPX_PARAM_DPRIIND, Parameter);
        break;
    case 3: // IterationLimit
        setIntParam(CPX_PARAM_ITLIM, Parameter);
        break;
    case 4: // ObjectiveLowerLimit
        setDblParam(CPX_PARAM_OBJLLIM, Parameter);
        break;
```

```
        case 5: // ObjectiveUpperLimit
            setDblParam(CPX_PARAM_OBJULIM, Parameter);
            break;
        case 6: // Feasibility Tolerance
            setDblParam(CPX_PARAM_EPRHS, Parameter);
            break;
        }
        return NOERROR;
    }
```

The functions setIntParam and setDblParam are members of CRLPSolver, and they simply convert the VARIANT Parameter into an int or a double, respectively, and set the appropriate parameter in CPLEX:

```
HRESULT CRLPSolver::setIntParam(int param, VARIANT& var)
{
    VARIANT v;
    VariantInit(&v);
    VariantChangeType(&v, &var, 0, VT_I4);
    CPXsetintparam(cplex_env(), param, v.lVal);
    VariantClear(&v);
    return NOERROR;
}

HRESULT CRLPSolver::setDblParam(int param, VARIANT& var)
{
    VARIANT v;
    VariantInit(&v);
    VariantChangeType(&v, &var, 0, VT_R8);
    CPXsetdblparam(cplex_env(), param, v.dblVal);
    VariantClear(&v);
    return NOERROR;
}
```

The SIDL for this solver's parameter description interface had the custom attribute GUID_RSOLVERIMPLEMENTEDINTERFACE, which means that the solver supports the interface IRSolverParametersInterface for accessing the parameters dispatch interface. Handling this is slightly more complicated than the IRSolverParameters interface itself. Because of a bug in the COM remoting layer, the dispatch interface must be implemented by a sub-object of the solver so that it does not conflict with any dispatch interfaces implemented by the solver.

This sub-object can be generalized into a class that can work with any parameters dispinterface, and hence could be incorporated into the core services. Its class declaration for the RLPWrapper solver is:

```
class ATL_NO_VTABLE CRLPWrapperParameters :
    public CComObjectRootEx<CComSingleThreadModel>,
    public IDispatch
{
protected:
    LPRSOLVERPARAMETERS m_pParameters;
    LPTYPEINFO m_pTI;
    void init(LPRSOLVERPARAMETERS pParameters, LPTYPEINFO pTI) {
        if( pTI ) pTI->AddRef();
        m_pTI = pTI;
        m_pParameters = pParameters;
    }

    CRLPWrapperParameters() : m_pParameters(0) {}

BEGIN_COM_MAP(CRLPWrapperParameters)
    COM_INTERFACE_ENTRY(IDispatch)
    COM_INTERFACE_ENTRY_IID(DIID_RLPWrapperParametersDisp,
        CRLPWrapperParameters)
END_COM_MAP()

protected:
    STDMETHOD(GetTypeInfoCount)(UINT* pctInfo);
    STDMETHOD(GetTypeInfo)(UINT itInfo, LCID lcid, LPTYPEINFO* ppITypeInfo);
    STDMETHOD(GetIDsOfNames)(REFIID riid, OLECHAR** rgszNames, UINT cNames, LCID
        lcid, DISPID *rgDispID);
    STDMETHOD(Invoke)(DISPID dispID, REFIID riid, LCID lcid, WORD wFlags,
        DISPPARAMS* pDispParams, VARIANT* pVarResult, EXCEPINFO* pExcepInfo, UINT*
        puArgErr);

public:
    static HRESULT create(LPUNKNOWN pParent, LPRSOLVERPARAMETERS pParameters,
        LPTYPEINFO pInfo, LPDISPATCH* ppDisp);
    ~CRLPWrapperParameters() {}
};
```

This is a straightforward class declaration of an object that will implement IDispatch. A sub-object is created by the solver by calling the static member function create, which creates and initializes the new sub-object. This function takes as its inputs a pointer to the parent object for the sub-object, which is the solver's IUnknown implementation, the implementation of the IRSolverParameters interface by the solver and the type info for the parameters dispinterface in the form of its ITypeInfo pointer. This function returns the IDispatch interface implemented by the sub-object.

```
HRESULT CRLPWrapperParameters::create(LPUNKNOWN pParent, LPRSOLVERPARAMETERS
        pParams, LPTYPEINFO pInfo, LPDISPATCH* ppDisp)
{
    if( !ppDisp ) return E_POINTER;
    CRLPWrapperParameters* p = new
        CComObjectChild<CRLPWrapperParameters>(pParent);
    if( p ) {
        p->init(pParams, pInfo);
        *ppDisp = static_cast<IDispatch*>(p);
        return NOERROR;
    }
    return E_OUTOFMEMORY;
}
```

The CComObjectChild<> template is a special class that manages the lifetime of a child object underneath a parent object, ensuring that as long as the child object is alive, the parent object remains alive.

The implementation of IDispatch is quite simple, because the sub-object already has the ITypeInfo of the parameters interface and the solver's IRSolverParameters interface:

```
STDMETHODIMP CRLPWrapperParameters::GetTypeInfoCount(UINT* pctInfo)
{
    if( !pctInfo ) return E_POINTER;
    *pctInfo = 1;
    return NOERROR;
}

STDMETHODIMP CRLPWrapperParameters::GetTypeInfo(UINT itInfo, LCID lcid, LPTYPEINFO*
        ppITypeInfo)
{
    if( !ppITypeInfo ) return E_POINTER;
    if( itInfo ) return TYPE_E_ELEMENTNOTFOUND;
    if( m_pTI ) m_pTI->AddRef();
    *ppITypeInfo = m_pTI;
    return NOERROR;
}

STDMETHODIMP CRLPWrapperParameters::GetIDsOfNames(REFIID riid, OLECHAR**
        rgszNames, UINT cNames, LCID lcid, DISPID *rgDispID)
{
    if( IID_NULL != riid ) return DISP_E_UNKNOWNINTERFACE;
    LPTYPEINFO pTI = NULL;
    HRESULT hr = GetTypeInfo(0, lcid, &pTI);
    if( SUCCEEDED(hr) )
```

```
    {
        hr = pTI->GetIDsOfNames(rgszNames, cNames, rgDispID);
        pTI->Release();
    }
    return hr;
}

STDMETHODIMP CRLPWrapperParameters::Invoke(DISPID dispID, REFIID riid, LCID lcid,
        WORD wFlags, DISPPARAMS* pDispParams, VARIANT* pVarResult, EXCEPINFO*
        pExcepInfo, UINT* puArgErr)
{
    if( riid != IID_NULL ) return DISP_E_UNKNOWNINTERFACE;
    if( NULL == m_pParameters ) return E_UNEXPECTED;

    // if the dispid is <0 or >6 the invalid
    if( dispID<0 || dispID>6 ) return DISP_E_MEMBERNOTFOUND;

    // if a property get/method call, then get a parameter
    if( wFlags & DISPATCH_PROPERTYGET || wFlags & DISPATCH_METHOD )
    {
        if( NULL==pVarResult ) return E_INVALIDARG;
        VariantInit(pVarResult);
        m_pParameters->GetParameter(dispID, pVarResult);
        return NOERROR;
    }

    // property put. Make sure there's one argument
    if( pDispParams->cArgs != 1 ) return DISP_E_BADPARAMCOUNT;
    int c = pDispParams->cNamedArgs;
    if( 1!=c || (1==c && DISPID_PROPERTYPUT!=pDispParams->rgdispidNamedArgs[0]))
        return DISP_E_PARAMNOTOPTIONAL;

    m_pParameters->SetParameter(dispID, pDispParams->rgvarg[0]);
    return NOERROR;
}
```

To use the sub-object, the solver implements the IRSolverParametersInterface:: GetParametersInterface member to create and cache a pointer to the sub-object CRLPWrapperParameters. This requires loading the type info for the parameters dispinterface, but is otherwise a straightforward implementation of a child object.

```
STDMETHODIMP CRLPSolver::GetParametersInterface(LPDISPATCH* ppDisp)
{
    if( !ppDisp ) return E_POINTER;
    *ppDisp = NULL;
```

```
    // create the cached RLPWrapperParameters implementation if necessary
    if( !m_parameter )
    {
        // load the parameters dispinterface typeinfo
        LPTYPEINFO pTI = NULL;
        LPTYPELIB plTypeLib = NULL;
        LoadRegTypeLib (LIBID_RDBWrapperInfoLib), 1, 0, LOCALE_USER_DEFAULT,
        &plTypeLib);
        plTypeLib->GetTypeInfoOfGuid(DIID_RLPWrapperParametersDisp, &pTI);
        plTypeLib->Release();
        LPDISPATCH pDisp = NULL;
        CRLPWrapperParameters::create(GetUnknown(), this, pTI, &pDisp);
        pTI->Release();
        m_parameter = pDisp;
    }
    if( ! !m_parameter )
    {
        m_parameter->AddRef();
        *ppDisp = m_parameter;
        return NOERROR;
    }
    return E_OUTOFMEMORY;
}
```

The solver is thus completely wrapped. Wrapping this basic functionality of the CPLEX solver required just over 1000 lines of C++ code. Some of the items, such as the parameter dispatch implementation and the conversion of data elements to vectors and sparse matrices suitable for CPLEX, can be reused relatively easily, reducing future coding efforts. Regardless, 1000 lines of code is a small price to pay to transform a subset of the CPLEX Callable Library into a solver object that can easily be embedded in solution networks, accessed from Visual Basic and Excel, and even the web.

A.2 Creating a Solver That Wraps Model Knowledge

Section A.1 detailed the development of a solver. This section presents only the SolverInfo file that specifies the inputs and outputs to the Monsanto solver, described in section 4.2.1.4, page 232, as shown here (attribute tags have been removed for readability):

```
library RMonsantoInfoLib
{
    typedef BSTR Technical;     // i
    typedef BSTR Formulation;   // j
    typedef BSTR Package;       // k
    typedef BSTR RMIStorage;    // l
```

```
typedef BSTR WIPStorage;   // l
typedef BSTR FGIStorage;   // l
typedef BSTR ProdResource; // m
typedef BSTR PackResource; // n
typedef long Segment;      // r
typedef long Scenario;     // s
typedef BSTR Time;         // t

interface Inputs
{
double RMIProduction(Technical i, Time t);     // P(i,t)
double RMIUsage(Technical i, RMIStorage l, Formulation j);     // A(i,l,j)
double WIPUsage(Formulation j, Package k); // A(j,k)
double RMIStorage(Technical i, RMIStorage l);        // S(i,l)
double WIPStorage(Formulation j, WIPStorage l);      // S(j,l)
double FGIStorage(Formulation j, Package k, FGIStorage l);   // S(j,k,l)
double RMICapacity(RMIStorage l, Time t);   // W(l,t)
double WIPCapacity(WIPStorage l, Time t);   // W(l,t)
double FGICapacity(FGIStorage l, Time t);   // W(l,t)
double Demand(Formulation j, Package k, Time t, Scenario s);// D(j,k,t,s)
double ProdCapacity(ProdResource m, Time t, Segment r);      // Q(m,t,r)
double PackCapacity(PackResource n, Time t, Segment r);      // Q(n,t,r)
double ProdGoesInto(Formulation j, ProdResource m); // V(j,m)
double PackGoesInto(Formulation j, Package k, PackResource n);        // V(j,k,n)
double RMIHoldingCost(Technical i, RMIStorage l);   // H(i,l)
double WIPHoldingCost(Formulation j, WIPStorage l); // H(j,l)
double FGIHoldingCost(Formulation j, Package k, FGIStorage l);        // H(j,k,l)
double ProductionCost(ProdResource m, Time t, Segment r);    // C(m,r,t)
double PackageCost(PackResource n, Time t, Segment r);       // C(n,r,t)
double RMIDistributionCost(Technical i, RMIStorage l);// X(i,l)
double WIPDistributionCost(Formulation j, WIPStorage l);      // X(j,l)
double FGIDistributionCost(Formulation j, Package k, FGIStorage l);// X(j,k,l)
double DemandCost(Formulation j, Package k, FGIStorage l);  // Y(j,k,l)
double LostSalesCost(Formulation j, Package k, Time t);      // U(j,k,t)
double RMIInitial(Technical i, RMIStorage l);  // I(i,l,0)
double WIPInitial(Formulation j, WIPStorage l, Scenario s);     // I(j,l,0,s)
double FGIInitial(Formulation j, Package k, FGIStorage l, Scenario s); // I(j,k,l,0,s)
double RMIMinEnding(Technical i, RMIStorage l);      // E(i,l)
double WIPMinEnding(Formulation j, WIPStorage l);  // E(j,l)
double FGIMinEnding(Formulation j, Package k, FGIStorage l);// E(j,k,l)
double ScenarioWeight(Scenario s); // Weight(s)
};

interface Outputs
{
double ObjectiveValue();
```

```
        // decision variables primary values
        double Demand(Formulation j, Package k, FGIStorage l, Scenario s, Time t);
        double RMI(Technical i, RMIStorage l, Time t);
        double FGI(Formulation j, Package k, FGIStorage l, Scenario s, Time t);
        double FormulationInventory(Formulation j, WIPStorage l, Scenario s, Time t);
        double TechnicalProduction(Technical i, WIPStorage l, Time t);
        double FormulationProduction(Formulation j, WIPStorage l, Time t);
        double PackagingProduction(Formulation j, Package k, FGIStorage l, Scenario s, Time t);
        double ProductionLevel(ProdResource m, Time t, Segment r);
        double PackagingLevel(PackResource n, Time t, Segment r);
        double LostSales(Formulation j, Package k, FGIStorage l, Scenario s, Time t);
    // the decision variable reduced costs
        double DemandReducedCost(Formulation j, Package k, FGIStorage l, Scenario s, Time t);
        // ... the other nine reduced cost variables here ...
    // constraint slack values
        double ProdTechSlack(Technical i, Time t);
        double ProdTechBalSlack(ProdResource m, Time t);
        double PackSlackSlack(PackResource n, Scenario s, Time t);
        double RMIBalanceSlack(Technical i, RMIStorage l, Time t);
        double RMICapacitySlack(RMIStorage l, Time t);
        double WIPBalanceSlack(Formulation j, WIPStorage l, Scenario s, Time t);
        double WIPCapacitySlack(WIPStorage l, Scenario s, Time t);
        double FGIBalanceSlack(Formulation j, Package k, FGIStorage l, Scenario s, Time t);
        double FGICapacitySlack(FGIStorage l, Scenario s, Time t);
        double DemandMetSlack(Formulation j, Package k, Scenario s, Time t);
    // constraint shadow prices
        double ProdTechShadowPrice(Technical i, Time t);
        // ... the other nine constraint shadow prices here ...
        };

        dispinterface RMonsantoParameters
        {
        properties:
           [id(0)] double TimeLimit;
           [id(1)] long IterationLimit;
        methods:
        };
    };
```

This solver is an example of a solver that contains another solver, namely the CPLEX wrapper solver. It would be a trivial task to allow specification via a parameter of which LP engine to use, assuming that it exposes the same solver structure as the CPLEX wrapper does.

A.3 Custom Extensions to the Framework: A Random Variable Specification

One direction in which the framework can be extended is through the specification of new data types. Many data types are characterized by complex structures rather than simple types. This section proposes interfaces for two complex data types: random variables and queueing systems.

A.3.1 Specifying a data type

There are two ways to define a data type. One is to use the structure capabilities of COM IDL to create a complex structure from simple data types. For instance, the Microsoft IDL file for Windows types, wtypes.idl, includes a data type corresponding to a point:

```
typedef struct tagPOINT
{
    LONG x;
    LONG y;
} POINT, *PPOINT, *LPPOINT;
```

These data types are, like the simple data types, static objects with no functionality of their own. A POINT is two longs, and nothing else. Using a POINT requires knowing the layout of the POINT object in memory, which requires knowing how to interpret IDL structures. The scripting languages VBScript and JScript cannot, so this POINT data type would be unavailable to them.

The second way is to define interfaces that objects that implement the data type will support. For example, an interface for a point structure might be:

```
interface IRPoint : IUnknown
{
    HRESULT get([out] long* px, [out] long* py);
    HRESULT set([in] long x, [in] long y);
};
```

This version, because it does not derive from IDispatch, also cannot be used by the simpler scripting languages VBScript and JScript. However, this data type can easily be defined by a dual interface:

```
interface IRPointDisp : IDispatch
{
    [id(0), propget] HRESULT x([out, retval] long* retval);
```

```
        [id(0), propput] HRESULT x([in] long newValue);
        [id(1), propget] HRESULT y([out, retval] long* retval);
        [id(1), propput] HRESULT y([in] long newValue);
    };
```

Now, VB clients can refer to the X and Y components of the point separately, as in point.x and point.y.

The interface versions of a data type seem like significantly more work than just using a structure because the interfaces provide no implementation themselves. Someone would need to create a component that implements those interfaces in order to actually use the data type. Furthermore, the object would swell in size. The structure version of a point is eight bytes (two longs, four bytes each). The interface version of a point would require these same eight bytes for the point values themselves, as well as at least eight more bytes for the vtable pointer and the reference count. That is, implementing a point as a COM object would at least double the size of each point. In environments where hundreds of thousands of points will be manipulated, this might be undesirable. Either using the structure version of a point or using some container interface (something akin to the flyweight pattern of Gamma et al. [30]) would be preferable.

Nonetheless, there are three cases where the interface definition is preferred. One case is when the data type is volatile, in the sense that the values of the data type can change without any directives from the client. An example is stock prices. A structure for a stock price data type could be:

```
typedef tagSTOCKPRICE
{
    BSTR    bstrSymbol;        // Like "MSFT"
    double  dPrice;            // price of the stock at last update
    DATE    dtLastUpdate;      // contains data and time of last update
} STOCKPRICE;
```

However, for a client to update the stock price, it needs a function that can query the stock price service providers, such as:

```
// On input, pPrice->bstrSymbol contains symbol to look up
// On output, dPrice and dtLastUpdate contain price and time
HRESULT getStockPrice([in, out] STOCKPRICE* pPrice);
```

An alternative solution is to wrap the data type by an interface that queries the stock price whenever the price is requested:

```
interface IRStockPriceDisp : IDispatch
{
    [id(0), propget] HRESULT Symbol([out, retval] BSTR* retval);
    [id(0), propput] HRESULT Symbol([in] BSTR newValue);
    // Dynamically queries the stock price
    [id(1), propget] HRESULT Price([out, retval] double* retval);
    // returns time of last Price query
    [id(2), propget] HRESULT Time([out, retval] DATE* retval);
};
```

Now, the query mechanism is implicitly wrapped into the data type. Furthermore, the client can no longer *set* the value of the price or time. This is the second case for using interfaces. The data type can prohibit changing values that are derived attributes, such as the stock price (a function of the symbol and the query time) or the area of a rectangle when the length and width can be specified. This read-only ability of properties in an interface can aid development efforts by ensuring that clients do not misbehave by changing values they should not.

The third case is similar to that for derived attributes. Some data types require such complex calculations, or offer such sufficiently complex operations, that they warrant their own interfaces. In particular, when the copy, assignment, and equality semantics are sufficiently complex, an object might need its own interface[1]. An example provided by Microsoft is the font data type, wrapped by the IFont and IFontDisp interfaces. These interfaces provide property access, comparison, and cloning capabilities for a font, as well as the ability to query the text metrics for the font, which is computed upon request. The random variable and queueing system data types are examples of the third case.

A.3.2 Using a custom data type in the framework

Ideally, a custom data type could be stored in a data element just as a simple data type can. The data element interfaces make this trade-off by specifying parameters of type VARIANT, which incurs a performance and space overhead (VARIANTs are 16 bytes and have deep copy semantics), but permits the greatest flexibility for data flow. Fortunately, VARIANTs can store pointers to IUnknown, which means that any object can be stored in data elements. In particular, custom data types that are described by COM interfaces and implemented by components can be stored in data elements under the current specification.

Complex data types based on structures are not so fortunate, as VARIANTs cannot store structures. Using structures in the current framework would require adding new data element interfaces that support the structure, and then clients would have to know to use that

[1] For example, to clone a POINT requires a shallow copy of two longs. This is the default assignment behavior. However, to clone a string requires a deep copy of the string data. The default assignment behavior is not sufficient for this task (in C and C++), so a special string copy function is necessary.

interface specifically. This is certainly possible, but it currently limits the structure's applicability with generic clients that are only aware of the basic data element interfaces. Data elements containing these complex structures could certainly be passed around a network without any client interaction, if all of the solvers know how to use the structure, so for internal networks this could be a good solution. Structures might offer optimizations over using VARIANTs or component implementations.

Given that a pointer to an object's IUnknown can be stored in a standard data element, it remains to specify how the client knows that a solver input or output is a complex data type rather than a simple type. Ideally, it should be as simple as specifying in the solver's SIDL file (see section 3.6.1.4, page 151) the new data type. So, the original description of an input that conceptually is a vector of points might be:

```
// From SIDL library block
interface Inputs
{
    long LatticeX(PointIndex i);
    long LatticeY(PointIndex i);
};
```

Using a custom data type, this becomes the more logically pleasing:

```
// From SIDL library block
interface Inputs
{
    // Takes as input a vector of points
    IRPointDisp* Lattice(PointIndex i);
};
```

This works fine for generating the SITL. However, for a client to determine the type of this data element, a new SolverInfo interface needs to be defined. Whereas previously the client could determine the raw data type of the data element as an enumeration of the possible VARIANT types using the GetDataType method of the IRSolverInputInfo, IRSolverOutputInfo, and IRSolverParamInfo interfaces, now the client needs to be able to browse the interface definition of the custom data type. That is, the client needs the ITypeInfo* of the interface. A new SolverInfo interface might look like this:

```
interface IRSolverInfoProvideTypeInfo
{
    HRESULT GetTypeInfo([out] ITypeInfo** ppTInfo);
};
```

This interface would be implemented by the SolverOutputInfo, SolverInputInfo, and SolverSetInfo classes, and would be exposed by those objects that describe custom data types. A client can determine if a particular SolverInputInfo object, for instance, has a custom data type by checking to see if its data type is type VT_UNKNOWN or VT_DISPATCH. If so, it can QueryInterface for IRSolverInfoProvideTypeInfo and retrieve the type library information for the interface that describes the custom data type.

A.3.3 A random variable specification

A random variable data type, as specified by this extension, is an object that implements the IRRandomVariable interface, defined as follows:

```
[
    object, dual, pointer_default(unique),
    uuid(69CB66A0-BF8E-11D1-9197-00207810C741),
]
interface IRRandomVariable : IDispatch
{
    // Mean
    [propget, id(0)]  HRESULT Mean([out, retval] double* retval);
    [propput, id(0)]  HRESULT Mean([in] double Mean);
    // Variance
    [propget, id(1)]  HRESULT Variance([out, retval] double* retval);
    [propput, id(1)]  HRESULT Variance([in] double Variance);
    // Standard deviation
    [propget, id(2)]  HRESULT Stdev([out, retval] double* retval);
    [propput, id(2)]  HRESULT Stdev([in] double Stdev);
    // Any moment
    [propget, id(3)]  HRESULT Moment([in] short Moment, [out, retval] double* retval);
    [propput, id(3)]  HRESULT Moment([in] short Moment, [in] double Value);
    // Any central moment
    [propget, id(4)]  HRESULT CentralMoment([in] short Moment, [out, retval] double* retval);
    [propput, id(4)]  HRESULT CentralMoment([in] short Moment, [in] double Value);
    // Calculate the PDF
    [id(5)]  HRESULT PDF([in] double x, [out, retval] double* retval);
    // Calculate the PMF
    [id(6)]  HRESULT PMF([in] double x, [out, retval] double* retval);
    // Calculate the CDF
    [id(7)]  HRESULT CDF([in] double x, [out, retval] double* retval);
    // Calculate the InverseCDF
    [id(8)]  HRESULT InverseCDF ([in] double F, [out, retval] double* retval);
    // The name of the distribution; maybe a stringified CLSID or ProgID
    [propget, id(9)]  HRESULT Type([out, retval] BSTR* retval);
    [propput, id(9)]  HRESULT Type([in] BSTR Distribution);
    // Number of parameters that parameterize this random variable
```

```
        [propget, id(10)] HRESULT ParameterCount([out, retval] short* retval);
        // Description of a parameter
        [propget, id(11)] HRESULT ParameterDesc([in] short Parameter, [out, retval] BSTR* retval);
        // The type of a parameter
        [propget, id(12)] HRESULT ParameterType([in] short Parameter, [out, retval] VARTYPE* vt);
        // The value of a parameter
        [propget, id(13)] HRESULT Parameter([in] short Parameter, [out, retval] VARIANT* retval);
        [propput, id(13)] HRESULT Parameter([in] short Parameter, [in] VARIANT Value);
    };
```

This interface has several groups of properties and methods. The first five properties return or set standard random variable moment-based properties. The methods PDF, PMF, CDF, and InverseCDF calculate functions of the random variable. The Type property returns the name of the distribution, such as "Geometric." The final four methods provide a generic interface for parameterizing a random variable. For instance, an exponential random variables has a single parameter of type double with the description "The rate of the exponential process," while a normal random variable has two parameters of type double with the descriptions "The mean of the normal random variable" and "The variance of the normal random variable."

Note that neither the name or the parameter choice is unique for any given random variable. Hence, it is up to the client to ensure that the correct interpretations are given. For instance, one normal random variable implementation might use the mean and variance as two parameters, while another might use the mean and standard deviation. Without direct dimension semantics, it is difficult to automatically ensure the proper interpretation of the parameters is followed.

Random variables further can support sampling from their distributions. This optional capability is expressed and exposed by a random variable object implementing the IRRandomSample interface, described here:

```
    [
        object, dual, pointer_default(unique),
        uuid(69CB66A1-BF8E-11D1-9197-00207810C741),
    ]
    interface IRRandomSample : IDispatch
    {
        // Returns a single sample from the distribution
        [id(201)]    HRESULT Sample([out, retval] double* retval);
        // Returns/sets the UniformGenerator for the sampling
        [propget, id(202)] HRESULT UniformGenerator([out, retval] IRRandomGenerator** retval);
        [propputref, id(202)] HRESULT UniformGenerator([in] IRRandomGenerator* Generator);
        // Returns a vector in a SAFEARRAY
        [id(203)]    HRESULT SampleVector([in] short Count, [out, retval] VARIANT* retval);
    };
```

The interface defines the property UniformGenerator, which is a reference to another object that implements a uniform 0-1 random number generation engine. The random variable object uses the UniformGenerator to generate a CDF value, and then can call InverseCDF to calculate the value of the random variable. The client can specify any UniformGenerator object, thereby giving the client complete control over the underlying random number generator. In particular, a single UniformGenerator object, and hence a single sequence of uniform random numbers, could be used for many random variable sampling engines. The two methods Sample and SampleVector should use the UniformGenerator to generate a single sample and a vector of samples, respectively.

The IRRandomGenerator interface is relatively simple:

```
[
    object, dual, pointer_default(unique),
    uuid(69CB66A2-BF8E-11D1-9197-00207810C741),
]
interface IRRandomGenerator : IDispatch
{
    // Returns a single sample
    [id(300)]    HRESULT Sample([out, retval] double* retval);
    // Sets the seed for randomization
    [id(301)]    HRESULT Seed([in] double seed);
    // Returns a sample vector in a SAFEARRAY
    [id(302)]    HRESULT SampleVector([in] short Count, [out, retval] VARIANT* retval);
    // Randomizes the seed
    [id(303)]    HRESULT RandomizeSeed();
};
```

The ability to sample an entire vector enables multidimensional random variables with correlated components. The IRRandomGenerator could also be implemented to generate non-uniform samples or correlated sequences.

A.3.4 A queueing system specification

The queueing system specification is a pair of interfaces that define characteristics of another custom data type, one that represents a simple single-stage queueing system. The system has a single arrival process, any number of servers operating according to a single service process, any maximum system size, and any population size. The interfaces use the random variable specification defined in section A.3.3, above.

The queueing system data type is very much like a solver. It has an "input" interface, IRQueueData, and an "output" interface, IRQueueStatistics. In fact, it would be easy for the queueing system implementation to be a solver object rather than just a custom data type, by exposing the solver interfaces described in section 3.5. Often, in fact, a custom data type is

indistinguishable from a very simple solver, and how the custom data type is used might determine how it is interpreted.

The input side of the custom data type is described by the IRQueueData interface:

```
[
    object, dual, pointer_default(unique),
    uuid(69CB66A3-BF8E-11D1-9197-00207810C741),
]
interface IRQueueData : IDispatch
{
    // Interarrival process
    [propget, id(401)]   HRESULT Interarrival([out, retval] IRRandomVariable** retval);
    [propput, id(401)]   HRESULT Interarrival([in] IRRandomVariable* Process);
    // Service process
    [propget, id(402)]   HRESULT Service([out, retval] IRRandomVariable** retval);
    [propput, id(402)]   HRESULT Service([in] IRRandomVariable* Process);
    // Number of servers (default 0=infinite)
    [propget, id(403)]   HRESULT ServerCount([out, retval] long* retval);
    [propput, id(403)]   HRESULT ServerCount([in] long Count);
    // Maximum allowable in the system (default 0=infinite)
    [propget, id(404)]   HRESULT MaxInSystem([out, retval] long* retval);
    [propput, id(404)]   HRESULT MaxInSystem([in] long Max);
    // Population size (default 0=infinite)
    [propget, id(405)]   HRESULT PopulationSize([out, retval] long* retval);
    [propput, id(405)]   HRESULT PopulationSize([in] long Count);
};
```

The output side of the custom data type is described by the IRQueueStatistics interface:

```
[
    object, dual, pointer_default(unique),
    uuid(69CB66A4-BF8E-11D1-9197-00207810C741),
]
interface IRQueueStatistics : IDispatch
{
    [propget, id(101)]   HRESULT Utilization([out, retval] double* retval);
    [propget, id(102)]   HRESULT NumberInSystem([out, retval] IRRandomVariable** retval);
    [propget, id(103)]   HRESULT NumberInQueue([out, retval] IRRandomVariable** retval);
    [propget, id(104)]   HRESULT TimeInSystem([out, retval] IRRandomVariable** retval);
    [propget, id(105)]   HRESULT TimeInQueue([out, retval] IRRandomVariable** retval);
    [propget, id(106)]   HRESULT ProbOfWaiting([out, retval] double* retval);
    [propget, id(107)]   HRESULT FractionOfTimeBusy([out, retval] double* retval);
};
```

Both interfaces derive from IDispatch, making it possible for an alternative implementation to support only the input or output side of the queueing system data type. However, if the implementation supports both, it should implement a version of IDispatch that exposes all of the properties of both IRQueueData and IRQueueStatistics. That is why none of the dispatch IDs collide in the two interfaces.

Within the context of the single arrival and single service process restrictions, the queueing system specification is fairly powerful. As most of the queue statistics are returned as their own random variables, a queueing system engine can provide an arbitrary degree of complexity in its results. For instance, one particular engine might accept only exponential arrival processes and service processes as inputs, but generate fully defined output random variable objects that can return their PDF and CDF values. Another engine might accept exponential arrival processes and general service processes but output random variable objects that implement only the mean and variance properties. Exponential networks could easily be created by tying together queueing system objects.

A.4 M/M/k Queue Examples

This section presents code samples associated with the M/M/k queueing analysis in Microsoft Excel, presented in section 4.2.3, page 244.

A.4.1 VBA macros

This section presents the two macros described in section 4.2.3.3, page 246. These are the ExpWaitMMk macro and the ServiceTimeFromWaitTimeMMk macro, and their associated helper macros.

```
'
' Checks arrival rate and service rates
' returns 0 if all okay, an error value otherwise
'
Function CheckRates(ArrivalRate, ServiceRate, NumServers) As Variant
    CheckRates = 0
    If ArrivalRate < 0 Or ServiceRate <= 0 Then
        CheckRates = CVErr(xlErrNum)
    ElseIf NumServers < 1 Then
        CheckRates = CVErr(xlErrNum)
    ElseIf ArrivalRate / (NumServers * ServiceRate) >= 1 Then
        CheckRates = CVErr(xlErrNum)
    End If
End Function
```

```
' Probability of queueing for an M/M/k queue.
' Kleinrock vol. 1 (3.40)
'
Function ProbOfQueueingMMk(ArrivalRate, ServiceRate, NumServers)
    Dim rhoRatio As Double                        ' = rho*k = lambda/mu
    Dim K As Integer
    Dim InvProbOfZeroCustomers As Double      ' = 1/P0
    Dim UnnrmlzdPrbOfQueueing As Double       ' = Pq/P0
    Dim UnnrmlzdPrbOfNotQueueing As Double ' = (1-Pq)/P0
    Dim factorial As Double
    Dim rhoPower As Double
    Dim rho As Double
    Dim errorCode

    ' check parameters for errors
    errorCode = CheckRates(ArrivalRate, ServiceRate, NumServers)
    If IsError(errorCode) Then
        ProbOfQueueingMMk = CVErr(errorCode)
        Exit Function
        End If

    ' truncate fractional part of number of servers
    NumServers = Int(NumServers)

    rhoRatio = ArrivalRate / ServiceRate
    rho = rhoRatio / NumServers

    UnnrmlzdPrbOfNotQueueing = 1
    factorial = 1
    rhoPower = 1

    For K = 1 To NumServers - 1
        rhoPower = rhoPower * rhoRatio
        factorial = factorial * K
        UnnrmlzdPrbOfNotQueueing = UnnrmlzdPrbOfNotQueueing + rhoPower / factorial
    Next K

    rhoPower = rhoPower * rhoRatio
    factorial = factorial * NumServers

    UnnrmlzdPrbOfQueueing = rhoPower / (factorial * (1 - rho))

    InvProbOfZeroCustomers = (UnnrmlzdPrbOfNotQueueing + UnnrmlzdPrbOfQueueing)
    ProbOfQueueingMMk = UnnrmlzdPrbOfQueueing / InvProbOfZeroCustomers
End Function ' ProbOfQueueingMMk
```

```
' Expected waiting time for M/M/1 queue function
'
' = (rho/mu)/(1-rho)
' = lambda/[mu*(mu-lambda)]
'
' (which is more robust when (mu-lambda) goes to zero?
'
Function ExpWaitMM1(ArrivalRate, ServiceRate)
    Dim errorCode
    errorCode = CheckRates(ArrivalRate, ServiceRate, 1)
    If IsError(errorCode) Then
        ExpWaitMM1 = CVErr(errorCode)
    Else
        ExpWaitMM1 = ArrivalRate / (ServiceRate * (ServiceRate - ArrivalRate))
    End If
End Function ' ExpWaitMM1

' Expected waiting time for M/M/k queue function
' NumServers defaults to 1 if left out.
'
Function ExpWaitMMk(ArrivalRate, ServiceRate, Optional NumServers)
    Dim ProbOfQueueing

    ' for one server, use the one-server routine
    If IsMissing(NumServers) Or NumServers = 1 Then
        ExpWaitMMk = ExpWaitMM1(ArrivalRate, ServiceRate)
        Exit Function
    End If

    ' truncate fractional part of number of servers
    NumServers = Int(NumServers)

    ' ProbOfQueueing is an external routine, so does its own error checking
    ' of parameters.
    ProbOfQueueing = ProbOfQueueingMMk(ArrivalRate, ServiceRate, NumServers)
    If IsError(ProbOfQueueing) Then
        ExpWaitMMk = ProbOfQueueing
        Exit Function
    End If

    ExpWaitMMk = ProbOfQueueing / (NumServers * ServiceRate - ArrivalRate)
End Function ' ExpWaitMMk
```

```
Function ServiceRateFromWaitTimeMMk(ArrivalRate As Double, NumServers As Integer, _
                                    DesiredWait As Double, Optional Tolerance)

Dim ServiceRate As Double        ' mu
Dim stepsize As Double
Dim CurrWait As Double
Dim CurrIteration As Integer

    ' do some error checking
    If DesiredWait <= 0 Then
       ServiceRateFromWaitTimeMMk = [#NUM!]
       Exit Function
    End If
    If NumServers <= 0 Then
       ServiceRateFromWaitTimeMMk = [#NUM!]
       Exit Function
    End If
    If ArrivalRate <= 0 Then
       ServiceRateFromWaitTimeMMk = [#NUM!]
       Exit Function
    End If

    ' initialize Tolerance if left out
    If IsMissing(Tolerance) Then
       Tolerance = 0.0000001
    End If

    ' initial service rate set to minimum to satisfy ( rho < 1 )
    ServiceRate = ArrivalRate / NumServers

    ' initial step size value and iteration number
    stepsize = 1
    CurrIteration = 0

    ' iterate through, increasing service rate until wait time satisfied
    CurrWait = ExpWaitMMk(ArrivalRate, ServiceRate + stepsize, NumServers)
    While (Abs(DesiredWait - CurrWait) / DesiredWait) > Tolerance
       CurrWait = ExpWaitMMk(ArrivalRate, ServiceRate + stepsize, NumServers)
       If CurrWait >= DesiredWait Then
        ServiceRate = ServiceRate + stepsize
        CurrIteration = CurrIteration + 1

       ' if too many iterations at same step size, take larger steps
        If CurrIteration = 10 Then
            stepsize = stepsize * 10
            CurrIteration = 0
```

```
            End If
        Else

        ' current step would be too large, so make it smaller and reset counter
            stepsize = stepsize / 10
            CurrIteration = 0
        End If
    Wend

    ' output final correct value
    ServiceRateFromWaitTimeMMk = ServiceRate
End Function
```

The macro ServiceRateFromWaitTimeMMk initializes the service rate to its minimum value to ensure that the utilization equals one. Then it repeatedly increases the service rate, in varying step sizes depending on progress, until the desired waiting time is found (within the specified tolerance). In particular, for each iteration the service time is increased by the step size, which decreases the current waiting time. If the current waiting time is decreased ten times without becoming less than the desired waiting time, then the step size is presumed to be too small and it is increased by a factor of ten. Once the service level is so high that the current waiting time is less than the desired waiting time, then the algorithm backs up a step and then decreases the step size by a factor of ten.

A.4.2 Framework solver

This section presents the VBA macro necessary to invoke the framework solver version of the M/M/k queue, as described in section 4.2.3.5, page 247. The framework solver uses the random variable and queueing system extensions from section A.3.4, page 317, for M/M/k queueing systems. The VBA code to use this calculation looks like this:

```
'
' Expected waiting time for M/M/k queue function
' NumServers defaults to 1 if left out.
'
Function FrameExpWaitMMk(ArrivalRate, ServiceRate, Optional NumServers)
    Dim QueueSystem As MMQueue
    Dim QueueStats As IRQueueStatistics
    Dim interarrival As IRRandomVariable
    Dim service As IRRandomVariable

    If IsMissing(NumServers) Then
        NumServers = 1
    End If
```

```
    Set interarrival = CreateObject("RANDVAR.Exponential.1")
    Set service = CreateObject("RANDVAR.Exponential.1")
    Set QueueSystem = CreateObject("MMQUEUE.Queue.1")

    interarrival.Parameter(1) = ArrivalRate
    service.Parameter(1) = ServiceRate
    Set QueueSystem.interarrival = interarrival
    Set QueueSystem.service = service
    QueueSystem.ServerCount = NumServers

    Set QueueStats = QueueSystem
    FrameExpWaitMMk = QueueStats.TimeInQueue.Mean
End Function
```

This code seems longer than it might otherwise need to be, but that is the trade-off for the flexibility provided by a queueing system that allows arbitrary interarrival and service processes as inputs and generates arbitrary random variables as outputs. Furthermore, the savvy developer can include an Excel Add-in that hides this code, thereby making the solver as easy to use now as in the Add-in case while also leaving the solver in a separate executable and thereby making it available to all clients that understand the framework.

Bibliography

1. Adler, R.M., "Emerging Standards for Component Software," *IEEE Computer,* 28:3 (1995), 68–77.

2. Banerjee, S., and A. Basu, "A Knowledge Based Framework for Selecting Management Science Models," *Proceedings of the Twenty-Third Annual Hawaii International Conference on System Sciences,* Vol. III, IEEE Computer Society Press, Los Alamitos, CA, (1990), 484–493.

3. Banerjee, S., and A. Basu, "Model type selection in an integrated DSS environment," *Decision Support Systems,* 9:1 (1993), 75–89.

4. Bhargava, H.K., S.O. Kimbrough, and R. Krishnan, "Unique Names Violations, a Problem for Model Integration or You Say Tomato, I Say Tomahto," *ORSA Journal on Computing,* 3:2 (1991), 107–120.

5. Bhargava, H.K., and R. Krishnan, "A Formal Approach for Model Formulation in a Model Management System," *Proceedings of the Twenty-Third Annual Hawaii International Conference on System Sciences,* Vol. III, IEEE Computer Society Press, Los Alamitos, CA, (1990), 453–462.

6. Blanning, R.W., "Model management systems: An overview," *Decision Support Systems,* 9:1 (1993), 9–18.

7. Booch, G., *Object-Oriented Analysis and Design With Applications,* 2nd ed., Addison-Wesley, Reading, MA, 1994.

8. Box, D., "Q&A: ActiveX/COM," *Microsoft Systems Journal,* 12:5 (1997), 95–110.

9. Box, D., *Essential COM,* Addison-Wesley, Reading, MA, 1998.

10. Bradley, G.H., and R.D. Clemence, Jr., "Model Integration With a Typed Executable Modeling Language," *Proceedings of the Twenty-First Annual Hawaii International Conference on System Sciences,* Vol. III, IEEE Computer Society Press, Los Alamitos, CA, (1988), 403–410.

11. Brockschmidt, K., *Inside OLE*, 2nd ed., Microsoft Programming Series, Microsoft Press, Redmond, WA, 1995.

12. Brown, K., "VB can leverage iid_is and GUIDs – here's how!" 1997, <http://microsoft.ease.lsoft.com/scripts/wa.exe?A2=ind9710a&L=dcom&D=0&P=21462> (4 March 1998).

13. Brown, S., "The Fall of Software's Aristocracy: Realizing the Potential of Development," in *The Future of Software*, The MIT Press, Cambridge, MA, (1995), 157–175.

14. Buschmann, F., R. Meunier, H. Rohnert, P. Sommerlad, and M. Stal, *Pattern-Oriented Software Architecture: A System of Patterns*, John Wiley & Sons, West Sussex, England, 1996.

15. Chappell, D., *Understanding ActiveX and OLE*, Microsoft Press, Redmond, WA, 1996.

16. Coffee, P., "JVM: Is It Living Up to Java's Potential?" *PC Week*, 15:11 (1998), 25, 32.

17. CPLEX Optimization, Inc., *Using the CPLEX Callable Library, including Using the CPLEX Base System, with CPLEX Barrier and Mixed Integer Solver Options*, version 4.0, 1995.

18. De, S., and A.B. Whinston, "A Framework for Integrated Problem Solving in Manufacturing," *IIE Transactions*, 18:3 (1986), 286–297.

19. Deger, R., "Worldwide effort cracks DES," *PC Week Online*, 19 June 1997, <http://www.zdnet.com/pcweek/news/0616/19mdes.html> (26 January 1998).

20. Dellarocas, C.N., "A Coordination Perspective on Software Architecture: Towards a Design Handbook for Integrating Software Components," Ph.D. thesis, Massachusetts Institute of Technology, Department of Electrical Engineering and Computer Science, 1996.

21. Dellarocas, C.N., "Toward a Design Handbook for Integrating Software Components," presented at the Fifth International Symposium on Assessment of Software Tools and Technologies, 1997.

22. Digital Equipment Corporation, "DIGITAL Demonstrates Commitment to Microsoft COM," 26 January 1998, <http://www.digital.com/PRW03Q/> (19 April 1998).

23. Dolk, D.R., "An introduction to model integration and integrated modeling environments," *Decision Support Systems*, 10:3 (1993), 249–254.

24. Dolk, D.R., and J.E. Kottemann, "Model integration and a theory of models," *Decision Support Systems*, 9:1 (1993), 51–63.

25. Dyck, T., "OLE DB: A Worthy Successor to ODBC," *PC Week*, 14:3 (1997), 69.

26. Eck, R.D., A. Philippakis, and R.G. Ramirez, "Solver Representation for Model Management Systems," *Proceedings of the Twenty-Third Annual Hawaii International Conference on System Sciences*, Vol. III, IEEE Computer Society Press, Los Alamitos, CA, (1990), 474–483.

27. Finkelberg, H., and S.C. Graves, "Franz Edelman Award for Management Science Achievement," *Interfaces*, 27:1 (1997), 1–6.

28. Fourer, R., "Software Survey: Linear Programming," *OR/MS Today*, 24:2 (1997), 54–63.

29. Fowler, M., and K. Scott, *UML Distilled: Applying the Standard Object Modeling Language,* Addison-Wesley, Reading, MA, 1997.

30. Gamma, E., R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software,* Addison-Wesley, Reading, MA, 1995.

31. Gates, W.H., III, *The Road Ahead,* 2nd ed., Penguin Books, London, England, 1996.

32. Geoffrion, A.M., "An Introduction to Structured Modeling," *Management Science,* 33:5 (1987), 547–588.

33. Geoffrion, A.M., "Computer-based Modeling environments," *European Journal of Operations Research,* 41:1 (1989), 33–43.

34. Geoffrion, A.M., "Reusing Structured Models via Model Integration," *Proceedings of the Twenty-Second Annual Hawaii International Conference on System Sciences,* Vol. III, IEEE Computer Society Press, Los Alamitos, CA, (1989), 601–611.

35. Geoffrion, A.M., "The Formal Aspects of Structured Modeling," *Operations Research,* 37:1 (1989), 30–51.

36. Geoffrion, A.M., "A Taxonomy of Indexing Structures for Mathematical Programming Modeling Languages," *Proceedings of the Twenty-Third Annual Hawaii International Conference on System Sciences,* Vol. III, IEEE Computer Society Press, Los Alamitos, CA, (1990), 463–473.

37. Geoffrion, A.M., "Structured Modeling: Survey and Future Research Directions," *Interactive Transactions of OR/MS,* 1:1 (1996), <http://catt.bus.okstate.edu:80/itorms/currvol/vol1/papers/geoffrion/docs/csts/index.htm> (22 April 1998).

38. Goldberg, A.V., "Andrew Goldberg's Network Optimization Library," <http://www.neci.nj.nec.com/homepages/avg/soft/soft.html> (19 May 1998).

39. Gonsalves, A., and M. Moeller, "IONA Out to Bridge Gap," *PC Week,* 14:52 (1997), 6.

40. Goul, M., C. Kuo, and T.E. Sandman, "Towards Synergizing the Active Object, Software Maintenance, and Algorithm Synthesis Metaphors for Integrated Modeling Environments," *Proceedings of the Twenty-Fifth Annual Hawaii International Conference on System Sciences,* Vol. III, IEEE Computer Society Press, Los Alamitos, CA, (1992), 437–448.

41. Grau, R., and J. Barcelo, "GETRAM: A Generic Environment for Traffic Analysis and Modeling," *IFAC Transportation Systems: Theory and Application of Advanced Technology,* Vol. 2, Elsevier Science, Oxford, UK, (1995), 701–706.

42. Graves, S.C., C. Gutierrez, M. Pulwer, H. Sidhu, and G. Weihs, "Optimizing Monsanto's Supply Chain Under Uncertain Demand," *Annual Conference Proceedings—Council of Logistics Management,* Orlando, FL (1996), 501–516.

43. Graves, S.C., and S.P. Willems, "Strategic Safety Stock Placement in Supply Chains," *Proceedings of the 1996 MSOM Conference,* Dartmouth College, Hanover, NH, (1998), 299–304.

44. Gutierrez, C.J., "Development and Application of a Linear Programming Model to Optimize Production and Distribution of a Manufacturing Company," M.S. thesis, Massachusetts Institute of Technology, Department of Civil and Environmental Engineering, 1996.

45. Huh, S., "Modelbase Constructions with Object-Oriented Constructs," *Decision Sciences,* 24:2 (1993), 409–434.

46. Huhns, M.N., N. Jacobs, T. Ksiezyk, W. Shen, M.P. Singh, and P.E. Cannata, "Enterprise Information Modeling and Model Integration in Carnot," *Enterprise Integration Modeling: Proceedings of the First International Conference,* The MIT Press, Cambridge, MA, (1992), 290–299.

47. IBM Corporation, Inc., *Optimization Subroutine Library: Guide and Reference, Release 2,* 3rd ed., 1991.

48. Islam, N., and R.H. Campbell, "Latest Developments in Operating Systems," *Communications of the ACM,* 39:9 (1996), 38–40.

49. Jones, C.V., "An Integrated Modeling Environment Based on Attributed Graph and Graph-Grammars," *Decision Support Systems,* 10:3 (1993), 255–275.

50. Jones, C.V., "Attributed Graphs, Graph-Grammars, and Structured Modeling," *Annals of Operations Research,* 38 (1992), 281–324. (Special volume on Model Management in Operations Research, edited by B. Shetty, H. Bhargava, and R. Krishnan.)

51. Jones, C.V., "Visualization and Optimization," *ORSA Journal on Computing,* 6:3 (1994), 221–257.

52. Jordan, W.C., and S.C. Graves, "Principles on the Benefits of Manufacturing Process Flexibility," *Management Science,* 41:4 (1995), 577–594.

53. Jorgenson, B.R., "Model Repository Technology for Model Integration," *Enterprise Integration Modeling: Proceedings of the First International Conference,* The MIT Press, Cambridge, MA, (1992), 419–429.

54. Karsai, G., "A Configurable Visual Programming Environment," *Computer,* 28:3 (1995), 36–44.

55. Klein, M., and R. Traunmüller, "Architecture and User Interface of KB-DSS Development Environment," *Proceedings of the Twenty-Sixth Annual Hawaii International Conference on System Sciences,* Vol. III, IEEE Computer Society Press, Los Alamitos, CA, (1993), 108–118.

56. Kleinrock, L., *Queueing Systems,* Volume 1: Theory, John Wiley & Sons, New York, 1975.

57. Kottemann, J.E., and D.R. Dolk, "Model Integration and Modeling Languages: A Process Perspective," *Information Systems Research,* 3:1 (1992), 1–16.

58. Lakos, J., *Large Scale C++ Software Design,* Addison-Wesley, Reading, MA, 1996.

59. Larson, R.C., "Perspectives on Queues: Social Justice and the Psychology of Queueing," *Operations Research,* 35:6 (1987), 895–905.

60. Lazimy, R., "Object-Oriented Modeling Support System: Model Representation and Incremental Modeling," *Proceedings of the Twenty-Sixth Annual Hawaii International Conference on System Sciences,* Vol. III, IEEE Computer Society Press, Los Alamitos, CA, (1993), 445–459.

61. Leebaert, D., editor, *The Future of Software,* The MIT Press, Cambridge, MA, 1995.

62. Lenard, M.L., "An object-oriented approach to model management," *Decision Support Systems,* 9:1 (1993), 67–73.

63. Liang, T., "Integrating Model Management with Data Management in Decision Support Systems," *Decision Support Systems,* 1:3 (1985), 221–232.

64. Ma, P., F.H. Murphy, and E.A. Stohr, "An Implementation of LPFORM," *ORSA Journal on Computing,* 8:4 (1996), 383–401.

65. McCright, J., "OLAP Council Readies Interoperability Standard," *ZDNet,* 16 January 1998, <http://www.zdnet.com/zdnn/content/pcwk/1503/270779.html> (21 March 1998).

66. McKay, K.N., D.B. Kletter, and S.C. Graves, "OMAC: A System for Operations Modeling and Analysis," *Annals of Operations Research,* 72 (1997), 241–264. (Special volume on Interface between IS and OR, Part II: Systems for Models, edited by R. Ramesh and H.R. Rao.)

67. McLaren, B.M., P.M. Neuss, and O. De Groote, "The Integrated Modeling Package (IMP): An Object Oriented Module for Manufacturing Simulation," *Simulation Environments and Symbol and Number Processing on Multi and Array Processors: Proceedings of the European Simulation Multiconference,* Society for Computer Simulation, Ghent, Belgium, (1988), 231–236.

68. Mertins, K., W. Süssenguth, and R. Jochem, "An Object Oriented Method for Integrated Enterprise Modeling as a Basis for Enterprise Coordination," *Enterprise Integration Modeling: Proceedings of the First International Conference,* The MIT Press, Cambridge, MA, (1992), 249–258.

69. Meyer, B., *Object-Oriented Software Construction,* 2nd ed., Prentice Hall PTR, Upper Saddle River, NJ, 1997.

70. Microsoft Corporation, "Microsoft Scripting Technologies: Hosting Information," 1998, <http://www.microsoft.com/scripting/hosting/hosting.htm> (23 March 1998).

71. Microsoft Corporation, "Microsoft Visual Basic Advances RAD Industry Leadership Position," 28 January 1998, <http://www.microsoft.com/corpinfo/press/1998/Jan98/vb5mompr.htm> (29 March 1998).

72. Microsoft Corporation, "OLE DB for OLAP," <http://www.microsoft.com/data/oledb/olap/> (19 April 1998).

73. Microsoft Corporation, "OLE Industry Solutions," 1997, <http://www.microsoft.com/oledev/olemkt/olesol.htm> (15 April 1998).

74. Mili, F., and F.A. Cioch, "Documenting Decision Models for Informed and Confident Decisions," *Proceedings of the Twenty-Third Annual Hawaii International Conference on System Sciences,* Vol. III, IEEE Computer Society Press, Los Alamitos, CA, (1990), 494–503.

75. Mili, F., and I. Szoke, "Assisted Model Selection Evaluation and Comparison," *Proceedings of the Twenty-Fifth Annual Hawaii International Conference on System Sciences,* Vol. III, IEEE Computer Society Press, Los Alamitos, CA, (1992), 485–493.

76. Mowbray, T.J., and W.A. Ruh, *Inside CORBA: Distribution Object Standards and Applications,* Addison-Wesley, Reading, MA, 1997.

77. Muhanna, W.A., "An object-oriented framework for model management and DSS development," *Decision Support Systems,* 9:2 (1993), 217–339.

78. Myers, W., "Taligent's CommonPoint: The Promise of Objects," *Computer,* 28:3 (1995), 78–83.

79. Nahmias, S., *Production and Operations Analysis,* Richard D. Irwin, Inc., Homewood, IL, 1989.

80. Object Management Group, "Object Management Group Adopts Unified Modeling Language and Meta Object Facility Specifications," 1997, <http://www.omg.org/news/pr97/umlpr.htm> (17 February 1998).

81. Park, S.J., and H.D. Kim, "Constraint-based metaview approach for modeling environment generation," *Decision Support Systems,* 9:4 (1993), 325–348.

82. Piela, P., R. McKelvey, and A. Westerberg, "An Introduction to ASCEND: Its Language and Interactive Environment," *Proceedings of the Twenty-Fifth Annual Hawaii International Conference on System Sciences,* Vol. III, IEEE Computer Society Press, Los Alamitos, CA, (1992), 449–461.

83. Pietrek, M., *Windows® 95 System Programming SECRETS™,* IDG Books Worldwide, Inc., Foster City, CA, 1995.

84. Raghunathan, S., "Towards Computer-assisted Qualitative Analysis for Model Development: Theory and Algorithms," *Proceedings of the Twenty-Fifth Annual Hawaii International Conference on System Sciences,* Vol. III, IEEE Computer Society Press, Los Alamitos, CA, (1992), 473–484.

85. Ramirez, R.G., and E. Lin, "Subscript-Free Indexing in a Mathematical Programming Language," *Proceedings of the Twenty-Sixth Annual Hawaii International Conference on System Sciences,* Vol. III, IEEE Computer Society Press, Los Alamitos, CA, (1993), 424–433.

86. Ramirez, R.G., C. Ching, and R.D. St. Louis, "Independence and mappings in model-based decision support systems," *Decision Support Systems,* 10:3 (1993), 341–358.

87. Rational Software Corporation, "UML resource center," 1998, <http://www.rational.com/uml/index.shtml> (17 February 1998).

88. Raymond, E., comp., "Jargon Dictionary – daemon," *The Jargon Dictionary,* 24 July 1996, <http://www.netmeg.net/jargon/terms/d/daemon.html> (17 February 1998).

89. Richter, J., *Advanced Windows™: The Developer's Guide to the Win32® API for Windows NT™ 3.5 and Windows 95,* Microsoft Press, Redmond, WA, 1995.

90. Rogerson, D., *Inside COM,* Microsoft Press, Redmond, WA, 1997.

91. Ruark, J., "Using the Monsanto Software," Operations Research Center working paper OR 325-98, Massachusetts Institute of Technology, 1998.

92. Ruark, J., "Reference Guide to the Framework Presented in 'Implementing Reusable Solvers: An Object-Oriented Framework for Operations Research Algorithms'," Operations Research Center working paper OR 328-98, Massachusetts Institute of Technology, 1998.

93. Rumbaugh, J., M. Blaha, W. Premerlani, F. Eddy, and W. Lorensen, *Object-Oriented Modeling and Design,* Prentice Hall, Englewood Cliffs, NJ, 1991.

94. Savage, S., "Weighing the PROS and CONS of Decision Technology in Spreadsheets," *OR/MS Today,* 24:1 (1997), 42–48.

95. Smith, T.R., J. Su, A.E. Abbadi, D. Agrawal, G. Alonso, and A. Saran, "Computational Modeling Systems," *Information Systems,* 20:2 (1995), 127–153.

96. Steiger, D., R. Sharda, and B. Leclaire, "Graphical Interfaces for Network Modeling: A Model Management System Perspective," *ORSA Journal on Computing,* 5:3 (1993), 275–291.

97. Sun Microsystems, Inc., "JavaBeans: The Only Component Architecture for Java TM," 1997, <http://www.javasoft.com/beans/> (17 February 1998).

98. Sventek, J., "The Distributed Application Architecture," *Enterprise Integration Modeling: Proceedings of the First International Conference,* The MIT Press, Cambridge, MA, (1992), 481–492.

99. Tolvanen, J., P. Marttiin, and K. Smolander, "An Integrated Model for Information Systems Modeling," *Proceedings of the Twenty-Sixth Annual Hawaii International Conference on System Sciences,* Vol. III, IEEE Computer Society Press, Los Alamitos, CA, (1993), 470–479.

100. Tung, L., R.G. Ramirez, and R.D. St. Louis, "Model Integration In An Object-Oriented Model Management System," *Proceedings of the Twenty-Fourth Annual Hawaii International Conference on System Sciences,* Vol. III, IEEE Computer Society Press, Los Alamitos, CA, (1991), 284–290.

101. Varhol, P.D., "Trends in Operating System Design," *Dr. Dobb's Journal,* 19:5 (1994), 18–20, 22, 26–27.

102. Weston, R., "The Fifth Generation," *PC Week,* 14:1 (1997), 103.

103. Wilkes, M.V., "The Long-Term Future of Operating Systems," *Communications of the ACM,* 35:11 (1992), 23–24, 112.

104. Wolfram von Eschenbach, *Parzival,* translated by A.T. Hatto, Penguin Books, London, England, 1980.

Intentionally blank, this page left.

BIBLIOGRAPHY OF APPLIED SOLUTIONS

[AAASAS]
Vasquez-Marquez, A., "American Airlines Arrival Slot Allocation System (ASAS)," *Interfaces,* 21:1 (1991), 42–61.

[AACREW]
Anbil, R., E. Gelman, B. Patty, and R. Tanga, "Recent Advances in Crew-Pairing Optimization at American Airlines," *Interfaces,* 21:1 (1991), 62–74.

[AAYIELD]
Smith, B.C., J.F. Leimkuhler, and R.M. Darrow, "Yield Management at American Airlines," *Interfaces,* 22:1 (1992), 8–31.

[ABB]
Gensch, D.H., N. Aversa, and S.P. Moore, "A Choice-Modeling Market Information System That Enabled ABB Electric to Expand Its Market Share," *Interfaces,* 20:1 (1990), 6–25.

[AT&T]
Spencer, T., III, A.J. Brigandi, D.R. Dargon, and M.J. Sheehan, "AT&T's Telemarketing Site Selection System Offers Customer Support," *Interfaces,* 20:1 (1990), 83–96.

[AT&TCAPS]
Brigandi, A.J., D.R. Dargon, M.J. Sheehan, and T. Spencer III, "AT&T's Call Processing Simulator (CAPS) Operational Design for Inbound Call Centers," *Interfaces,* 24:1 (1994), 6–28.

[AT&TCLS]
Curnow, G., G. Kochman, S. Meester, D. Sarkar, and K. Wilton, "Automating Credit and Collections Decisions at AT&T Capital Corporation," *Interfaces,* 27:1 (1997), 29–52.

[BELLCORE]
Hoadley, B., P. Katz, and A. Sadrian, "Improving the Utility of the Bellcore Consortium," *Interfaces,* 23:1 (1993), 27–43.

[BELLCOREPDSS]
Katz, P., A. Sadrian, and P. Tendick, "Telephone Companies Analyze Price Quotations with Bellcore's PDSS Software," *Interfaces,* 24:1 (1994), 50–63.

[BETHLEHEM]
Vasko, F.J., F.E. Wolf, K.L. Stott, J.W. Scheirer, "Selecting Optimal Ingot Sizes for Bethlehem Steel," *Interfaces,* 19:1 (1989), 68–84.

[CAROLINA]
Secton, T.R., S. Sleeper, and R.E. Taggart, Jr., "Improving Pupil Transportation in North Carolina," *Interfaces,* 24:1 (1994), 87–103.

[CHINA]
Kuby, M., S. Qingqi, T. Watanatada, S. Xufei, X. Zhijun, C. Wei, Z. Chuntai, Z. Dadi, Y. Xiaodong, P. Cook, T. Friesz, S. Neuman, L. Fatang, R Qiang, W. Xusheng, and G. Shenhuai, "Planning China's Coal and Electricity Delivery System," *Interfaces,* 25:1 (1995), 41–68.

[CITGO]
Klingman, D., N. Phillips, D. Steiger, and W. Young, "The Successful Deployment of Management Science through Citgo Petroleum Corporation," *Interfaces,* 17:1 (1987), 4–25.

[DELTA]
Subramanian, R., R.P. Scheff, Jr., J.D. Quillinan, D.S. Wiper, and R.E. Marsten, "Coldstart: Fleet Assignment at Delta Air Lines," *Interfaces,* 24:1 (1994), 104–120.

[DESERTSTORM]
Hilliard, M.R., R.S. Solanki, C. Liu, I.K. Busch, G. Harrison, and R.D. Kraemer, "Scheduling the Operation Desert Storm Airlift: An Advanced Automated Scheduling Support System," *Interfaces,* 22:1 (1992), 131–146.

[DIGITAL]
Arntzen, B.C., G.G. Brown, T.P. Harrison, and L.L. Trafton, "Global Supply Chain Management at Digital Equipment Corporation," *Interfaces,* 25:1 (1995), 69–93.

[DRG]
Fetter, R.B., "Diagnosis Related Groups: Understanding Hospital Performance," *Interfaces,* 21:1 (1991), 6–26.

[EGYPT]
El Sherif, H., "Managing Institutionalization of Strategic Decision Support for the Egyptian Cabinet," *Interfaces,* 20:1 (1990), 97–114.

[ENGLAND]
Carr-Hill, R.A., G. Hardman, S. Martin, S. Peacock, T.A. Sheldon, and P.C. Smith, "A New Formula for Distributing Hospital Funds in England," *Interfaces,* 27:1 (1997), 53–70.

[EPRI]
Chao, H., S.W. Chapel, C.E. Clark, Jr., P.A. Morris, M.J. Sandling, and R.C. Grimes, "EPRI Reduces Fuel Inventory Costs in the Electric Utility Industry," *Interfaces,* 19:1 (1989), 48–67.

[GECAPITAL]
Makuch, W.M., J.L. Dodge, J.G. Ecker, D.C. Granfors, and G.J. Hahn, "Managing Consumer Credit Delinquency in the U.S. Economy: A Multi-Billion Dollar Management Science Application," *Interfaces,* 22:1 (1992), 90–109.

[GM]
Blumenfeld, D.E., L.D. Burns, C.F. Daganzo, M.C. Frick, and R.W. Hall, "Reducing Logistics Costs at General Motors," *Interfaces,* 17:1 (1987), 26–47.

[GRI]
Burnett, W.M., D.J. Monetta, and B.G. Silverman, "How the Gas Research Institute (GRI) Helped Transform the U.S. Natural Gas Industry," *Interfaces,* 23:1 (1993), 44–58.

[GTE]
Jack, C., S. Kai, and A. Shulman, "NETCAP—An Interactive Optimization System for GTE Telephone Network Planning," *Interfaces,* 22:1 (1992), 72–89.

[HANSHIN]
Yoshino, T., T. Sasaki, T. Hasegawa, "The Traffic-Control System on the Hanshin Expressway," *Interfaces,* 25:1 (1995), 94–108.

[HARRIS]
Leachman, R.C., R.F. Benson, C. Liu, and D.J. Raar, "IMPReSS: An Automated Production-Planning and Delivery-Quotation System at Harris Corporation—Semiconductor Sector," *Interfaces,* 26:1 (1996), 6–37.

[HASTUS]
Blais, J., J. Lamont, and J. Rousseau, "The HASTUS Vehicle and Manpower Scheduling System at the Société de transport de la Communauté urbaine de Montréal," *Interfaces,* 20:1 (1990), 26–42.

[HOMART]
Bean, J.C., C.E. Noon, and G.J. Salton, "Asset Divestiture at Homart Development Company," *Interfaces,* 17:1 (1987), 48–64.

[IBMLMS]
Sullivan, G., and K. Fordyce, "IBM Burlington's Logistics Management System," *Interfaces,* 20:1 (1990), 43–64.

[IBMOPT]
Cohen, M., P.V. Kamesam, P. Kleindorfer, H. Lee, and A. Tekerian, "Optimizer: IBM's Multi-Echelon Inventory System for Managing Service Logistics," *Interfaces,* 20:1 (1990), 65–82.

[ISRAEL]
Spector, Y., and L. Marom, "SQOM-2: The Israeli Air Force's Air Power Multiplier," *Interfaces,* 26:1 (1996), 75–84.

[KEYCORP]
Kotha, S.K., M.P. Barnum, and D.A. Bowen, "KeyCorp Service Excellence Management System," *Interfaces,* 26:1 (1996), 54–74.

[KODAK]
Farley, A.A., "Planning the Cutting of Photographic Color Paper Rolls for Kodak (Australasia) Pty. Ltd.," *Interfaces,* 21:1 (1991), 92–106.

[KUWAIT]

Elimam, A.A., M. Girgis, and S. Kotob, "A Solution to Post Crash Debt Entanglements in Kuwait's al-Manakh Stock Market," *Interfaces,* 27:1 (1997), 89–106.

[LLBEAN]

Quinn, P., B. Andrews, and H. Parsons, "Allocating Telecommunications Resources at L. L. Bean, Inc.," *Interfaces,* 21:1 (1991), 75–91.

[LTVSTEEL]

Box, R.E., and D.G. Herbe, Jr., "A Scheduling Model for LTV Steel's Cleveland Works' Twin Strand Continuous Slab Caster," *Interfaces,* 18:1 (1988), 42–56.

[MARRIOTT]

Wind, J., P.E. Green, D. Shifflet, and M. Scarbrough, "*Courtyard by Marriott:* Designing a Hotel Facility with Consumer-Based Marketing Models," *Interfaces,* 19:1 (1989), 25–47.

[MERIT]

Flowers, A.D., "The Modernization of Merit Brass," *Interfaces,* 23:1 (1993), 97–108.

[METRON]

Stone, L.D., "Search for the *SS Central America:* Mathematical Treasure Hunting," *Interfaces,* 22:1 (1992), 32–54.

[MOBIL]

Brown, G.G., C.J. Ellis, G.W. Graves, and D. Ronen, "Real-Time, Wide Area Dispatch of Mobil Tank Trucks," *Interfaces,* 17:1 (1987), 107–120.

[MONSANTO]

Graves, S.C., C. Gutierrez, M. Pulwer, H. Sidhu, and G. Weihs, "Optimizing Monsanto's Supply Chain Under Uncertain Demand," *Annual Conference Proceedings—Council of Logistics Management,* Orlando, FL (1996), 501–516.

[MOSLS]

Eiger, A., J.M. Jacobs, D.B. Chung, and J.L. Selsor, "The U.S. Army's Occupational Specialty Manpower Decision Support System," *Interfaces,* 18:1 (1988), 57–73.

[NATIONAL]

Geraghty, M.K., and E. Johnson, "Revenue Management Saves National Car Rental," *Interfaces,* 27:1 (1997), 107–127.

[NAVANLINES]

Powell, W.B., Y. Sheffi, K.S. Nickerson, K. Butterbaugh, and S. Atherton, "Maximizing Profits for North American Van Lines' Truckload Division: A New Framework for Pricing and Operations," *Interfaces,* 18:1 (1988), 21–41.

[NEWHAVEN]

Kaplan, E.H., and E. O'Keefe, "Let the Needles Do the Talking! Evaluating the New Haven Needle Exchange," *Interfaces,* 23:1 (1993), 7–26.

[NHFIRE]

Swersey, A.J., L. Goldring, and E.D. Geyer, Sr., "Improving Fire Department Productivity: Merging Fire and Emergency Medical Units in New Haven," *Interfaces,* 23:1 (1993), 109–129.

[NYC]
Larson, R.C., M.F. Cahn, and M.C. Shell, "Improving the New York City Arrest-to-Arraignment System," *Interfaces,* 23:1 (1993), 76–96.

[NYNEX]
Barnea, T., D. Benanav, K. Dutta, I. Eisenberg, J. Euchner, E. Gilbert, A. Goodarzi, E. Lee, Y. Lin, J. Martin, J. Peterson, R. Pope, R. Salgame, S. Sardana, and G. Sevitsky, "Arache: Planning the Interoffice Facilities Network at NYNEX," *Interfaces,* 26:1 (1996), 85–101.

[P&G]
Camm, J.D., T.E. Chorman, F.A. Dill, J.R. Evans, D.J. Sweeney, and G.W. Wegryn, "Blending OR/MS, Judgment, and GIS: Restructuring P&G's Supply Chain," *Interfaces,* 27:1 (1997), 128–142.

[PLANETS]
Breitman, R.L., and J.M. Lucas, "PLANETS: A Modeling System for Business Planning," *Interfaces,* 17:1 (1987), 94–106.

[PONTIS]
Golabi, K., and R. Shepard, "Pontis: A System for Maintenance Optimization and Improvement of U.S. Bridge Networks," *Interfaces,* 27:1 (1997), 71–88.

[PRUDENTIAL]
Ben-Dov, Y., L. Hayre, and V. Pica, "Mortgage Valuation Models at Prudential Securities," *Interfaces,* 22:1 (1992), 55–71.

[REYNOLDS]
Moore, Jr., E.W., J.M. Warmke, and L.R. Gorban, "The Indispensable Role of Management Science in Centralizing Freight Operations at Reynolds Metals Company," *Interfaces,* 21:1 (1991), 107–129.

[SADIA]
Taube-Netto, M., "Integrated Planning for Poultry Production at Sadia," *Interfaces,* 26:1 (1996), 38–53.

[SAINSBURYS]
Ormerod, R.J., "Information Systems Strategy Development at Sainsbury's Supermarkets Using 'Soft' OR," *Interfaces,* 26:1 (1996), 102–130.

[SANDF]
Botha, S., I. Gryffenberg, F.R. Hofmeyr, J.L. Lausberg, R.P. Nicolay, W.J. Smit, S. Uys, W.L. van der Merwe, and G.J. Wessels, "Guns or Butter: Decision Support for Determining the Size and Shape of the South African National Defense Force," *Interfaces,* 27:1 (1997), 7–28.

[SANFRAN]
Taylor, P.E., and S.J. Huxley, "A Break from Tradition for the San Francisco Police: Patrol Officer Scheduling Using an Optimization-Based Decision Support System," *Interfaces,* 19:1 (1989), 4–24.

[SANTOS]
Dougherty, E.L., D. Dare, P. Hutchison, and E. Lombardino, "Optimizing SANTOS's Gas Production and Processing Operations in Central Australia Using the Decomposition Method," *Interfaces,* 17:1 (1987), 65–93.

[SHUTTLE]
Paté-Cornell, M., and P.S. Fischbeck, "Risk Management for the Tiles of the Space Shuttle," *Interfaces,* 24:1 (1994), 64–86.

[SIPMODEL]
Graves, S.C., and S.P. Willems, "Strategic Safety Stock Placement in Supply Chains," *Proceedings of the 1996 MSOM Conference,* Dartmouth College, Hanover, NH, (1998), 299–304.

[SO]
Erwin, S.R., J.S. Griffith, J.T. Wood, K.D. Le, J.T. Day, and C.K. Yin, "Using an Optimization Software to Lower Overall Electric Production Costs for Southern Company," *Interfaces,* 21:1 (1991), 27–41.

[SONET]
Cosares, S., D.N. Deutsch, I. Saniee, and O.J. Wasem, "SONET Toolkit: A Decision Support System for Designing Robust and Cost-Effective Fiber-Optic Networks," *Interfaces,* 25:1 (1995), 20–40.

[SPAIN]
Dembo, R.S., A. Chiarri, J.G. Martin, and L. Paradina, "Managing Hidroeléctrica Española's Hydroelectric Power System," *Interfaces,* 20:1 (1990), 115–135.

[SYNTEX]
Lodish, L.M., E. Curtis, M. Ness, and M.K. Simpson, "Sales Force Sizing and Deployment Using a Decision Calculus Model at Syntex Laboratories," *Interfaces,* 18:1 (1988), 5–20.

[TATA]
Sinha, G.P., B.S. Chandrasekaran, N. Mitter, G. Dutta, S.B. Singh, A.R. Choudhury, and P.N. Roy, "Strategic and Operational Management with Optimization at Tata Steel," *Interfaces,* 25:1 (1995), 6–19.

[TEXACO]
DeWitt, C.W., L.S. Lasdon, A.D. Waren, D.A. Brenner, and S.A. Melhem, "OMEGA: An Improved Gasoline Blending System for Texaco," *Interfaces,* 19:1 (1989), 85–101.

[TINKERAFB]
Ravindran, A., B.L. Foote, A.B. Badiru, L.M. Leemis, and L. Williams, "An Application of Simulation and Network Analysis to Capacity Planning and Material Handling Systems at Tinker Air Force Base," *Interfaces,* 19:1 (1989), 102–115.

[USARMY]
Durso, A., and S.F. Donahue, "An Analytical Approach to Reshaping the United States Army," *Interfaces,* 25:1 (1995), 109–133.

[USPOSTAL]
Cebry, M.E., A.H. deSilva, and F.J. diLisio, "Management Science in Automating Postal Operations: Facility and Equipment Planning in the United States Postal Service," *Interfaces,* 22:1 (1992), 110–130.

[VILPAC]
Nuño, J.P., D.L. Shunk, J.M. Padillo, and B. Beltrán, "Mexico's Vilpac Truck Company Uses a CIM Implementation to Become a World Class Manufacturer," *Interfaces,* 23:1 (1993), 59–75.

[YASUDA]

Cariño, D.R., T. Kent, D.H. Myers, C. Stacy, M. Sylvanus, A.L. Turner, K. Watanabe, and W.T. Ziemba, "The Russell-Yasuda Kasai Model: An Asset/Liability Model for a Japanese Insurance Company Using Multistage Stochastic Programming," *Interfaces,* 24:1 (1994), 29–49.

[YELLOW]

Braklow, J.W., W.W. Graham, S.M. Hassler, K.E. Peck, and W.B. Powell, "Interactive Optimization Improves Service and Performance for Yellow Freight System," *Interfaces,* 22:1 (1992), 147–172.

This left page blank intentionally.

INDEX OF AUTHORS

Adams, S., 85
Banerjee, S., 44
Basu, A., 44
Bhargava, H.K., 42, 47, 48, 85
Blanning, R.W., 41
Booch, G., 70, 107
Box, D., 115, 151, 168, 216, 281
Bradley, G.H., 47, 48, 87
Brockschmidt, K., 156, 168, 277
Brown, K., 116
Buschmann, F., 100
Chappell, D., 216, 281
Ching, C., 46, 284
Cioch, F.A., 44
Clemence, Jr., R.D., 47, 48, 87
Coffee, P., 267
Dellarocas, C.N., 46, 212, 279
Dolk, D.R., 45, 46, 48, 58
Dyck, T., 119
Eck, R.D., 43, 44, 284
Finkelberg, H., 52
Gamma, E., 98, 169, 312
Gates, III, W.H., 18
Geoffrion, A.M., 35, 41, 42, 45, 47, 56
Goldberg, A.V., 224
Graves, S.C., 52, 229, 239, 249, 251
Gutierrez, C.J., 229
Hatto, A.T., 287
Huh, S., 28, 44, 45
Jones, C.V., 27, 42, 286
Jordan, W.C., 239
Kant, I., 51
Kimbrough, S.O., 47, 48, 85
Kottemann, J.E., 46, 48, 58
Krishnan, R., 42, 47, 48, 85
Larson, R.C., 82
Lazimy, R., 28
Lenard, M.L., 28, 44
Ma, P., 27
McCright, J., 119
McKay, K.N., 239, 240
McKelvey, R., 27
Meyer, B., 263
Mili, F., 44
Mowbray, T.J., 36
Muhanna, W.A., 28, 47, 56
Murphy, F.H., 27
Nahmias, S., 61
Philippakis, A., 43, 44, 284
Piela, P., 27
Pietrek, M., 75
Ramirez, R.G., 43, 44, 46, 47, 48, 284
Richter, J., 97
Rogerson, D., 97
Ruark, J.D., 25, 115, 229
Ruh, W.A., 36
Rumbaugh, J., 51
St. Louis, R.D., 46, 47, 48, 284
Stohr, E.A., 27
Szoke, I., 44
Tung, L., 47, 48
Westerberg, A., 27
Willems, S.P., 249, 251
Wolfram von Eschenbach, 286–87

Blank, this page left intentionally.

INDEX

56-bit key, 96
911, 60
acquiring running solver, 175–76
acquiring SolverDescription, 168
acquiring SolverInfo, 161–62
Active Scripting, 203
Active Server Pages, 270
Active Template Library, 220, 291
ActiveX control containers, 184
ActiveX controls, 116, 118, 222, 258, 277
ActiveX interfaces, 220
actors, 70
Ada, 76
add-ins. *See* Microsoft Excel add-ins
advise, 99
advise helpers, 102
ALCOA, *253–58*
al-Manakh stock market, 62, 66
AMPL, 26, 27, 28, 70, 222, 230, 271
Anfortas, 287
Antigua, 5
apartment, 213
API, 38
Apple MacOS, 20, 24, 241, 250
application programming interface, 38
application-centric computing, 20
applications
  ALCOA, *253–58*
  FlexCap, 224, *239–44*, 258, 264
  M/M/k queue, *244–49*, 264, 265, 289–324
  Monsanto, 62, 67, *229–39*, 255, 258, 262, 308
  SIPModel, 67, 224, *250–53*, 270
asynchronous call, 97
Athena, 74
ATL. *See* Active Template Library
Autodesk AutoCAD, 228, 274
automation-compatible, 116, 222, 244, 274, 283
AvgGapPerIteration, 82
AvgTimePerIteration, 82
batch solution, 64
BeanInfo, 141
beef jerky, 138
benefits, 261–72
blue screen of death, 85
Borland Delphi, 35, 76
Business Object Framework Facility, 36
C++, 76, 228
CanMaintainSolverData, 201
cat stalking bird, 81
CATID, 216
CBT hooks, 89–90, 109, 276–77
Chrétien de Troyes, 287
CLSCTX, 192
CLSIDFromProgID, 193
CLSIDFromString, 158

CoCreateInstance, 135, 171, 192, 215, 216, 280
COleControl, 118
COM, 31, 36–37, 38, 40, 76, 97, 112, 115, 116–17, 217, 267
  Active Scripting, 203
  circular reference counts, 191
  COM+, 275
  component categories, 258. *See* component categories
  CORBA bridges, 273
  CORBA, and, 273
  custom marshaling, 213
  DCOM, 272
  dispatching, 229
  framework, 107, 273
  IDL, 151, 311
  integration with Visual C++, 252
  interface pointers, 200
  interfaces, 118
  marshaling, 97, 213
  OLE. *See* Object Linking and Embedding
  QueryInterface, 168
  running object table, 175
  smart pointers, 242
  standard marshaling, 213
  type libraries, 151
  use of GUIDs, 278
COM+, 117
comma separated value file, 231, 234
common dialogs library, 102
component categories, 23, 216
Component Object Model. *See* COM
component-based modeling environments, 27–30
components of solution archetypes, 53–67
compound documents, 104
  container, 23
  evolution, 20–24
  future, 23–24
  in-place activation, 22–23
  out-of-place editing, 21–22
  pasting a picture, 20–21
connection points, 277
control flows, 71
CORBA, 36–37, 76, 244, 273
  COM, and, 273
CORBAfacilities, 36
core services, 101–4, 214–17, 286
  advise helpers, 102, 216
  DataAdviseHolder, 216
  dimension and type support, 102, 104
  dispatch wrappers, 216
  running solver table, 176, 217
  solver database, 102, 103–4, 109, 215–16
  SolverAdviseHolder, 216, 291
  SolverDescription helpers, 167–68
  SolverInfo helpers, 160–61
  SolverInfo, generic implementation, 216
  SolverRegistrar, 160, *216*, 277
Corel WordPerfect, 33
CPLEX, 26, 27, 54, 55, 56, 70, 77, 230, 271, 295
  Callable Library, 24, 28, 220, 226–29, 230, 270, 289, 308
  wrapping into framework. *See* RLPWrapper solver
CPXloadlp, 289
CPXoptimize, 289
CPXsolution, 289, 300
CSV file, 231, 234
Ctrl-Alt-Delete, 84
Ctrl-Command-Reset, 84
custom() attribute, 152
customization, 74
daemons, 58
DAMS. *See* Data and Algebraic Management System
DAO, 119
Data and Algebraic Management System, 46
data corruption, 85
data elements, 120–32
  change notifications, 135
  creating and acquiring, 131–32
  data sources, and, 132
  dimensions, 123–24
  example code for iteration, 297–98
  generic interfaces, 124–28
  properties, 129–31
  sets, 121–23
  special interfaces, 128–29
  VARIANT storage type, 313
Data Encryption Standard, 96
data flow, 71, 110, 118–37, 278–79
  data element properties, 129–31
  data elements, 120–32
  data sources, 120, 132–37
  dimensions, 123–24

sets, 121–23
data integrity, 91–96
data source maintains data, 200, 207–8
data sources, 120, 132–37
change notifications, 135
DataAdviseHolder, 135–37
DataAdviseHolder, 135–37, 216
database requirements, 105
datamart, 43, 85
datastore, 43, 79
DCOM, 272
decision-based directed graph architecture, 57–59
Delta, 66
DES. *See* Data Encryption Standard
description property IDs, 164–66
DescriptionPropManager, 164, 165
Dilbert, 85
dimension and type support, 102, 104
dimension manipulators, 87, 104, 108, 275
dimensions, 123–24
dimensions and typing, 85–88, 108–9, 275–76
manipulation, 87
directed acyclic graph architecture, 55–57
dispatch wrappers, 216
dispinterface, 156–58, 156, 203, 274, 304, 305
distribution, 96–97
DLL. *See* dynamic-link library
DllRegisterServer, 161
DllUnregisterServer, 161
documentation, 76–81, 87, 141
automating through introspection, 81
reducing complexity, 80
selecting a solver, 78–79
suitability of the algorithm, 78
suitability of the solver, 79
using a solver, 80
documentation framework, 44
document-centric computing, 20
domain analysis, 52–69
actors, 70
control flows, 71
data flow, 71
participants, 70–72
solvers, 70
stores, 70
DSS, 18, 119
dual interface, 203
dynamic-link library, 74, 115
ease of integration, 17, 28
Edelman papers, 52, 67–68
elevators, 81
enumeration, solvers, 103
enumeration, to crack code, 96
estimated time remaining, 82
evolution of application implementation, 31–37
evolution of compound documents, 20–24
executable, 115. *See* executable file
requirement, 73–75
executable file, 74
execution periodicity, 53, 61–65
one-shot solution, 62
operational solutions, 64, 279
strategic solutions, 61–62
tactical solutions, 63
finite first fit, 256
finite next fit, 256
finite state machine, 179
first-order logic, 42
Fisher King, the, 287
FlexCap, 224, *239–44*, 258, 264
flyweight pattern, 312
framework
benefits, 261–72
core services, 214–17
data flow, 110, 118–37, 278–79
data integrity, 211–12
dimensions and typing, 108–9
distribution, 213–14
entities, 111–12
example extension, 311–19
global/local control, 212
interfaces, 112–15
introspection, 109, *141–68*
issues, 272–81
life cycle control. *See* life cycle control
networking solvers, 178–214
organization, 107–15
progress updates. *See* progress updates
queueing system specification, 317–19
reference, 115
specifying a data type, 311–13
synchronization, 214
target audience, 19
using a custom data type, 313–15

framework interfaces, 116–18
Framework IV, 42
Franz Edelman finalists. *See* Edelman papers
future of modeling environments, 30–31
FW/SM, 42
FX!32, 273
GAMS, 28
global control, 178–79
global workflow manager, 58
global/local control, 99–101, 110
Goal Seek, 246
Grail, the, 287
Gral, the, 287
graphical modeling environments, 27, 282
GUID, 118, 284
Hanshin Expressway, 61, 66
homonyms, 47, 285
Howzat!, 288
IClassFactory, 116, 123
IClassFactory2, 123
IConnectionPointContainer, 277
ICreateTypeInfo, 154
ICreateTypeLib, 154
IDataAdviseHolder, 216
IDataObject::DAdvise, 135, 170
IDataObject::DUnadvise, 135, 170
IDispatch, 116, 117, 158, 216, 222, 229, 274, 280, 305, 311, 319
IEnumCLSID, 215
IEnumRDESCRIPTIONPROPERTY, 164
IEnumRREGSOLVERINFO, 215
IFont, 313
IFontDisp, 313
IMarshal, 213, 214
inbound solver sites, 185, *189–93*
  creating and destroying solvers, 191–93
  executing solvers, 190–91
  managing mappings, 190
informal scheduler, 63
in-process server, 115
Insignia Solutions SoftWindows 95, 273
integrated modeling, 56
integrated modeling environments, 27
Integrated Supply Chain Management Consortium, MIT, 229
integrity of data. *See* data integrity
interface-based programming, 217, 284
interfaces, 112–15
Interfaces, 52
interfaces, framework, 116–18
International System, 86
introspection, 76–81, 109, *141–68*
  QueryInterface, 168
  requirements, 76–81
  SolverDescription, 142
  SolverDescription, *162–68*. *See* SolverDescription
  SolverInfo, 141, *142–62*. *See* SolverInfo
InverseCDF, 316, 317
InverseNormalCDF, 223
InverseNormalPDF, 223
Invoke, 229
invoking from different applications, 75–76
IOleClientSite, 184
IOleControl, 116, 118
IOleControlSite, 184
IOleInPlaceSite, 184
Iowa, 253
IPersist, 116, 123
IPersistStream, 220
IProvideClassInfo2, 277
IRDataAdvise, 133, 134, *135*, 136
IRDataAdviseHolder, 133, *135–37*
IRDataElement, 118, 124, *125–26*, 127, 134, 279
IRDataElement1, *128–29*
IRDataElement2, *129*
IRDataElementAccess, 124, *126–28*
IRDataElementClone, 124, *126*, 127
IRDataElementCreator, 124, *128*
IRDataElementProperties, 130
IRDataElementPropertiesChange, 130, *131*
IRDataElementScalar, *128*
IRDataSource, *133–34*, 135, 136
IRDescriptionPropEnumeration, 164, *165–66*
IRDescriptionPropRegistration, 164–*65*
IRDimension, *124*
IRDimensionCreator, 123, *124*
IRMappingAdvise, *191*
IRQueueData, 247, *318*, 319
IRQueueStatistics, *318*, 319
IRRandomGenerator, *317*
IRRandomSample, *316*
IRRandomVariable, *315–17*

IRSet, *121–22*, 124
IRSetCreator, 121, 122, *123*, 124
IRSetModifier, 121, *122–23*
IRSolver, 137, *139–40*, 170, 228
IRSolverAdvise, 169, *173–75*
IRSolverAdviseHolder, *170–72*
IRSolverAsynchSolve, *140*, 145, 228
IRSolverBaseInfo, *144–45*, 158
IRSolverControl, 169, 177
IRSolverDescription, 162–*63*, 164, 166
IRSolverDescriptionProperties, 162, *163–64*, 166
IRSolverDimInfo, 147, 148, *149*
IRSolverEnumeration, *215–16*, 277
IRSolverInfo, *145–46*, 166
IRSolverInfoProvideTypeInfo, *314*
IRSolverInputInfo, 146–*47*, 314
IRSolverInputs, 137, *140*, 147, 189–*90*, 199, 217, 228, 289
IRSolverMapping, *199–200*, 201
IRSolverMappingMechanism, 199, *202*
IRSolverOutputInfo, 146, *148*, 314
IRSolverOutputs, 137, *140–41*, 190, 193, 199, 228
IRSolverParameters, 137, *141*, 157, 276, 302, 304, 305
IRSolverParametersInterface, *156–58*, 304
IRSolverParamInfo, 146, *148–49*, 314
IRSolverProvideInfo, *149*, 151, 161, 166, 167, 168
IRSolverRegistration, *215*, 216, 277
IRSolverSetInfo, 146, *148*
IRSolverSiteIn, 189–*90*, 193
IRSolverSiteInMappings, 189, *190*
IRSolverSiteInSolverFactory, 189, *191–93*, 204, 211
IRSolverSiteOut, 193
IRSolverStatistics, 276
IRSolverStatus, 175, 176–*77*
IRunningObjectTable, 176
ITypeInfo, 152, 305, 314
ITypeLib, 152
IUnknown, 116, 155, 168, 220, 280, 305, 313, 314
IUnknown::QueryInterface. *See* QueryInterface
IViewObject, 116
Java, 267, 273
  SDK, 77
  virtual machines, 36, 74, 221, 244, 267
Java Virtual Machine
  see Java virtual machines, 36
JavaBeans, 22, 81
  introspection, 141
javadoc, 77
JavaScript, 117, 203
JDBC, 119
JDK, 77
JNI, 244
JScript, 117, 267, 311
JVM. *See* Java virtual machines
killer application, 18, 28, 281
knapsack, 72, 78, 86, 138, 143, 154, 155–56
knowledgebase, 44, 79
Kuwait al-Manakh stock market, 62, 66
Leaders for Manufacturing, MIT, 41
LFM, 41
life cycle control, 84, 111, 177
LINDO, 26, 54, 230, 231, 232, 238
linear programming solver, 53
listener. *See* observer
live solution, 64
Liz, 126
LoadRegTypeLib, 160
LoadTypeLib, 160
local control, 180–84
local procedure calls, 97
local server. *See* executable
LockRequired, 202
lockstep flows, 279
Lotus 1-2-3, 18, 76
M/M/k queue model, *244–49*, 264, 265, 289–324
MaintainPreference, 200, *201–2*
make it so, 73
mapping data, 199
mapping maintains data, 200, 208–10
mappings, 184–89, *199–203*
  aggregating mapping, 203
  data source maintains data, 200, 207–8
  extensions and versatility of, 203, 285
  MaintainPreference, 200, *201–2*
  mapping data, 200–202
  mapping maintains data, 200, 208–10
  solver maintains data, 201, 210–11

SolverInputNum, 202
Mathematica, 39, 222
Matlab, 222
matrix multiplication, 82
MBTA, 81
MCSS, 34–36
metafile, 21
Microsoft Access, 72, 76, 196, 230, 254
Microsoft Developer Network Library, 77
Microsoft Excel, 35, 39, 40, 56, 72, 76, 222, 228, 230, 231, 252, 270, 274, 308, 319
add-ins, 75, 228, 247, 265, 324
Goal Seek, 246
Microsoft Internet Explorer, 102
Microsoft Java Virtual Machine, 76, 152
Microsoft Office, 75, 117
Microsoft Query, 72
Microsoft Visual Basic, 76, 116, 152, 154, 239, 241–42, 281, 308
Microsoft Visual C++, 220, 242, 252, 291
Microsoft Visual Studio, 76
Microsoft Windows, 20, 36, 38, 39, 74, 102, 116
logo program, 24
Windows 95, 75
Microsoft Word, 33, 103, 268
MIDL, 151, 154, 161
MINOS, 26
MIPS, 74
MIT Athena, 74
MMQueue solver, 228, *247–48*, 264
model integration, 45–49
model selection, 43, 78
model selectors, 278
modelbase, 43, 44
model-centric computing, 25
model-data independence, 46
ModelInfo, 257
modeling component services system. *See* MCSS
modeling languages, 26–27
modelstore, 43, 44
monikers, 123, 176
Monsanto, 62, 67, *229–39*, 255, 258, 262, 308
Monsanto solver, 308–10
Mosaic, 18
Mountain View, 121
MPL, 222
MPS, 26, 27, 28, 236, 269
MSDN Library. *See* Microsoft Developer Network Library
multiple-use solutions, 66–67
Munsalvæsche, 287
Muswell Hill, 121
naming and typing, 48–49
National Car Rental, 66
NCSA Mosaic, 18
NetLib, 223
network flow algorithms, 90
networking solvers, 178–214
component interactions, 207–11
creating and wiring the network, 204–7
data integrity, 91–96, 211–12
destroying the network, 211
distribution, 96–97, 213–14
fulfilling requirements, 211
global control, 178–79
global/local control, 99–101, 110, 212
inbound solver sites. *See* inbound solver sites
interaction diagrams, 207–11
limitations, 279
local control, 180–84
mappings. *See* mappings
notifications, 98–99, 276–77
outbound solver sites. *See* outbound solver sites
requirements, 211
sample code, 203–11
synchronization, 96–97, 110, 214
transparency, 97
wiring the network, 189
networks of solvers, 90–101
Networks/SM, 42
NormalCDF, 223
NormalPDF, 223
notifications, 98–99, 110, 276–77
advise, 99
unadvise, 99
Object Linking and Embedding, 22, 24, 32, 36
structured storage, 32
Object Management Group, 36
observer, 98
Observer pattern, 98, 169
ODBC, 26, 28, 71, 105, 119
ODBMS, 45, 47

ODEs, 54
oil of bergamot, 73
OLAP, 119, 120
  OLE DB for, 120, 278
OLE. *See* Object Linking and Embedding
OLE Automation, 116
OLE DB, 36, 105, 119, 134, 231
OLE DB for OLAP, 120, 278
OMAC, *241*
OMEGA project, Texaco, 63
OMG, 36
one-shot solution, 62
online help, 76
Open dialog, 102
OpenDoc, 22, 23, 24
operational solutions, 64, 279
  batch, 64
  live, 64
  responsive, 64
OSF DCE RPC IDL, 151
OSL, 24, 26, 70, 77
outbound solver sites, 186, *193–98*
  exposing outputs, 195
  locking solver, 193–95
  making decisions, 195–97
  resetting the solver, 195
  state diagram, 197–98
packaging solvers, *220–23*
parameterization, 74
participants in solutions, 70–72
Parzival, 287
Penny, 126
Penster, 5
percentage complete, 82
persistent flows, 279
per-solver implementation, SolverInfo, 151
plug-and-play modeling, 29
polymorphism, 47
postconditions, 43
process, 97
process homicide, 84
Procter & Gamble, 56
production-environment scheduler, 63
progress bar, 83
progress updates, 81–84, 98, 109, 169–77
  acquiring running solver, 175–76
  assumptions during push notifications, 175
  presentation, 83
  progress bar, 83
  pull queries, 175–77
  push notifications, 169–75
  registering for notifications, 170–72
  sending and receiving notifications, 173–75
  SolverAdviseHolder, 170–72
Prolegomena to Any Future Metaphysics, 51
proof of concept, 40–41
publisher, 98
QueryInterface, 120, 142, 168, 175, 176, 216, 277, 315
queueing system specification, 317–19
quiddity, 48, 85
random variable specification, 311–19
RandVar module, *223–24*, 240, 248, 264, 270
RBinPack module, 256–57
RDBElements module, *234–36*, 256
RDBLoad solver, 159, 231, *234–36*, 238
RDBStore solver, 231, *234–36*
RDESCPROPID, 162, 163, *164–66*, 164
RDESCPROPIDINFO, 165
RDESCRIPTIONPROPERTY, 164
readme, 77
real-time directed graph architecture, 60–61, 279
receiving progress updates, 173–75
reference manual, 115
registering for progress updates, 170–72
registration, solvers, 103
registry settings for SolverInfo, 160
remote procedure calls, 97
replicator, 73
requirements, 17–106
  CBT hooks, 89–90, 109, 276–77
  client, 105
  core services, 101–4
  data integrity, 91–96, 211–12
  database, 105
  dimension and type support, 102, 104
  dimensions and typing, 85–88, 275–76
  distribution, 96–97, 213–14
  documentation, 76–81, 78–79, 80
  domain analysis, 52–69
  executable, 73–75
  global/local control, 99–101, 212
  introspection, 76–81

invoking from different applications, 75–76
life cycle control, 84
networks, 90–101
notifications, 98–99, 110, 276–77
progress updates, 81–84, 98
solver database, 102, 103–4
solvers, 72–90
synchronization, 96–97, 214
testing and validation, 88–89, 276
residual networks, 90
responsive solution, 64
rewards, 18
RGUID_ANSWERTOLIFETHEUNIVERSE-ANDEVERYTHING, 152
RKnapAlg solver, 257
RLoadRegSolverInfo, *160*, 161, 167
RLoadSolverInfo, *160*, 162, 167
RLoadSolverInfoClsid, *160*, 161, 167
RLookupInfoGuid, *160*
RLPWrapper solver, *226–29*, 232, 233, 238, 258, 270, 289–308
RNetOpt module, *224–26*, 240, 257, 258, 263, 270
rowset, 119, 234, 235, 236
RPC. *See* remote procedure calls
RREGSOLVERINFO, 215
RSML solver, *236–37*, 258
RTFM, 77, 81
run time, 78
running object table, 175, 217
running solver table, 176, 217
sandbox, 97
SANTOS, Ltd., 59
Scudder, Stevens, and Clark, Inc., 14, 244
sending progress updates, 173–75
separation of orthogonal functionality, 17, 28, 224
service, 107
sets, 121–23
SIDL. *See* SolverInfo Definition Language
simplex algorithm solver, 53
simplex method, 90
simplicity of interface, 17, 28
single-stage architecture, 54–55
single-use solutions, 66
SIPEng solver, *252*
SIPModel, 67, 224, *250–53*, 270
SITL. *See* SolverInfo Type Library
slogging, 14
SML. *See* Structured Modeling Language
solar eclipse, 5
Solaris, 39
solution archetypes, 52
categorization of samples, 67–68
components, 53–67
decision-based directed graph architecture, 57–59
directed acyclic graph architecture, 55–57
execution periodicity, 53, 61–65
multiple-use solutions, 66–67
operational solutions, 64, 279
real-time directed graph architecture, 60–61, 279
single-stage architecture, 54–55
single-use solutions, 66
solution architecture, 53
solution reuse, 53, 65–67
strategic solutions, 61–62
tactical solutions, 63
solution architecture, 53–61
decision-based directed graph architecture, 57–59
directed acyclic graph architecture, 55–57
real-time directed graph architecture, 60–61, 279
single-stage architecture, 54–55
solution reuse, 53, 65–67
multiple-use solutions, 66–67
single-use solutions, 66
solutions
participants, 70–72
SolveAdvise, 170
solver database, 102, 103–4
solver interaction, 112
solver maintains data, 201, 210–11
solver representation language, 43
solver selectors, 278
solver sites, 184–89. *See also* inbound solver sites and outbound solver sites
solver structure, 137–39
SolverAdviseHolder, 216, 291
SolverAdviseHolder object, 170–72
SolverDescription, 142, *162–68*
acquiring, 168
description property IDs, 164–66
generic implementation, 166–67
per-solver implementation, 166, 167

RDESCPROPID, 164–66
registry entries, 165
solvers, adding to, 167–68
SolverDispatch, 217
SolverInfo, 141, *142–62*, 168, 189, 215
acquiring, 161–62
Definition Language. *See* SolverInfo Definition Language
example use, 150
example, generating, 289–91
generic implementation, 151–61, 216
per-solver implementation, 151
registry settings, 160
solvers, adding to, 160–61
Type Library. *See* SolverInfo Type Library
SolverInfo Definition Language, *151–61*, 285, 289, 304
custom attributes, 158–59
custom data types, 314
custom() attribute, 152
including SolverDescription, 166–67
input and output interfaces, 155–56
knapsack example, 155–56
parameter interface, 156–58
set typedefs, 154
specifying flags on elements, 158
SolverInfo Type Library, 151–61, 285
custom attributes, 158–59
including SolverDescription, 166–67
input and output interfaces, 155–56
parameter interface, 156–58
set typedefs, 154
specifying flags on elements, 158
SolverInputNum, 202
SolverRegistrar, 160, *216*, 277
solvers
acquiring running solver, 175–76
CBT hooks, 89–90, 109, 276–77
customers, 185
customization and parameterization, 74
data integrity, 91–96
database of, 215–16
dimensions and typing, 85–88, 108–9
distribution, 96–97
documentation, 87, 141
documentation, for selecting, 78–79
documentation, for using, 80
documentation, suitability, 79
domain analysis, in, 70
executable, 73–75, 115
global/local control, 99–101
implementing inputs, 295–98
implementing outputs, 301–2
implementing parameters, 302–8
interaction, 112
interfaces, 116–18
introspection, 76–81, 109, *141–68*
invoking, 75–76
Java development, 221–22
life cycle control, 84, 111. *see* life cycle control
locking mechanism, 193
MMQueue, 228, *247–48*, 264
networking. *See* networking solvers
networks, 90–101
packaging, 220–23
primary interfaces, 137–41
progress updates, 81–84, 98, 109. *See* progress updates
QueryInterface, discovering capabilities with, 168
RandVar module, *223–24*, 240, 248, 264, 270
RBinPack module, 256–57
RDBElements module, *234–36*, 256
RDBLoad, 159, 231, *234–36*, 238
RDBStore, 231, *234–36*
registration, 102, 103–4
requirements, 72–101
RKnapAlg, 257
RLPWrapper, *226–29*, 232, 233, 238, 258, 270, 289–308
RNetOpt module, *224–26*, 240, 257, 258, 263, 270
RSML, *236–37*, 258
run time, 78
running solver table, 176
SIPEng, *252*, 270
solver interaction, 137–41
SolverAdviseHolder, 170–72
SolverDescription, 142, *162–68*
SolverDescription, acquiring, 168
SolverDescription, adding, 167–68
SolverInfo, 141, *142–62*
SolverInfo Definition Language, 151–61
SolverInfo registry settings, 160
SolverInfo, acquiring, 161–62
SolverInfo, adding, 160–61

source code, importance of, 75
state diagram, 91
structure, 137–39
suppliers, 185
synchronization, 96–97
testing and validation, 88–89
using, 228–29
validity of outputs, 193
Visual Basic development, 221–22
wrapping CPLEX. *See* RLPWrapper solver
SolveUnadvise, 170
Somerville, 120
source code, 75
specification, 107
specifying a data type, 311–13
splash screens, 82
Sporkinator, 5
spreadsheets, 70, 89
modeling, 55
SQL, 26, 28, 71, 72, 105, 119, 196
SS Central America, 66
Star Trek, 31
stochasticity, 42
stores, 70
strategic solutions, 61–62
one-shot solution, 62
structured modeling, *41–43*
Structured Modeling Language, 28, 35, 41–43, 42, 43, 45, 70, 236
structured storage, 32
subject. *See* publisher
subproblem, 52, 53–54, 65
SunSoft, 76
Sybase PowerBuilder, 76
synchronization, 96–97, 110
synonyms, 47, 285
T, the, 81
tactical solutions, 63
informal scheduler, 63
production-environment scheduler, 63
target audience, 19
task-centric computing, 24
Tea, Earl Grey, hot, 73
testing and validation, 88–89, 276
TeX, 163
Texaco OMEGA project, 63
Thalia, 126
trade-offs, 75
transient flows, 279
transparency, 97
type libraries, 151
typing. *See* dimensions and typing. *See* dimensions and typing
UML. *See* Unified Modeling Language
unadvise, 99
Underground, London, 81
Unified Method. *See* Unified Modeling Language
Unified Modeling Language, 43
United States customary units, 86
UNIX, 74, 267
URL, 163
Use the source, Luke, 77
using a custom data type, 313–15
using solvers, 228–29
UTSL, 77
validation. *See* testing and validation
VARIANT, 125, 147, 148, 149, 274, 304, 307, 313
VBScript, 117, 203, 267, 274, 283, 311
Visio Corp. Visio, 228
Visual Basic for Applications, 72, 117, 228, 230, 247, 267, 270, 274
voter registration, 103
WebBrowser control, 102
Win16Mutex, 75
Windows Scripting Host, 117
with events keywords, 277
Wolfe-Dantzig method, 59
Wolfram von Eschenbach, 287
wtypes.idl, 311

7124.8