SCHEDULING AND RESOURCE ALLOCATION IN AUTONOMOUS SERVICE NETWORKS

A DISSERTATION

SUBMITTED TO THE DEPARTMENT OF COMPUTER SCIENCE

AND THE COMMITTEE ON GRADUATE STUDIES

OF STANFORD UNIVERSITY

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS

FOR THE DEGREE OF

DOCTOR OF PHILOSOPHY

Neal Joseph Sample

March 2005

© Copyright by Neal Joseph Sample 2005
All Rights Reserved

I certify that I have read this dissertation and that in my opinion it is fully adequate, in scope and quality, as dissertation for the degree of Doctor of Philosophy.

____

Gio Wiederhold Principal Advisor

I certify that I have read this dissertation and that in my opinion it is fully adequate, in scope and quality, as dissertation for the degree of Doctor of Philosophy.

____

Hector Garcia-Molina

I certify that I have read this dissertation and that in my opinion it is fully adequate, in scope and quality, as dissertation for the degree of Doctor of Philosophy.

____

Michael Franklin

Approved for the University Committee on Graduate Studies

____

ABSTRACT

Fee-based computing models are gaining attention in both cooperative and commercial computing environments. Grid computing and web services are models where organizations can charge a fee or trade resources with clients for use of the service, making it possible for clients to use resources and services that are too expensive, too esoteric, or simply inefficient for individual clients to manage or maintain.

Unfortunately, benefits of the Grid model come with a loss of control over job execution and real uncertainty about job completion: distributed services are not under the control of the client. Clients cannot control resource allocation to recover from inaccurate runtime estimates, runtime delays, and execution hazards.

We introduce “surety” as the probability that a composed program will finish execution within a deadline window forecast by the service provider. Short runtime services with a high surety (high confidence of success) would be more valuable than services with longer running times or lower surety. Client selection of service providers depends on what values they place on time, cost, and surety, simultaneously. Earlier schedulers treated the remote service problem as a multivariate optimization accounting for only two variables: cost and time. We extend these schedulers by accounting for the uncertainty introduced when services are not under the client's control.

Ultimately, control over the service remains at the provider's site. This is the distinction from software models where components are subservient to calling routines. Distributed computation schedulers often assume a cooperative environment where delays are rare, and that initial estimates come from oracles. Our work addresses systems that may be rife with uncertainty, affecting the reliability of schedules generated a priori.

We present a new scheduler (the Surety-Based Scheduler (SBS)) that uses a statistical notion of surety to mitigate uncertainty in distributed computing environments and to select strategies to recover from runtime hazards. The goal of our scheduler is to take programs composed of multiple distributed services and completes them within client-specified soft deadlines by targeting a minimum level of surety throughout program execution. We demonstrate scheduling, monitoring runtime progress, and schedule repair when surety drops below a threshold value. Monitoring coupled with reactive rescheduling is the key to providing clients with the surety of distributed job completion, similar to performance they would expect from a program running on local resources.

By making programs composed of distributed services more reliable, these compositions become an increasingly viable solution for a wide range of problems, and become an appropriate solution for a larger class of clients. Our work is broadly applicable to any system where estimates may be gathered *a priori* and where clients may monitor runtime progress.

In this work, we cover several broad contributions to service computing, including the CLAM compositional language, the CPAM runtime, an address of data extraction and mediation considerations, and (of course) scheduling and rescheduling in uncertain environments.

ACKNOWLEDGMENTS

The CHAIMS project was supported by DARPA order D884 under the ISO EDCS program, with the Rome Laboratories being the managing agent and by Siemens Corporate Research, Princeton, NJ. Is life was extended by CIA funding for Neal Sample under the ORD's Postdoctoral Fellowship program.

Thanks are due to my academic mentors over the years, including Matt Haines, Mark Arnold, Hector Garcia-Molina, and Mike Franklin. They all have, at various points, put me on the right track, and also BACK on the right track as required. Without those four folks, there was not much chance of getting to this point!

First order thanks go, of course, to Gio Wiederhold, my primary research advisor on CHAIMS. In terms of research, Gio knew where I was going long before I did, which made life a lot easier. He first published on the CHAIMS paradigm before I was even looking for an undergraduate institution, though he managed to remain both active and interested in the research area long after I arrived at Stanford. His support was instrumental in the classroom, in the lab, and beyond. He set a remarkable example of broad-based success that I aspire to achieve… especially beyond the university walls.

Thanks to all my cronies! ☺ From the foosball folk, to the debaters, to the gamers, to the startup crowd, to the poker monsters, and beyond! This list is a long one, but an important one. This group was truly staggering in the sheer magnitude of experience lent and advice given on myriad topics, not limited to just Stanford and research. Their influence brought me to where I am today. It was good stuff! Thanks to aron, bartlett, byang, cacioppo, cgarver, charlie2na, cooperb, cpeth, cstaehlin, dan, dlopez, eytanb, farm, gingerw, goyo, jlogue, laurence, levyc, marinemj, michaelr, nazx, nstafford, paepcke, paulr, picton, ricks, rothfuss, tyson, walkers, and (not to miss anyone) *.

Finally, thanks to my family, all of ‘em. This includes my parents (Ken and Sharon Sample), my grandparents, my extended family, my wife (Meredith) and son (Aidan), my wife’s parents (Roger and Cathy Marine), and to my extended family, too.

DEDICATION

I dedicate this dissertation to the two who are fast asleep in the next room, Meredith and Aidan. You guys are the motivators that got me over the finish line. Lova ya.

TABLE OF CONTENTS

List of tables ... **xiv**

List of figures ... **xv**

1 CHAIMS Project Overview ... **1**

1.1 Introduction ... 4

1.1.1 Background ... 5

1.1.2 Architecture ... 6

1.1.3 Objectives ... 7

1.2 Language ... 8

1.2.1 Primitives ... 10

1.2.2 Clarity ... 11

1.2.3 Parallel Computation ... 12

1.2.4 Summary ... 13

1.3 CPAM Invocation Protocol ... 13

1.3.1 CLAM Primitives ... 13

1.3.2 CPAM Execution Alternatives ... 14

1.3.3 Interoperation Protocols ... 15

1.3.4 Adaptability to Changing Standards ... 16

1.4 Role Assignment ... 17

1.4.1 Services ... 17

1.4.2 Composition ... 18

1.4.3 Application Customers ... 19

1.5 Dissertation Overview ... 20

2 The CLAM language ... **23**

2.1 Introduction ... 23

2.2 CLAM in a sea of Languages ... 25

2.2.1 Objectives ... 25

2.2.2 Other Approaches to Composition ... 26

2.2.3 Manifold ... 27

2.2.4 Sample CLAM Program 28
2.3 CLAM Language Semantics 29
2.3.1 Parallelism 29
2.3.2 Clam-Enabled Optimizations 33
2.3.3 Simple Control Flow 37
2.4 CLAM IMPLEMENTATION 40
2.4.1 CLAM Repository 40
2.4.2 Data Types 41
2.5 CLAM Summary 43
3 The CPAM Runtime 45
3.1 Introduction 45
3.2 CPAM Support For Service Heterogeneity 47
3.2.1 Data Heterogeneity 47
3.2.2 Opaque Data 48
3.2.3 Distribution Protocol Heterogeneity 48
3.2.4 CHAIMS System Architecture 50
3.3 CPAM Preserves Service Autonomy 50
3.3.1 Information Repository 51
3.3.2 Service Connections in CPAM 52
3.3.3 Consistency 52
3.4 Large Service Composition 53
3.4.1 Invocation Sequence Optimization 53
3.4.2 Minimizing Data Flow between Services 58
3.5 using CPAM 60
3.5.1 Primitives Ordering Constraints 61
3.5.2 The CHAIMS Wrapper 62
3.5.3 Example of a Client Using CPAM 63
3.6 CPAM Summary 64
4 Result Extraction 66
4.1 Overview 66

4.1.1 Traditional RPC 67
4.1.2 Additional Extraction Models 68
4.1.3 Partial and Progressive Extraction Today 69
4.2 Result Extraction Within CLAM 71
4.2.1 EXAMINE and EXTRACT Revisited 72
4.2.2 Extraction Model Enabled by EXAMINE and EXTRACT 74
4.3 Comparisons 79
4.3.1 Partial and Progressive Result Extraction in Web Browsing 80
4.3.2 Incremental Result Extraction and Progress Monitoring in JointFlow 82
4.3.3 Incremental Result Extraction and Progress Monitoring in SWAP 84
4.3.4 Incremental Result Extraction and Progress Monitoring in CORBA 87
4.4 Data Extraction Summary 89
5 Surety-Based Scheduling 90
5.1 Introduction 90
5.2 Autonomous Service Providers 95
5.3 CHAIMS 97
5.4 Scheduling in an Uncertain Environment 98
5.4.1 Simple Planning 101
5.4.2 Monitoring and Repairing Schedules 109
5.4.3 Hazards and Repairs 112
5.4.4 Scheduling Observations 125
5.5 Additional Related Work 126
5.5.1 Mariposa 127
5.5.2 NOW (Networks of Workstations) 127
5.5.3 ePert and Extensions 128
5.5.4 Grid Computing 128
5.6 Summary 130
6 Experimental Results 131
6.1 Experimental Setup 131

6.1.1 Sandbox Description ... 131
6.1.2 Test Data Description ... 138
6.2 Metrics ... 142
6.2.1 Comparisons to Alternative Schedulers ... 142
6.2.2 Judgments about Program Execution Cost ... 144
6.3 Results ... 145
6.3.1 Scheduler Success Rates ... 147
6.3.2 Comparing Schedule Costs ... 149
6.3.3 Comparing Schedule Completion Times ... 153
6.3.4 Sample Student's Matched-Pair t Test ... 157
6.4 Analysis ... 159
6.4.1 Highly Constrained Budgets ... 160
6.4.2 Loosely Constrained Budgets ... 163
6.5 Summary ... 164
7 Future Work ... 166
7.1 Open Questions ... 166
7.2 Impacts of User Preferences ... 166
7.2.1 User Preferences Used in Repair Selection ... 166
7.2.2 Global Impact of Local Schedule Preferences ... 167
7.3 Questioning Assumptions about the Execution Environment ... 167
7.3.1 Progress is a Secondary Indicator of Work Rate ... 168
7.3.2 Monitoring Resolution Adjustments ... 169
7.3.3 Independence of Events ... 170
7.3.4 Gaussian or Power-Law Environment ... 171
7.4 Limitations of the Surety-Based Scheduler Implementation ... 171
7.4.1 Simultaneous Rescheduling ... 172
7.4.2 Execution Penalties ... 172
7.4.3 Second Order Cost Effects ... 173
7.4.4 Is there a benefit for non-repairable schedules? ... 175
8 APPENDIX A: Critical Path Method (CPM) ... 176

8.1 Steps in CPM Project Planning....177
8.1.1 Specify the Individual Activities 178
8.1.2 Determine the Sequence of the Activities....178
8.1.3 Draw the Network Diagram....178
8.1.4 Estimate Activity Completion Time 178
8.1.5 Identify the Critical Path....179
8.1.6 Update CPM Diagram....180
8.2 CPM Limitations....180
8.3 Example CPM Usage....180
9 APPENDIX B: Service Mediation....184
9.1 Introduction....184
9.1.1 Background 184
9.1.2 Objectives 187
9.2 Active Mediation in FICAS 190
9.2.1 Mediation 190
9.2.2 Mobile Classes 191
9.2.3 Active Mediation Architecture 193
9.2.4 Autonomous Service in FICAS 194
9.3 Applications of Mobile Classes 197
9.3.1 Relational M-class 198
9.3.2 Type Conversion M-class 199
9.3.3 Extraction Model Mediation....201
9.4 Performance Optimization for Active Mediation 204
9.4.1 Placement of Mobile Classes....204
9.4.2 LDS Algorithm 207
9.4.3 Data-flow Optimization for Sample M-classes....210
9.5 Conclusions....212
10 Bibliography 213

LIST OF TABLES

Number *Page*

Table 1. EXAMINE/EXTRACT relationships 76
Table 2. Computational models and resource management 92
Table 3. Bids received for each service 109
Table 4. Sample execution data points 147
Table 5. Overall completion rates 147
Table 6. Cost of generated schedules 150
Table 7. Comparing overall average cost and successful cost vs. cost of ideal scheduler 151
Table 8. Time of generated schedules 154
Table 9. Comparing overall average time and successful time vs. time of ideal scheduler 156
Table 10. Sample *t* test data 158
Table 11. Sample program completion with no extra resources 160
Table 12. Sample program completion with 50% extra resources 163
Table 13. Reserving budget example 174
Table 14. CPM Example Job Table 181
Table 15. Gantt Chart for Sample CPM Example Job Table 183
Table 16. Relational operators and their corresponding M-classes 199
Table 17. Sizing functions for relational M-classes 212

LIST OF FIGURES

Number *Page*

Figure 1 Components of the CHAIMS compositional paradigm 6
Figure 2 CHAIMS programming process and information flow.....................................9
Figure 3 Data transfer, opaque data 48
Figure 4 A view of the CHAIMS architecture..49
Figure 5 Split of the procedure call in CPAM and parallelism on synchronous calls.......57
Figure 6 Hierarchical setting of parameters..60
Figure 7 Primitives in CPAM and invocation ordering..62
Figure 8 CLAM program to calculate the best route between two cities 65
Figure 9 Overloading PARTIAL with additional semantic information.........................79
Figure 10 Program execution in the face of hazards..96
Figure 11 The scheduling process...101
Figure 12 Sample program...102
Figure 13 Dependency graph of sample program...103
Figure 14 Schedule search space in 2-D 107
Figure 15 Sample Schedule 111
Figure 16 Probability distribution of original program schedule................................112
Figure 17 Progressive hazard..113
Figure 18 Catastrophic hazard 114
Figure 19 Pseudo-hazard...115
Figure 20 Effect of "do nothing" on surety...116
Figure 21 Effect of service replacement on surety 117
Figure 22 Effect of service duplication on surety..118
Figure 23 Surety-Based Scheduler architecture..125
Figure 24 Sample programs successfully executed 148
Figure 25 Overall cost average 150
Figure 26 Relative cost of a successfully completed schedule.....................................152
Figure 27 Overall time average...155

Figure 28 Relative time of a successfully completed schedule 157
Figure 29 Sample programs successfully completed with no extra resources................161
Figure 30 Sample program completion with 50% extra resources 164
Figure 31 Progress versus finish time 169
Figure 32. Sample CPM diagram...177
Figure 33 Composing autonomous services into megaservices 187
Figure 34 Mediation architecture...191
Figure 35 Active mediation architecture..194
Figure 36 Autonomous service metamodel 197
Figure 37 Type conversion for autonomous services 201
Figure 38 Sample megaservice program segment with M-classes 206
Figure 39 Execution plans for the sample megaservice using M-class *Sum* 207
Figure 40 Data-flow diagrams for the sample megaservice using M-class *Sum* 207
Figure 41 LDS algorithm for optimizing M-class placement...209

1 CHAIMS PROJECT OVERVIEW

The "Grid" has received significant attention from computer scientists in the last decade, and is about to achieve even more attention in the dissertation! However, before we can set off on an in depth exploration of even the remotest corner of the Grid, we must have a shared understanding of what we will discuss. Because of substantial research and commercial funding for Grid projects, the term has been applied sloppily, at best, frequently polluting the concept and associated discussions. Ian Foster proposes in [109] a three-point checklist for determining whether a system is a Grid. He notes the stretching of meaning:

> "Grids have moved from the obscurely academic to the highly popular. We read about Compute Grids, Data Grids, Science Grids, Access Grids, Knowledge Grids, Bio Grids, Sensor Grids, Cluster Grids, Campus Grids, Tera Grids, and Commodity Grids. The skeptic can be forgiven for wondering if there is more to the Grid than, as one wag put it, a 'funding concept'—and, as industry becomes involved, a marketing slogan. If by deploying a scheduler on my local area network I create a 'Cluster Grid,' then doesn't my Network File System deployment over that same network provide me with a 'Storage Grid?' Indeed, isn't my workstation, coupling as it does processor, memory, disk, and network card, a 'PC Grid?' Is there any computer system that isn't a Grid?" [109]

He argues that the Grid is not evaluated through its architecture, but rather in terms of the applications, business value, and scientific results that it delivers. Early definitions of the Grid were numerous, and vary somewhat from what we consider today. In 1998, Foster and Kesselman had defined the grid this way:

> "A computational grid is a hardware and software infrastructure that provides dependable, consistent, pervasive, and inexpensive access to high-end computational capabilities." [110]

Much earlier (1969), Len Kleinrock issued a press release suggesting a vision of ubiquitous computing:

> "We will probably see the spread of 'computer utilities', which, like present electric and telephone utilities, will service individual homes and offices across the country." [111]

Later, Foster, et al. reconsidered Foster's original definition of the Grid, and revised it to include social and policy issues, stating that Grid computing is concerned with "coordinated resource sharing and problem solving in dynamic, multi-institutional virtual organizations" [112]. Their new definition is differentiated by considering that Grid participants have the ability to negotiate resource-sharing arrangements among a set of participating actors and then to use the resulting resource pool for some purpose. They noted:

> "The sharing that we are concerned with is not primarily file exchange but rather direct access to computers, software, data, and other resources, as is required by a range of collaborative problem solving and resource-brokering strategies emerging in industry, science, and engineering. This sharing is, necessarily, highly controlled, with resource providers and consumers defining clearly and carefully just what is shared, who is allowed to share, and the conditions under which sharing occurs. A set of individuals and/or institutions defined by such sharing rules form what we call a virtual organization." [112]

Foster synthesizes the definitions above into a 3-part checklist, thus determining that a Grid system, for purposes of our discussion, is one that meets the following criteria:

> "1) coordinates resources that are not subject to centralized control ... (A Grid integrates and coordinates resources and users that live within different control domains—for example, the user's desktop vs. central computing; different administrative units of the same company; or different companies; and addresses the issues of security, policy, payment, membership, and so forth that arise in these settings. Otherwise, we are dealing with a local management system.)
> 2) ... using standard, open, general-purpose protocols and interfaces ... (A Grid is built from multi-purpose protocols and interfaces that address such fundamental issues as authentication, authorization, resource discovery, and resource access. As I discuss further below, it is important that these protocols and interfaces be standard and open. Otherwise, we are dealing with an application specific system.)

> 3) ... to deliver nontrivial qualities of service. (A Grid allows its constituent resources to be used in a coordinated fashion to deliver various qualities of service, relating for example to response time, throughput, availability, and security, and/or co-allocation of multiple resource types to meet complex user demands, so that the utility of the combined system is significantly greater than that of the sum of its parts.)" [109]

We proceed in our discussion of Grid computing, testing the boundaries of "what is the Grid" by questioning whether suspected Grid applications, models, architectures, and actors meet the above definition.

Creating and managing large-scale software remains a task which requires many levels of expertise, well-defined processes, adherence to standards, and careful documentation. Even when all these prerequisites are in place, risk of failure is significant. We support a paradigm of composing software services for the creation of large-scale software. We have seen composition used in practice for a long time, but often in an ad-hoc fashion by experts. Composition is becoming more common as large-scale services, databases, mathematical modeling tools, and web-browsers are combined to create substantial systems. However tools to support this shift are somewhat marginal and fragmented.

To address these issues, the CHAIMS project provides an approach based on a high-level programming language (CLAM) for software service composition. This language supports the composition of remote services (typically provided by autonomous suppliers). A real-world analogue of service composition is the management of a manufacturing fulfillment workflow. At various stages, the manufacturer makes decisions about inventory levels, suppliers to use, and production schedules. These decisions are guided by customer expectations. For instance, when the manufacturer is ready to deliver the final product, it can send shipping specifications to various vendors (FedEx, USPS, UPS, etc.) and receive several bids for different levels of service (e.g., insurance, shipping time guarantees). The manufacturer then chooses the best service for the job, and this service may change depending on the outcome and timing of earlier manufacturing stages. Composition of large software services can be done in a similar manner to the manufacturing process, selecting the best software service vendor for each

customer requirement. A compiler for CLAM already generates variety of invocation sequences for current and developing standards for software interoperation, including CORBA, COM, and JAVA RMI.

CHAIMS is intended for building applications using "large services," where "large" may include massive data sizes, substantial runtimes, and significant cost. (We have demonstrated large software compositions for airframe design, multi-stage logistics planning, image processing, and several artificial workloads.) The size of service we envision typically justifies a dedicated host, although services can share single hosts when performance demands permit. A repository links suppliers of services and composers of applications, giving details of location, protocols, and data-types. In this work we focus on the composition of heterogeneous services, in contrast to the composition of multiple instances of like/identical services.

1.1 INTRODUCTION

Large software systems are difficult to build in a timely manner by just collecting requirements, analyzing the problem, and then partitioning the many functional components to programmers, and finally integrating and testing modules built by multiple programmers. This waterfall-style development of new software systems from the ground up is increasingly untenable as the scale of software grows. We know that in the near future large systems will typically be composed using libraries and existing legacy code to reduce development risk and cost.

These new composition programmers will use tools that differ from tools used by base programmers [69]. In composition, existing resources are catalogued, assessed, and selected, and systems are assembled by writing code to combine them. If the resources are distributed, the "glue" incorporates transmission protocols for control and data. Considerable expertise is needed for success: the composer has to understand the application domain, judge to what extent the requirements of the customers can be covered from existing resources, and often negotiate compromises. And then the

composer has to understand and manage the details of interfaces, options in the available resources, transmission protocols, and scheduling.

The CHAIMS project intends to fill the gap between a new compositional programming paradigm and current composition tools. By providing a language to support composition, CHAIMS supports a paradigm shift that is already occurring in industry: a move from coding as the focus of programming to a focus on composition. This shift may be invisible to some enterprises and educators, as there is no clear boundary in moving from subroutine usage to remote service invocation, though both tools and education are emerging to deal with this change.

1.1.1 BACKGROUND

Ten years ago, composition of large-scale software was performed by experienced groups in large companies, as in IBM, Unisys, Fujitsu, Arthur Andersen and the other "Big Five," and system contractors as SAIC, Lockheed, MITRE, Lincoln Labs, etc. Today, software composition has moved to *nix (UNIX and Linux variants) and PC platforms and an increasing fraction of the software workforce is engaged in composition, often without focused tools.

Several general-purpose programming languages have had some composition facilities included within their basic capabilities. Examples are the LEAP feature (an associative store) in SAIL, services in PLITS, the Courier Protocol in Interlisp, rendezvous in Ada, and tasks in PL/1. These facilities were often unwieldy and did not enter common usage. The complexity introduced by these features may have contributed to their decay. Other special-purpose languages (examined later) have had much more success for composition, but have very limited user bases.

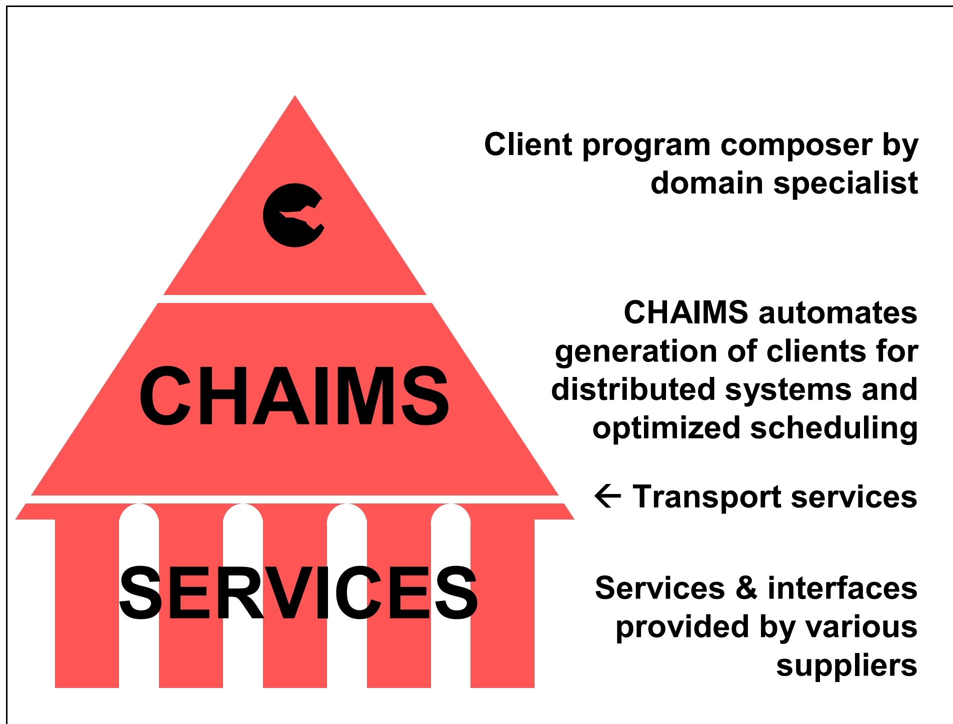


Figure 1 Components of the CHAIMS compositional paradigm

1.1.2 Architecture

At the highest level, and looking coincidentally (and only vaguely) like certain Stanford buildings, the components of CHAIMS are symbolically depicted in Figure 1. The topmost component is the client, represented by a program defining the composition. The client program only manages the invocation of the services, according to precedence constraints that typically represent dataflow dependencies. The client program is compiled by the CHAIMS compiler, developed by previous investigators [65]. Many service invocations will involve remote accesses, since we assume that services reside on multiple sites. Any needed data type or computational conversions to be performed in the dataflow are invoked externally to the client program, assuring that the client program is a clean representation of the architectural intent for this application (information on these conversions is the foundation of Appendix B). The actual services are either wrapped to serve CHAIMS primitives or written in a suitable programming language.

An example that captures the issues of invocation, dataflow, type conversions, and computational requirements that are all scheduled by a client program is the distributed query processing system embodied in Mariposa [43, 44]. The genesis of Mariposa was based on a series of "sites" that had various data available and different processing capabilities that could be combined to perform SQL-like database queries. Clients participating in the Mariposa environment could combine various sites (by paying for their services) to perform interesting data processing tasks. Because different sites had different availability, cost and performance characteristics, decisions had to be made at runtime about which sites were optimal to perform a particular query. The distributed query processing in Mariposa requires proper invocation timing and ordering, converting data types between sites, and balancing the dollar-cost of a query execution against the execution-time requirements.

1.1.3 Objectives

We observe that large-scale composition requires functionalities not available in current mainstream programming languages. Connections to remote sites have to be set-up, scheduling of computations that can operate on parallel has to be managed, data must be shipped among programs that are written in diverse languages, and a variety of transmission protocols have to be managed.

1. By defining a high-level language (CLAM) in which an application specialist can compose resources, the issues involved in composition are clarified. A compiler for the language allows us to test and assess the concepts needed.
2. To support the composition concept we need to define a protocol for communication that can be driven from a high-level language. To broaden the range of resources, our CPAM protocol can access services using any of a variety of existing interoperation standards, exploiting ongoing work in client-server technology. By linking to remote computational resources, we demonstrate the concepts to a broad audience and provide guidelines to deal with this paradigm change.

3. We also have to consider the roles that people play at the various points of the system. Like any large-scale organization, we recognize specialized roles. These roles are seen as long term, since the large-scale software will likely be long-lived, so that adaptation and maintenance is more important than the initial creation of a system. (People involved include program composers, service programmers, clients, etc.)

The CHAIMS project has a limited scope: there is no plan for automatic programming, CLAM supports only a few data types, and the CHAIMS environment uses many defaults rather than give the programmer a rich palette of choices in the invocation of software services. By focusing CHAIMS on interoperation in a multi-site environment, rather than on platform-specific code, we gain high-level support of software composition concepts.

We will explore these three issues in turn.

1.2 LANGUAGE

The CHAIMS programming language, CLAM, serves only service composition and scheduling. Its narrow focus allows it to remain simple, although some significant new concepts were introduced. By isolating the concepts related to composition and thus reducing our conceptual scope, CLAM addresses the specific issues within a research project of modest size.

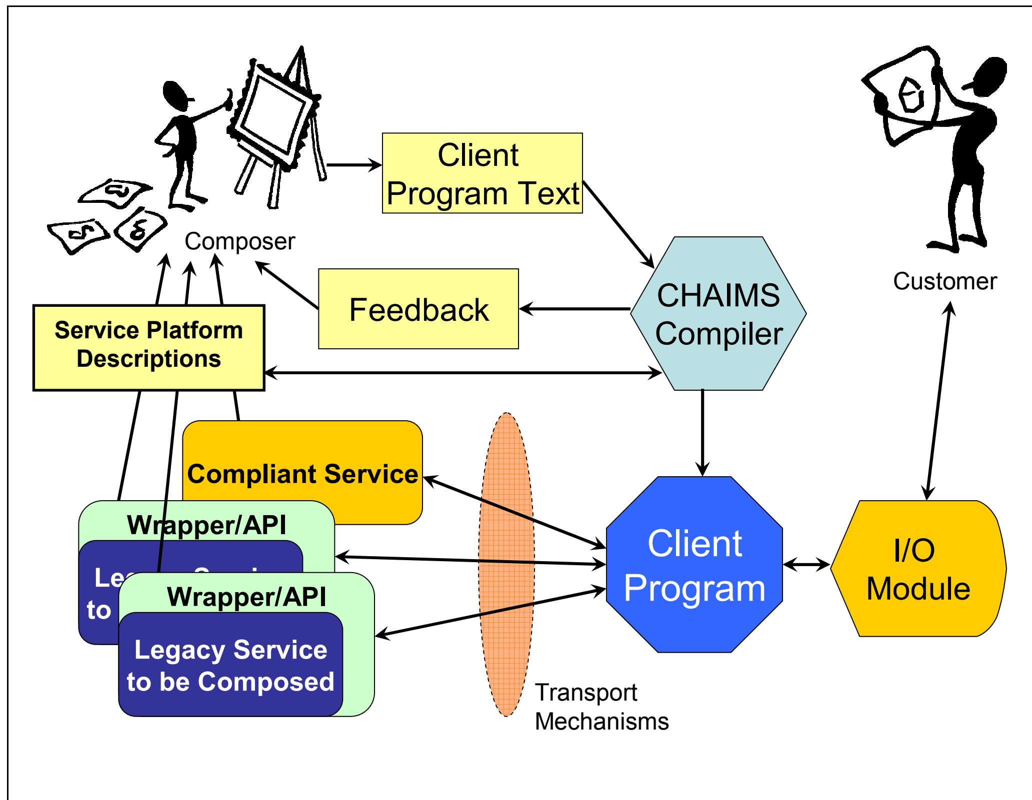


Figure 2 CHAIMS programming process and information flow

Since CHAIMS does not provide for automatic programming or knowledge-based techniques to automatically generate programs directly from service specifications, the composer should have access to a well-maintained repository of services. The process of CHAIMS client program generation and use is illustrated in Figure 2. The composer builds the source code of the client program (labeled "Client Program Text") in the composition environment. During that process the programmer will access service descriptions from the repository ("Service Platform Descriptions") and also obtain some initial feedback from the CHAIMS compiler. The final stage is the generation of the client program, a program capable of composing the services (legacy or native) needed to meet the customer's needs.

1.2.1 PRIMITIVES

The statements in the CLAM language fall into three categories:

1. Initialization and termination statements, which control connections to the services: SETUP, GETPARAM, SETPARAM, TERMINATE, and TERMINATEALL
2. Service invocation statements, which cause processing results or intermediate results for control to be generated: INVOKE, ESTIMATE, EXAMINE, and EXTRACT
3. Statements to control the flow of the client program: WHILE, IF .. THEN .., ELSE..

The specific functions of these statements are sketched in section 1.3.1. (The curious reader may skip forward to Figure 8 (page 65) for a sample program written in CLAM). CLAM places some logical and data dependency constraints on the execution sequence of these primitives. The major logical constraint is that each service being invoked requires an initial SETUP. An identifier generated by SETUP is used for all subsequent accesses to this service. INVOKE, starting the execution of a service method, also generates an identifier, and that identifier is needed for EXAMINE and EXTRACT statements to check and return results.

The services being invoked are assumed to be autonomous and independent from each other, but must individually support the CHAIMS protocol, CPAM. This allows suppliers to develop new services, to be easily accessed by CHAIMS programs. Such services can be programmed in the most suitable local language and use the best representation for the given sub-task and solution approach. However, in the beginning there will be few compliant software services available, so we provide wrapper templates for legacy software to satisfy CPAM.

CLAM supports only a few data types, for two distinct functions:

1. For control flow, CLAM supports identifiers, integers, Boolean, real, string, and a date type. The identifier type provides the handles needed for services and service invocations. Computations on the other types may be performed using a CHAIMS supplied native service module.

2. For processing data, CLAM supports a bag, which holds a complex object type, encoded using the ASN.1 standard (an updated version would likely prefer XML to ASN.1). A CLAM program cannot discern the content of a bag; it can only move it between services, including native services for input and output.

The native service modules, supplied with the CHAIMS system, appear identical to remote service modules, so that their behavior can be replaced. Such replacements may be needed if the client program lives in an unusual environment, such as an embedded system.

The ASN.1 standard supports a recursive data structure of triplets, where each data element consists of a descriptive name, type information and a value. The value is either again a triplet or a simple type, like a bit-map, an array of bytes, date-time, string, real, etc. CHAIMS wrappers convert this general object format into local service formats.

The CHAIMS compiler is not burdened with having to deal with many types, type conversions, or with code for fancy end-user input/output. Interaction with the customer running the client program is dealt with through native or substituted input/output services.

The binding of client programs to services can be delayed arbitrarily. Delayed binding simplifies maintenance, adaptation to changing network conditions, and changes in service capabilities and costs. Delayed binding also relaxes commitments to specific interface standards. Choices among standard interfaces such as CORBA, ActiveX, JavaBeans, and .NET can be deferred until the scale of the problem being addressed is known and the platform capabilities can be assessed.

1.2.2 Clarity

The structure of a client program within CLAM defines the architecture of a system, independent of its implementation. In effect, the program architecture will be clearly visible, and is not buried in interface and application-specific code. The client program defines an architectural instance of an application, while delegating all computational

activities to the services it invokes. The architecture instance defined in a client program is reusable, not as a paper document, but when invoked with appropriate services, as a complete re-instantiation of the domain architecture. Since at runtime the compilation can bind to alternate interface specification languages and to alternate services, scalability and platform independence can be achieved, as long as equally competent services can be acquired. The client program may be limited by the capabilities of current interface languages, but these are improving rapidly.

1.2.3 Parallel Computation

Modeling the activity of real world objects, parallel operation of services is an underlying assumption in CHAIMS. This vision means that the increasing availability of distributed computing can be exploited without resorting to parallel computing features applied to sequential programming languages. By considering any sequential dependency as an exceptional constraint, the composer will naturally think of parallel execution. This concept of natural parallelism is the alternative to the common paradigm seen today, where the programmer codes the actions to be performed in a world where everything lives in parallel into a sequential format, which is then subsequently analyzed by parallelization tools to extract possibilities for parallel execution. Since activities in a natural, distributed world actually occur in parallel, we hope that CHAIMS client programs can capture their essential parallelism without dual translations.

CLAM achieves new flexibility in service scheduling by having split the functions of the traditional CALL statement into units that can be scheduled independently: SETUP, INVOKE, and EXTRACT. The ESTIMATE and EXAMINE statements, described in more detail below, provide the information needed for their scheduling. In addition to exploiting parallel execution of services, the communication bandwidth requirement among them can also be significantly reduced. Documentation of the optimizations is covered in later chapters.

1.2.4 SUMMARY

One thrust of the programming language CLAM and the CHAIMS interface drivers is to reduce the cost of long-term maintenance and software evolution, and reduce the numbers of errors occurring in this process, without reducing the extent of maintenance and evolution actually being performed. Since CLAM only serves service composition and scheduling, client programs remain relatively simple, although efficient distributed computation is supported.

1.3 CPAM INVOCATION PROTOCOL

CPAM (CHAIMS Protocol for Autonomous Services) is our protocol for accessing and using the methods offered by services. Services offer their interfaces independent of a specific client. A client does not have to "own" a service; it just can use remote services, offered as methods to be invoked. The effect is that CPAM allows process composition.

CPAM has characteristics that specifically address the composition and reuse of autonomous, distributed and computationally intensive services [13]. CPAM calls allow for the client to be simple while exploiting the parallelism of methods invoked from different services. We will briefly describe the CPAM primitives, and then explain the benefits of their separation.

1.3.1 CLAM PRIMITIVES

We briefly summarize the primitives here, more detail is provided in the next chapter. CPAM, for the obvious reasons of co-development and shared objectives, closely parallels CLAM.

Establishing a connection to a service: The primitives SETUP and TERMINATEALL establish and end the connection from a client to a service.

Cost estimation: ESTIMATE allows a client to ask a service for cost estimates for a specific invocation. It is available both prior and during execution. The output is to be a name-tuple list (name of the cost factor, value of the cost factor and its uncertainty). If

the service cannot provide estimates, its wrapper returns as reasonable values as possible, perhaps based on past executions. Cost estimates allow the composer or a CLAM optimizer to choose among alternative services and/or optimal execution paths.

Executing service methods: Methods are executed by the following four calls: INVOKE, EXAMINE, EXTRACT and TERMINATE. INVOKE starts the execution of a method, which then proceeds asynchronously; multiple INVOKES with different parameters can occur within a single SETUP. EXAMINE reports the status of an execution, EXTRACT returns selected results, and TERMINATE deletes an invocation (but does not disable access to the service).

Presetting of attributes: The call SETPARAM is used to set default values for invocation attributes and global variables in a client-specific way. The complementary call GETPARAM simply allows checking of default values in a service or values set by SETPARAM.

1.3.2 CPAM Execution Alternatives

The CPAM protocol enables a variety of optimization patterns by application of its primitives:

1. As expected, multiple services can be SETUP in parallel, and their invocations interleaved. If useful, a single service can be SETUP multiple times, for alternate conditions.
2. SETUPs can be performed early and in parallel, so that subsequent data-constrained sequential INVOKEs can be rapidly executed. In some distributed computations, SETUP can take more time than method invocation.
3. EXAMINE permits traditional polling prior to EXTRACT, allowing the client to overlap execution of INVOKES. Nothing novel here, but EXAMINE can return progress indications from simulations and similar long-running or continuous executions (say weather predictions), so that the client can balance precision and result delivery time.

4. Multiple INVOKES to a service method can be made in parallel, typically with differing input parameters for the service, for instance to bracket a range of decision parameters.
5. A high uncertainty of costs of alternate optimization choices during compiling can be resolved by moving the ESTIMATE statement into the execution flow, when the values should have a lower variance.
6. ESTIMATE can be used to decide on scheduling or canceling of alternative services.
7. ESTIMATE can be used to make choices among service execution orderings, especially when pre-execution compilation can not resolve significant uncertainties.
8. EXTRACT can specify a few control values to be reported, to let the client program decide if a satisfactory solution has been obtained or more iterations of INVOKE with new input parameters are necessary.
9. Dynamic services can be interrogated using GETPARAM.

Many of these alternatives exist now somewhere, either as a feature of a specific programming system or through clever programming of an application. We do not know any approach which has provided the complete mix-and-match capability show in the CPAM protocol. (These features are explained in much more detail in Chapter 3. In that chapter, Figure 7 (page 62) shows the paths of execution that are possible with a single service; an arbitrary number of such paths can be interleaved.)

1.3.3 Interoperation Protocols

CPAM is a protocol which is mapped by CHAIMS onto a variety of existing client-server protocols. We have used CORBA, JAVA RMI, DCI, and DCOM with various degrees of success and happiness. However, the strength of CPAM is the flexibility of interface standard choice, changeable at any time.

The size of the services we envision typically justifies a dedicated processor, although services can share a single processor when performance demands are modest. Services written in C, C++, Ada, FORTRAN, etc., will need interfaces (similar to APIs) to allow interoperation in the CHAIMS setting. Interoperation protocols from standards as

CORBA, ActiveX, JavaBeans and DCE provide, in effect, machine languages that make the CHAIMS concept implementable today. By moving up one level of abstraction in programming, we are moving to a new paradigm where the focus is on "composition of services," rather than "assembly of code."

1.3.4 Adaptability to Changing Standards

The move to composed software is clear, but somewhat poorly focused. We now have as many proposed standards for software interoperation as there were computer hardware architectures thirty years ago. Modern programming languages now provide platform independence and programs can be recompiled for new hardware. The effect is that application software typically lives about 15 years, much longer than its hardware.

The same stability does not exist yet for interface software, especially when connecting computers of different types. For instance, SQL is undergoing changes in its transition to SQL-2 and -3. Selection and implementations of feature sets varies widely. OMG's CORBA 2 will have many features not available in CORBA. KQML concepts are being suggested for future versions of CORBA, while JAVA is taking over server-managed client computing. The intent of XML is to replace HTML for processing applications, but depends on an unknown variety of DTD definitions to achieve depth at the cost of consistency. New object-libraries arise, but are rarely compatible with those of other suppliers. We must surmise that interfaces for distributed computing are likely to stay fluid.

Just as traditional programming languages provide an insulation from platform differences, the essence of CHAIMS is to provide insulation for the composer from the differences in today's interoperation architectures and standards. Such independence is especially crucial for larger systems which need to operate on multiple sites, utilizing networks and a variety of services. These systems represent major investments, and have a long lifetime.

1.4 ROLE ASSIGNMENT

In a composition programming environment, we distinguish two primary functions, namely the servers' and the clients' roles. One application system is likely to use multiple servers, but be controlled by a single client. Other clients can be composed of the same and other services. Figure 4 shows the main components of an instance of a CHAIMS system. The repository provides the means for sharing information about all the available services, and links supplier, composer, and end-user.

1.4.1 SERVICES

The concept of services implies autonomous ownership. The control over the component or service remains at the provider's site. This is a major distinction from traditional software models, where components or services, no matter how large, are subservient to the calling routine. A prime example of services today is provided by databases, but there are also some computational services currently available. The Internet provides a wide variety of services, although they were rarely envisioned for composition. Examples include weather services, airline ticketing, and book sales. Other potential services include simulation programs, design and construction programs, services for genomics [75, 77], for manufacturing [74], and business services [76]. Many more are expected to come into existence. But there exist yet few protocols supporting an integrated vision and allowing easy reuse and composition into a larger system.

The economics of a service are based on reuse. By collecting fees or sales for each use, it becomes a benefit to the service provider to maintain the service. The rules and processes encoded in a service represent knowledge. This knowledge is subject to change, not just the data it operates on. As the world changes, recourses broaden and algorithms improve. Bringing this updated knowledge to the client, either by reusable components or as updated concepts and requirements to be integrated and implemented by the customer into their programs, is cumbersome. Legal issues for service providers in this context have been analyzed by [73].

Time and fee can be estimated by the service, and can then be directly taken into account when calculating the cost of using a service. This is often not true for the third cost factor, data volume. Though the amount of data flow can be estimated by the service, the effective cost is not the *amount* of data but the *time* the data needs to be transferred. This time is client specific because it not only depends on the amount of data but also on the capacity of service interconnections, i.e., quality of connections, distance, and bandwidth.

Note that for optimizing client processes, a high precision in estimation is not crucial. When scheduling alternatives are limited, and if choices are close in cost, then the "choice" does not really matter. If the choices differ greatly, then the proper choice may also be obvious, even if precision is low. Variance is more of an issue, since a high variance means that more choices must be deferred to execution time.

1.4.2 COMPOSITION

The programmer who writes the client programs becomes a composer of services. The composer's tools are knowledge of an application domain, an understanding of services at a high level, the specific interface information captured in a service repository, the CLAM compiler, and perhaps wrapper templates [13]. Our repository implementation is currently a simple text file with a graphical browser. It contains a description of all available services, their methods, and their attributes. All the valid service, method and attribute names are posted in the repository. The repository is the only formal information flow necessary between providers, composers, the end-users or customers using a client program, the compiler, and other tools in CHAIMS. The current compiler compiles a client program written in the composition language CLAM into a client side run-time (CSRT), including the generation and compilation of all necessary stubs for various distribution systems. The wrapper templates are provided as part of the CHAIMS system to facilitate the wrapping of legacy services into CPAM compliant services.

The composition language CLAM, together with the CHAIMS system, disengages domain experts from technical details like the use of complex programming languages and the programming of distribution systems. This can be compared to today's use of

database management systems, where the SQL programmers are quite distinct from the programmers who work at the DBMS provider, and it is unlikely that they have ever met. We hence assume that these two roles may be occupied by different persons with differing skills and objectives.

CHAIMS should make it easier to train domain specialists in composition rather than "untrain" conventional programmers that are used to all forms of restrictions. We still have to learn to what extent CLAM should not be similar to an existing sequential programming language, so that it will be clear to the composer that the environment is inherently parallel and that much prior experience and training will have to be unlearned.

1.4.3 APPLICATION CUSTOMERS

The customers of the client-programs live in the same domain as the composers, so that interaction is easy, and adaptation to changing customer need is rapid. If the customer is technically confident, and the tools are reliable, the customer and the composer may be the same person. When new services are required, searches through the repositories are the first task. If no suitable service is found, then wrapping of a legacy service has to be negotiated.

The focus of CHAIMS is mainly on computational services, complementing the now dominant information services. Especially in the domain of web-information, information is downloaded from an information server, and then put semi-manually (cut and paste plus maybe some cumbersome conversions, screen-scraped, etc.) into another program that performs actual processing, say, a spreadsheet. Our early demonstrations focused on logistics and scheduling applications, since in that area suppliers of transport, warehousing, and breakdown services have a motivation to provide assistance to their customers, while the customers have many choices to assemble these services (even if no fee is being paid). Our case studies include a logistics example ("find the best route from city A to city B under certain circumstances") using several services for the various parts of the computation, and an aircraft design example with services for the computation of the structure, the control elements and the static of an aircraft wing.

1.5 DISSERTATION OVERVIEW

In this dissertation, we describe a system that supports a non-traditional, but realistic approach to software systems development. Rather than following the waterfall model or its variations by starting from specifications, through design, etc., to code generation, CHAIMS assumes that large programs can best be composed from existing services. This limits the application client programs to the composition of available resources [71]. A substantial benefit of the approach is in terms of long-term maintenance of computer systems, which typically consume 70-90% of total systems costs.

In a compositional programming approach, composers are willing to give up control for the benefit of expert maintenance at the source sites in a collaborative setting [5]. CHAIMS work distinguishes itself from database integration by incorporating knowledge embedded in programs, rather than being limited to declarative knowledge applied to databases. Database functionality can certainly be incorporated into compositional programming through server programs that execute SQL SELECT statements, but these languages - focusing on a single verb - are known to have inherently limited computational capabilities [78].

Compositional (or, again "Grid") programming can also be viewed as large-scale object-oriented (OO) technology. OO increases the procedural capabilities of distributed objects [72], but is restricted in practice to single protocols and coherent libraries [68]. The high-level language approach of CHAIMS scales the object-oriented paradigm to autonomous service objects. Programming through service composition is an OO exercise.

Chapter 2 describes the CLAM compositional programming language in detail. The development of CLAM was an important piece of the investigation of compositional programming, and represents a purely compositional language. CLAM harnesses the asynchrony present when dealing with independent information suppliers. The inclusion of cost estimation methods allows easy online examination of invocation costs; that

language-level support means that CLAM can easily handle runtime optimizations not possible in current language approaches based on the traditional CALL routine.

Chapter 3 describes CPAM, our runtime protocol to support service composition. CPAM enables efficient composition of large-scale services by optimizing the invocation sequence, and minimizing data flow between client and service. The CPAM protocol gives sufficient information to a compiler or a client program to enable automated scheduling of composed software at compile-time, and more significantly, again at run-time.

Chapter 4 advances the argument that a simple asynchronous RPC-style of data extraction was not effective when building a composition-only language for remote, autonomous services. After considering different extraction models used in multiple programming domains, we developed a language and access protocol to support the myriad result extraction models in use. With this increased functionality, intelligent schedulers can now perform optimizations not previously possible. Early result extraction, fine-grained progress monitoring, and successive refinements all become scheduling options enabled by our model.

Chapter 5 present an overview of our scheduling technique, Surety-Based Scheduling, that overcomes much of the uncertainty naturally present in distributed computing environments. We demonstrate how to take a program composed of multiple distributed services and complete it within a client-specified soft deadline. Our scheduler generates initial schedules, collects information about runtime progress, and further repairs a running schedule if runtime difficulties are encountered. This scheduling work is broadly applicable to systems whose distributed nature is impacted by uncertainty.

Chapter 6 shows that our scheduler is successful more often than any alternative scheduler in normal, highly, and loosely constrained environments. Our scheduler generally finds *some* schedule that will run to completion, and the successful schedules that it finds have cost and execution times that are statistically indistinguishable from ideal schedules. No other scheduling mechanism provides our level of *schedule quality*,

though even the notion of scheduler quality is novel to most systems. We also show that even in resource rich environments, if the cost of failure is high, there is additional reason to favor our scheduler over alternatives.

Chapter 7 discusses future directions for research.

Appendix A describes the Critical Path Method (CPM), a well-known scheduling technique that is leveraged by the Surety-Based Scheduler.

Appendix B offers a conceptual framework for active mediation to increase the customizability and flexibility of autonomous services. Our evolutionary approach allows coexistence of active mediation and static mediation, making it feasible to build a service composition infrastructure that supports active mediation. Through the discussion of the application scenarios, we reveal the importance of active mediation in conducting service composition, and show a spectrum of scenarios that can be effectively addressed by our active mediation techniques. Appendix B also introduces an algorithm to determine the optimal placement of mobile classes for active mediation.

2 THE CLAM LANGUAGE

Advances in computer networks that support the invocation of remote services in heterogeneous environments have enabled new levels of software composition. We envision a purely compositional language to manage composition at such a high level. To this end, the CLAM composition language was introduced, a language designed for specifically for the improved control of large services.

The CLAM language focuses on asynchronous composition of large-scale, autonomous services and is guided by the decomposition of the traditional CALL statement. The language has the capability to support various optimizations specific to large scale software composition.

2.1 INTRODUCTION

Component software revolutionizes the traditional, programmatic method of building software by introducing a new paradigm in which software is created by leveraging a market of reliable and secure software components [1]. While the component model of software engineering has been promoted for several years [2], it is only in the past few years that infrastructure technology enabling such composition has become stable and sufficiently widespread to encourage serious adoption of the model. In the past 5 years, we have observed dozens of projects aimed at squarely grid computing (such as Globus [38], or the $10 billion internal “investment” in grid computing at IBM [108]), and at least as many paying lip-service to the promise of the grid (e.g., Oracle’s 10g database has been billed as “The World's First Self-managing Grid-Ready Database.”).

Distributed component software relies on protocols such as DCE [3], CORBA [4], DCOM [1], .NET, and Java RMI [4]. Using these protocols, a core software service can be made available to many clients. In order to use software services with these protocols, a client program must conform to the Application Program Interface (API) of the desired distribution protocol. However, different services will use different distribution

protocols, and these protocols do not effectively interoperate (it may be that "web services" will be the answer to interoperability, but that remains to be seen). In order to use distributed software services as part of a large-scale composition today, a client programmer must be familiar with heterogeneous distributed systems, client-server programming, large-scale software engineering, as well as the application domain. While these requirements are not unrealistic for modest programs composed of small-scale components (such as ActiveX controls or JavaBeans [4]), as components and assemblies scale upwards, and as more not-technically skilled domain experts would like to make use of distributed components and compose them, these become increasingly unrealistic expectations for most practitioners.

In the CHAIMS (Composing High-level Access Interfaces for Multi-site Software) project, we have attempted to reduce the knowledge requirements for composition by distinguishing between *composition*, *computation*, and *distribution*. We have attempted to free the compositional expert from the need to know about distributed systems or computational programming. (We refer to a "compositional expert" as an expert in some non-programming discipline, say genetics, that is interested in composing grid services to solve some problem within their discipline.) We recognize that the compositional expert needs added control over the execution and timing of remote method calls as remote server software scales in size. Furthermore, the ability to integrate components that are only accessible by differing distribution protocols is necessary. In later chapters, we present the Surety-Based Scheduler to remove even these mundane considerations from the purview of the compositional expert, but for now assume that the expert is interested in all facets of the composition task.

With this in mind, we introduce a programming language, CLAM, designed for the composition of large-scale modules or services. A client program that composes various services has been alternatively known as a "megaprogram," a "grid program," an "ensemble," or even just a "composition" [5, 7, 38, 103]. Services are likely written in different programming languages, reside on different machines, and integrated through different distribution systems. A runtime system is required to bridge the differences in

architectures, languages, and distribution systems. We believe that the CPAM (CHAIMS Protocols for Autonomous Megamodules) component of CHAIMS adequately addresses the major differences at the runtime level. The support provided by CPAM presented a unique opportunity to investigate a language that strictly addresses composition, without worrying about the aforementioned differences in architectures, module implementation languages, and distribution systems. We discuss CPAM in greater detail in Chapter 3.

We would also like to stress the differences between *coordination* and *composition*. Coordination languages have been studied and used for years, and include languages like SAIL, Ada, and Jovial. Coordination is closely related to the concepts of synchronization, proper ordering, and timing. Composition, on the other hand, is concerned with the act of combining parts or elements into a new whole. Appropriate *composition* of autonomous services is an important step that must be taken before specific *coordination* events occur.

2.2 CLAM IN A SEA OF LANGUAGES

2.2.1 OBJECTIVES

The structure of the CLAM language is motivated by the features a grid programming language should support. A language intended for a large-scale environment should implicitly take advantage of parallelism, rather than assume sequential execution. We also expect such a language to support compile-time as well as run-time optimization. The control structures within the language should reflect the simple elegance achievable when composition is the *only* goal, rather than the traditional composition coupled with computation. Finally, we assume that the environment in which a general grid programming language operates is somewhat heterogeneous, and therefore the runtime architecture for the language must bridge competing distribution protocols. Language specific requirements should not impose additional limitations on the supporting runtime system.

2.2.2 Other Approaches to Composition

With base-level runtime support for heterogeneous service interoperability provided by a system like CPAM, the focus of the language may turn to service composition. There are myriad languages designed for *computation*, some of which also have support for *coordination*. Several such languages and language extensions include FX, Orca, and Opus [10, 14]. These languages were not designed to be purely (or seemingly even "primarily") compositional languages, however. As such, each of these more "traditional" languages suffers from deficiencies when used for service composition. For instance, FX requires that all services are written to conform to the FX language, including data representations. FX codes are also *static* compositions, and do not have the dynamism expected in an online system [9]. Orca has its own distinct runtime support for use with its composition primitives, thus radically limits the potential for use of legacy codes in a composed program. The Opus extensions to High Performance Fortran (HPF) take a data-centric view that requires a programmer to have intimate knowledge of all data in a process, including typing and semantic meaning, to set up a composition [9, 10]. This would be a heavy burden to place upon a compositional expert.

Developing a language with unique features particularly suited to specific problems is not new. LISP was developed to be good at list processing and has seen widespread application where that ability is critical. FORTRAN was designed with significant support for scientific computation and performed well in that domain, even before additional features such as dynamic allocation of memory were added. CLAM is somewhat unique in its unwavering and exclusive focus on one problem domain: composition. CLAM does not limit its application potential by focusing on specific implementation schemes or computational models. To that end, list processing, scientific computing, database access and updating, and all other work performed on user data is left to composed services. CLAM only composes services and does not restrict itself in ways that any particular service's implementation language must necessarily do.

2.2.3 MANIFOLD

Like CLAM, the MANIFOLD language is a compositional language [11]. MANIFOLD has a control-driven semantic (via "events") which makes it quite different from the data-centric approaches of the languages mentioned in 2.2.2. The MANIFOLD language has the following building blocks: processes, ports, streams, and events [12, 46]. Processes are black box views of objects that perform useful work. Processes are most similar in concept to the work performed by a grid service. Ports are access points to processes. Ports are most similar in concept to the access interface of a grid service. Data is passed into and out of ports. Streams are the means to interconnect processes and ports. We discuss the CHAIMS equivalent of streams, the mediation among composed services, in Chapter 9. Events are independent of streams, and convey information through a different mechanism.

MANIFOLD assembles processes using the Ideal Worker Ideal Manager (IWIM) model [12, 47]. In IWIM, compositional codes (known as "managers") select appropriate processes ("workers," in IWIM), even online, for a given problem. The processes selected by a manager are workers for that manager. Nesting is possible in the IWIM model, so that one manager's worker may manage workers below it.

IWIM can be viewed like a contractor relationship: a general contractor may hire subcontractors to perform a certain job, and the subcontractors in turn may have additional subcontractors. The general contractor is not concerned with additional layers of contractors/workers, only that the original subcontractor completes the assigned task. MANIFOLD and CLAM both follow this model.

Unlike MANIFOLD, CLAM is not restricted to the simple black box object view of service providers. CLAM treats objects that provide services as entities with exposed methods. These methods can be accessed in a traditional way: invoked with input parameters and with available return values. This can be done in MANIFOLD as well. However, CLAM extends this simple notion of composition and task atomicity by decomposing the traditional CALL statement, as described in 2.3.1 and 2.3.2. In this

way, CLAM allows for increased control over scheduling, and allows runtime inspection of processes. With similar composition features to MANIFOLD, coupled with further CALL statement decomposition, CLAM achieves greater levels of expressiveness than MANIFOLD. This is shown in more detail in 2.3.2, where we examine optimization opportunities available when the traditional CALL statement is extended with additional functionality.

All MANIFOLD programs can be expressed in CLAM, but the reverse is not true. Because of the special scheduling primitives available in CLAM, more control over process scheduling is possible. Without these scheduling primitives, and without extensions in CLAM to reuse input parameters to remote service invocations, CLAM would be very similar to MANIFOLD.

2.2.4 Sample CLAM Program

The intentions of CLAM are not muddied by complexities of computational and runtime issues because CLAM limits itself to being a *composition only language*. CLAM provides a clean and simple way to achieve the composition achieved with difficulty in other languages. CLAM is not just a mental experiment or toy exercise. It has been used to write working programs that were later compiled with CPAM runtime support. One such program example that frequently appears in the CHAIMS literature is a composition that finds the least expensive route for transporting goods between two cities [8, 13]. This transportation program composes five services to achieve the goal.

The following fragment is from one version of the transportation program, though for clarity, certain trivial elements have been omitted:

```
// Fragments from megaprogram for finding the cheapest
// route for transporting certain goods between cities
// bind to the megamodules
best_handle         = SETUP ("PickBest")
route_handle        = SETUP ("RouteInfo")
cost_handle         = SETUP ("AirGround")
io_handle           = SETUP ("InputOutput")
best_handle.SETPARAM (criterion = "cost")

// get information from the user about the goods to be
// transported (start and end time, size and weight)
// and the two desired cities
input_cities_handle = io_handle.INVOKE ("input_cities")
input_goods_handle  = io_handle.INVOKE ("info_goods")
WHILE (input_cities_handle.EXAMINE() != DONE) {}
cities = input_cities_handle.EXTRACT()

// terminate call to "input_cities" within InputOutput
input_cities_handle.TERMINATE()

// get all routes between the two cities
route_handle = route_handle.INVOKE("AllRoutes",
  Pair_of_Cities = cities)

...

//terminate all invocations with "InputOutput" module
io_handle.TERMINATE()
```

2.3 CLAM LANGUAGE SEMANTICS

2.3.1 PARALLELISM

The compositional programming language CLAM is for scheduling and organizing remote, autonomous services. Since remote services can all operate in parallel, the programming language coordinating these services should be as asynchronous wherever possible, mirroring the everyday world where tasks are often done in parallel. Asynchrony is particularly important for large-scale services in which each invocation can be expected to be both long running and resource intensive. Synchronous invocation works well for small-scale services. However, as remote services increase in size and power, it is increasingly important that the client has the ability to decide when it wishes to start a specific invocation. In addition, the client must have control returned

immediately after starting one invocation so that it can initiate and monitor other invocations. When the client needs the results from a particular service, it can wait for that service as necessary, explicitly synchronizing as necessary.

In traditional programming languages the primitive used to invoke a particular routine, which we refer to as the CALL statement, typically assumes synchrony in execution, forcing possibly parallel tasks into a sequential order or requiring explicit parallelism at the client side. By breaking up the CALL statement into several primitives, we can capture the asynchrony necessary to support parallel invocation of remote services from within an apparently sequential client program. There is another reason for breaking up the CALL statement. The CALL statement typically performs many functions: handling the binding to a remote server, setting local parameters, invoking the desired method, and retrieving the results. For large-scale composition, thinking of the CALL statement as an atomic primitive often makes the system somewhat unwieldy [5]. Therefore, dividing the CALL statement into several primitives gives the programmer more control over the timing and the execution of the various functions of a CALL statement.

To harness the potential parallelism within a grid program, we decompose the traditional CALL statement into the following primitives:

2.3.1.1 SETUP

The purpose of the SETUP call is to establish communications with a service. SETUP can mean different things depending on the runtime system supporting CLAM. When a CLAM program is compiled to CPAM, SETUP establishes communications to services using whatever service location scheme is appropriate (e.g. CORBA). Using CLAM with a runtime system like MARS [7], SETUP would likely be required to start a server, rather than simply establish communication to a static one. SETUP is necessary in the language to direct the runtime system to service descriptions at compile time, and to open a communications channel at runtime.

The SETUP calls in the sample CLAM program introduce CLAM handles, e.g.:

```
best_handle  = SETUP ("PickBest")
```

The call to SETUP returns a handle to the method requested, without regard to the locations or implementation of the method or service. Messy details such as network location and transportation protocols, unimportant to the compositional expert, are not used in the language, but briefly introduced in section 2.4.1 for the interested reader.

2.3.1.2 SETPARAM

The SETPARAM call is used to establish parameters referred to in any method of any particular service that has already been "SETUP." It can also be used to set global variables within a service, i.e. variables used by multiple methods within a single service. If no parameters are set by SETPARAM, the assumption is that the service has suitable default values for an invocation. Many remote procedure calls to major services include various environment variables that are repeated with every invocation. CLAM eliminates that overhead with SETPARAM.

SETPARAM is called using the method handle returned from a SETUP. Again, we see this in the sample program:

```
best_handle.SETPARAM (criterion = "cost")
```

Here, SETPARAM is used to set a global value for criterion. Whenever criterion is required in any method within a particular service, the value set by SETPARAM may be used for free, or overridden for a particular invocation.

2.3.1.3 GETPARAM

The GETPARAM call can return the value of any parameter of any method of a particular service. It can also return any value of that service's global variables. GETPARAM can be done immediately following SETUP to examine initial and/or default values within a service. It can also be done after a method invocation to inspect changes to global variables, if needed.

2.3.1.4 INVOKE

The INVOKE call starts the execution of a specified method. Any parameters passed to a method with a call to INVOKE take precedence over parameters already set using SETPARAM. Execution of a specific method returns an "invocation handle." This handle can be used to examine a specific instance of an invocation, and to terminate it as well. We see invocation handles returned in the sample code from section 2.2.4:

```
input_cities_handle = io_handle.INVOKE ("input_cities")
input_goods_handle  = io_handle.INVOKE ("info_goods")

route_handle = route_handle.INVOKE("AllRoutes",
  Pair_of_Cities = cities)
```

With this expressiveness, programs can request multiple instances of the same method from a single service provider. Single instances can also be terminated with handles. Also, invocation is not a synchronous activity, so we perceive the above pairs of services as starting concurrently.

2.3.1.5 EXTRACT

The EXTRACT call collects the results of an invocation into a list. Any subset of all parameters returned by an invocation may be extracted. Extraction can also occur at any point in the execution, including partial data extractions or extractions of incomplete data. It is up to the programmer/compositional expert to understand when (and even *if*) partial extraction is meaningful. We later discuss how the EXAMINE primitive reveals that an invocation is DONE or NOT_DONE, giving the programmer some insight into execution progress.

Extraction of data is also shown in the transportation code. Note that the return value is *not* another handle type, but rather a storage location for the returned data:

```
cities = input_cities_handle.EXTRACT()
```

2.3.1.6 TERMINATE

The TERMINATE call ends either a running invocation or a connection from a client program to a specific service. Terminating a connection to a service necessarily ends all running invocations within that service initiated by the client issuing the TERMINATE. Invocations belonging to other client programs accessing the same service are unaffected, of course. Termination of a specific method from a single client, without termination of the connection, does not invalidate values set by SETPARAM.

Calls of the two possible termination types are shown in section 2.2.4:

```
input_cities_handle.TERMINATE() // acts on a method
io_handle.TERMINATE()           // acts on a service
```

The first TERMINATE shown above acts on a single method invocation. The second type of TERMINATE show above ends all invocations within a particular service instance.

With this breakdown of the method CALL process, CLAM can take advantage of both implicit and explicit parallelism. Assuming that all traditional CALL statements are asynchronous, some parallelism can be achieved without the decomposition found in CLAM. However, some decomposition is implicit in asynchrony as callbacks are required to retrieve data from invocations. Also, traditional asynchronous CALL statements do not yield the scheduling benefits from the primitives described in 2.3.2. Nor do they allow for SETPARAM type operations; execution is an “all or nothing” proposition.

2.3.2 CLAM-Enabled Optimizations

As components increase in size, the ability to schedule and plan for the execution of remote services becomes increasingly important. Traditional optimization methods have generally focused (quite successfully) on compile-time optimization. The CLAM language, coupled with the CPAM architecture (or another suitable grid-enabled runtime) can support both run-time and compile-time optimization, thus conforming to the

dynamic nature of a distributed environment. Here we extend the work done on dynamic query optimization for databases into software engineering [6].

There are both compile-time and runtime optimizations possible with CLAM. The possible compile-time optimizations are unique to each specific supporting system (RMI, CORBA, etc.), so we focus here on the runtime optimizations with CLAM. There are four main runtime optimizations that are enabled by the language: simple selection among competitive services, optimization of call setup parameters, parallel scheduling of services based on cost functions, and partial data extractions.

2.3.2.1 ESTIMATE for Service Selection

In a widely distributed environment, the availability of services and the allocation of resources they need is beyond the control of the programmer. Furthermore, several competing services may offer the same functionality at program runtime. Therefore a client must be able to check the availability of services and get performance and cost estimates from services prior to invocation. This must be done at run-time, as any compile-time estimations may change by the time a program is executed. Traditional CALL statements do not consider execution cost estimates. As performance and cost estimates become increasingly important to both service users and providers, explicit language support becomes essential.

The ESTIMATE primitive in CLAM helps service composers take advantage of runtime selection. An ESTIMATE call returns an estimation of the cost of a specific service invocation. CLAM recognizes three metrics for estimation: invocation cost, invocation time, and invocation data volume. (When we introduce Surety-Based Scheduling, we will concentrate on cost and time estimates, leaving issues of data to the data flow work presented in Chapter 3 and in Chapter 9.) At this point in our story, it is up to the programmer to use the cost and time estimates in a meaningful way. With the information returned from ESTIMATE, programs can choose to use invocations from specific service providers based on costs of their services.

An example of using values from ESTIMATE to steer execution (note, INVEX is short for "invoke and extract," and shown in 2.3.3):

```
cost1 = method1_handle.ESTIMATE()
cost2 = mathod2_handle.ESTIMATE()
IF (cost1 < cost2) THEN result = method1_handle.INVEX()
ELSE result = method2_handle.INVEX()
```

ESTIMATE provides the CLAM language programmer more ability to explicitly schedule services than a language like MANIFOLD. This makes ESTIMATE a useful language addition for scheduling, without limiting CLAM in other ways.

By using ESTIMATE, program users (or compiled programs) can make online, runtime choices about which services to use. These decisions may be based on whichever factors are most important during that particular service invocation. In some instances, importance may reside with the cost of the service, the time necessary to deliver an acceptable solution, or the data volume returned from a particular query.

Further uses of ESTIMATE may include compile-time scheduling hints. With a primitive like ESTIMATE, compilers can generate requests for estimates of costs for specific services at compile-time. This information can be used as "hints" to schedulers at runtime. ESTIMATE is an "estimate" in the truest sense of the word: the Surety-Based Scheduler does not assume that estimates are infallible. The SBS considers available reliability information when making scheduling decisions, including historical data on the accuracy of previous ESTIMATEs for a service.

2.3.2.2 Parallel Scheduling

It can be difficult at compile time to appropriately schedule distributed services for optimal performance, especially as the execution environment changes. Even with compile time estimates for service execution times/costs, actual runtime performance may differ significantly. To combat the problem of compile time naiveté, we include a runtime service examination primitive in CLAM.

2.3.2.3 EXAMINE

The EXAMINE call is used to determine the current state of a service invocation. EXAMINE returns an enumerated status value to the megaprogram. It can return DONE, PARTIAL, NOT_DONE, and ERROR. The examination of a service before invocation is essentially meaningless, and should always return NOT_DONE. Discussions about extension of CLAM indicated that an important addition to EXAMINE was an indicator of the *degree* of completion: a value (usable by a programmer) whose meaning is *not* enforced by the language. Such an additional return value may be used to indicate the level of progress of an invocation (or anything else of interest), but is not specifically required of services.

EXAMINE is used in the transportation example in section 2.2.4 to synchronize the megaprogram after section of parallel code:

```
WHILE (input_cities_handle.EXAMINE() != DONE) {}
```

EXAMINE may be called with the name of a result value as a parameter, to retrieve information about a specific parameter.

In many cases, compile time scheduling is possible. However, with EXAMINE, distributed services can be scheduled based on much more accurate runtime information. Programmers have language level scheduling ability with CLAM. This is not found in other languages like MANIFOLD.

2.3.2.4 Progressive Data Extraction

The EXAMINE primitive allows for online process steering. User-steered speculative scheduling may be done based on estimated times of completion returned from EXAMINE. More advanced scheduling and steering can be achieved as well, as we will cover later. First, recall that EXAMINE has a second parameter, the meaning of which is understood by the programmer. A call to EXAMINE may return “PARTIAL” in the first field and “70%” in this second field, indicating a 70% completion of the result at that point. When a surety level of less than 100% for program completion is acceptable for a

particular problem, then early extraction based on an EXAMINE statement can save users on potentially many fronts, including process time and cost. This is another case where Surety-Based Scheduling can benefit program composers.

The progress information provided EXAMINE allows users to extract partial results at various stages of execution. This means client programs can better steer executions based on estimated completion times. Also, if a result does not appear to be converging to an appropriate solution as a service approaches completion, the service can be terminated early, saving the client both time and cost.

2.3.2.5 Optimizing Setup Parameters

A programmer may also wish to dynamically check the performance of various services by optimizing various setup parameters, e.g., search parameters or simulation parameters. These parameters may influence the speed and quality of the results, and the programmer may need to try several settings and retrieve overview results before deciding on the final parameter settings. This is best done during run-time execution. This type of language support is also included in the CLAM specification.

The SETPARAM primitive makes perturbation of input sets easy. By inspecting service progress using ESTIMATE (shown in section 0), users can perform online tests of invocation status. When “tweaking” input parameters to expensive services, partial extraction of data (shown in section 2.3.2.2) and early termination may yield substantial cost savings. With certain classes of problems, particular setup parameters may not converge on a solution. Using EXAMINE with partial extractions can save users significant costs by *not* computing unacceptable solutions. With many languages, users must wait until a process is complete before testing the viability of the results. There are no primitives for interim process examination.

2.3.3 Simple Control Flow

CLAM was designed with service composition in mind, and seemingly presents a very limited set of primitives for programmers to work with. In reality, the CLAM primitives

are a rich set tailored specifically to service composition, with little regard for providing functionality beyond composition. Even complex comparisons and altering control flows based on service output require helper modules to analyze and reason about user data.

CLAM strives to remain a language solely for *composition* and not for *computation*. As such, the use of simple CLAM data types for activities other than control flow is quite restricted. Programmers can assign to a limited set of simple data types (mentioned in section 2.4.2), and make comparisons, but little else. Because of this, the language is very compact.

Control flow is achieved through the use of simple IF and WHILE constructs. Both are used with Boolean expressions composed of comparisons among simple data types. The WHILE loop has the following form:

```
WHILE (Boolean Expression) Statement_list
```

The WHILE is a control mechanism used before making invocations that depend on termination of previous event(s). There are many forms of loops available in different languages, but only the simple WHILE in CLAM. Other loop types, such as DO…WHILE and DO…UNTIL can be constructed from the simple WHILE loop.

The IF statement is a control mechanism used before making invocations which depend on the value of previous result(s) or meta-information such as performance data. The IF statement has the following well-known form:

```
IF (Boolean Expression) THEN Statement_list
[ELSE Statement_list]
```

Boolean expressions take many forms. The simplest Boolean expression is a single Boolean variable. Equality tests between integers, strings, and Booleans also yield Boolean results (a == b). Comparison tests may be done for integers and reals ($a<b$, $a>b$, $a<=b$, $a>=b$).

By using the IF and WHILE statements, programmers can write programs in CLAM that have conditional execution paths. These paths may change based on multiple factors, including service availability (at program setup time), estimations of invocation costs (before potential execution), and progress of particular services (during execution). The sample program shown above shows this use of a WHILE statement within a CLAM program, and the example for optimization shows the use of an IF statement.

There are two shortcut primitives in CLAM: INVEX (for INVOKE-EXTRACT) and EXWDONE (for EXTRACT when DONE). These two shortcuts are useful for synchronizing parallel executions, and stalling pipelined programs until critical results become available.

INVEX

The INVEX call starts the execution of a specified method the same way the INVOKE primitive does. However, INVEX does not return until it collects the results of the service invocation. There is no possibility to EXAMINE or TERMINATE an INVEX call, though this is still under investigation (with INVEX, there is no reason to EXAMINE or TERMINATE, except in case of error). The INVEX shortcut, operating similarly to the traditional CALL statement, blocks until method execution is complete and results have been collected. This can be an important short cut when strict synchronicity is desired.

EXWDONE

The EXWDONE call waits until all desired results from a particular service are available and then collects those results. There is a clear relationship between EXWDONE and INVEX; INVEX is equivalent to INVOKE immediately followed by EXWDONE. As such, EXWDONE facilitates the addition of instructions between an INVOKE and a corresponding EXTRACT from a single method, with the same possibility for synchronization as INVEX.

2.4 CLAM IMPLEMENTATION

The complete CHAIMS system is based on three main components: CLAM (the language), the CHAIMS compiler, and CPAM (for runtime support). It is a working implementation of the CLAM language. Because a compiler and system supporting the current version of CLAM was developed parallel to CPAM, they coexist naturally. However, the CLAM language is not restricted to use with CPAM and the CHAIMS system. CPAM provides runtime support for myriad communication protocols to CLAM, but is not critical to the language.

As a language, CLAM can compile to alternate runtime systems, such as MARS (Multidisciplinary Application Runtime System), developed while the author was at the University of Wyoming [7]. MARS, like CPAM, is a self-contained runtime system capable of exchanging data between heterogeneous, autonomous services. As such, CLAM is an independent part of CHAIMS, with unique language features specifically appropriate for compositional programming.

2.4.1 CLAM REPOSITORY

To keep implementation details out of the language, there must be a clear mapping of names between a service and information needed by the runtime system to invoke the service. For instance, in the sample used throughout this chapter, there are several hidden details even within the single statement:

```
route_handle = SETUP ("RouteInfo")
```

"`RouteInfo`" is a service that the programmer wishes to use, but where is it located on the grid? What protocols does it use to communicate? The issues of location and communication protocols are of no real concern to a programmer, but there must be a way for the service specified in CLAM language statements to be located by the runtime system.

To resolve these issues, there is a repository where services are registered. The repository provides critical information to the runtime system about location and protocol to the compiler, and provides service and method signatures to programmers.

The entry in the repository for the service “RouteInfo” shown in the sample program has the following information:

```
MODULE "RouteInfoMM" CORBA ORBIX sole.stanford.edu
"RouteInfoModule"
```

This repository entry, typical for a CORBA-based service, consists of the tag “MODULE,” the module name, the tag “CORBA,” the object request broker type (“ORBIX” here), the hostname, and the service provider. This information is important to the runtime system, but not to the programmer. Services (a.k.a. “methods”) available to the programmer and the parameters used also appear in the repository:

```
METHOD RouteInfoMM.GetRoutes (IN CityPair, RES Routes)
  /* gets all routes between 2 cities */
PARAM RouteInfoMM.CityPair     OPAQUE
  /*data containing a city pair    */
PARAM RouteInfoMM.Routes       OPAQUE
  /*data containing routes         */
```

The service and parameter entries in the repository allow for type checking within the program. The use of any full-featured standard repository system would be appropriate (e.g., CORBA, X.500 realms, UDDI). There is no intrinsic contribution to this project from our particular repository system; it must simply be available, in some form, to the programmer and to the client program at runtime. A repository that additionally maintains information about past service executions would be an invaluable source of additional information for a Surety-Based Scheduler. The most udeful repositories would maintain a complete, precise, and accurate record of service executions.

2.4.2 Data Types

There are six data types in CLAM:

- opaque
- integer
- string
- Boolean
- real
- datetime

The first type, *opaque*, is a complex type. Elements of type opaque are exclusively used as user data (i.e., data used with a service) and are not modifiable with CLAM language statements. Opaque data is transferred between services but never processed in the program. That data generally cannot be directly examined by a program written in a compositional language without violating the notion of being "composition only." Opaque data follows the concepts also found in other compositional languages like MANIFOLD.

The other five types are simple CLAM types. Data of simple type is primarily used for control, and it can be received from services with the primitive EXAMINE (the time parameter of type datetime, data volume of type integer, costs of type real, etc.) and the primitive EXTRACT. EXTRACT allows programs to have special functions that take opaque types as input and have simple CLAM types as results. Though CLAM does not provide any arithmetic on complex types, it does provide comparisons for simple types.

There are no classes or amalgamated data types in CLAM. This makes sense as CLAM aims to be a purely compositional language. There are no complex calculations or comparisons in a CLAM program, nor are there large data structures within the program itself that are queried or updated within the megaprogram. Literals may be assigned to the five simple types but, true to its compositional nature, there are no other (non-comparison) operators in CLAM (such as an integer increment operator).

By restricting the available data types in this way, the CLAM language remains true to its purpose. If control decisions are to be made using CLAM data types (perhaps from a call to EXAMINE), they can be done within the program. If decisions must be made about opaque user data, rather than extend CLAM to handle myriad possible data

representations for any arbitrary service, the decision is pushed to a helper service. The helper service will be able to inspect data, and pass back control information as a CLAM data type.

2.5 CLAM SUMMARY

The CHAIMS project was started to investigate a new programming paradigm: compositional programming. As increasing numbers of autonomous services become available to compositional programmers, the overall importance of the composition process rises as well. CLAM is an important piece of the investigation of compositional programming, and represents a purely compositional language.

Does CLAM take advantage of the opportunities presented within a programming paradigm shifting towards composition? We believe it does. Breaking down the traditional CALL statement as CLAM does provides new opportunities for parallelism within programs composed of distributed services. Such parallelism, intrinsically important because of the nature of autonomous service, is not achievable with standard languages not designed for composition. CLAM provides a tool to harness the asynchrony necessarily present when dealing with independent information suppliers.

CLAM is simultaneously suited to take advantage of novel optimization opportunities. Simple compile-time optimizations are possible with CLAM, but the possible runtime optimizations will likely prove to be much more significant. As a language, CLAM delivers explicit language support for user-controlled runtime optimizations. The inclusion of cost estimation methods allows easy online examination of invocation costs. That language-level support means that CLAM can easily handle runtime optimizations not possible in current language approaches based on the traditional CALL routine.

A new composition language should not place new restrictions on service heterogeneity. Differences in communication protocols used by different services should not affect programmers; the composition language of choice should not be fundamentally tied to any specific protocol. As mentioned before, the CLAM language does not distinguish

between the different protocols at a semantic level. Any need to separate features is done at the level of the compiler or support system. Furthermore, by not committing itself to computations on user data, CLAM is not restricted in its ability to pass data between arbitrary services. The opaque data type in CLAM can handle whatever objects are returned by a service, without restriction.

The compositional programming paradigm represents a new level of abstraction in programming. Machine code can be mapped to assembly language instructions, and assembly can be generated by high level programming languages. High level languages enhance a programmer's ability to generate increasingly complex sets of assembly language code, all without adding new assembly language instructions or changing their meaning. High level programming languages are a powerful means to abstract away the tedious details of the assembly language and machine code below them; composition languages are to high level programming languages what those languages were to assembly and machine code. Composition languages enable programmers to use services written in high level languages as a useful construct whose implementation details are not of great concern (like assembly language is to the high level language). CLAM is a step toward this next level of programming.

To realize the full power of scheduling concepts expressible in the CLAM language, compositional programmers could further become scheduling experts, using EXAMINE, EXTRACT, and other primitives to their own runtime adjustment routines directly into their compositions. However, direct use of CLAM's scheduling primitives is an *option* for composers, not a *requirement*: the Surety-Based Scheduler automates the possibly difficult task of manually scheduling large-scale composed programs.

3 THE CPAM RUNTIME

In this chapter, we again consider the composition of services offered by heterogeneous, autonomous and distributed external sources. An ongoing objective is to compose these services and build new applications while preserving the autonomy of the service providers. We describe our high-level protocol that enables software composition: CPAM, CHAIMS Protocol for Autonomous Megamodules. The CPAM runtime may be used on top of various distribution systems. It offers features to support service heterogeneity and preservation of service autonomy, and also directly implements several optimization concepts such as cost estimation for services and partial extraction of results.

3.1 INTRODUCTION

CPAM, the **C**HAIMS **P**rotocol for **A**utonomous **M**egamodules, is a high-level protocol for realizing software composition. Like CLAM, CPAM is part of the CHAIMS project [5], and a tool for building extensive applications through composition of large, heterogeneous, autonomous, and distributed software services.

In the context of CPAM, software services are "large" if they are computationally intensive (computation time may range from seconds in the case of information requests, to days in the case of simulations) or/and data intensive (the amount of data can not be neglected during transmissions) or/and they incur significant external costs for their execution. They are "heterogeneous" if they are written in different languages (e.g., C++, Java), use different distribution protocols (e.g., CORBA [60], RMI [4], DCE [3], DCOM [61]), or run on diverse platforms (e.g., Windows NT, Sun Solaris, HP-UX, Linux). Service are "autonomous" if they are developed and maintained independently of one another, and independently of the programmer who uses them. Finally, software services are distributed when they are not located on the same server as the client and are available to more than one client.

Compositional programming consists of combining remote services in order to produce new functionality. This chapter emphasizes that "composition" differs from "integration" in the sense that composition preserves service autonomy.

Composition of services is becoming increasingly important for the software industry. As business competition and software complexity increases, companies have to shorten their software cycle (development, testing, and maintenance) while offering ever more functionality. Because of high software development or integration costs, there is increasing incentive to build large-scale applications by reusing external services and composing them. Global information systems such as the Web and global business environments such as electronic commerce foreshadow a software development environment where developers would access and reuse services offered on the Web, combine them, and produce new services which, in turn, would be accessed through the Web. In addition to the grid moniker, these ideas combine under the banner of "web services."

Existing distribution protocols such as CORBA, RMI, DCE, or DCOM allow users to compose software with different legacy codes but using CORBA, RMI, DCE, or DCOM as just the distribution protocol underneath. The Horus protocol [62] composes heterogeneous protocols in order to add functionality at the protocol level only. The ERPs, Enterprise Resource Planning systems, such as SAP R/3, BAAN IV, and PeopleSoft, integrate heterogeneous and initially independent systems, but do not preserve software autonomy. None of these systems simultaneously supports heterogeneity and preserves software autonomy during the process of composition in a distributed environment.

CPAM has been defined for accomplishing service composition. In the following sections, we describe how CPAM supports service heterogeneity, how it preserves service autonomy, and how it enables optimized composition of large-scale services. Finally, we provide an illustration of a client doing composition in compliance with the CPAM protocol.

3.2 CPAM SUPPORT FOR SERVICE HETEROGENEITY

Composition of heterogeneous and distributed software services has several constraints: it has to support heterogeneous data transmission between services as well as the diverse distribution protocols used by different services.

3.2.1 DATA HETEROGENEITY

In order for services to exchange information, data need to be in a common format. (A separate project at Stanford explored ways to map different ontologies, rather than the more mundane issue of data formats [63].) Data used by services needs to be machine and architecture independent (32-bit architecture versus 64-bit architecture, for instance), and transferred between services regardless of the distribution protocol at either end (source or destination). For these reasons, CPAM manipulates data as ASN.1 structures encoded using BER rules [64]. We could have opted for an XML format, but chose ASN.1 for its compact binary representation and superior compiled performance (at the time). Given the rise of XML tools and support, future use of XML is likely. With ASN.1/BER-encoding rules:

1. Simple data types as well as complex data types can be represented as ASN.1 structures,
2. data can be encoded in a binary format that is interpreted on any machine where ASN.1 libraries are installed, and
3. data can be transported through any distribution system.

It has not been possible to use other definition languages such as CORBA Interface Definition Language or Java classes to define data types because these definitions respectively require that the same CORBA ORB or the RMI distribution protocol be supported at both ends of the transmission.

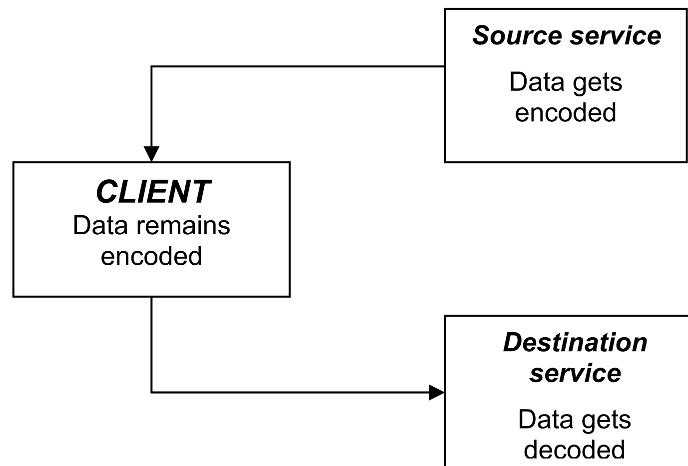


Figure 3 Data transfer, opaque data

3.2.2 Opaque Data

Because ASN.1 data blocks are encoded in a binary format, we refer to them as BLOBs (Binary Large OBjectS). BLOBs are opaque and thus not readable by CPAM. As mentioned in the previous chapter, it is assumed that a client doing composition only does not need to interpret the data it receives from a service, or sends to another service. Therefore, as shown in Figure 3, before being transported, data is encoded in the source service; it is then sent to the client where it remains a BLOB, and gets decoded only when it reaches a destination service.

A client with knowledge of the service definition language could convert the BLOBs into their corresponding data types, and read them. It would then become the client's responsibility to encode the data before sending it to another service. However, this behavior is precluded lack of encoding/decoding facilities in CLAM, and would have to be implemented as a utility service.

3.2.3 Distribution Protocol Heterogeneity

Both data transportation and client-server bindings are dependent on the distribution system used. CPAM is a high-level protocol that is currently implemented on top of other existing distribution protocols. Since its specifications may be implemented on top of more than one distribution protocol within the composed application, CPAM has to

support data transmissions and client-server connections simultaneously across multiple systems.

We mentioned that encoded ASN.1 data could be transferred between the client and the services independently of the distribution protocols used at both ends. Regarding client-server connections, CPAM assumes that the client is able to simultaneously support the various distribution systems of the servers it wishes to talk to. The CHAIMS architecture, along with the CHAIMS compiler [65], enables the generation of such a client. This process is described in next section. Currently, in the context of CHAIMS, a client can simultaneously support the following protocols: CORBA, RMI, *local* C++ and local Java (*local* qualifying a server which is not remote).

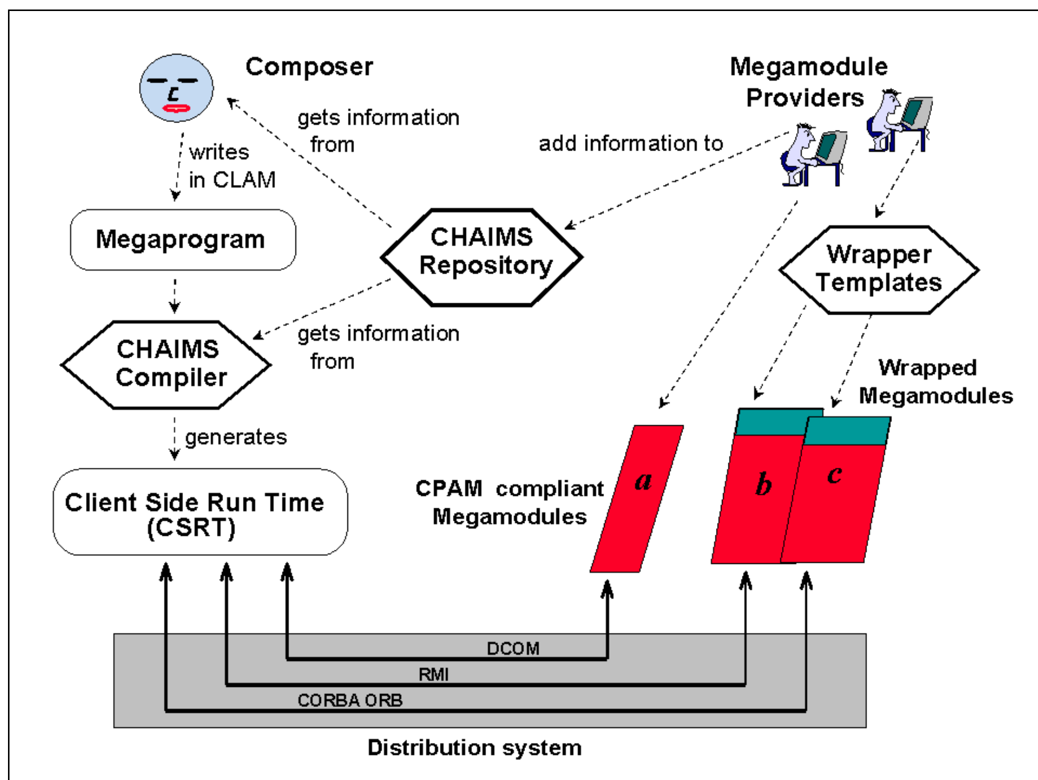


Figure 4 A view of the CHAIMS architecture

3.2.4 CHAIMS SYSTEM ARCHITECTURE

Figure 4 describes the CPAM interaction with the other CHAIMS components. In CHAIMS, the client program is written in CLAM and the compiled client program is the CSRT (Client Side Run Time), compiled against CPAM. Server information repositories are currently merged into one unique CHAIMS repository.

The distribution protocol used during a specific communication between the client and a remote server is the protocol supported by the server, and must be supported by the client. In the context of CHAIMS, the composer is writing their program in CLAM [19], a composition only language. The CLAM program contains the sequence of invocations for the services that the composer wishes to compose (another example CLAM program is given in section 3.5.3). The CHAIMS compiler parses the program and generates the client code necessary to simultaneously bind to the various servers (CSRT). Server specifications, such as the required distribution protocol, are contained in the CHAIMS repository and are accessible to the CHAIMS compiler.

Both the client and services have to follow CPAM specifications. As it is indicated in Figure 4, services that are not CPAM compliant need to be wrapped. The process of wrapping is described in section 3.5.2.

3.3 CPAM PRESERVES SERVICE AUTONOMY

Besides being heterogeneous and distributed, services are assumed to be autonomous. They are developed and maintained independently from the composer who has no control over their execution. How can the composer be aware of all available services, and of the latest versions of these services without compromising autonomy? Also, how do the connections between the client and the server take place? Which of the client or the server controls the connection? After answering these questions, we briefly describe several consistency rules that will ensure offered services are not updated by the server without the client being aware of it.

3.3.1 Information Repository

Composition can not be achieved without knowing what services are offered and how to contact them. The composer could refer to an application user's guide to know what the purposes of the services available are. He/she could also refer to the application programmer's guide to get the implementation details about the services. Nonetheless, the composer would only get static information such as service description and input/output parameter names and types. Services, being autonomous and distributed, make it compulsory to also retain dynamic information, such as the names of the hosts where the services are currently located.

CPAM requires that the necessary service information, both static and dynamic, be gathered into one information repository. Each service provider is responsible for making such a repository available to external users, and for keeping the information up-to-date. It is also the service provider's responsibility to actually offer the services and the quality it advertises.

Information Repository Content. The information repository has to include the following information:

1. Logical name of the service, along with the machine location and the distribution protocol used, in order for the client to bind to the server, and
2. Names of the services offered, along with the names and nature (input or output) of parameters, in order for the client to make invocations or preset parameters before invocation.

Scope of Parameter Names. The scope of parameter names is not restricted to one method where the parameters are used, but rather to an entire service. For services offering more than one method, this implies that if two distinct methods have the same parameter name in their list of parameters, any value preset for this parameter will apply to any use of this parameter within the service scope. CPAM enlarges the scope of parameter names in order to offer the possibility of presetting all parameters of a service using one call only in the client, hence minimizing data flow.

3.3.2 SERVICE CONNECTIONS IN CPAM

Another issue when composing autonomous services is the ownership of the connection between a client and a server. Autonomous servers do not know when a client wishes to initiate or terminate a connection. In CPAM, clients are responsible for making a connection to a service and for terminating the connection. Servers should be able to handle simultaneous requests from multiple clients, and must be started before such requests arrive. Certain distribution protocols (like CORBA) include an internal timer that stops a server execution process if no invocations occur after a set time period, and instantly starts it when a new invocation arrives.

CPAM defines two primitives in order for a client to establish or terminate a connection to a service instance. These are *SETUP* and *TERMINATEALL*. SETUP tells the service that a client wants to connect to it; TERMINATEALL notifies the service that the client will no longer use it (the service kills any ongoing invocations initiated by this client). If for any reason a client does not explicitly terminate a connection to a service, we can assume the service itself will do this after a time-out, and a new SETUP will be required from the client before any future invocation.

3.3.3 CONSISTENCY

Because services are autonomous, they can update their offerings without clients or composers being aware of the modifications brought to the services. The best way for a client to become aware of updates with a service is still under investigation (one option is to have all such changes posted to the repository). Nevertheless, it should not be the responsibility of the service provider to directly notify clients and composers of all changes since we want to preserve service autonomy.

Once a composer knows what modifications were made to a service, he/she can accordingly upgrade the client program. In CHAIMS, the composer updates the CLAM program, which the compiler recompiles in order to generate the final client program.

It is the responsibility of the service provider to avoid updating a service while there are still clients connected to it. A service provider should first request that clients disconnect or wait until their disconnection before upgrading services.

The information repository and the connection and consistency rules ensure that server autonomy is preserved and that clients are able to use offered services.

3.4 LARGE SERVICE COMPOSITION

Composition of large services presents the additional client objective of efficient use of limited resources. Two ways that CPAM contributes to efficient composition of large services are:

- Invocation sequence optimizations
- Data flow minimization between service [13, 97]

3.4.1 INVOCATION SEQUENCE OPTIMIZATION

Because the invocation cost of a large service is high and services are distributed, an arbitrary composition of random services could be very expensive. The invocation sequence has to be optimized. CPAM has defined its own invocation structure in order to allow parallelism and easy invocation monitoring. Such capabilities, coupled with the possibility of estimating service cost prior to invocation, enable optimization of the invocation sequence from the client.

Invocation Structure in CPAM. A traditional procedure call generally consists of invoking a method and getting its results back in a synchronous way: the calling client waits during the procedure call, and the overall structure of the client program remains simple. In contrast, an asynchronous call avoids client waiting, but makes the client program much more complex, and it has to be at least implicitly multithreaded. To support the CLAM language, CPAM splits the traditional call statement into four synchronous remote procedure calls that make the overall call behave asynchronously, while keeping the client program sequential and simple. These procedure calls have also

enabled the implementation of interesting optimizations within CPAM, such as partial extraction and progress monitoring.

The four main procedure calls are INVOKE, EXAMINE, EXTRACT, and TERMINATE. Like the corresponding analogues presented in CLAM, here we present the procedure calls CPAM uses to support composition. Historically, CPAM was partially implemented before CLAM was fully specified so that researchers could experiment with test codes. After CLAM came into being, much of CPAM was reshaped to exclusively support CLAM. We present the main procedure calls and their implementation nuances here:

1. *INVOKE* starts the execution of a service method applied to a set of input parameters. Not every input parameter of the method has to be specified, as the service takes client-specific values or general hard-coded default values for any and all missing parameters (more information about the hierarchical setting of parameters in section 3.4.1). An INVOKE call returns an *invocation identifier*, which is used in all subsequent operations on this invocation (EXAMINE, EXTRACT, and TERMINATE).
2. Clients check if the results of an INVOKE call are ready using the *EXAMINE* primitive. EXAMINE returns two pieces of information: an *invocation status* and an *invocation progress*. As in CLAM, the invocation status can be one of {DONE, NOT_DONE, PARTIAL, ERROR}. If it is either PARTIAL or DONE, then respectively part or all of the results of the invocation are ready and can be extracted by the client. Invocation progress' semantics is service specific. For instance, progress information could be quantitative and describe the degree of completion of an INVOKE call, or qualitative (e.g., specify the degree of resolution a first round of image processing would give).
3. The results of an INVOKE call are retrieved using the *EXTRACT* primitive. Only the parameters specified as input are extracted, and only when the client wishes to extract results, can it do so. CPAM does not prevent a client from repeatedly extracting an identical or a different subset of results.

4. *TERMINATE* is used to inform a service that the client is no longer interested in a specific invocation. TERMINATE is necessary because the service has no other explicit way to know whether an invocation (or its resultant data) will be referred to by the client in the future. In case the client is no longer interested in an invocation's results, TERMINATE makes it possible for the server to abort an ongoing execution. In case the invocation has generated persistent changes in the server, it is the responsibility of the service to preserve consistency.

Parallelism and Invocation Monitoring. The benefits of having the call statement split into the four primitives mentioned above are parallelism, simplicity, and simple invocation monitoring:

- *Parallelism*: thanks to the separation between INVOKE and EXTRACT in the procedure call, different service can be executed in parallel, even if they are synchronous, with the only restrictions being data flow dependencies. The client program initiates as many invocations as desired and begins collecting results when it needs them. Figure 5 illustrates the parallelism that can be induced on synchronous calls using CPAM directly. Of course, similar parallelism could also be obtained with asynchronous methods.

- *Simplicity*: the intermediate client program using CPAM consists of sequential invocations of CPAM primitives, and is simple. It does not have to manage any callbacks of asynchronous calls from services. The client initiates all the calls to a service, including call for retrieving invocation results.

- *Ease of invocation monitoring*: CPAM mirrors CLAM in this regard.

- *Progress monitoring*: a client can check a method execution progress (EXAMINE), and abort a method execution (TERMINATE). Consider the case where a client has the choice between vendors offering the same service and arbitrarily chooses one of them for invocation. EXAMINE allows the client to confirm or revoke its choice, even ending an invocation if another seems more

promising. This capability is shown to be essential to the automated scheduling presented later.

- *Partial extraction*: a client can extract any subset of the results of a service. CPAM also allows progressive extraction: the client can incrementally extract results. This is feasible if the service makes a result available as soon as its computation is completed (and before the computation of the next result is), or results that become significantly "more accurate" as time goes on. Incremental extraction can also be used for terminating an invocation as soon as its results are satisfying, or conversely for verifying the adequacy of large service invocations and terminating them if results are not satisfying.

- *Ongoing processes:* separating service invocation from result extraction and service termination enables clients to monitor ongoing processes (processes that continuously compute or complete results, such as weather services).

CPAM provides one more function that enables the optimization of an invocation sequence in the client program: runtime cost estimation.

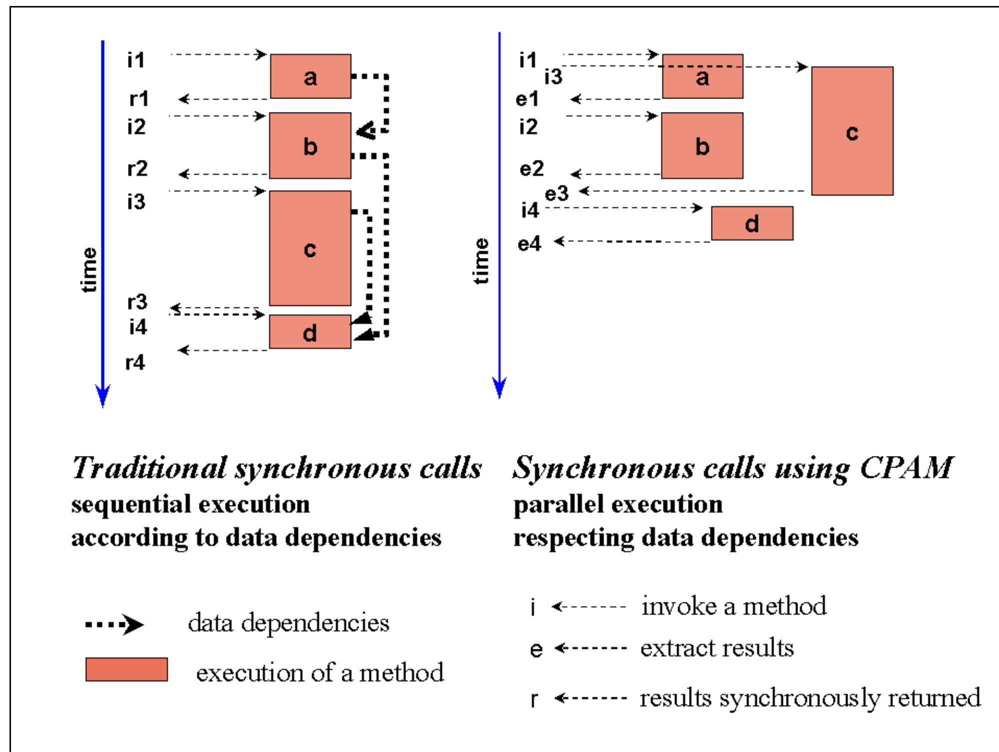


Figure 5 Split of the procedure call in CPAM and parallelism on synchronous calls

Cost Estimation. Estimating the cost of a service prior to its invocation augments the probability of making the most favored invocation at any given time. This is enabled in CPAM through the ESTIMATE primitive. Due to the autonomy of services, the client has no foreknowledge of or influence over the availability of resources at runtime. The ESTIMATE primitive, which much be directly supported by the service, is the only way a client can get the most accurate service performance and cost information.

At runtime, a client requests that a service delivers cost estimation, and then can "decide" whether or not to use the service based upon the estimate received. ESTIMATE is very valuable in the case of identical or similar large services offered by more than one vendor. Indeed, for expensive methods offered by several vendors, it could be very fruitful to first get estimates of the service costs before choosing one service offering. Of course, there is no guarantee that the estimate received is completely accurate. We can at least expect that a service invocation which is not in concordance with the estimate

previously provided to a client will not be reused by the client. (But these are quality of service questions best answer in the chapter on scheduling, and not within the context of CPAM.)

Cost estimates are described in CPAM as cost (amount of money to pay for the service), expected execution time, and/or data volume (amount of data resulting from a service invocation). Since the last two factors (time and data volume) are highly runtime dependent, their estimation should be *at runtime*, as close as possible to the time the service would be invoked. Other application specific parameters (like server location, quality of service, and accuracy of estimations) may be added to the output estimate list in the server (and in the information repository), without changing the CPAM specification.

Parallelism, invocation estimates and invocation examinations are very helpful functions of CPAM which, when combined, give enough information and flexibility to get an optimized sequence of invocations at run-time. Though perhaps somewhat mundane in purpose, CPAM functions as a universal platform (and glue) for service providers and for higher-level programmers alike.

A final factor for optimized composition concerns data flow between services. We address the question of data flow more fully in Chapter 9, but introduce the basics data flow support in CPAM.

3.4.2 Minimizing Data Flow between Services

Partial extraction enables clients to reduce the amount of data returned by an invocation. CPAM also makes it possible to avoid parameter redundancy when calling INVOKE thanks to parameter presetting and hierarchical setting of parameters.

Presetting Parameters. CPAM's SETPARAM primitive sets method parameters and global variables before a method is invoked. For a client that executes a service with the same parameter value several times consecutively, or executes several services which have a common subset of parameter names with the same values, it becomes cost-

effective to not transmit the same parameter values repeatedly. Let us recall that large services are often likely to be data intensive. Also, in the case of services that have a very large number of parameters, only a few of which are modified at each call (very common in statistical simulations), the SETPARAM primitive becomes very advantageous. Finally, presetting parameters is useful for setting a specific context before estimating the cost of a method.

GETPARAM, the primitive dual, returns client specific settings or default values of the parameters and global variable names specified in its input parameter list.

Hierarchical Setting of Parameters. CPAM establishes a hierarchical setting of parameters within services (see Figure 6). A parameter's default value (most likely hard-coded) defines the first level of parameter settings within the services. The second level is the client specific setting (set by SETPARAM). The third level corresponds to the invocation specific setting (parameter value provided through one specific invocation, by INVOKE). Invocation specific settings override client specific settings for the time of the invocation, and client specific settings override general default values for the time of the connection. When a method is invoked, the service takes the invocation specific settings for all parameters for which the invocation supplies values; for all other parameters, the service takes the client specific settings if they exist, and the service general default values otherwise. For this reason, CPAM requires that services provide meaningful default values for all parameters or global variables they contain.

A client does not need to specify all input data or global variables used in a service in order to execute that service, nor does it need to repeatedly transmit the same data for all method invocations which use the same parameters' values. Also, a client need not retrieve all available results. This potentially reduces the amount of data transferred between service.

With large and distributed services, invocation sequence optimization and data flow minimization are necessary for efficient composition.

Example: parameters of method “reservation”

	start-date	end-date	from	to	seats
general default values	1JAN1998	1JAN1998	LAS	BWI	1
client-specific settings for client *A*	4OCT1998	6OCT1998	SJO	ZRH	
client-specific settings for client *B*				FRA	2
invocation-specific settings for client A					
actual values used during invocation	4OCT1998	6OCT1998	SJO	ZRH	1
invocation-specific settings for client A		7OCT1998	SFO		
actual values used during invocation	4OCT1998	7OCT1998	SFO	ZRH	1
invocation-specific settings for client B	1DEC1998	9DEC1998			
actual values used during invocation	1DEC1998	9DEC1998	LAS	FRA	2

Figure 6 Hierarchical setting of parameters

3.5 USING CPAM

We have so far discussed the various CPAM primitives supporting compositional programming. Still, we have not yet described the primitives’ syntax or the constraints a composer would have to follow in order to write a correct client program. Another point would concern the service provider: what does he/she need to do in order to convert a service which is not CPAM compliant to a CPAM compliant service that supports CPAM specifications?

CPAM primitives’ syntax is fully described on the CHAIMS Web site at http://www-db.stanford.edu/CHAIMS/Doc/Details/index.html#The%20CHAIMS%20Protocols; that location contains the specifications for CPAM CORBA, RMI, DCE, native TCP/IP, local Java services, and local C++ services. In this section, we describe the primitive ordering constraints, CHAIMS wrapper templates, and provide a client program example which complies with CPAM, and is written in CLAM.

3.5.1 PRIMITIVES ORDERING CONSTRAINTS

CPAM primitives cannot be executed in any arbitrary order, but they only have to follow two constraints:

- All primitives apart from SETUP must be preceded by a connection to the service through a call to SETUP (which has not yet been terminated by TERMINATEALL),
- An invocation referred to by EXAMINE, EXTRACT, or TERMINATE must be preceded by an INVOKE call (which has not yet been terminated by TERMINATE).

Figure 7 summarizes CPAM primitives and the ordering relations.

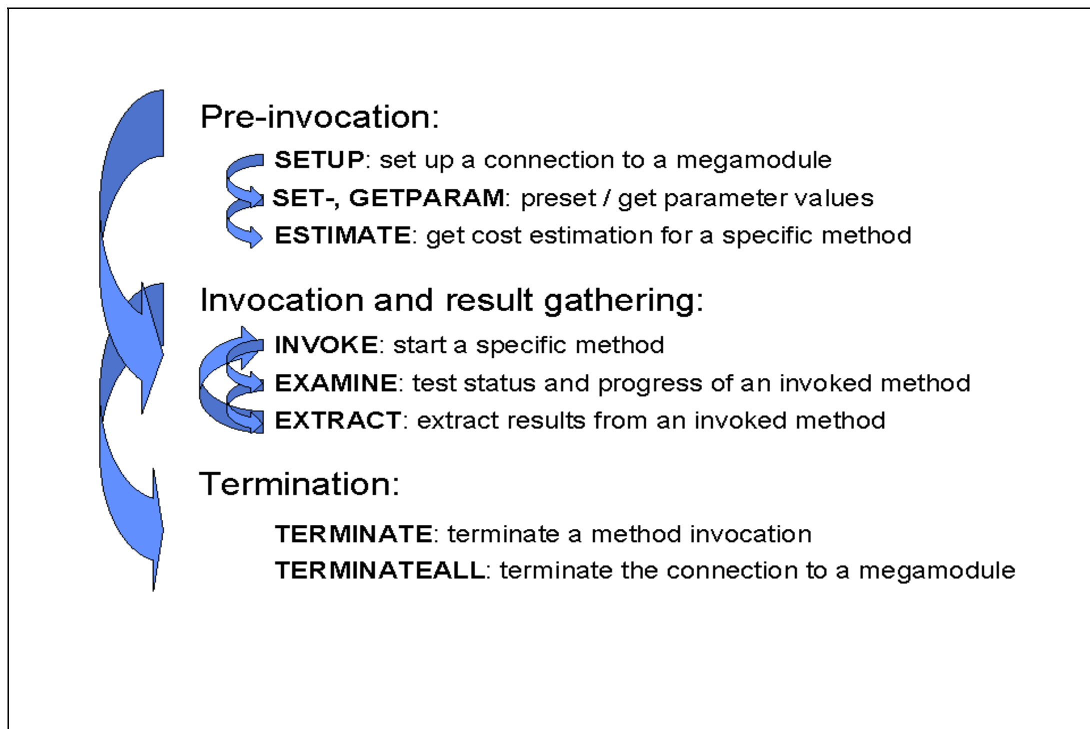


Figure 7 Primitives in CPAM and invocation ordering

3.5.2 The CHAIMS Wrapper

If a service does not comply with the CPAM specification, it must to be wrapped in order to use the CPAM protocol. The wrapper provides the semantic information needed to support the CPAM runtime, as we assume that protocol issues can be ameliorated by the CSRT. Existing CHAIMS wrapper templates allow a service to become CPAM compliant with a minimum of additional work.

The CHAIMS wrapper templates are currently implemented as a C++ or Java object which serves as a middleman between the network and non-CPAM compliant servers. They implement CPAM specifications in the following way:

1. *Mapping of methods [66, 67] and parameters*: the wrapper maps methods specified in the information repository to one or more methods of a legacy service. It also maps any parameters to ASN.1 data structures, preserving default values assigned in the legacy modules (or adding them if they were not originally assigned).

2. *Threading of invocations*: to ensure parallelism and respect asynchrony in the legacy code, the CHAIMS wrapper spawns a new thread for each invocation.
3. Generation of internal data structures to handle client invocations and connections, and support hierarchical setting of parameters: each call to SETUP generates the necessary data structures to store client and invocation related information in the wrapper. Such information includes client-specific preset values, and the status, progress and results for each invocation. The generated data structures are deleted when a call to TERMINATEALL occurs.
4. *Implementation of the ESTIMATE primitive for cost estimation*: for each method whose cost estimation is not already provided by the service, the ESTIMATE primitive by default returns an average of the costs of the previous calls of that service. It could also include dynamic information about the service and the network, such as service host load, network traffic, etc.
5. *Implementation of the EXAMINE primitive for invocation monitoring*: by default, only the status field is returned. For the progress information to be set in the wrapper, the service has to make some significant information available.
6. *Implementation of all other CPAM primitives*: SETUP, GET/SETPARAM, INVOKE, EXTRACT, TERMINATE, and TERMINATEALL.

The current CHAIMS wrapper templates automatically generate the code to ensure points 2 through 6. Only requirement 1 needs manual coding (except for the BER encoding and decoding, which is automatically done by ASN.1 libraries).

3.5.3 Example of a Client Using CPAM

A successful utilization of CPAM for realizing composition is the "Transportation" example implemented within the CHAIMS system. The example consists of finding the best way for transporting goods between two cities (and is similar to the code shown previously in section 2.2.4). In this example, the composer uses services from five heterogeneous and autonomous services. The client program is written in CLAM, and the CSRT generated through the CHAIMS compiler is in compliance with CPAM. Another implemented example is computes the best design model for an aircraft system,

and includes optimization functionality as cost estimation, incremental result extraction, and invocation progress examination.

Below is a simplified version of the Transportation program (Figure 8).

3.6 CPAM SUMMARY

CPAM is a high-level protocol that supports service composition. It supports heterogeneity by transferring data as encoded ASN.1 structures, and preserves service autonomy by collecting service information from an information repository, and by subsequently using the generic INVOKE primitive of CPAM in order to execute services.

Most importantly, CPAM enables efficient composition of large-scale services by optimizing the invocation sequence (parallelism, invocation monitoring, cost estimation), and minimization of data flow between client and service (presetting of parameters, hierarchical setting of parameters, partial extraction). As CPAM is currently tailored for composition, it does not provide support for recovery or security. These services could be obtained by orthogonal systems or by integrating CPAM into a larger protocol.

Embedding the Surety-Based Scheduler (described in a later chapter) within the CPAM layer will discharge the composer from worries about parallelism or lower level scheduling tasks, while not disabling optimizations that are based on domain expertise. In any large-scale and distributed environment, resources are likely to be relocated, and their available capacity depends on aggregate usage. Invocation scheduling and data flow optimization need to take into account such constraints. The CPAM protocol can give sufficient information to the compiler or the client program for enabling automated scheduling of composed software at compile-time, and more significantly, at run-time. It currently operates as a target for programs written in the CLAM language, but would be a suitable runtime for other compositional languages, like MANIFOLD [11, 46].

```
BEGINCHAIMS
 io      = SETUP ("io")              // Remote, Java, CORBA ORBACUS
 math  = SETUP ("MathMM")      // Local, Java
 am     = SETUP ("AirMM")        //  Remote, C++, CORBA ORBIX
 gm     = SETUP ("GroundMM")  //  Remote, C++, CORBA ORBIX
  ri      = SETUP ("GroundMM")  //  Remote, C++, CORBA ORBIX
 // Get type and default value of the city pair parameter
 (cp_var = CityPair) = ri.GETPARAM()

 // Ask the user to confirm/modify the source and destination cities
 ioask = io.INVOKE ("ask", label = "which cities", data = cp_var)
 WHILE ( ioask.EXAMINE() != DONE )  {}
 (cities_var = Cities) = ioask.EXTRACT()
 // Compute costs of the route by air, and by ground, in parallel
 acost = am.INVOKE ("GetAirTravelCost", CityPair = cities_var)
 gcost = gm.INVOKE ("GetGdTravelCost", CityPair = cities_var)
   // Make other invocations (e.g., Check weather)
   ….
 // Extract the two cost results
 WHILE ( acost.EXAMINE() != DONE )  {}
 (ac_var = Cost) = acost.EXTRACT()
 WHILE ( gcost.EXAMINE() != DONE )  {}
 (gc_var = Cost) = gcost.EXTRACT()
 // Compare the two costs
 lt = math.INVOKE ("LessThan", value1 = ac_var, value2 = gc_var)
 WHILE ( lt.EXAMINE() != DONE )  {}
 (lt_bool = Result) = lt_ih.EXTRACT()
 // Display the smallest cost
 IF ( lt_bool == TRUE ) THEN
 { iowrite = io.INVOKE ("write", data = ac_var) }
 ELSE
  { iowrite = io.INVOKE ("write", data = gc_var)  }
 am.TERMINATE()
 ri.TERMINATE()
 gm.TERMINATE()
 io.TERMINATE()
 math.TERMINATE()
ENDCHAIMS
```

Figure 8 CLAM program to calculate the best route between two cities

4 RESULT EXTRACTION

Scheduling is a concern for both workflow management and grid computing systems. Most schedulers balance time and cost to fit within a client's budget, while accepting explicit data dependencies between services as the best resolution for scheduling. Execution results are extracted from one service in total, and then forwarded to the next service. However, distributed objects and remote services adhere to various standards for data delivery and result extraction. There are multiple means of requesting results and multiple ways to deliver those results. By examining several popular and idiosyncratic methods data extraction methods, we have developed a comprehensive model that combines the functionality of all component models. Our model for result extraction from distributed objects provides increased flexibility for object users, and an increased audience for service providers. In turn, intelligent schedulers may leverage these result extraction features to optimize data flows (issues more deeply covered in Chapters 9 and 5). In this chapter, we present and defend the claims that the CLAM/CPAM data extraction model supports a superset of functions of other widely used models, and that our extraction model also enables useful optimizations.

4.1 OVERVIEW

Schedulers for grid computing and workflow management are similar in both goals and problem spaces. Both take component objects or services and compose them into useful "programs" or "workflows." Chains and graphs of services are constructed to meet a goal not achievable by a solitary service. In these programs and workflows, interactions between the component objects are typically not fine-grained [20, 29, 30, 31, 32, 50]. Data elements passed between components are central to the scheduling process: explicit dependencies typically arise when one component's data inputs depend on another component's results. By utilizing a richer model of result extraction, finer-grained scheduling becomes possible, and new optimization techniques present themselves [33].

A "comprehensive" model for result extraction should consider and account for all major existing result extraction techniques and functionalities, even if some techniques have orthogonal objectives. The realization of a truly comprehensive model implies that such the model captures the functionality of all of its precursors, and that any such precursor model can be expressed in the more complete model. We present such a result extraction model, a model that arises from the CLAM semantics and supported by CPAM functions, and we compare and contrast it to result extraction in web browsing, JointFlow, SWAP, and CORBA (and briefly generic RPCs). We focus on the main issues of *partial* and *progressive* data extraction, but also demonstrate how result-progress inspection as part of the model also increases expression.

We define "partial extraction" as the extraction of a subset of available results. There are obvious benefits mentioned in the last chapter to this type of extraction. For instance, users can save time and cost with result subset extractions, and can obtain specific results at the earliest possible time. We define "progressive extraction" as the extraction of the same result parameter with different content/data at different points in a computation. The different points of computation can deliver different accuracy of specific results, as is typical in simulations or complex calculations for images. Or different points of the extractions may signify data freshness and timeliness, as is the case for weather data or stock data that changes over time.

4.1.1 TRADITIONAL RPC

We address the problem of obtaining results from any of multiple computational servers in response to requests made by a client program. The simplest form of result extraction is the synchronous remote procedure call (RPC). Parameters are passed in, the client waits patiently for the results, and finally all results are simultaneously available. Only a single object is returned with certain function calls, but most languages offer procedure calls with multiple OUT-parameters. Multiple OUT-parameters are possible in C++ through the use of pointers, and it is cleanly implemented in CORBA. RPC has been the dominant form of result extraction in most programming languages and for many

distributed systems (e.g. CORBA, where a typical procedure call is defined in the IDL syntax like `void mymethod(in TYPE1 param1, out TYPE2 param2, out TYPE3 param3)`).

4.1.2 Additional Extraction Models

There are other models of data extraction that have received little attention in most programming languages and paradigms. These alternative models tend to be strictly more functional than the typical remote procedure call, and they are being used increasingly as a replacement to RPC-style result extraction. Each extraction model, in at least some aspect, provides more functionality or flexibility than the RPC.

4.1.2.1 Asynchrony

One of the most important enhancements to RPC has been the addition of asynchrony. Clients do not have to stall while results are computed or delivered. Asynchronous result extraction has been used even in many sequential computing languages, like Java and Ada, and is enabled in various ways (through message passing, threading, rendezvous, etc.). Asynchrony as a tool to delay or reorder result extraction is well understood and widely used [22, 23].

4.1.2.2 Partial Extraction

Partial extraction of data results has become almost ubiquitous with the advent of web browsing, and may feature prominently within web services as well. Partial extraction is taking only the desired portions of a result set, thus saving the costs associated with extracting the entire set. For instance, almost all modern web browsers have the ability to download only text, without images, to speed browsing on slower connections. Thi is partial extraction of data. Many browsers also allow users to filter unwanted objects (i.e., embedded audio) from HTML documents. The web has brought “partial extraction” to the desktop as the default browsing model.

4.1.2.3 Progressive Extraction

Another model for result extraction, strictly more powerful than the traditional RPC, is the "progressive extraction" model. In many scientific computations, such as adaptive mesh refinements, answers become "better" (e.g., more precise) over time. It can be quite useful to extract results as progress is made toward an acceptable solution for steering, early termination, or other reasons. A typical application of this type is the design of an aircraft wing [14, 10, 16, 22]. Certain scientific and mathematical processes, like Newton's method of successive approximation for roots of an equation, can also utilize and benefit from this type of extraction [15]. The traditional RPC result extraction model does not lend itself well to progressive extractions.

4.1.2.4 Survey

Decomposition of the traditional call model has been discussed for some time [5]; we advocate a further breakdown of the extraction phase of this model. We propose an extraction model, developed in the course of research and development of the language CLAM [19] and within the CHAIMS [13], which encompasses these three important augmentations (asynchronous, partial, and progressive extractions) to RPC-style result extraction models. The amalgamation of these three extraction paradigms leads to a more general model, more expressive than any of the three taken alone.

4.1.3 Partial and Progressive Extraction Today

For many languages and systems, the generic asynchronous remote procedure call model of result extraction provides enough flexibility. In cases where it does not, custom solutions abound. Occasionally, these custom solutions become widespread, often circumstantially, e.g., because of runtime support rather than language specification. For example, partial extraction within the available community of web-browsers does not arise from HTML, *per se*. It is a consequence of parsing HTML pages and making only selective *HTTP* requests. In effect, the web-browser implements the partial extraction of results while the HTML provides a "schema" of the results available. For instance, when web users choose not to download certain types of content, the web-browser implements

the filtering and is not impacted by the HTML or the HTTP protocol. Additional tricks performed by web browsers include result caching, reducing the number of HTTP requests to recall the same object referred to in multiple HTML pages.

We accept this HTML/HTTP/browsing model as appropriate for partial extractions when a schema for the data is available (such as the object schema represented by the HTML document). Of course, without a schema, partial extractions would be meaningless (without a schema for a result set, what constitutes a "part" of that set?!). At some level, this constrains the domain for which partial extraction is semantically meaningful or even practical. However, since the model of partial extraction wholly encompasses the traditional RPC method of result extraction, this is not a problem; progressive extracts can always be used to implement simpler RPCs.

Progressive extractions are occasionally the consequence of elaborate development projects or again arise because of the specific nature of the data involved. We can look again to web browsing to find (albeit an extremely limited) form of progressive extraction, one that arises from the *data* at hand, rather than the HTML or HTTP requests. Within certain result sets, like weather maps, the results change over time. By simply re-requesting the same data, progressive updates to the data may be seen by a client. On the other hand, to see a broader picture of progressive result extraction, we turn to (frequently) hand-tooled codes specialized for single applications.

As far as we know, there is currently no language with primitives supporting progressive extractions. Additionally, what marginal runtime and protocol support for progressive extractions there is seems to only exist because of specialized data streams (like browsing weather services), and not because of intentional support. We have found examples of hand-coded systems that allow partial result extractions, at predefined process points, to allow early result inspection [10]. Such working codes have been built to allow users to steer executions and to test for convergence on iterative solution generators. However, a language and runtime system to develop arbitrary codes that allow progressive

extractions is not to be found. This is especially true of composition languages geared toward distributed objects and services.

Herein we outline a set of language primitives expressive enough to capture each of these result extraction models simultaneously. We also review an implementation of this general result extraction model, as encompassed within the megaprogramming language CLAM and the supporting components within the CHAIMS system [18]. There are many ways to do data selection, transformation, and extraction from data sets (e.g., XML data, XQuery selection, and XSLT transformation). However, we focus on distinctions introduced when data sets are remote and elements become available over time. We examine the consequences of available modes for monitoring and extracting results from data sets provided by remote services.

4.2 RESULT EXTRACTION WITHIN CLAM

In Chapter 3, we described how the CPAM runtime could save users time and data transfer by using parameter presetting and reuse *before service invocation*. In this chapter, we will extend these benefits by underscoring the power of partial and progressive extractions to reduce the data costs *after service invocation*. Interesting results often come from computational services, rather than simple data services like databases, web servers, etc. Computational services add value to their input data by processing transformations that significantly update values based on those inputs. Results from computational services are tailored to their inputs (e.g., weather modeling under specific input circumstances), unlike data services where results are often already available before and after the service is used (e.g., static pages from a web server). Different data extraction methods have potentially more value in the context of computational services than in the simpler context of data services.

We present CLAM as one possible language and implementation of our generic result extraction model. CLAM is aptly suited to the presentation of this extraction model, though not necessarily a programmer’s language of choice for a given problem. We

advocate this extraction model independent of the language, though we present it within the CHAIMS framework. Our model is "client-centric" because, unlike an exception or callback- type model, clients initiate all data inspection and collection. In this "data on demand" approach, clients expect that servers do not "push" either data or status, but rather must make requests for either. We specifically contrast this client-centric approach to the CORBA event model seen in section 4.3.4.

4.2.1 EXAMINE AND EXTRACT REVISITED

Two CLAM primitives define how users can retrieve results from a service: EXAMINE and EXTRACT. Recall that we use EXAMINE to inspect the progress of a service (or calculation, method invocation, procedure, etc.) by requesting data status or, when available, richer information about the invocation progress.

4.2.1.1 EXAMINE

The EXAMINE primitive is used to determine the state and progress of an invocation. EXAMINE has two status fields: state and progress. State can be any of {DONE, NOT_DONE, PARTIAL, ERROR}. Consider the progress field an integer, showing the progress of individual result values, as well as of an entire invocation. The various pieces of status and progress information are only returned when the client requests them, in line with the client-centric approach.

Syntax

```
(mystatus=status) = invocation_handle.EXAMINE()
(mystatus=status) = invocation_handle.EXAMINE(parameter)
(mystatus=status, myprogess=progress) =
            invocation_handle.EXAMINE()
(mystatus=status, myprogess=progress) =
            invocation_handle.EXAMINE(parameter)
```

In this section, we describe in detail how the EXAMINE primitive functions in support of the partial and progressive data extraction model. Imagine a service "*foo*" which returns three distinct results, "*A*," "*B*," and "*C*." As *foo* is invoked, and before any work has been completed, *A*, *B*, and *C* will all be incomplete. A call to `foo_handle.EXAMINE()`

will return NOT_DONE. When the execution is completed, a call to `foo_handle.EXAMINE()` will return DONE, because all work has been performed. If there are *some* meaningful results available for extraction before all results are ready, `foo_handle.EXAMINE()` returns PARTIAL. In case of error with the invocation of *foo*, ERROR is returned.

If the service supports invocation progress information, `(mystatus=status, myprogess=progress) = invocation_handle.EXAMINE()` may be used instead of the simpler `(mystatus=status)=invocation_handle.EXAMINE()` to get additional progress information about the invocation. This progress information indicates how much headway the computation has made so far in terms of time used and estimated remaining time needed to complete execution. (Ideally, progress information is in accordance with pre-invocation cost estimates provided by the ESTIMATE primitive [19].) Because execution environment conditions can change rapidly, it is essential to be able to track the progress of the computation from its invocation to its completion. Tracking execution progress is essential to Surety-Based Scheduling; supporting per-element data availability progress information gives the best resolution possible to a scheduler.

After receiving an invocation status of "PARTIAL," the client knows that some subset of the results are extractable, though not that any *particular* element of the result data is ready for extraction (nor if the result set contains progressive and thus temporary values) or has been finalized. PARTIAL indicates to the client that it would probably be worth while to inspect individual result parameters to get their particular status information.

Once again, return to *foo* with return data elements *A*, *B*, and *C*. At this point, *A* is done, *B* is partially done with extractable results available, and no progress has been made on return value *C*. A call to `foo_handle.EXAMINE()` would return PARTIAL, because some subset of the available data is ready. Subsequently, a client issue of `foo_handle.EXAMINE(A)` will return DONE, `foo_handle.EXAMINE(B)` will return {PARTIAL, 50}, and `foo_handle.EXAMINE(C)` will return NOT_DONE.

Interpretations of the results from the examination *A* and *C* are obvious. In the case of result *B*, assuming that the result value is 50% completed, and the service makes this additional information available to the client, a return tuple such as {PARTIAL, 50} would express this.

Remember, the second parameter returned by EXAMINE is, by default, an integer. CLAM places no restriction on its general use. Each service is free to impart its own meaning on such parameters; however, recommended usage is for the value to indicate a "percentage (0-100) value of completion." The semantic meaning of progress values returned by a particular service should be available to clients via the service repository.

4.2.1.2 EXTRACT

In CLAM, the EXTRACT primitive collects the results of an invocation. Any subset of all parameters returned by the invocation can be extracted. Result values which are extracted are ones specified on the left hand side of an EXTRACT command.

Syntax

```
(myvar1=outvar1, myvar2= ...) = invocation_handle.EXTRACT()
```

Returning to our previous example of the service *foo*, we look at EXTRACT. The return fields are name/value pairs, and may contain any subset of the available data. For the example service *foo*, we might do the following: `(tmpa=A, tmpb=B) = foo_handle.EXTRACT()`. This call would return the current values for *A* and *B*, leaving *C* on the server.

This EXTRACT primitive allows extraction just those results that are ready as well as wanted. This in contrast to a simpler EXTRACT, where all results would be returned with each EXTRACT call, independent of any result's state.

4.2.2 Extraction Model Enabled by EXAMINE and EXTRACT

Different flavors of extraction are available with different levels of functionality by combining EXTRACT and EXAMINE. We present here the various achievable

extraction modes when portions of the complete set of CLAM data extraction primitives are available. Our analysis shows how EXAMINE and EXTRACT interact to provide each of the resultant extraction flavors outlined in this chapter.

To complete the extraction model, we present two augmentations to the EXAMINE primitive. The first augmentation is the addition of a second parameter, the progress indicator. The second augmentation is the ability to inspect individual parameters. These two augmentations provide four distinct possible result examination schemes:

(1) *without* parameter inspection, and *without* progress information

(2) *without* parameter inspection, but *with* progress information which is invocation specific

(3) *with* parameter inspection, but *without* progress information

(4) *with* parameter inspection, and *with* progress information which is parameter specific

There are two types of EXTRACT operations in CLAM:

(a) EXTRACT returns all values from an invocation (like RPC)

(b) EXTRACT retrieves specific return values (like requesting specific embedded HTML objects)

The two EXTRACT options, when coupled with the four possible types of EXAMINE, form eight possible models of data extraction.

Error! Reference source not found. Table 1 shows the eight possible combinations and outlines basic functionality achievable with each potential scheme. Even for the simplest case as shown in table entry **1a** (an EXAMINE that works on a per invocation basis only (cannot examine particular parameters) and an EXTRACT that only returns a complete set of results), the extraction model is more powerful than a typical C++/Fortran-like procedure call because of the viable asynchrony achieved. The model retains its client-centric orientation. Clients polls at selected times (using EXAMINE) and can extract the

data whenever desired. Recall that an assumption in CLAM is that a client may also extract the same data multiple times, placing some burdens on the support system that we discuss in the next section.

Table 1. EXAMINE/EXTRACT relationships

	(a) single EXTRACT	**(b)** per return element EXTRACT
(1) per *invocation* EXAMINE, One parameter	**1a.** Like an asynchronous procedure call **e.g., Java RMI**	**1b. Limited partial extraction**. Like 1a, with the added ability to extract a subset of return data at a time indicated by PARTIAL (limited to one checkpoint) or after all results are completed.
(2) per *invocation* EXAMINE, Two parameters	**2a.** Very limited. ***Progressive*** extraction becomes possible, with data completion level indirectly indicated by second parameter. Must retrieve entire data set.	**2b.** Very limited. Progressive extraction still possible, no legitimate potential for partial extractions other than a unique set as in 1b.
(3) per *result* EXAMINE, One parameter	**3a.** Allows for data retrieval as particular return values are complete, but entire set must be retrieved each time (***semantic*** **partial extraction**).	**3b. *True partial extraction*** becomes possible here. No real progressive extraction. **e.g., Web-browsing, SQL cursors**
(4) per *result* EXAMINE, Two parameters	**4a. *Progressive*** **extraction** becomes possible, with data completion level indicated by second parameters. Must retrieve entire data set. More detail than 2a. **e.g., JPEG**	**4b. *Partial*** **and *progressive*** **extraction** are both possible. Single results may be extracted at various stages of completion. **e.g., CLAM**

The mechanism described by Table 1 entry **1b** is mainly used for partial result extraction when all results are completed, yet not all results are needed. The client has the possibility to extract only those results needed right away, and can leave the other results on the server until they are extracted at a later point or time, or until the client indicates it is no longer interested in the results. The main advantage of this level of partial extraction is to avoid transmitting huge amounts of data when the data is not needed. This is particularly useful when one result parameter contains some meta-information

that tells the client if it needs the other result parameters at all. This mechanism can be compared to the partial download of web pages by a web-browser. In a first download, the browser might exclude images. At a later point, the person using the web-browser might request the download of some specific (or all) images based on the textual information received in the first download. This usage of partial extraction has become very important as it allows to partially download small amount of information, and only download (mainly time when using slow connections) images as needed.

There is a limited capability for process progress inspection even with only the single return parameter in Table 1 entry **1b**. With the set {DONE, NOT_DONE, PARTIAL, ERROR}, there is room for limited process inspection. The return of "PARTIAL" from a server may indicate a unique meaningful checkpoint to a client. It could be used to indicate some arbitrary level of completion, or that some arbitrary subset of data was currently meaningful. This single return value is really a special binary case of the second return value from EXAMINE. Together with a per-return-element EXTRACT, this model can only reasonably be used to extract one specific, pre-defined subset of results before final extraction of the entire return data set. Figure 9 shows this use of the PARTIAL value for creating a single binary partition of results this way: PARTIAL indicates in this case that the results *A* and *B* are ready, yet *C* is not yet ready. If e.g., *A* and *C* were ready but *B* was not, this cannot be uniquely identified, and the status NOT_DONE would be returned.

For clarity, we examine more closely the power associated with each entry in the above table. Table 1 entry **1a**, when there is only one status parameter per invocation (i.e., for all of *foo*) and only the ability to extract the entire return data set, is clearly the most limited case. It is very much like an asynchronous remote procedure call, with the distinction that the client polls for result readiness. Also, data are only delivered when the client requests it, as noted previously.

In Table 1 entry **1b**, we see that adding per element extraction capability allows clients to reasonably extract one portion of the data set if it is done earlier than the whole return set.

This capability is still very limited. This model can only reasonably be used to extract one specific, pre-defined subset of results before final extraction of the entire return data set. The return value “PARTIAL” can be used to indicate that a partial set of results are done, whereas the examination return value "DONE" may indicate that *all* results are ready. This use of PARTIAL is shown in Figure 9. Again, it is only possible to create a single binary partition of results this way.

In Table 1 entry **2a**, we see that very limited progressive extraction of the data is made possible by the addition of the second parameter to EXAMINE. The status of the results can be indirectly derived from the progress of the invocation. This indirect progress indication only applies to the entire result return set, which is really only useful if there is only a single result parameter or all the result parameters belong tightly together. This is actually a superset of the web-browser extraction method. With the web-browser extraction method for weather data to be computed, images, or simulations, no meta-information about the data is returned to the client (i.e., status about the data being 20% complete, etc.). To add such meta-data to web browsing, the browser must be further extended to actively process return values through another mechanism like Java or JavaScript.

In Table 1 entry **2b**, we see that very limited progressive extraction of the data is again made possible by the addition of the second parameter to EXAMINE. There is really no more power in this examination and extraction model than Table 1 entry **2a**. However, creative service programmers could take advantage of a scheme in a way similar to that shown in Figure 9. Still, even under such circumstances, programmers are again limited to one predefined subset for extraction and are not allowed the flexibility seen in Table 1 entries **3b** and **4b**.

In Table 1 entry **3a**, we see the addition of per return value examination information. When there is only one return value from a method, the functions of entries **3a**, **3b**, **4a**, and **4b** are identical to entries **1a**, **1b**, **2a**, and **2b**, respectively. We refer to entry **3a** as “semantic partial extraction” because the per-result examination allows the user to know

exactly when individual return results are ready, but the entire data set must be extracted to get any particular element. This can cause unnecessary delay and overhead, especially with very large result sets.

In Table 1 entry **3b**, we see the first fully functional version of partial extraction. Whereas in entries **1b** and **2b** individual return values could be selected from the return set, partial extraction was not meaningful without a per return value EXAMINE primitive as offered here.

In Table 1 entry **4a**, progressive extraction is possible with progress indicators for each data return value. On the other hand, it still suffers from the same overhead as table entry **3a**: all data must be extracted at each stage. This expense cannot be avoided.

In Table 1 entry **4b**, we see the manifestation of the complete CLAM examine/extraction model, one that has all facets of the general result extraction model. Both partial and progressive extractions are possible, including partial-progressive extractions (where arbitrary individual elements may be extracted at various stages of completion). Without exception, this model is strictly more powerful than any of the others.

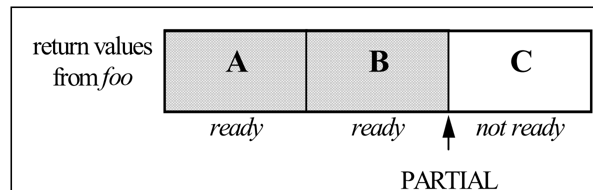


Figure 9 Overloading PARTIAL with additional semantic information

4.3 COMPARISONS

In the following subsections we compare the extraction model as defined in CLAM to the extraction models found in the following protocols: Web browsing, JointFlow, SWAP, and CORBA-DII.

4.3.1 Partial and Progressive Result Extraction in Web Browsing

Web browsing generally falls into the category of services that we referred to as "data services." Data services primarily deliver specific data requested by clients, in contrast to "computational services," which add value to client supplied data. Clients are usually represented by one of many available web browsers or crawlers while web servers deliver data to those clients. Clients request data using the HTTP protocol. Data extracted from servers (e.g., documents delivered) are often written in HTML, and often have components/references to varying types of information, including images, audio, and video.

Web browsing occurs in batch and interactive modes. Web crawlers batch browse for many reasons: indexing, archiving, etc. Likewise, humans browse interactively for numerous reasons: information gathering, electronic commerce, etc. A browser of either sort, batch or interactive, makes a request to a server for a specific document. That document is returned to the client, and serves as the prompt for further requests. If the document is HTML, the browser may parse the document and determine that there are other elements that form the complete document (i.e., images). The HTML document serves as a schema for the entire webpage, describing the other elements that may be extracted from the server.

After a document has been fetched from a web server, we can consider the possible partial and progressive extractions that can take place. To extract a sub-element of a web page, an additional HTTP request is sent to a server, and the data (or an error message) is returned. In batch browsing, the textual information contained in the page is frequently enough to be meaningful. This is very different from the generalized result extraction model we presented, in that the schema of the results is not meaningful in itself. But, in web browsing, the page retrieved is often meaningful, not just for the sub-elements it describes. This aside, we consider result extraction in terms of gathering sub-elements from pages.

In interactive browsing, partial extraction is a simple process, and is at least marginally exploitable in the most widely used interactive browsers (Netscape Navigator, Internet Explorer). Both feature an "auto-load" feature that can be toggled (to varying degrees) to automatically load (or not load) different content type such as images, audio, or video. For instance, some users are concerned with images and text, but do not with to be disturbed by audio. Their browser makes the HTTP requests for all sub-elements, save audio. This is partial extraction! In other cases, especially with slower internet connections, images are expensive to download; users may choose to not automatically download images until determining that a particular image or set of images is important enough to invest time in.

Partial extraction in web browsing is a special case of the general partial extraction model in that the first result to be extracted always contains information about the other results to be extracted. Based on this first result, the client not only determines its interest in the other elements of the page, but also gets the information about what other results are available at all. This is in contrast to our general model, where a result parameter *may* but is not required to provide information about other result parameters, and where all possible result parameters are specified in a repository beforehand.

The most commonly found progressive extraction in web browsing is quite different from progressive extraction in a computational service, though progressive extraction of computational services over the web (e.g. improving simulation data) is also feasible. In a computational service, progressive extraction refers to extracting various transformations of input data over the life of a computation. In web browsing, progressive extraction is actually repeated extraction of a changing data stream. Weather services on the web often provide continuous updates and satellite images. Stock tickers provide updated information so users can know about their investments. Repeated extractions from the same stream show the stream's progress through time. Sometimes these repeated extractions may be done by manually reloading the source, or they may be pulled from servers by HTML update commands, JavaScript code, embedded Java-code, etc. Such data is retrievable any time and its progress status is always DONE and 100%

complete, yet we expect the data to also contain information about which point of time it refers to. (The same properties hold if the data is an XML document multiply retrieved; when retrieving the entire document is required to determine result status, optimization opportunities are radically reduced.)

4.3.2 Incremental Result Extraction and Progress Monitoring in JointFlow

JointFlow is the Joint Workflow Management Facility of CORBA [17]. It is an implementation of the I4 protocol of the workflow reference model of WfMC [21] on top of CORBA. JointFlow adopts an object oriented view of workflow management: processes, activities, requesters, resources, process managers, event audits etc. are distributed objects, collaborating to get the overall job done. Each of these objects can be accessed over an ORB; the JointFlow specification defines their interfaces in IDL. We have chosen JointFlow for comparison because it is a protocol that supports requests to remote computational units which can have some degree of autonomy. Also, the protocol is also based on asynchronous invocation of work and extraction of results, having special primitives for invocation, monitoring, and extraction.

JointFlow work is started when a *requester* asks a *process manager* to create a new *process*. The requester then communicates directly with the new process, setting *context attributes* in the process and invoking the start operation of the process. A process my be a physical device, a wrapper of legacy code, or it may initiate several *activity objects* which might in turn use *resources* (e.g., humans) via *assignments* or itself act as requesters for other processes. Our interest here is the interaction between the requester and the process concerning result extraction and progress monitoring.

4.3.2.1 Monitoring the Progress of Work in JointFlow

In JointFlow, processes and activities are in one of the following states: running, not_running.not_started, not_running.suspended, completed (successfully), terminated (unsuccessfully), aborted (unsuccessfully). A requester can query the state of a process, the states of the activities of the process (by querying and navigating the links from

processes to activities), and the states of assignments (by querying and navigating the links from activities to assignments). If the requester knows the workflow model with all its different steps implemented by the process, the requester might be able to interpret the state information of all sub-activities and assignments and figure out what the actual progress of a JointFlow process is. If the full workflow is not known, e.g., due to an autonomy boundary (which are assumed in CHAIMS), the only status information provided by the JointFlow protocol is essentially “service completed” or “service not yet completed.” In contrast, CLAM supports the notion that certain services may support progress information (e.g. “I am 40% done”) that can be monitored. This information is more detailed than just “running” or “complete,” and yet more aggregated and better suited for autonomous services than detailed information about individual component activities.

In contrast to CHAIMS, which polls all progress information, in JointFlow a process signals its completion to the requester by an audit event. These **audit events** could be used to implement CHAIMS-like progress monitoring on top of JointFlow: a process can have a special result attribute for progress information and the process is free to update that attribute regularly. It then can send an audit event with the old and new value of the progress indicator result to its requester after each update. Unfortunately, this result attribute cannot be polled by a requester (in contrast to CPAM or SWAP), because *get_result* only returns results if all results are available at least as intermediate results.

4.3.2.2 Extracting Results Incrementally in JointFlow

Both processes and activities have an operation get_result():ProcessData (returning a list of name value pairs). Get_result does not take any input parameter and thus returns all results. The get_result operation may be used to request **intermediate result data**, which may or may not be provided, depending upon the work being performed. If the results cannot yet be obtained, the operation get_result raises an exception and returns garbage. The results are not final until the whole unit of work is completed, resulting in a state change to the state “complete” and a notification of the container process or the requester. This kind of extraction of intermediate results corresponds to the progressive

extraction of all result attributes in CHAIMS. The following features found in CHAIMS are **not available** in JointFlow:

- **Partial extraction with get_result**: only all or none of the result values can be extracted by get_result, and there is no mechanism to return an exception only for some of the values.
- **Progressive extraction with get_result of just one result attribute** when not all other results are ready for intermediate or final extraction
- There is no **accuracy information** for intermediate results, unless it is in a separate result attribute. There is no possibility to find out about the availability or the accuracy of intermediate results unless requesting these results.

Though partial and progressive result extraction were not part of the design of JointFlow, they can be achieved by using audit events and by pushing progressive and partial results to the requester, instead of letting the requester poll for them. A process or an activity can send out an audit event to its requester or to the containing process whenever one of the result values has been updated. This event would then contain the old as well as the new result value. In case of large data and frequent updates, this messaging mechanism could result in huge amounts of traffic. The mechanism would have to be extended by special context attributes that tell the process or activity in advance which results should be reported and at what intervals. Even this strategy results in a very static and server-centric approach, in contrast to the client-centric approach in CHAIMS. Also, as partial and progressive result extractions are not mandated by the JointFlow protocol itself, it is questionable how many processes and activities would actually offer it.

4.3.3 Incremental Result Extraction and Progress Monitoring in SWAP

SWAP (Simple Workflow Access Protocol) is a proposal for a workflow protocol based on extending HTTP. It mainly implements I4 (to some extend also I2 and I3) of the WfMC reference model. SWAP defines several interfaces for the different components

(which are internet resources) of the workflow system that interact via SWAP. The three main components are of type ProcessInstance, ProcessDefinition and Observer. The messages exchanged between these components are extended HTTP-messages with headers defined by SWAP. The data to be exchanged is encoded as text/XML in the body of the message. As in JointFlow and in CHAIMS, the process instance creation, setting of context attributes, start of the process, and the extraction of results are done asynchronously. SWAP can be considered a precursor to web services.

4.3.3.1 Result Extraction and Result Monitoring in SWAP

Results are extracted from a *process instance* by sending it the message "PROPFIND" at any time during the execution of the process instance. This message either returns all available results, or if it contained a list of requested result attributes, it only returns the selected ones. Only result attributes are returned that are complete and available. If requested attributes are not yet available, presumably an exception should be returned for these result attributes. SWAP does not specify if the results returned by PROPFIND have to be final or not. Given the possibility to ask for specific result attributes, and to get exceptions for specific result attributes in case they are not available, allows some degree of partial and maybe even progressive extraction. SWAP exceptions are encoded in XML, instead of having just one possible exception per procedure call, as in the CORBA based JointFlow protocol.

A process instance signals the completion of work to an observer with the "COMPLETE" message. This message also contains the result data: all the name value pairs that represent the final set of data as of the time of completion. After sending the COMPLETE message, a resource does not have to exist any longer; this in contrast to CHAIMS where no result data is lost until the client sends a TERMINATE.

A process instance can also send "NOTIFY" messages to an observer resource. These messages transmit state change events, data change events, role change events, and data change events containing the names and values of data items that have changed.

4.3.3.2 Incremental Result Extraction in SWAP

The mechanisms of SWAP allow the following kind of result extraction and progress monitoring:

- **Partial result extraction**: Either pushing results via NOTIFY messages or pulling results via PROPFIND messages is possible. NOTIFY sends all new result data, PROPFIND returns all available result data whether or not they have already been returned by a previous PROPFIND. Notification of result changes without sending all new values is not possible unless additional result attributes are explicitly added. The same is true for getting the status of individual results: asking for the status of results, without also retrieving the results, is not possible unless a state attribute is added for each data attribute to the set of result attributes.

- **Progressive result extraction**: The SWAP specification does not explicitly specify if progressive result updates in a process instance are allowed or not. If not, the result attributes would not be available until their values are final. If yes, then progressive results can be extracted either by pushing results via NOTIFY messages or by pulling results via PROPFIND messages. Accuracy indication is not provided; it could be implemented via additional result attributes.

4.3.3.3 Process Progress Monitoring in SWAP

PROPFIND not only returns all result values available, it also returns the state of the process instance and additional descriptive information about the process. As possible states can be specified by the process itself, PROPFIND also returns the list of all possible state values, yet in most cases it would probably just be *not_yet_running*, *running*, *suspended*, *completed*, *terminated*, etc (the basic set of states defined by I4). A process instance can be asked for the URI of all the processes it has delegated work to, and an observer can then directly ask all sub-processes about their status. This is analogous to the model in JointFlow, and thus has the same drawbacks concerning autonomy and concerning amalgamated progress information.

Overall progress information is not specified by SWAP, but it could be implemented by a special result attribute, assuming that result attributes can be changed over time (are non-final after extraction). Such result attributes could be extracted any time by PROPFIND, independent of the availability of other result attributes.

Though SWAP does not support incremental result extraction as defined in CLAM, it could readily be added to the SWAP protocol itself or mimicked by using the SWAP protocol as defined and applying the simple workarounds mentioned above. As SWAP has very similar goals in accessing remote processes as the CHAIMS system, and as it is a very open and flexible protocol, its result extraction model is closest to the complete CHAIMS model and could be easily extended to encompass all aspects of the CHAIMS extraction model. Yet, because SWAP was not designed with incremental extraction in mind, it lacks the strong duality between extract and monitoring commands found in CLAM between EXAMINE and EXTRACT.

4.3.4 Incremental Result Extraction and Progress Monitoring in CORBA

4.3.4.1 CORBA- DII

CORBA offers two modes for interaction between a client and remote servers: the static and the dynamic interface to an ORB. For the static interface, an IDL must exist, and is compiled into stub code that can be linked with the client. The client then executes remote procedure calls as if the remote methods were local.

The dynamic invocation interface (DII) offers dynamic access where no stub code is necessary. The client has to know (or can ask for) the IDL from the remote object, i.e., the names of methods and the parameters they take. The CORBA client then creates a request for a method of that object. In the request the method name appears as a string and the parameters appear as a list of named values, with each named value containing the name of the parameter, the value as type “any” (or a pointer to the value and a CORBA type code), the length of the parameter, and some flags. Once the request is

created, the method can be invoked. This is either done synchronously with *invoke* or asynchronously with *send* (some flags allow more elaborate settings). Invoke returns after the remote computation has completed, and the client can read all OUT parameters in the named value list. In case of a send, the client is not blocked. After using send, to figure out when the invocation has finished, the client can use get_response, either in a blocking (it waits until invocation is done) or a non-blocking mode. As soon as the return status of get_response indicates that the remote computation is done, the client can read OUT parameters from the named value list.

In case of the asynchronous method invocation in CORBA-DII, the progress of an invocation can be monitored and asked for by the client as far as completion is concerned, but no further progress information is available. Progressive extraction of results is not supported by DII. Of course, a client is free not to read and use all results after the completion of an invocation, yet while the computation is going on no partial extraction is supported.

4.3.4.2 CORBA Notification Service

In order to mimic the incremental result extraction of CHAIMS, one could use asynchronous method invocation with DII coupled with the event service of CORBA. The client could be implemented as a PullConsumer for a special event channel CHAIMSresults, and the servers could push results into that channel as soon as they are available, together with accuracy information. Though event channels could be used for that purpose (we could require that every service uses event channels for this), an integration of incremental result extraction and invocation progress monitoring into the access protocol itself is definitely more reasonable, as we consider this data extraction model to be an integral part of CHAIMS protocols. The same is true for languages used to program the client: while CLAM directly supports incremental extraction and progress monitoring, this is not the case for any of the languages in used for programming CORBA clients.

4.4 DATA EXTRACTION SUMMARY

An oft-repeated goal of the CHAIMS project was to build a composition-only language for remote, autonomous services. To that end, many different extraction models used in different service domains had to be considered. This examination led to the realization that a simple asynchronous RPC-style approach was not enough.

Building a language and access protocol to support arbitrary result extraction models took careful consideration of the myriad ways extractions were currently being used in widespread as well as hand-crafted systems. Building support and primitives for all of these result extraction methods (asynchronous, progressive, and partial) and then binding them within one system led to the formulation of this comprehensive model for arbitrary result extraction.

Our model captures the notions of traditional result extraction, partial extraction, and progressive extraction. Through two simple primitives (EXAMINE and EXTRACT) in the CLAM language, the full expressiveness of each of these extraction types can be achieved. This extraction model is appropriate as a template for existing systems and future languages as well. It is generic to result extraction, and only assumes that the necessary asynchrony can be achieved among components through distributed communication, threading, or some other available means.

With this increased functionality, intelligent schedulers may perform optimizations previously not possible. Early result extraction, fine-grained progress monitoring, and successive refinements all become scheduling options enabled by our model. However, even if some components within a scheduling system do not fully adhere to this result extraction model, understanding the limitations of particular components, and how those restrictions interact with more flexible components, can still enable interesting scheduling optimizations [23]. We cover these options more fully in Chapter 5 when we describe the Surety-Based Scheduler.

5 SURETY-BASED SCHEDULING

As discussed in earlier chapters, computational Grid projects are ushering in an environment where clients make use of resources and services that are far too expensive for single clients to manage or maintain. Clients compose a *megaprogram* with services offered by outside organizations. However, the benefits of this paradigm come with the loss of control over job execution with added uncertainty about job completion. Current techniques for scheduling distributed services do not simultaneously account for autonomous service providers whose performance, reliability, and cost are not controlled by the service user. We propose an approach to scheduling that compensates for this uncertainty. Our approach builds initial schedules based on cost estimates from service providers and during program execution monitors job progress to determine if future deadlines will be met. This approach enables early hazard detection and facilitates schedule repairs to compensate for delays.

5.1 INTRODUCTION

Advances in the speed and reliability of computer networks in combination with distribution protocols (such as CORBA and Java RMI) allow clients to abstract away heterogeneities in the network, platform, language, etc., and make use of distributed services and resources that were previously unavailable. Remote services and resources can be utilized as if they were locally available. There are still complications that arise from geographic distance, security concerns, service autonomy, and compensation. In order to complete the abstraction to transparently use remote services and resources, it is necessary to have a mechanism to deal with the uncertainties introduced by scheduling services not under local control.

The development of Grid computing has enabled a model where organizations can develop services and charge a fee for their use to clients. Fee-based computing models are gaining success in both cooperative and commercial computing environments [35, 37,

39, 40, 43]. In these grids, service providers charge fees or trade resources for the use of their service. The value of the service offered is a combination of the software itself and the execution of the software. This is an attractive opportunity for service providers because they can amortize their development and maintenance costs and share these expenses with clients, while protecting their proprietary interests in the service. Also, under the CHAIMS model, service updates can be performed in a central location and not need to be propagated to end-users.

Clients also gain from this model in that they can access services that they do not have to develop or maintain [22]. Many clients do not have the resources to develop sophisticated software or purchase the high-end machinery necessary to accomplish their tasks. For example, suppose a small university's genomic research lab had a digitized DNA sequence from which they wanted to isolate a certain gene. Instead of developing the necessary software "in house" they hire a service provider that has the computational power and appropriate genomic software to analyze their data and give them the result they seek. Contracting for the service has the same result as purchasing the computational hardware and proprietary software, but at a fraction of the cost (contracting for services also saves on staffing, maintenance, and other costs). In an open market, valuable software services may have multiple service providers competing for the same pool of customers. A natural pricing structure would evolve based on the time to completion and the *surety* of the service providers. (Recall from earlier chapters that "surety" is the probability that a job will finish execution within a deadline window forecast by the service provider.) Quickly executing services with a high surety would of course be more valuable than the same services that have longer running times or a low surety. A customer's choice of service providers would depend on what value they place on time, cost, and surety, simultaneously. Until now, schedulers have treated the remote service problem as a multivariate optimization involving only two variables: cost and time [34, 36, 44]. We extend the worldview by accounting for the uncertainty introduced when services are not under the client's control.

Access to an array of services provides many opportunities for service composition. The ability to compose services is an especially powerful tool for multi-disciplinary projects where no single client has expertise in all sub-problem areas [22]. By allowing for composition of existing modules, researchers can devote less effort to software development and more time to central research questions. But there is a pitfall: distributed services are not under the control of the client. This means that estimates for job completion time may be inaccurate and clients cannot control resource allocation to recover from hazards. An inaccurate estimate in the completion time of single service is undesirable; in a program comprised of multiple services this can quickly become untenable.

The main research problem we address in this chapter is the decreased level of scheduling surety that comes from composing a program from multiple distributed services [22, 45]. By making programs composed of distributed services more reliable, these compositions become an increasingly viable solution for a wide range of problems, and become an appropriate solution for a larger class of clients. Compositional programs differ in both computational model and resource management compared to smaller scale programming. Table 2 presents some of these differences (though it is by no means exhaustive).

Table 2. Computational models and resource management

Programming Level	**Computational Model**	**Resource Management**
End Systems	Multithreading Automatic parallelization	Process creation OS signal delivery OS scheduling
Clustered Computing	Synchronous communication Distributed shared memory	Parallel process creation Gang scheduling OS-level signal propagation
Intranet Computing	Client/server Loosely synchronous: pipelines IWIM	Resource discovery Signal distribution networks
Internet Computing	Collaborative systems Remote control Data mining	Brokers Trading Mobile code negotiation

At a finer resolution, the goal of the Surety-Based Scheduler is to take a program composed of multiple services and complete it within a soft deadline and cost budget, while guaranteeing a client-specified minimum level of surety. The scheduling process begins with the selection of an initial schedule based on service provider estimates for completion time and fee [59]. The initial schedule is driven by dependencies (data, control, etc.) between service invocations and estimates from service providers. At runtime, job monitoring detects misbehaving services that can jeopardize the completion of the entire program. During the monitoring phase, surety is recalculated whenever progress is made (or not made, though time has advanced). If surety drops below the minimum threshold determined by the client, the scheduler takes action to recover from the delay and increase surety to an acceptable level. Any repair measures taken require finding alternative schedules for the remainder of the program that restore surety without exceeding the program's budget. Monitoring coupled with reactive rescheduling is key to providing clients with the surety of distributed job completion, as they would expect from a program running solely on local resources.

Systems such as CHAIMS (Compiling High-level Access Interfaces for Multi-site Software) allow clients to abstract away heterogeneity and service autonomy while simultaneously compensating for pitfalls associated with both [5]. We focus on scheduling with CHAIMS because its preferred development language (CLAM – **C**omposition **L**anguage for **A**utonomous **M**egamodules) provides key language primitives that enable dynamic scheduling with the possibility of recovery from hazards. For instance, CLAM contains a primitive to get estimates of the job completion time and cost from a service provider (ESTIMATE), and a primitive to examine job progress from a service provider (EXAMINE) [16]. These two capabilities used in concert allow for scheduling a program with more confidence in execution time. Other coordination languages and frameworks, such as MANIFOLD, are also appropriate for this type of composition, but lack the EXAMINE and ESTIMATE primitives found in CLAM [46, 47].

The current supported runtime system for CHAIMS is known CPAM (**C**HAIMS **P**rotocol for **A**utonomous **M**egamodules) [18]. The protocol removes the barriers imposed by different programming languages and distribution protocols, while providing support for the scheduling primitives in CLAM. Programs written in CLAM are known as megaprograms, though the class of programs is known by myriad names in other scheduling literature (ensembles, compositions, grid programs, workflows, etc.) [18, 32, 37, 50]. Within CHAIMS nomenclature, "megaprograms" (compositional programs using remote services) were initial composed from "megamodules." "Megamodules" refers to what we have otherwise simply called "services." We make few assumptions about services; they may possibly come from multiple programming language sources, distinct hosts, and have different native distribution protocols [19]. (The first megaprogramming papers were written in the early 1990s (E.g., [5, 71]), and while the accepted nomenclature has moved on, the concepts remain the same.)

An early objective of CHAIMS was to simply develop a language and runtime support for the programs composed from distributed services. The focus of the work presented in this chapter is to add a dynamic scheduler that can deal with the issues that arise in an unreliable environment to CHAIMS. We build on that prior effort because the language and runtime support overlap well with the requirements of runtime testing and surety monitoring. However, our work is broadly applicable to any system where estimates may be gathered *a priori* and where clients may monitor runtime progress.

Current distributed service scheduling research has not presented a complete solution that incorporates uncertainty. Most distributed computation schedules assume a cooperative environment where delays are rare, and that initial estimates come from oracles. The foundation of this work is that distributed systems (in practice) are rife with uncertainty that affects the reliability of schedules generated *a priori*. Section 5.2 discusses the characteristics of autonomous service providers and the attributes central to the scheduling task. Section 5.3 underlines the CHAIMS facilities that enable composition and coordination of distributed services. Section 5.4 explains the scheduling techniques and job monitoring that we advocate. Section 5.5 covers related work and explains how

our techniques may be leveraged in distributed architectures other than CHAIMS. Experimental results of applying the schedule designed in this chapter are shown in Chapter 6. Further questions sparked by this research are covered in Chapter 7.

5.2 AUTONOMOUS SERVICE PROVIDERS

The Internet has made distributed services a reality and opened up a completely new scheduling problem area to explore [31, 36, 49, 51, 52]. Without careful consideration, as computations are moved farther and farther from client control, it is increasingly likely that hazards will slow or halt progress. These hazards may arise from hardware or software failures, to power outages, to resource mismanagement by a service provider. Additionally, in competitive markets, service providers try to maximize profits. Greedy service providers could mistakenly take on more jobs than they can handle and delay the finishing time of all jobs. Alternately, unscrupulous service providers might stop the execution of low-paying jobs, however unfair it may seem, for more lucrative jobs that arise. Running into delays for a single service can be costly, but when programs are composed from multiple distributed services, delays in one service can have an undesirable cascade effect that destroys scheduling commitments for the entire client program. One aid to avoid such problems comes in the form of contracts [43, 58]. In the simplest contracts, clients use initial estimates of job completion time to bind service providers. This still does not *guarantee* that service providers will be able to meet the deadline of their contract. As such, it is also necessary to monitor job progress during execution to determine if the contract will be met (and to react swiftly and appropriately to recover if it is not).

A timeline showing a program execution with hazards might look like Figure 10. In that figure, we see that a program runs from a start time (t_0) to finish time (t_{finish}). A program does useful work until it encounters a "hazard." After a period to respond to and reschedule around the hazard, useful work resumes until either the next hazard is encountered, or the program finishes.

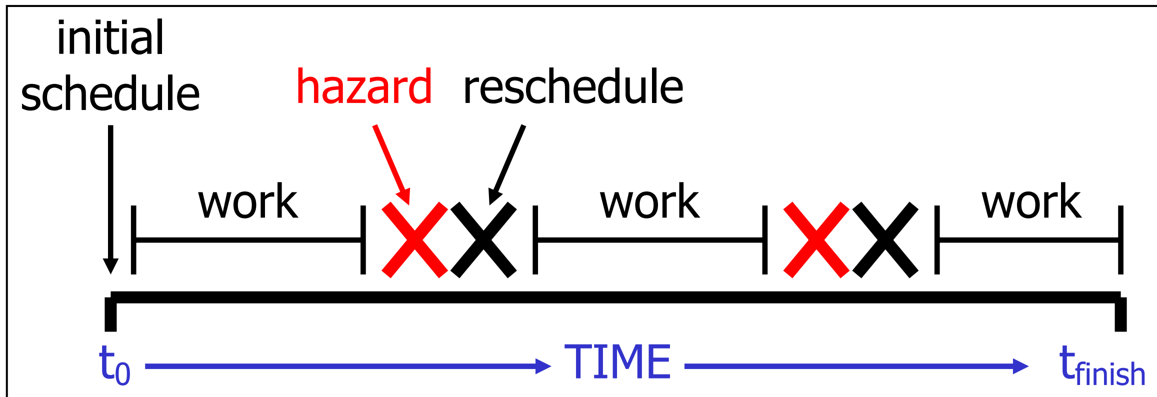


Figure 10 Program execution in the face of hazards

In this uncertain environment it is necessary to leverage contracts to motivate clients and service providers to meet their mutual obligations. At its core, the contract enables two parties who do not trust each other to enter into a mutually beneficial agreement. While contracts are a tool to promote accountability, we again stress that they do not *enforce* it. In this work, we do not discuss contract negotiation or enforcement, as they are implementation details that each distribution model must decide on. More information on contracts and negotiations within distributed systems can be found in [32, 34, 48]. We expect contracts to consider:

- *Cost* – what a service provider will charge for the service.
- *Completion time* - the estimated length of time to complete the job.
- *Variance* - the amount of time before or after the completion time that the job may finish. (It is assumed no service provider can be completely accurate in job estimation.) Variance may be presented symmetrically or asymmetrically. In our work, we assume symmetric variance in the examples, but provide equations to handle asymmetric variance.
- *Late fee* – a credit the service provider returns to the client per unit of time that the job is not finished after completion time plus variance.

- *Reservation* – an amount the client pays after negotiating the contract to hold resources until it exercises the contract and invokes the service. This reservation fee guarantees that the client will be given access to the service provider's resources, and confidence that the client can have some control over job start time. Reservations are a complicated issue themselves, and are more extensively discussed in [32, 43]. We assume "American options," which allow a client to start a service at any point until expiry.
- *Cancellation fee* - a set amount that the service provider will charge (or return to) the client if the contract is broken. This value may be zero, but is of utility to both client and service provider.

This distributed computational model for accessing services closely parallels traditional economical models. Significant detail on how grid computing relates to (and can leverage) various economic mechanisms can also be found in [31, 32, 36, 43]. We do not make any limiting assumptions about the trading or cash economies that will lubricate the grid; we simply examine the general considerations that economic factors place on scheduling under uncertainty.

5.3 CHAIMS

Technological advancements and social change have made grid computing a feasible option for computation. We have chosen the CHAIMS platform as a test bed for our investigations for the reasons mentioned in section 5.1. Specifically, we leverage its compositional programming language (CLAM) and the runtime support system (CPAM) to enable composition and coordination of distributed services. The Surety-Based Scheduler is designed to be usable within other compositional and grid platforms. In this section, we review the CHAIMS features important for investigating scheduling.

CLAM is a declarative language intended for large-scale service composition in an environment where parallelism is important [18, 19]. Unlike many other languages for distributed or parallel computing (e.g., HPF - High Performance Fortran [53]), CLAM

does not specify what resources a service invocation uses. This permits service instances to be scheduled identically, regardless of the capabilities and resources of the service's execution system. Breaking free from resource specification enables flexibility in choosing service providers at runtime, rather than compile time, selecting providers based on costs at execution time. This late binding time is similar to the tactic introduced in [43, 44].

CPAM is a generic high-level protocol for remote service invocation. CPAM separates a service invocation into a multi-step process to enable asynchronous service invocation and parallel execution. The steps of invocation are SETUP, SETPARAM (parameter setting), ESTIMATE (garner cost estimates), INVOKE, EXAMINE (monitor progress), EXTRACT and TERMINATE [19]. (An extended discussion of each primitive is available in Chapter 3.) The key facilities that CPAM offers are the ability to get *estimates* on job completion and to *examine* job progress during service execution, mirrored in the ESTIMATE and EXAMINE primitives of CLAM. Further discussion in Section 5.4 will show how ESTIMATE and EXAMINE provide the capabilities for building an intelligent scheduler. (Note that enabling these primitives in any distributed runtime system provides a generic framework for scheduling under uncertainty. Runtime support provided by mature grid implementations such as Legion is also appropriate as an alternative to CPAM [7, 54].)

With the construction of the Surety-Based Scheduler, ESTIMATE and EXAMINE are no longer required in the CLAM language, only as part of the CPAM runtime. An additional goal then of the SBS is to be effective enough to deprecate the EXAMINE and EXTRACT primitives in CLAM by utilizing them directly within a scheduler built directly on top of CPAM.

5.4 SCHEDULING IN AN UNCERTAIN ENVIRONMENT

In order to deal with an uncertain environment it is necessary to consider *surety* when scheduling. A surety threshold is set by a program user before a program is started, and

is monitored throughout the execution lifetime of the program. When progress information indicates that the surety threshold has been breached, dynamic repair and rescheduling operations are triggered. As mentioned previously, the Surety-Based Scheduler makes use of the ESTIMATE and EXAMINE capabilities to build initial schedules and to monitor execution progress.

Before scheduling of a megaprogram can start, the client determines a budget for the program. A budget is made up of (1) the deadline that the program must be finished by, (2) the amount of consideration (money, credits, bartered resources, etc. depending on the economic model) that can be spent on the program execution, and (3) the minimum level of surety that must be met by the scheduler. Individual clients determine the amount of time and consideration available for a specific program's execution. Surety represents a limit on the risk the client will tolerate in meeting their budget. Once the constraints of the budget are determined, the client can select to optimize or balance these three budgetary concerns (time, cost, surety). Figure 11 shows a simplified view of the scheduling process; steps not central to this chapter are omitted. We discuss each of these steps in greater detail; in this section, we simply present an overview of the process steps.

First, estimates are collected for each service from potentially many service providers and used in the program to build possible schedules. In our current naïve implementation, we exhaustively enumerate all schedules and select one from the pool of best choices. Before discussing which schedules are "best," we will clarify our underlying schedule evaluation techniques.

Once a candidate schedule is created, the shortest expected running time of that schedule can be determined using CPM (Critical Path Method) [55]. (A more extensive examination of CPM appears in APPENDIX A: Critical Path Method (CPM).) With this information it is trivial to test whether a schedule meets the minimum time and budget criteria, however nothing is known at this point about surety. The longest path (in terms of expected execution time) in the program determines the runtime of the program. This longest path is called the "critical path" because any delay along the critical path will

affect the running time of the entire program. To determine surety, it is necessary to extend the CPM analysis to a probabilistic PERT (Program Evaluation and Review Techniques) analysis [29, 30, 55]. PERT extends CPM by accounting for the uncertainty in each estimated service duration to compute the surety of the entire program. We will discuss our use of PERT in significant detail in the next section.

Once a schedule is selected and contracts are finalized, the scheduler may invoke any ready services in the program. As services execute, their progress is monitored using the EXAMINE facility of CLAM/CPAM to ensure that completion times are being met; if the overall surety of program completion drops below the predetermined threshold, the scheduler begins the repair and reschedule phase. In the repair phase there are many options. New service providers may be found to replace the service module that is delaying overall progress. Or other services along the critical path may be substituted for alternative services that have shorter runtimes (though at an increased cost). Once repair and rescheduling is complete, the scheduler returns to monitoring the execution. A longer discussion of runtime hazards and schedule repair tactics appears in Section 5.4.3.

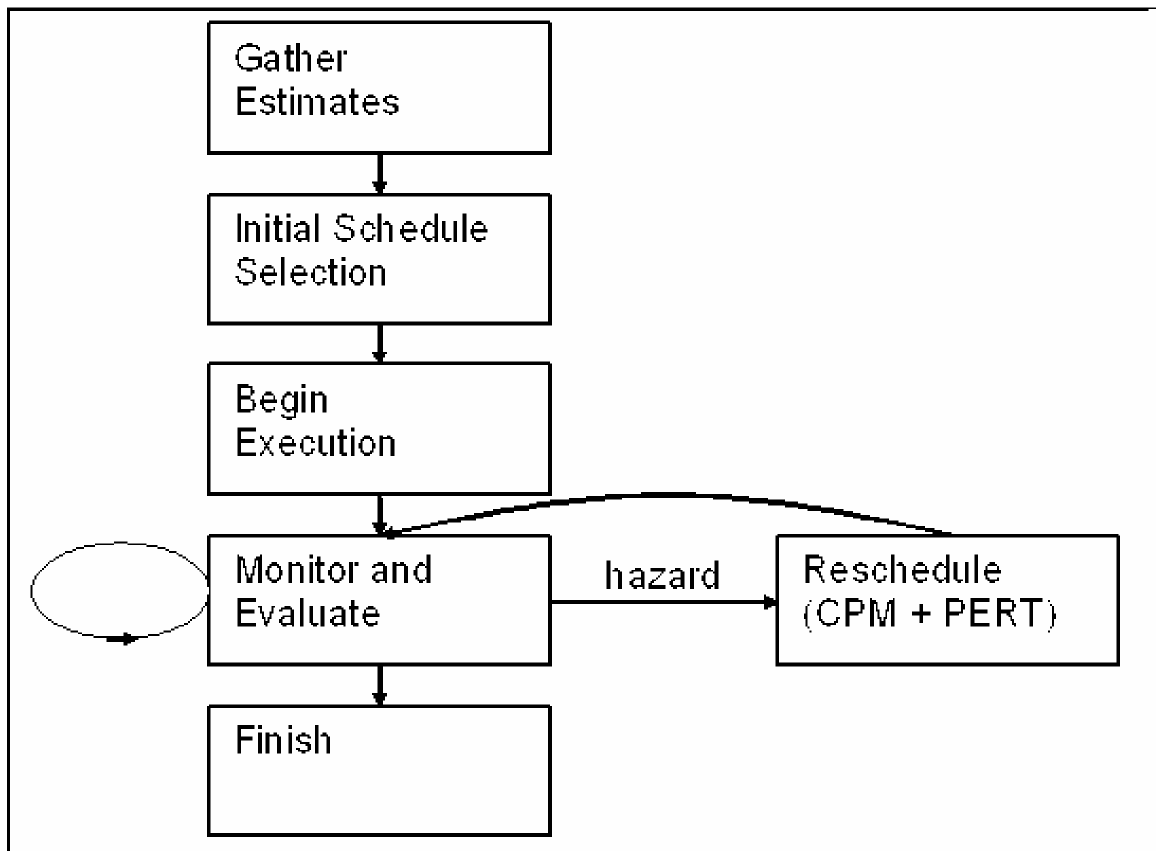


Figure 11 The scheduling process

5.4.1 SIMPLE PLANNING

The first step in scheduling is program analysis to discover any dependencies among component services and construct a dependency graph for the workflow. The very simple program text in Figure 12 shows implicit data dependencies between services. For instance, `service3` takes `A` and `B` as input. `A` and `B` are outputs of `service1` and `service2`, respectively. These dependencies are mapped into the workflow of Figure 13, where nodes represent services and dependencies are shown as arcs between nodes with the arrow pointing to the dependent node. These execution flows consist of paths that are created by dependencies between nodes. Once the dependencies are mapped, the scheduler requests bids from service providers in order to fill in cost values for the proposed schedule.

Before gathering bids from service providers, the scheduler contacts a well-known repository or directory that returns a list of service providers to perform a specific service. Based on this list, the scheduler contacts service providers and requests bids. The *bid request* is based on the service needed, the expected start time for the service, and information about the size and complexity of the input parameters to the service. For some services, the inputs cannot be known at runtime because they are the outputs of other services; in these cases, information about the size and complexity of parameters is currently based on heuristics that the service provider uses to make best estimates. Service providers collate this information and calculate a possible bid. The client receives a collection of bids and will either accept one or more bids for the schedule, or may attempt to renegotiate with the service providers [32, 34, 48]. The deciding factor in which bids are accepted is based on the Pareto optimality [56, 114] of the "best" schedules. For a bid to be Pareto optimal, there can be no other bid with an absolute advantage in terms of price, time and surety. The Pareto curve in this case is weighted by the optimization strategy presented by the client, for instance in soft real-time applications surety will have a high weighting. All decisions are based simultaneously on cost, time, and surety.

```
// begin program
A = service1();
B = service2();
C = service3(A,B);
D = service4(C);
E = service5(C);
// end of program
```

Figure 12 Sample program

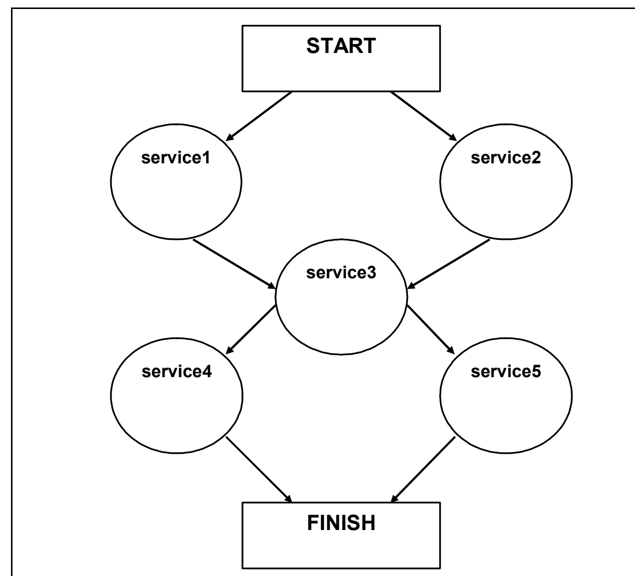


Figure 13 Dependency graph of sample program

Using CPM gives the scheduler the critical path and the expected runtime of the program. This information allows the schedule to select candidate schedules that meet the client's budget and to further optimize and refine the schedule. For instance, CPM indicates positions of "slack" in the schedule, places where cheaper, longer running processes may be used because they are not along the critical path [55]. By choosing slower services in these non-critical paths, the scheduler can possibly decrease the overall cost of the program, thus saving resources that may be used for schedule repairs at a later time. The total price cost for all services executed plus the cost of any reservations not kept is the total cost of the program.

The PERT technique extends CPM to account for uncertainty of individual service completion times and determines the probability of completing a complete program by an expected time [55]. PERT analysis also forms the basis for our rescheduling decisions. To perform the initial analysis, the scheduler uses three time estimates for each service: most likely(m), optimistic(a) and pessimistic(b) completion times. "Optimistic" and "pessimistic" times are derived from the expected time coupled with the variance. With this information, the *expected duration* e_i of a single service can be determined by a

weighted combination of the most likely duration m_i and the midpoint of the distribution $\frac{(a_i + b_i)}{2}$ for each service *i*:

$$e_i = \frac{2m_i + \frac{a_i + b_i}{2}}{3} \qquad \textbf{(1)}$$

For a normal distribution, there is a spread of about 6 standard deviations from a_i to b_i thus:

$$\sigma_i = \frac{b_i - a_i}{6} \qquad \textbf{(2)}$$

Activity durations are independent; hence the sum of the independent random expected functions e_i is normally distributed. From the expected completion time and standard deviation of each service on the critical path, we construct the expected completion time and standard deviation of the entire program as:

$$\bar{e} = \sum e_i \text{ and } \sigma_{program} = \sqrt{\sum \sigma_i^2} \qquad \textbf{(3)}$$

for all services *i* on the critical path. With this we can calculate the probability that the program completion time *T* is less than the deadline of time *t*. This $prob(T \le t)$ represents the surety level of the program completing execution by its deadline, *t*. We specify the completion time as:

$$t = \bar{e} + x * \sigma_{program} \quad \text{giving us} \quad x = \frac{t - \bar{e}}{\sigma_{program}} \qquad \textbf{(4)}$$

which is used to express the surety of the program as:

$$prob(T \le t) = \left[\frac{T - \bar{e}}{\sigma_{program}} \le \frac{t - \bar{e}}{\sigma_{program}} \right] \qquad \textbf{(5)}$$

This is the same as the probability of a random variable from N(0, 1) distribution being less than or equal to $\frac{t - \bar{e}}{\sigma_{program}}$.

Using CPM and PERT, the scheduler can evaluate the bids that form the final schedules. The high-level architecture of the Surety-Based Scheduler can be seen in Figure 23. Initial estimates to drive the CPM and PERT calculations are gathered by the estimator/bidder. The execution monitor gathers runtime observation to update the surety calculations. The dispatcher carries out scheduling decisions. Calculations and decisions are made in the scheduling logic module.

5.4.1.1 Choosing an Initial Schedule

With our Surety-Based Scheduler, program users can specify at runtime what the most important considerations are to determine the "best" available initial schedule. For example, certain users may place a higher premium on execution time and are willing to "spare no expense" to get the job done as quickly as possible. Other users may execute an identical program, but only care about minimizing the overall execution cost, without consideration of the time required to complete the program's execution. Still other users may favor surety above all other considerations. These competing user objectives complicate matters because different considerations (time, cost, or surety) often have orthogonal scheduling techniques. With our Surety-Based Scheduler, scheduling trade-offs are not determined solely by the program being executed, or even by available resources, but also by runtime user preferences. Shifting goals for task instances and users means significantly increased scheduling complexity.

Considering user preferences, how is an initial schedule selected from the space of possible schedules? First, the schedule space is constrained by the program user's budget. The budget consists of the three parameters mentioned previously: time, cost, and surety.

- **Time** provides an *upper-bound* on the runtime of acceptable schedules (e.g., 22 hours allowed for execution).
- **Cost** provides an *upper-bound* on the cost of acceptable schedules (e.g., $250 available to purchase required services)

- **Surety** provides a *lower-bound* on acceptable schedules (e.g., only consider schedules that are 90% likely to complete within the time allowed)

The triple of {*time, cost, surety*} is the user's budget. Using the examples in the bulleted list above, the budget may be represented as {22h, $250, 90%}. The user's budget *limits* the space of possible schedules, but we allow user preferences to *shape* the remaining space of schedules. Program users can specify weights for each scheduling constraint. For example, a user who places 10 times the importance on execution time more than execution cost, and five times more importance on surety than cost, can represent her preferences (or "utility") as a triple similar to the budget triple: <10, 1, 5>.

Any particular schedule in the space of all possible schedules then has a particular utility to a specific user, where that utility is determined by a linear combination of the user's budget, the user's preferences, and the initial estimates of execution time, cost, and surety. Initial estimates are also represented by a similar triple, [*estimated Time, estimated Cost, estimated Surety*]. A schedule's utility is determined by the following linear equation:

```
Overall Utility(Schedule[est. Time, est. Cost, est. Surety]) =

            (budgeted Time – est. Time)      * utilityOfTime   +

            (budgeted Cost – est. Cost)      * utilityOfCost   +

            (est. Surety – budgeted Surety) * utilityOfSurety
```

Readers may recognize this as a form of multiobjective query optimization, covered in greater depth in [100]. If we consider just time and cost as objectives (for sake of simplicity), we can see what the space of initial schedules looks like in Figure 14. The x-axis is labeled "budget time," and the y-axis is labeled "budget cost." First, individual schedules (labeled "plans" in the key) are represented by the round points that dot the space. Schedules that exceed the user's budget in terms of time or cost lie in the grey areas. As such, we only consider plans in the un-shaded region. The schedules will determine some Pareto curve, or curve of semi-optimal schedules [56, 100]. The Pareto

curve is shaped to the exclusion of all schedules for which there is another schedule holding an absolute advantage, an advantage in both cost and time. The Pareto curve for this space is the segmented line connecting the schedules for which there are no competing plans with an absolute advantage in both time and cost. Finally, the user's utility curve is shown as the smooth line intersecting the Pareto curve. We can be certain that if user-specified utility values are not allowed to go negative, the ideal schedule for any given user will be some schedule on the Pareto curve, inside the un-shaded region (assuming that such a plan exists). If no such schedule exists, then one of two things must be true:

1. There are no schedules that will complete the program within the user's budget, or
2. The user's budget in limited in such a way that he will have to settle for execution of a strictly suboptimal plan.

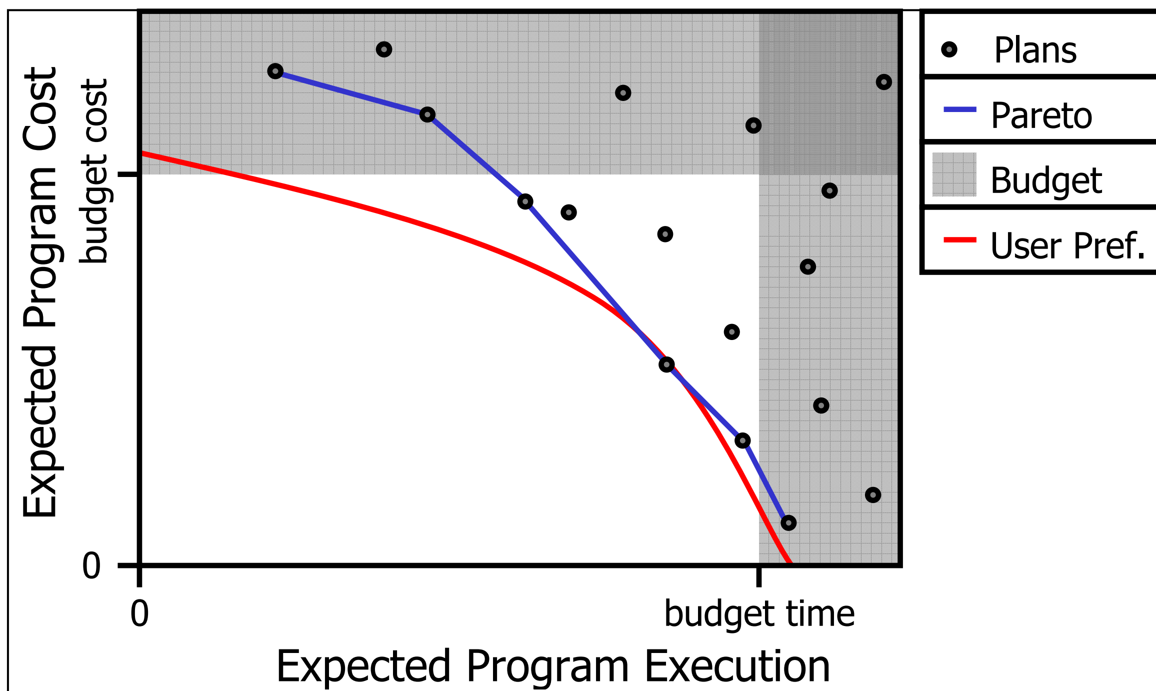


Figure 14 Schedule search space in 2-D

5.4.1.2 An Example of Repairing a Schedule at Runtime

What follows now is an example of the complete repair and scheduling activity. For this example, Table 3 shows a set of bids produced for the sample program presented in

Figure 12. In some cases, multiple service providers bid for a service, giving the scheduler some flexibility. In the case of `service3`, only one service provider has replied with a bid, thus it will have to be used.

After all initial service bids have been received, the scheduler searches for an optimal initial schedule, based on the client's budget (as described in 5.4.1.1). Assume the client has given a deadline of 20 hours for time, a cost budget of ¥65 and a minimum surety requirement of 90%. (We use the "¥" symbol simply to distinguish the cost numbers from the time numbers, and make no implication about any particular specie or economic model.) The bids from Table 3 are used to construct the initial schedule shown in Figure 15. In Figure 15, the critical path consists of nodes `service1`, `service3`, and `service4`. By application of CPM, this schedule has an expected finishing time of 18, an earliest finishing time of 14, and latest finishing time of 22 (assuming no hazards). The total budget estimated for this program is ¥38, allowing a reserve of ¥27, which can be used to repair the schedule in case of delay or hazard. Figure 16[1] shows the probability distribution of this program's completion time. The surety for the program is determined via PERT to be 98.31% (shown in Figure 16), an acceptable level (≥ 90%, the client-specified minimum surety level). Please note that these values for cost, time, and surety are from an *a priori* analysis based solely on service provider estimates. In the next section, we explain surety based monitoring and repair strategy.

[1] Note that in Figure 23 the user preference vector is shown as a curve, rather than a straight line. This has previously caused some confusion. If user preferences are a simple combination of the user's budget elements: cost, time, and acceptable surety, how can this yield a curve? User preferences determine the shape of a curve (rather than a line) because the left over cost and time for any given schedule directly impact surety calculations and as noted earlier it is not a simple linear combination. As time and cost vary linearly, *the impact on surety is not necessarily linear*. (If the user was selecting just a *single service*, the preference vector would in fact be a straight line!)

Table 3. Bids received for each service

Service	Cost	Time
service1	8	8 +/- 2
	9	10 +/- 1
	11	6 +/- 0
service2	10	5 +/- 2
	12	4 +/- 1
service3	5	6 +/- 0
service4	15	2 +/- 0
	10	4 +/- 2
service5	20	1 +/- 0
	10	2 +/- 2
	5	3 +/- 1

5.4.2 MONITORING AND REPAIRING SCHEDULES

Once the initial planning stage is complete and contracts have been drawn, execution starts. For our running example, we assign this start time a convenient value of time=0. The schedule for our example (shown in Figure 15) is very “tight” in terms of cost. There are alternative assignments of service instances that would give a lower overall cost. This trades off with overall runtime of the schedule to some extent, but the tradeoff is considered acceptable because the schedule does not fall below the surety threshold.

Delays along the critical path are likely to be the most damaging since they extend the run time of the entire program. Constant monitoring is required to ensure that single delays do not affect the entire program. However, delays not in the critical path can also impact surety. We illustrate this case next.

Consider at time=0 that `service1` and `service2` begin execution. Imagine that we monitor progress at each integer time interval. At time=1, time=2, and time=3, the scheduler observes no anomalies. However, at time=4, a hazard is detected!

According to the expected schedule, at time=4, `service2` should be 80% completed. Imagine that the scheduler observes that `service2` is only 50% complete. Based on this information, the scheduler projects that the `service2` will complete at time=11, thus deforming the critical path to now include `service2` instead of `service1`. This potential delay changes the expected running time of the overall program, which

subsequently lowers the surety of the overall execution to 14.44%, which is an unacceptable level (< 90%). This dip in surety is a hazard.

To counter this delay, the scheduler first contacts all service providers to get new bids. (It is interesting to note that in our model as system conditions change, initial estimates become moot. This is especially true when scheduling long-running services.) The scheduler determines that the most effective strategy is to accept a bid to attempt to finish `service2` at an earlier time. At time=4, a bid for an instance of `service2` that will cost ¥10 and complete in 5 units of time (with a variance of 2 (5±2)) is found and accepted as a reasonable repair attempt. Immediately, this second service provider for `service2` can begin work in parallel with the delayed instance (recall that `service2` has no dependencies on earlier executions). This repair strategy has increased the surety of the complete program, but we now expect the program to finish somewhere between time=15 and time=23 with a mean expectation of time=19. This repair increases the surety to 85.56%, and reduces the reserve budget to ¥17. A surety of 85.56% is still below the client-specified surety threshold. This schedule requires further repair.

After the hazard is initially detected and an alternative schedule is found, surety remains below the 90% threshold. To further increase surety, it is necessary to select a node along the critical path and either find alternate service providers that can perform the same service in less time, or contract with multiple service providers to execute the same service in parallel, thus increasing the probability that at least one of them will deliver results in time. The method used to increase surety depends on how much budget is left over, and if alternative service providers can be found with the required performance capabilities. In this example, assume that we discover another service provider that offers `service4` for ¥10 with a completion time of 3, and variance of 2. Using this service provider increases overall surety to 95.83%, which is now above the threshold for this execution.

Recall that "surety" represents the risk of a client program not meeting a deadline that the client will accept. Monitoring job execution at runtime allows our CHAIMS scheduler to

compare current surety to the limit established by the client. Falling below the surety threshold triggers the scheduler to repair or reschedule to counteract the effects of hazard. This is achieved primarily by finding alternative or duplicate services increase surety. If surety is set too high (e.g., 100%) or the budget too low, the space of acceptable schedules is radically reduced, and the likelihood of successful schedules decreases as well. In the next section, we present a taxonomy of runtime hazards and schedule repair strategies. We also consider the interactions between given hazard types and a set of possible runtime repairs.

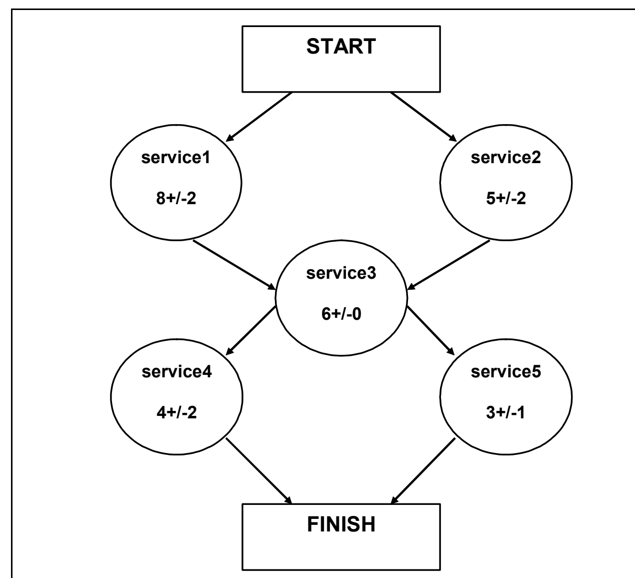


Figure 15 Sample Schedule

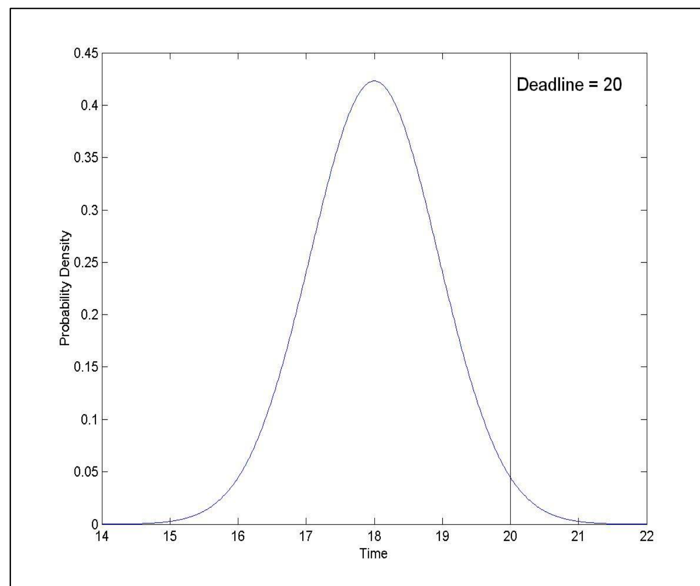


Figure 16 Probability distribution of original program schedule

5.4.3 Hazards and Repairs

5.4.3.1 Types of Hazards

Compile-time scheduling decisions have little bearing on the execution lifetime of a megaprogram. Megaprograms are really just templates or patterns for service invocations. Even though a megaprogram may compile, compilation only implies that the required services for a program are described in a service repository. It says nothing about the runtime cost, or even runtime *existence*, of a given service. If users had control over resource allocation during execution of megaprograms, or at least could be sure that there would never be any runtime hazards, then this scheduling problem would be trivial. However, the possible hazards encountered during execution in a distributed environment are numerous. Sample hazards include throughput delays and slowdowns, execution stoppages, inaccurate estimates from resource providers, communication failures, and the myriad complications that can arise from competitive interactions between various clients and service providers.

We place possible runtime hazards into three groups: progressive hazards, catastrophic hazards, and pseudo-hazards. Progressive hazards are those that clients can observe and

essentially "see them coming." An example would be a service invocation that proceeds at 50% of the work rate implied by its initial estimate. Through progress monitoring via the CPAM EXAMINE facility, a Surety-Based Scheduler can detect that a service provider is not on track to meet the estimated finish time. Figure 17 shows what surety (graphed against execution time) looks like when facing a progressive hazard. The surety level of the program drops slowly, until it dips below the user-specified surety threshold (the dotted line labeled "minimum surety").

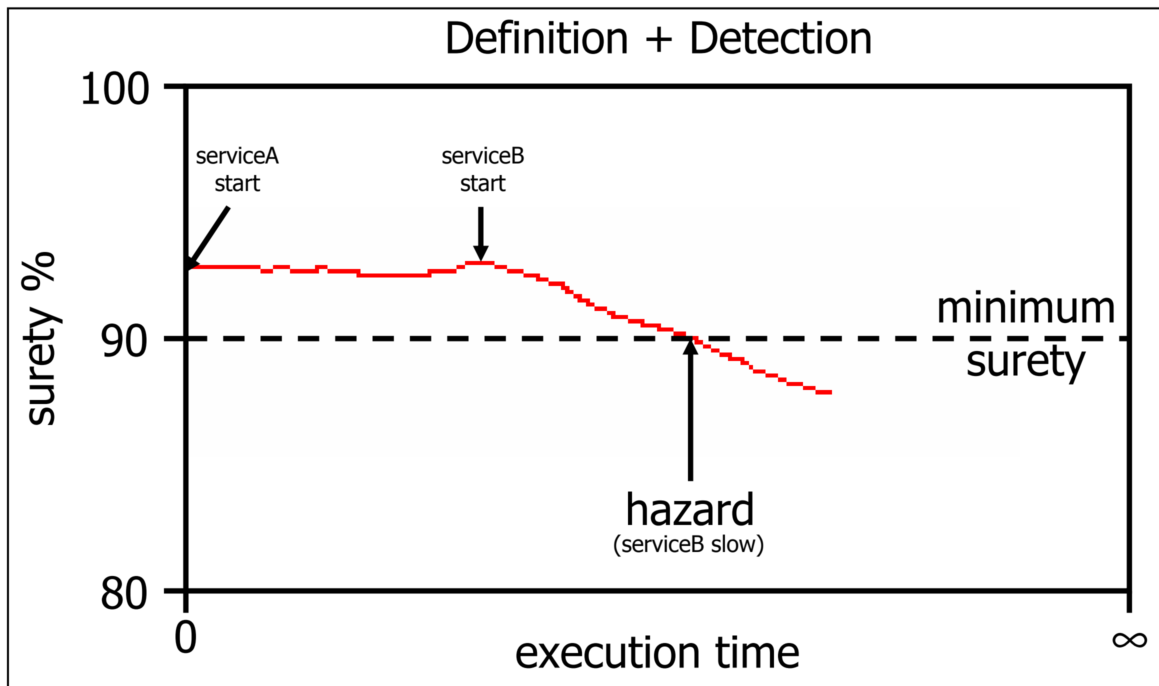


Figure 17 Progressive hazard

Catastrophic hazards are simpler to detect, though their surety impact looks very different when graphed against execution time. An example of a catastrophic hazard is an OS crash on a service provider's execution machine. Catastrophic failures drive program surety immediately to 0% chance of success. This effect is seen in Figure 18.

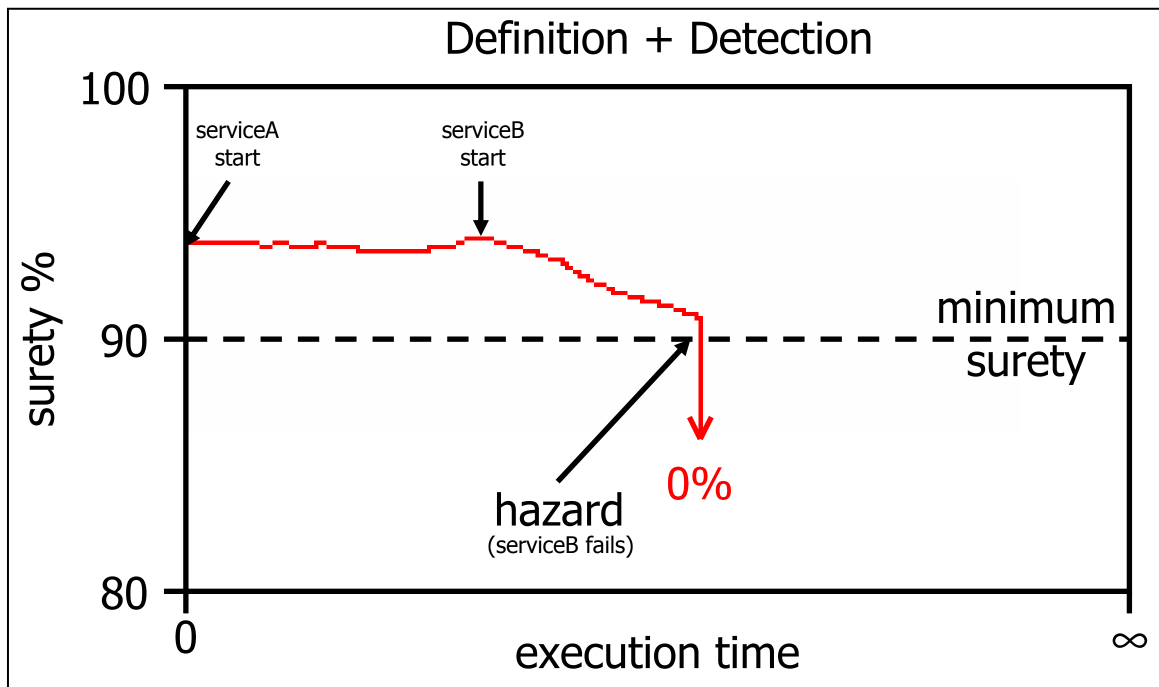


Figure 18 Catastrophic hazard

Pseudo-hazards present the most difficulty to any scheduler. A pseudo-hazard is an event that appears to be another class of hazard, but does not actually affect the execution surety of a program. An example of a pseudo-hazard is a communication failure between a client and service, where the communications failure is indistinguishable from a crashed service. Such a hazard looks catastrophic, and thus the scheduler reacts to compensate for the apparent 0% surety level. However, doing nothing may be the ideal "repair" for this hazard, if the communication link is restored before service completion. The external view of a pseudo-hazard is captured in Figure 19.

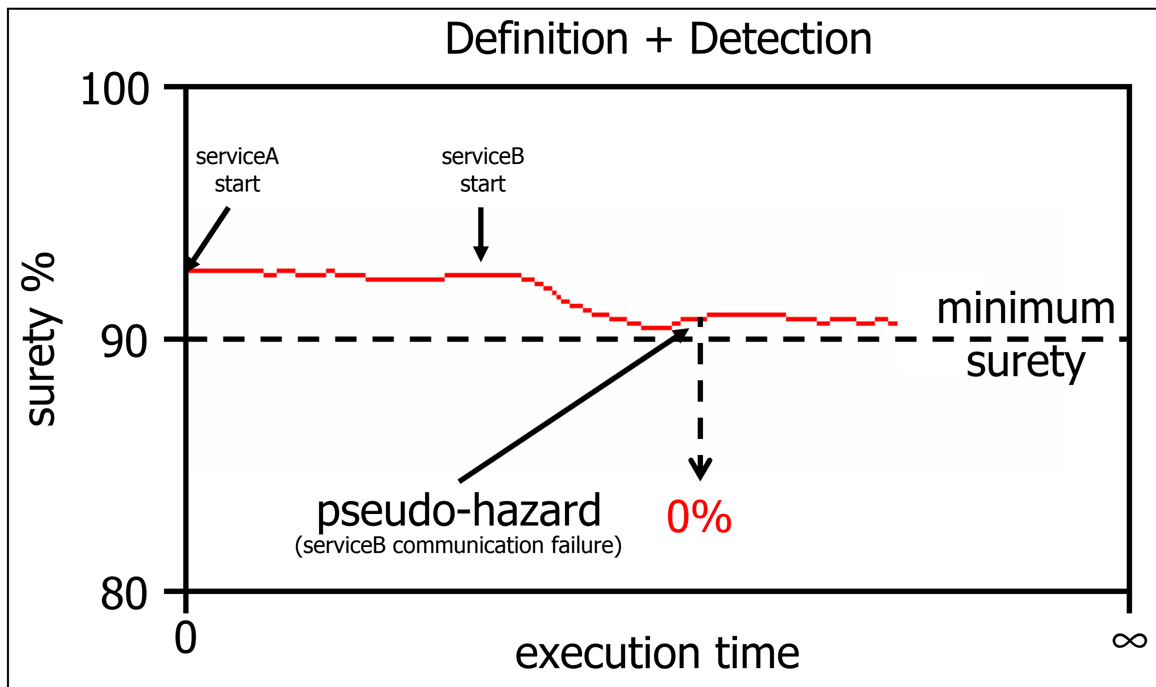


Figure 19 Pseudo-hazard

5.4.3.2 Types of Repairs

We offer a set of repairs sufficient to optimally repair any of the above hazards, where "optimally" is defined as engaging the least-cost repair possible. This set of repairs is not minimal. The set of schedule repair options is:

1. **do nothing** – doing nothing in the face of a hazard is an ideal solution for the class of pseudo-hazards, and also represents the "action" always chosen by static scheduler (a static scheduler is one that does not repair after the initial schedule is generated).
2. **service replacement** – is a tactic that replaces the service which is responsible for triggering a hazard event. It is not always possible to replace a service, nor always desirable, but it is the only meaningful repair for catastrophic hazards.
3. **service duplication** – duplicating a running service is a technique that offers the largest increase in surety for a program, but at the largest cost.
4. **pushdown repair** – a pushdown "repair" is not actually a repair, but a repair strategy that says the scheduler will consider all possible repairs, though only by repairing services other than the service directly responsible for the hazard. Pushing down

repairs can be a cost-effective strategy for progressive and pseudo-hazards, but offers no solution to catastrophic hazards.

By using these four types of repairs, an ideal scheduler can utilize the lowest cost repair strategy in the face of any hazard. The impact on surety of each repair type comes with very different costs and benefits. For example, the “do nothing” repair strategy is obviously the lowest cost strategy (lowest dollar-cost; the cost of the risk may be substantial). At the same time, it is the ideal strategy for handling pseudo-hazards, such as a network partitioning that will recover before the service’s expected finish time. Unfortunately, this “do nothing” strategy also depends on self-recovery; if the problem does not fix itself, it will result in a schedule that violates its budget constraints in *at least* one dimension. The effect of the “do nothing” strategy on overall program surety (at the time of the repair) can be seen by the dotted line in Figure 20. Program surety is not changed after the repair, and remains below the user-specified surety level.

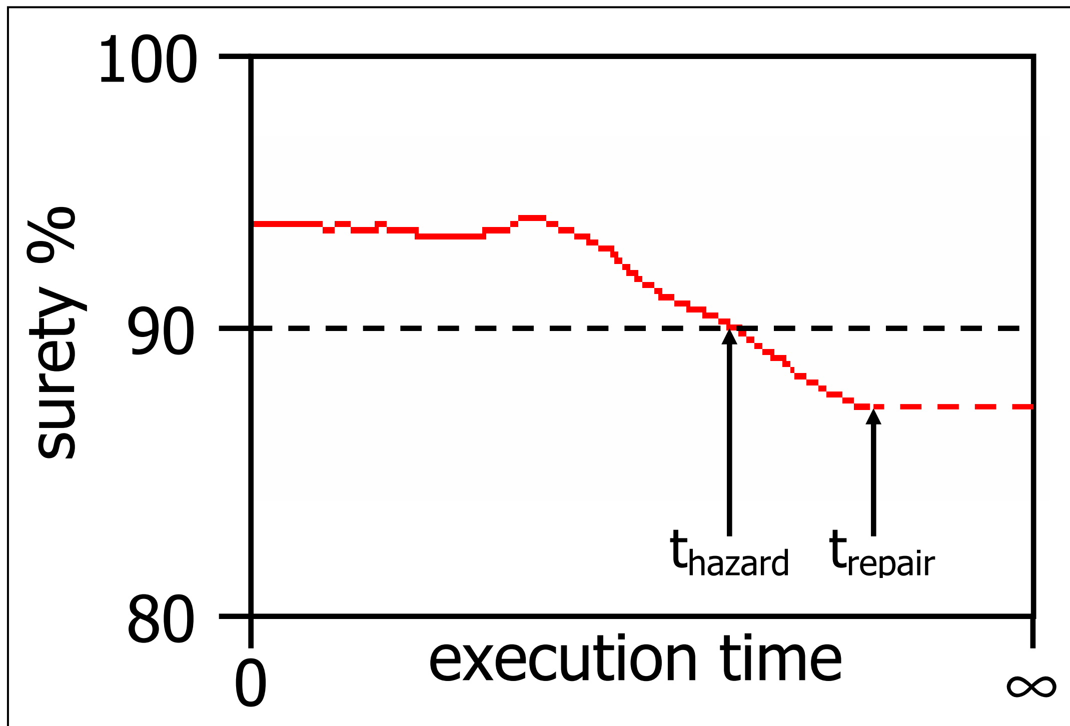


Figure 20 Effect of "do nothing" on surety

The service replacement strategy has a somewhat different effect on program surety, and provides a solution to catastrophic hazards not possible with the "do nothing" or pushdown repair strategies. By replacing failing services, we assume that some cost recovery is possible because the original service provider did not meet their obligation to complete the service on time. One drawback of this approach is that it concedes some investment in time and expense already sunk into the original service. And in a stance opposite of the "do nothing" approach, the service replacement strategy foregoes any chance of service recovery, making it the *worst* solution for pseudo-hazards. The effect of the service replacement strategy on overall program surety (at the time of the repair) can be seen by the dotted line in Figure 21. Program surety after the repair returns to the user-specified surety level, or above.

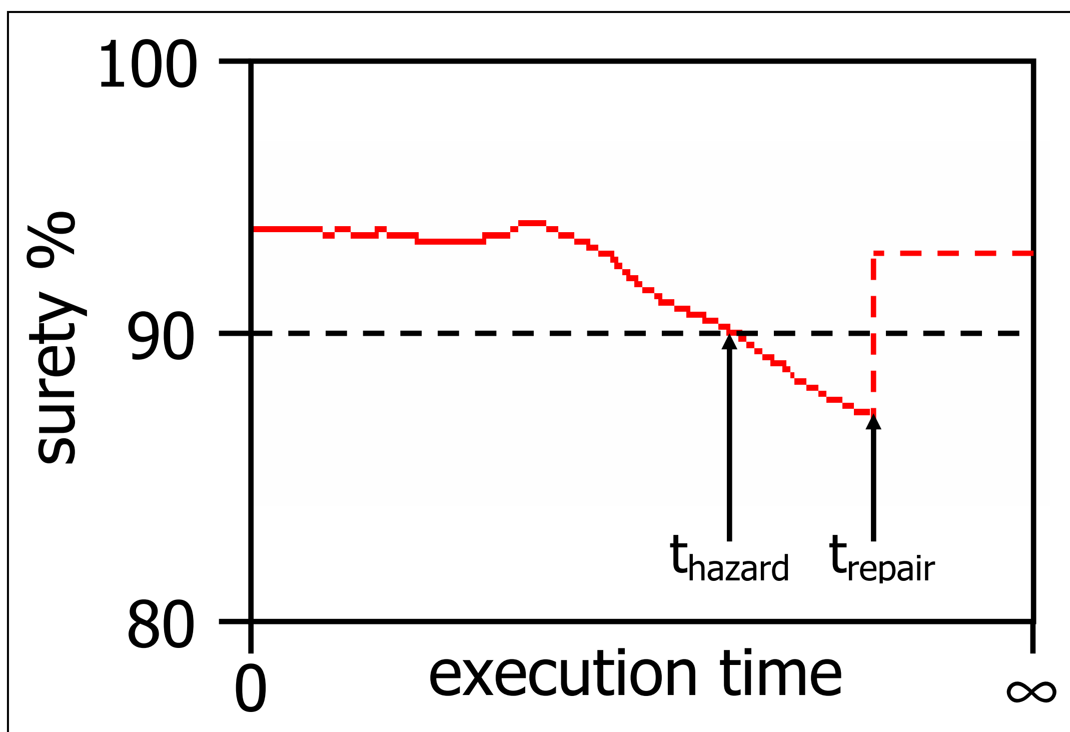


Figure 21 Effect of service replacement on surety

The service duplication strategy is an alternative to service replacement. Under the assumption that services operate independently, duplication provides the largest possible gain in program surety. It does this by leveraging the chance of service recovery that the

"do nothing" strategy depends on. One can think of the service duplication strategy like tossing extra coins to gain a desired outcome. We can model a service with a 50% likelihood of on-time completion as a fair, two-sided coin. Success (completion on time) is represented by a flip that yields heads, failure by tails. A successful outcome then means that the service coin comes up heads. Duplicating a service is statistically equivalent to tossing two fair coins, where the successful outcome is defined as having *at least one* positive outcome, heads. Flipping two fair coins, simultaneously, carries a 75% likelihood of coming up with *at least one head*. The downside of the duplication strategy is simple: it rapidly consumes the user's budget. The impact of a service duplication strategy can be seen in Figure 22. The highest dashed line represents some level of surety beyond that achieved by simple service replacement.

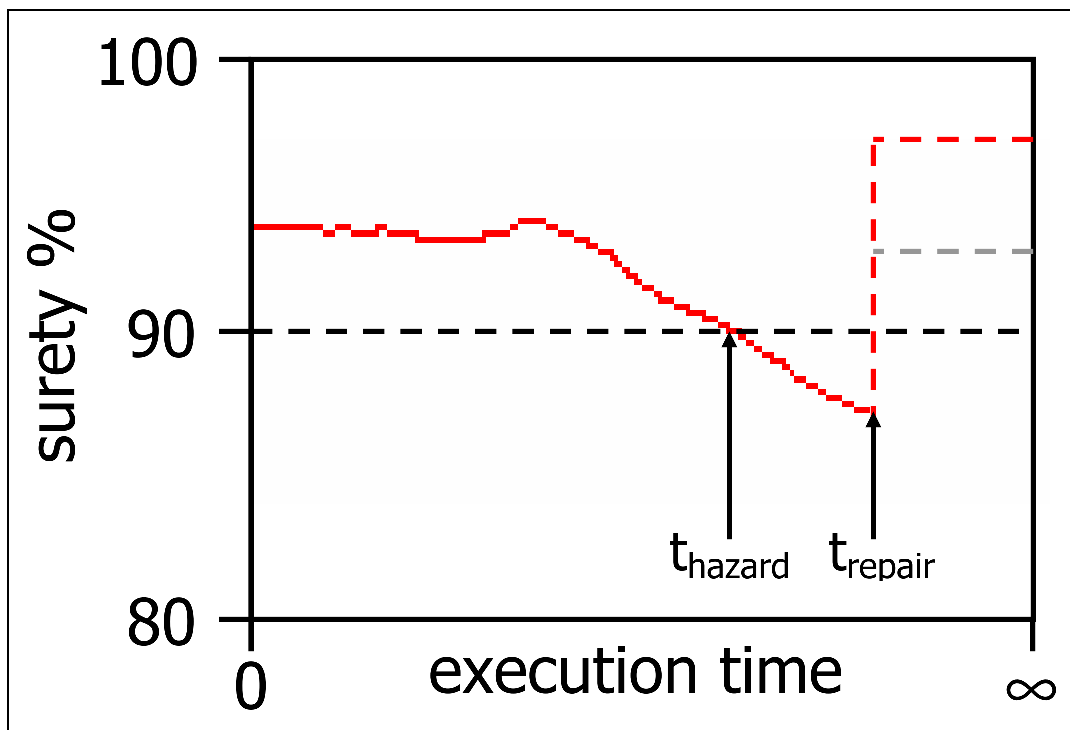


Figure 22 Effect of service duplication on surety

The last repair strategy is the pushdown repair, where the service experiencing the hazard is not rescheduled, but some later service on the critical path (assuming one exists) is repaired. To simplify matters, we only consider pushing down a service replacement or

service duplication (pushing down a “do nothing” repair is indistinguishable from doing nothing at the service experiencing the hazard). The pushdown strategy has some interesting implications. First, it may be relatively inexpensive compared to the other active repairs because it preserves all investment in time and expense of the service experiencing a hazard. Of course, this advantage only applies to progressive or pseudo-hazards. A pushdown strategy does nothing to repair a catastrophic failure. Also, the pushdown strategy depends on the availability of multiple alternative services for services not yet executed. The impact of the pushdown strategy can be represented by Figure 21, the same figure used to show the relative efficacy of the service replacement strategy. It should return the program surety level to someplace above the user-specified threshold. Pushdown repairs have a high time cost if the strategy fails, thus is of greatest utility when the value of execution time is lowest.

5.4.3.3 Multiple Repairs for Single Hazards

There may be times when multiple, simultaneous repairs are required to overcome the impact of single hazards. The scheduler always begins by considering the space of possible single repairs along the critical path. If this set of repairs does not find a solution to the hazard, then something true of the set of repair strategies considered thus far:

1. Each single repair considered is an inadequate solution, or
2. The impact of some single repairs considered is occluded by the formation of a new critical path(s).

To know when we should consider multiple repairs, we can first consider the second case above, where the contributions of potential repairs are occluded by the creation of a new critical path. This is readily detectible by running CPM after testing a potential repair, and observing that the critical path between the repaired and un-repaired schedules has changed. (We again refer readers to APPENDIX A: Critical Path Method (CPM) for more information about CPM.) Situations where suggested repairs change the critical path, but that the improved surety is still too low, are ideal for consideration of multiple repairs. The change of the critical path is a heuristic indicating that the scheduler has made progress towards a solution.

The first case listed above is much harder. In that case, we only know that no single repair is effective, but we have no indication of what a good repair strategy (if one exists) will be. How long does it take to find an ideal repair solution in that case? Unfortunately, the answer is that it is NP-hard to find the optimal solution. We rely on the proof shown in [100] for the query plans generated by the Mariposa scheduler: "*Theorem 6: The problem of finding a global solution that optimizes a given linear function of cost and delay for a given query tree is NP-hard.*" The query plans generated by Mariposa are equivalent to the initial schedules generated by the Surety-Based Scheduler. Fortunately, also presented in [100] is a solution that computes an approximation of the optimal solution in polynomial time: "*Theorem 7: There is an algorithm that computes the ε-Pareto curve of the bicriterion query plan problem in time polynomial in the size of the input and 1/ ε.*" If we are willing to accept approximations of optimality (i.e., "good enough" schedules) in our rescheduling process, we can find acceptable repair strategies in polynomial time.

At least for our experiments presented in Chapter 6, NP-hard was not particularly problematic. First, composed programs are often related through small sets. There may be a small set of services involved in a single user program, greatly limiting the space of potential repairs. There may also be a small set of available service alternatives in the execution environment available to reschedule with. Additionally, there are often very strong constraints in the scheduling environment. Recall that we do not consider any repairs that exceed the user specified budget for time or cost. As a program moves through its execution lifetime, these constraints become more and more restrictive. The passage of time and completion of various services rapidly contracts the space of considered repairs. Also, in any execution environment, only services along the Pareto curve need to be considered for repairs. Services that lie inside the curve have alternatives that are strictly cheaper to use in terms of time and cost. For programs with very large budgets, the space of *acceptable* repairs is actually very large. Finally, NP is the *worst* case, but perhaps not the *average* case to find effective solutions. For instance, because alternative services are very likely to have distinct cost characteristics, the search

space may be navigated using a Depth-First Branch and Bound (DFBB) technique that rapidly arrives at a good solution. Of course, for large enough programs, with relatively unconstrained and undifferentiated service offerings, and large user budgets, finding *the* optimal repair strategy will not be reasonable, as it will remain an *NP-hard* problem.

For completeness, we offer that this scheduling problem is equivalent to the knapsack problem, known to be in NP. The knapsack problem is as follows: *Given a set of items, each with a cost and a value, determine the number of each item to include in a collection so that the total cost is less than some given cost and the total value is as large as possible* [100]. Items are equivalent to services, where costs and values are the {time, cost} of each service. The total cost must be less than the remaining budget, while the total value is determined by linear combination of the cost, time, and surety with the user's preference vector. The authors in [100] draw a parallel equivalence between the Mariposa schedule and the knapsack problem.

5.4.3.4 Recovery from Pseudo-Hazards

Recovery from pseudo-hazards seems to be the least examined area in grid scheduling research. One reason for this omission is that recognizing pseudo-hazards is not possible in many scheduling situations. For example, with basic job-shop schedulers, scheduling threads have complete and accurate knowledge of all participating entities. In addition to explicit resource control, they assume oracular system knowledge, and thus do not distinguish a separate class of pseudo-hazards (i.e., there is no "pseudo-" because the *actual* cause of a hazard is known). Pseudo-hazards are particularly difficult to handle ideally because we cannot, when operating as external observers, detect the difference between a pseudo-hazard and another class of hazard.

However, for certain restricted types of pseudo-hazards, we have developed a working solution. The types of pseudo-hazards that the Surety-Based Scheduler readily handles are those that mimic a catastrophic failure, and are either *regular* or *limited* in effect. By regular, we mean pseudo-hazards that are similar across the system, or simply repeat in a somewhat predictable way. An example of a very regular pseudo-hazard might be a

communications link that is only available during alternating minutes. By limited in effect, we mean pseudo-hazards that last for a short amount of time, compared to the length of the affected megaprogram. An example of a limited pseudo-hazard might be a communications link that fails for five minutes during a service expected to run for two hours.

The key to working around regular and/or limited pseudo-hazards is to:

1. Avoid underestimating the expected length of the hazard, and
2. Favor recent historical data when determining the likelihood that a hazard is a pseudo-hazard (rather than a catastrophic hazard).

If the Surety-Based Scheduler can be informed about pseudo-hazards, and the information conforms to the two properties above, it can reasonably choose the "do nothing" repair strategy for dealing with pseudo-hazards with little added risk that the program execution will not finish within budget.

5.4.3.5 A Recognition Technique for Pseudo-Hazards

We can use the following method to estimate how a pseudo-hazard is *likely* to behave, and use that information to decide on the necessity of immediate action. First, we decide on a *history* value, an integer that represents the inverse rate of decay of the values characterizing our pseudo-hazard information. For example, we can choose:

$$n = 10$$

The value n=10 implies that each new characterization of a pseudo-hazard will update our information by about 10% ($1/n$), and that we will retain 90% of ($(n-1)/n$) of our historical information. In addition to knowing this history value, we maintain an approximate sum, Σ, of the durations of the last n observed pseudo-hazards. After initially observing n pseudo-hazards lasting for {3, 3, 5, 1, 5, 4, 4, 3, 2, 4} minutes, we know:

$$\Sigma = 34$$

With Σ, we calculate the mean, μ, of the observed hazard durations as follows:

$$\mu = \Sigma/n = 34/10 = 3.4$$

During these calculations, we maintain another value, the sum of the squares of the observed pseudo-hazard times. We label this value χ. For the values observed so far, we have:

$$\chi = 130$$

With these values, n, Σ, μ, and χ, we can calculate an estimate of the standard deviation of the recently observed pseudo-hazards. We favor our technique over the standard calculation of standard deviation because our calculation always provides an overestimate of the actual standard deviation (unlike other techniques which provide an error bound, resulting in a slight over- *or* underestimate):

$$\sigma = \sqrt{\chi / h - (\Sigma / h)^2} + \frac{\left(\sqrt{\chi / h - (\Sigma / h)^2}\right)^2}{\mu}$$

The resulting σ value for the observed data set is σ=1.62, somewhat larger than the expected value of 1.26 when using the standard equation. The statistically-minded reader will notice that the above equation biases the standard deviation calculation by a factor proportional to the coefficient of variation. Recall that the coefficient of variation measures the spread of a set of data as a proportion of its mean [101]. It is often expressed as a percentage. Thus far, these equations have yielded exact results for all observed n data points; we have not favored more recent data over historical data. To bias the calculation to get more temporally meaningful results, we introduce sliding window calculations, with a decay rate proportional to n, to find successive values for Σ and χ.

After observing the duration, l, of a recent pseudo-hazard, we calculate the successive Σ value, Σ', using the following formula:

$$\Sigma' = \frac{n-1}{n}\Sigma + l$$

We calculate successive values of χ as follows:

$$\chi' = \frac{h-1}{h}\chi + l^2$$

For example, after observing a new pseudo-hazard of length 2, we calculate the new value Σ=32.6. We also calculate a new χ, determining that χ=121. With this new information, we can also calculate a new σ=1.67.

How do these numbers inform the scheduler? First, they provide an *accurate*, weighted value, μ, for the mean expected duration of a pseudo-hazard. In this regard, accuracy is important. It also provides a potentially *inaccurate* value for the standard deviation, σ, of the expected duration of the pseudo-hazard. What is novel about the *inaccuracy* of our standard deviation calculation is that it is not bounded by some error value; rather it always provides an *overestimate* of the standard deviation. This property of overestimation, rather than an error bound, is important. An error bound implies that the true value of the standard deviation is potentially above or below the actual value. Unfortunately, the scheduler can be led astray by one side of the error range, where the value is an underestimate of the expected pseudo-hazard duration. In this regard, overestimating the duration of a pseudo-hazard is a better policy. Underestimation of the expected duration will yield repairs that occur "too early" to be optimal, actively repairing a pseudo-hazard before its true nature is revealed. Overestimation of the expected duration, on the other hand, will only effect a "do nothing" approach from the scheduler when there is the availability of an alternative repair strategy, should the expected pseudo-hazard later be determined to be a catastrophic hazard.

We have also learned to favor a service duplication strategy over termination and service replacement when we know that a pseudo-hazard is likely. By allowing the scheduler to speculate that an observed hazard is potentially a pseudo-hazard, we can often preserve

valuable budget resources to yield more cost-effective schedules, or to use this budget to recover from hazards experienced later in the program execution. If pseudo-hazards are neither regular nor limited in effect, there is no apparent ideal scheduling strategy.

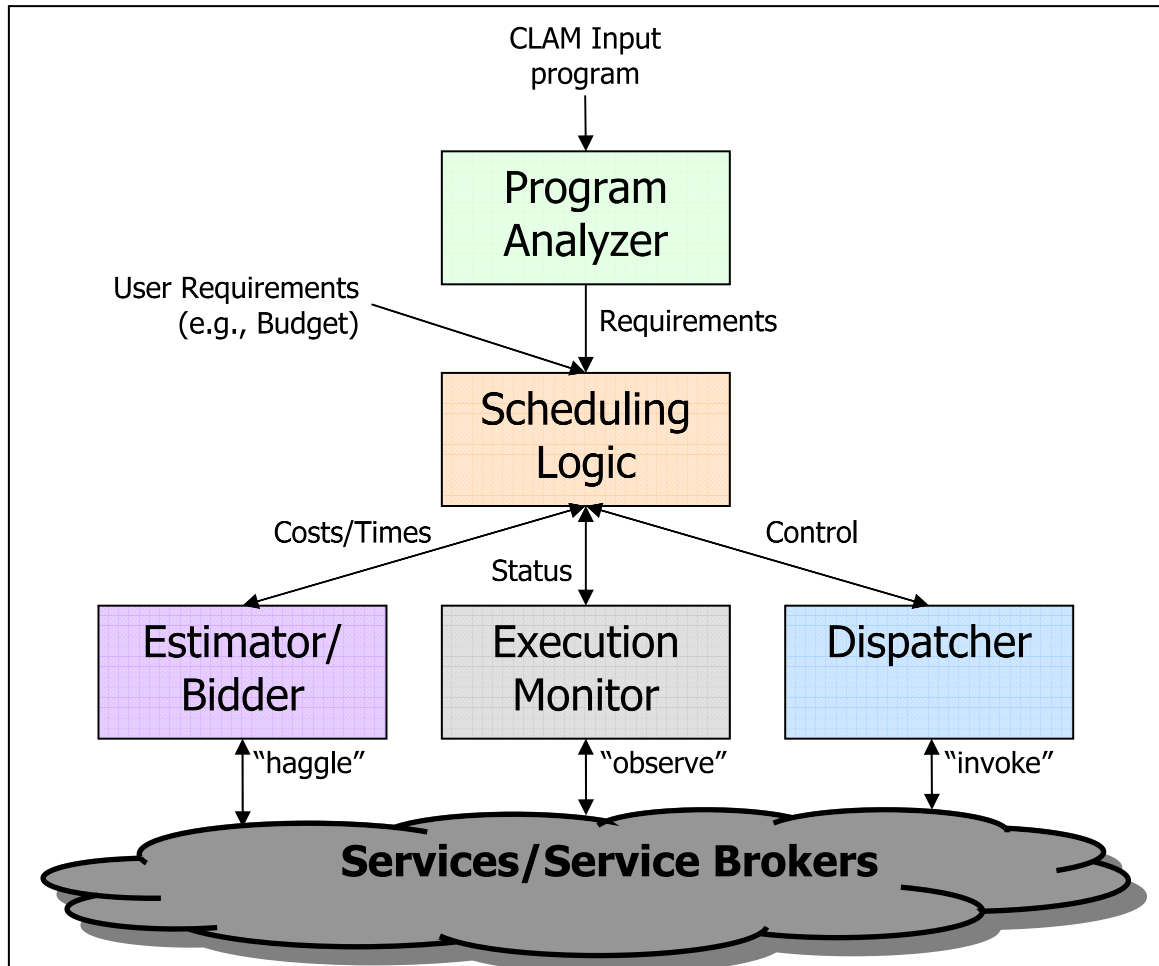


Figure 23 Surety-Based Scheduler architecture

5.4.4 Scheduling Observations

The Surety-Based Scheduler generates Pareto optimal initial schedules and selects a schedule based on the client-specified criteria. However, this does not address the central performance question: *how effective is the scheduler at working around delays and hazards?* Our evaluation strategy is to simulate network conditions and service providers and allow the scheduler to make its best attempt at scheduling a set of randomly

generated programs. Various hazards plague the scheduler during simulation, and we compare various combinations of hazards, repair strategies, execution environments, and scheduling parameters. Using dynamic programming, we also simulated each schedule and compared the ideal overall costs and completion times to our Surety-Based Scheduler's performance.

Some results are difficult to interpret. For instance, what happens when executing a program for which its last required service becomes permanently unavailable shortly after its execution begins? It turns out that the "ideal" schedule is counter-intuitive and would not be selected by any rational scheduler. For example, an "ideal" initial schedule (one that minimizes the overall expenditure) would utilize only very long running, inexpensive services, and would be required to fail because it exceeds its time budget long *before reaching the final service* that has gone offline. In similar cases, a schedule that has no chance to finish on time will waste fewer client resources during a futile attempt to solve the problem. Odd edge cases like those shows a real flaw in straightforward quantitative analysis: if a schedule cannot be completed, the *worst* initial schedules are rewarded. In this particular case, a null scheduler that constantly returns "failure to find any satisfying schedules" would perform best. We discuss the detailed experimental results in Experimental Results.

5.5 ADDITIONAL RELATED WORK

As described in Chapter 2, there has been significant work in the area of composition and scheduling, though the missing ingredient to move from laboratory conditions to real world systems has been *surety*. Our approach differs from previous research that operates under closed world assumptions where a) *a priori* scheduling estimates are provided by infallible oracles, or b) *a priori* scheduling estimates of cost will be valid at time=n, where n is possibly far in the future after the estimate was given. Finally, we see scheduling techniques for time and cost simultaneously, and repair strategies under the oracular estimates assumption, but we have not seen these techniques used in conjunction with surety analysis.

5.5.1 MARIPOSA

The Mariposa project developed a query planner for operations over large distributed databases. Entities negotiate with each other for different services such as queries, data transmission, and storage [43, 44]. Entities act through agents to process their requests. A key assumption is that estimates will be met, without exception. This assumption only holds if a central administrator manages all entities and the administrator ensures that each entity behaves properly. Our scheduler could provide Mariposa the intelligence to handle issues that arise when there is no resource overseer, as expected in a truly distributed environment. While Mariposa covers the multiobjective query optimization problem [100], their planner does not reschedule executions around runtime hazards. We can envision supplementing their approximation of the lowest cost initial schedule with Surety-Based Schedule repairs.

5.5.2 NOW (NETWORKS OF WORKSTATIONS)

The premise of NoW [57] is that collections of desktops working together have a much better price-performance ratio than mainframes and supercomputers of the same power. Applications considered highly suitable for NoW range from cooperative file caching to parallel computing within a network. Specific projects such as POPCORN [37] seek to take concepts of NoW and extend them to work on the entire Internet. POPCORN is providing programmers with a virtual parallel computer by utilizing processors that participate in this system. Similar to many Grid initiatives, and to our own market model of resource allocation, POPCORN is also based on the notion of a market where buyers and sellers come together and barter for resources.

POPCORN does assume that nodes ("services") will fail, and that it is easier to repeat work using backup nodes if a worker misses a deadline. Within the POPCORN system, failure decisions are binary: at the execution deadline, a node has either finished its work or missed the deadline. There is no concept progress monitoring, or of surety. Our scheduling system could better account for uncertainty in POPCORN programs by

monitoring job progress and rapidly migrating computation to alternative nodes as delays are detected, rather than waiting until the service deadline to test for completion.

5.5.3 ePERT AND EXTENSIONS

ePert extends traditional workflow management systems to include time management [29, 30]. ePert determines internal deadlines in the workflow and monitors progress at run-time. If deadlines are not met, alternate schedules are chosen. However, these alternative schedules must be *known at runtime*, and fully available during execution time. ePert extends the PERT method to include with it alternate execution paths that a process can take. These extensions allow for some level of pro-active scheduling by detecting time failures and recovering from them. Our scheduling techniques can contribute dynamic components to their work. However, closed-world assumptions are more likely to hold with workflow schedulers since there is often a strong command and control structure responsible for the workflow (e.g., workflows within a single corporation). While our Surety-Based Scheduler could be beneficial to workflow management systems, it is unlikely that the problem space is rich enough (in practice) that anything beyond obvious solutions will provide significant benefit.

5.5.4 GRID COMPUTING

Many computational grid projects are being developed simultaneously (ecogrid, DataGrid, power grid, etc.) [41, 42, 50]. Contributions to this field are coming from many different projects, each with tailored goals for grid computing. An overarching goal is to allow for resource sharing and services spread over large geographic, political, and economic distances. Projects like the European DataGrid currently focus significant attention on developing a network infrastructure that supports the rapid transport of multi-PetaByte datasets between different locations [31].

Other projects beyond CHAIMS, such as Globus, are providing tools to bridge the gap between heterogeneous grid participants [38]. Globus provides a low level toolkit to handle issues of network communication, authentication and data access. These tools can

be used to create high-level services such as intelligent schedulers that can be inserted into a computational grid. Thus, the Globus toolkit should make it possible to insert our scheduler into a grid infrastructure and provide execution surety to clients. The resulting increase in schedule dependability will extend the power and use of grid computing.

There is a significant body of work related to scheduling within grid systems. Most of the work related to the issues of surety and quality of service (QoS) has focused on networks and the flow of data [102, 103, 104]. The work focuses on optimizing high-bandwidth, dynamic network transmission. The Qualis project does traffic management at the packet level, and distinguishes among types of data flows such as "foreground media flows" and "background data flows" [103, 104]. The research also tends to consider the goal of response time and throughput on the network, rather than program completion. But there are two important features of this research that we find heartening. First, other researchers have come up with reasonable solutions for transmission of data among remote services, something we have thus far taken for granted in this work. (While we show a process for optimizing *logical* data flows in Chapter 9, until now, we have assumed that physical transport layers are optimal.) Second, the network QoS work depends on monitoring, and adaptation based on monitoring. Acceptance of both paradigms indicates a propensity for other grid programs to operate with the monitoring and rescheduling primitives that we present as part of CHAIMS.

Another important question that we did not directly address is the capability of service providers to provide accurate estimates. Without some ability to predict execution times with accuracy, Surety-Based Scheduling cannot be any better than a random selection of service instances. Fortunately, the problem of service provider estimation has been covered in some depth. The work in [105] deepens our ability to predict queue wait times on remote services, and provides some insight into discovery of optimal execution start times to maximize throughput, reduce latencies, etc. This is useful for increasing the accuracy of service estimations and underscores our contention that accurate estimates are possible in service provider networks.

Even more relevant to accurate service cost prediction is the work presented [106]. Local load predictions and CPU decisions are guided by heuristics and predictions about future workloads. They present algorithms for service providers to predict:

- load at future point
- average load for future interval
- variation of load over future interval

The second two predictions, “average load for future interval” and “variation of load over future interval,” are precisely the information that the Surety-Based Scheduler requires to build initial schedules, and to do pushdown repairs. Handy.

5.6 SUMMARY

We present scheduling techniques that use surety to overcome much of the uncertainty naturally present in distributed computing environments. We take a program composed of multiple distributed services and complete it within a client-specified soft deadline by guaranteeing that a minimum level of surety is maintained throughout the program execution. The scheduler described here makes initial schedules, collects information about runtime progress, and then repairs the running schedule if surety drops below a client-specified threshold value. This work is broadly applicable to systems whose distributed nature is impacted by uncertainty.

6 EXPERIMENTAL RESULTS

In this chapter, we describe the experiments that we performed to test the effectiveness of our Surety-Based Scheduler. Earlier chapters covered several substantial contributions in different areas of composition-oriented programming, but this chapter presents a more mundane, numerical validation of our particular Surety-Based Scheduling techniques.

We first describe the experimental setup, then the metrics for comparison, and finally present the results of simulations. Lastly, we analyze some variations of this particular experiment, but saving future work for Chapter 7.

6.1 EXPERIMENTAL SETUP

Because there is no robust, competitive software-as-a-service model in place today, our experiments were run in a simulation environment. We built a discrete event time simulation in Java, modeled around the basic components needed to express basic jobs and scheduling events.

6.1.1 SANDBOX DESCRIPTION

At the heart of the scheduling process, is the description of a service:

```
public class Service {
    String name;
    int    cost;
    int    timeToRun;
}
```

Service instances are identified by a simple name, and have a cost and a time to run. In our simulation, service descriptions themselves are essentially static (services do not suffer from Byzantine logic errors), and therefore they inherit their surety level from the work site on which they are executed. (The work site will also update the cost and time-to-run for a service instance during execution.)

```
public class WorkSite {
    //queues of working jobs {finished, running, suspended}
    ArraySequence finishedJobs  = new ArraySequence(10,true);
    ArraySequence runningJobs   = new ArraySequence(10,true);
    ArraySequence suspendedJobs = new ArraySequence(10,true);

    //queue of available services
    ArraySequence services      = new ArraySequence(10,true);

    String name;
}
```

Services execute on a work site. The work site has an array of available services that can be invoked, instances of which are added to the array of running jobs upon “purchase.” The work site maintains queues of running and finished jobs, but also maintains runable-but-not-executing jobs in a suspended queue. The suspended queue allows us to model hazards directly at the work site, but also allows us to install behaviors that simulate real “competition” among clients. (E.g., we can introduce a high priority job that suspends all other jobs until completion of the high priority job.)

```
public class Job {
    String name;
    double workLeft;
}
```

A “job” is just an instance of a service executing on a work site. (Obvious methods exist to access the values associated with a job, find it in the system by a unique ID, apply work to the job, etc.)

```
public class UserPreferenceVector {
    double cost, time, surety;
}

public class UserBudgetVector {
    double cost, time, surety;
}
```

User preferences and budgets (shown immediately above) are modeled with the three factors mentioned throughout this work: cost, time, and surety.

A user program (also referred to as a “schedule”) is a graph of services for execution. We present several of the non-trivial methods for schedule graph class because they are central to the scheduling process:

```
public class ScheduleGraph {
    String  name;

    Graph  schedule;
    Vertex start; //points to services with no incoming edges
    Vertex finish;//points to services with no outgoing edges


    /**
     * this is a special node, with no cost, that points to all
     * nodes that may be started simultaneously at the beginning
     * of a schedule's execution
     */
    public Vertex getStart()


    /**
     * this is a special node, with no cost, pointed to by all
     * nodes that must finish before the end a schedule's
     * execution
     */
    public Vertex getFinish()

    /**
     * Adds a service to a schedule
     */
    public Vertex insertService (Object service)

    /**
     * Adds an execution dependency between two services
     */
    public Edge insertDependency (Vertex fromService,
                                  Vertex toService )

    /**
     * initializes START and FINISH nodes for a given graph
     */
    private Vertex prepareForUse() {

        //make "start" point to anything without a parent,
        schedule.removeVertex(start);
        start = schedule.insertVertex(START_NAME);
```

```
        for (VertexIterator vi = schedule.vertices() ;
vi.hasNext() ; ) {
            Vertex v = vi.nextVertex();
            //no IN edges?, point START to this
            if (schedule.degree(v,EdgeDirection.IN) == 0 &&
!start.equals(v) && !finish.equals(v)) {
                insertDependency(start,v);
            }
        }

        //make "finish" pointed to by anything without a child
        schedule.removeVertex(finish);
        finish = schedule.insertVertex(FINISH_NAME);

        for (VertexIterator vi = schedule.vertices() ;
vi.hasNext() ; ) {
            Vertex v = vi.nextVertex();
            //no OUT edges?, make this point to FINISH
            if (schedule.degree(v,EdgeDirection.OUT) == 0 &&
!finish.equals(v)) {
                insertDependency(v,finish);
            }
        }
        return getStart();
    }

    /**
     * returns the complete schedule graph, including special
     * "start" and "finish" nodes
     */
    public Graph getSchedule()

    /**
     * returns all services waiting to be executed
     */
    public Vector getServiceInstancesToRun()

    /**
     * returns all paths that must be executed (eg, A->B->C)
     */
    public Vector getServicePathsToRun()
}
```

An instance of the ScheduleGraph class holds the description of the composed program that a user is interested in running. During execution, this same ScheduleGraph instance holds information about completed, running, and to-be-run services. Through inspection of the remaining services and their dependencies, we can determine and track program

surety during runtime. Once again, surety is determined by the algorithms presented in Chapter 5 (modifications of PERT) and in Chapter 8 (CPM). It should also be noted that the basic graph construction, traversal and management routines were not written by the author, but come from the JDSL code library, a non-commercial use library from Brown University [98].

The last piece of container code is for the execution sandbox. It is also very simple, as it contains no behavioral code:

```
public class ChaimsSim {
    String name;
    Vector workSites;
    int clock;
}
```

The simulation has a clock driving the simulation activities and a vector of work sites. The test harness, not the simulation sandbox, is finally responsible for injecting work sites, services, and adding jobs from hypothetical users in the running environment.

More active code that runs in conjunction with a user's virtual program execution include classes to "solicit" bids for services from the work sites in an active ChaimsSim instance, classes to permute and select from alternative initial or repaired schedules, and so on. The one interesting class in this space of code is the primary scheduler analyzer class, ScheduleGraphCosts. This class provides data about executing programs/schedules, including expected runtime of specific execution paths, budget expended thus far on cost and time, and other basic reliability measures. The information provided by this class can be used to drive scheduling and rescheduling decisions; it provides the required data to inform CPM and our modified PERT decisions. This class also provides a facility to calculate total user utility for an arbitrary graph of services, and is thus useful for analyzing initial schedules. The generic utility of this class comes from the fact that it can calculate cost and surety information for partially completed execution graphs as well, thus the same algorithms may be used to evaluate initial schedules, runtime

progress, and to weigh possible repairs. We present the most interesting portions of this class:

```
public class ScheduleGraphCosts {

    /**
     * returns execution cost (in $) for a graph, even for a
     * partially executed graph
     */
    static double executionCost (ScheduleGraph graph) {
        return executionCost(graph.getServiceInstancesToRun());
    }
    static double executionCost (Vector vector) {
        double cost = 0;

        for (int i = 0 ; i < vector.size() ; i++ ) {
            Vertex vertex = (Vertex) vector.get(i);
            Service service = (Service) vertex.element();
            cost += service.cost();
        }
        return cost;
    }

    /**
     * returns execution time (in h) for a graph, even for a
     * partially executed graph
     */
    static double executionTime (ScheduleGraph graph) {
        return executionTime(graph.getServicePathsToRun());
    }
    static double executionTime (Vector vector) {
        double time = 0;
        double tmpTime = 0;

        //iterate through paths
        for (int i = 0 ; i < vector.size() ; i++ ) {
            //get time for a single path
            Vector innerVector = (Vector)vector.get(i);
            tmpTime = 0;
            for (int j = 0 ; j < innerVector.size() ; j++) {
                Vertex vertex = (Vertex) innerVector.get(j);
                Service service = (Service) vertex.element();
                tmpTime += service.timeToRun();
            }
            if (tmpTime > time)
                time = tmpTime;
        }
        return time;
```

```
}

/**
 * returns execution surety (in %) for a graph, even for a
 * partially executed graph
 */
static double executionSurety (ScheduleGraph graph,
                               double timeToDeadline) {
    return executionSurety(graph.getServicePathsToRun(),
                           timeToDeadline);
}
static double executionSurety (Vector paths,
                               double timeToDeadline) {
    double execTime = executionTime(paths);

    if (execTime > timeToDeadline)
        return 0;
    else if (GLOBAL.USE_BASIC_SURETY == true)
        return (1 - Math.pow(execTime / timeToDeadline , 2));
    else
        return Policy.pertEval(paths, timeToDeadline);
}

/**
 * returns total user utility of a given graph
 * (described in section 6.4.11)
 */
static double totalUserUtility (ScheduleGraph graph,
              UserPreferenceVector preferences,
              UserBudgetVector budget) {
    double totalUtil = 0;
    double time   = executionTime(graph);
    double cost   = executionCost(graph);
    double surety = executionSurety(graph, budget.getTime());

    totalUtil += (budget.getTime() - time) / budget.getTime()
                    * preferences.getTime();
    totalUtil += (budget.getCost() - cost) / budget.getCost()
                    * preferences.getCost();
    totalUtil += (surety - budget.getSurety()) /
                    budget.getSurety() *
                    preferences.getSurety();
    return totalUtil;
}
```

```
    /**
     * check that a schedule is within budget
     */
    static  boolean scheduleWithinBudget (ScheduleGraph graph,
                                 UserBudgetVector budget) {
        double time   = executionTime(graph);
        double cost   = executionCost(graph);
        double surety = executionSurety(graph, budget.getTime());

        return (time   < budget.getTime() &&
                cost   < budget.getCost() &&
                surety > budget.getSurety());
    }
}
```

6.1.2 Test Data Description

To test our scheduler, we generated 10,000 sample programs and executed them in our sandbox with hazards externally injected into the environment. As noted in Chapter 5, sample programs are represented as directed acyclic graphs (DAGs). Vertexes in the graph are services to be executed, and edges are execution dependencies. For example, if an execution consists of graph vertex “A” that has an edge that points to vertex “B,” then the program consists of two services, “A” and “B,” and “A” must run to completion before “B” can start.

6.1.2.1 Services in the Sample Programs

The 10,000 sample programs feature 3 to 10 services each. We refer to the number of services in a sample program simply as *n*. The value of *n* is chosen with uniform probability to be in the range of 3 to 10, inclusive. In the results described in this chapter, the 10,000 sample programs had the following distribution of services:

```
 3 service programs: 1297
 4 service programs: 1250
 5 service programs: 1274
 6 service programs: 1177
 7 service programs: 1243
 8 service programs: 1257
 9 service programs: 1318
10 service programs: 1184
```

Simple addition shows that this means there were 64,782 services in play across the 10,000 sample programs, or about 6.5 services per program on average.

6.1.2.2 Service Layers in the Sample Programs

The 10,000 sample programs featured n=3 to 10 services, each connected in some fashion. One feature of the connections between services in a sample program is the depth of the service graph, where "depth" is the longest path among dependent services. We refer to the number of layers of services (depth) simply as l. The value of l was chosen with uniform probability to be from 2 to n, where n is again the number of services in the program. We did not allow for sample programs with only one layer, as they are trivial to schedule[2]. In the results described in this chapter, the 10,000 sample programs had the following distribution of layers:

```
 2 layer programs: 2342
 3 layer programs: 2223
 4 layer programs: 1686
 5 layer programs: 1304
 6 layer programs:  891
 7 layer programs:  654
 8 layer programs:  481
 9 layer programs:  294
10 layer programs:  125
```

The programs with 10 layers are perhaps the least interesting, in that they are necessarily "straight lines," with no parallel execution paths, and only single dependencies among participating services. However, they are still more interesting than 1 layer programs in that they allow us to test any of our possible repair strategies.

[2] A one layer program has no dependencies among executing services; no service depends on inputs or outputs from any other service. We did not include these graphs for two main reasons. First, these graphs are trivial to schedule, as each service is equally critical to success, and that overall completion time is simply the time of completion for the longest service. Second, without dependencies between services, several repair techniques cannot be tested, including push-down repairs. (There's no place to push a repair down to!) Finally, a one-layer graph would be a special case of gang-scheduling, and would not present any particularly interesting or unknown result.

6.1.2.3 Service Distributions in the Sample Programs

The number of services and the number of layers in a typical sample program is not enough information to describe the construction of sample programs graphs. After considering the number of services, n, in a sample program, and the number of layers, l, in that program ($l \leq n$), we must consider the specific distribution of the available services among layers.

To construct a sample program instance, we begin by distributing one service to each of the l layers in the program. This guarantees that each layer is at least partially filled. After this distribution process, the remaining services (if there are any), are assigned with a uniform probability to any of the l layers. This operation leaves the following distribution of services per layer:

```
Services on layer  1: 17696
Services on layer  2: 17585
Services on layer  3: 11313
Services on layer  4:  7344
Services on layer  5:  4695
Services on layer  6:  2890
Services on layer  7:  1736
Services on layer  8:   964
Services on layer  9:   434
Services on layer 10:   125
```

Again, by inspection, we can see that there are in fact 64,782 services in play. Some features to note in this view are that every program has at least one service in layer 1 and at least one service in layer 2. Also, only 125 services appear in layer 10, which is precisely the number of 10-layer services.

6.1.2.4 Service Dependencies in the Sample Programs

As noted in the section describing the layers of a sample program, there is always at least one linear path of dependencies in a sample program (at least one service in each layer, and each layer is connected at least to the layer above and below). We restrict our sample programs such that all services are reachable (or reach) this single dependency path in some manner. Without this restriction, we could have disjointed execution graphs.

While disjointed graphs are certainly solvable[3], their inclusion here would merely complicate the already less-than-simple description of the test cases!

Service dependencies within a program are established in a simple way. First, as noted in a previous section, there is a linear dependency path composed of at least one service at each layer. Additional services, if the number of services is greater than the number of layers in a program, are placed with uniform probability at any layer. These additional services then participate by connecting to a "higher" or "lower" layer (or both), if such a "higher" or "lower" layer exists. We fix the connection probabilities as follows:

```
P(connect to higher layer)           = 0.25
P(connect to lower layer)            = 0.25
P(connect to higher AND lower layer) = 0.50
```

If there is no higher layer, then we only connect to a lower layer, and vice versa. (Recall that there must be *at least* one or the other of higher/lower, as all sample programs have at least 2 layers.) Connections from a service not on the initially generated dependency path to another layer are assigned to a uniformly chosen random service in the destination layer. The destination layer itself is also chosen randomly from the set of all layers higher (or lower) than the connecting layer.

6.1.2.5 Hazards in the Sample Programs

Each sample program will, with equal likelihood, suffer from 1 to 3 hazards during an execution (thus the average number of hazards encountered is 2). There are three types of hazards that we inject: progressive, catastrophic, and pseudo-hazards. They are chosen as follows:

```
P(progressive hazard)  = 0.4
P(catastrophic hazard) = 0.4
P(pseudo hazard)       = 0.2
```

---

[3] In fact, the surety-based scheduler draws no distinction between disjointed graphs and connected graphs. Disjointed graphs have the same single virtual "start" node that points to all services that can start in parallel, and the same single virtual "finish" node that indicates a program has completed when *all services* have finished.

6.1.2.6 Service and Repair Considerations

In these experiments, we have two additional assumptions about the execution environment. First, we assume fixed service availability: services available at the start of program execution are available when the program needs to be repaired. We do not see substantial explanatory value in considering schedules which fail simply because there is no possible path to completion. The second environmental assumption is similar, but broader in scope: only repairable schedules are considered. For this experiment, a repairable schedule may not actually be repairable by the Surety-Based Scheduler, but must be repairable (within budget constraints) by *some ideal scheduler*.

6.2 METRICS

There are two essential comparisons when judging the impact of the Surety-Based Scheduler. First, we present several alternatives to compare the Surety-Based Scheduler to. Second, we present criteria for judging effectiveness, beyond a binary notion of sample program success or failure.

6.2.1 COMPARISONS TO ALTERNATIVE SCHEDULERS

The baseline for comparing a dynamic scheduling and repair system like the Surety-Based Scheduler is provided by a static scheduler. A static scheduler does its best to generate an initial schedule, but does not perform schedule repairs when faced with hazards during execution. Several fully static schedulers without repair have been discussed in earlier chapters (including Mariposa [43]), as have schedulers that prepare alternative execution paths during initial schedule generation (like ePert [29, 30]). For our baseline, we will assume a static scheduler similar to Mariposa, rather than ePert. The ePert option makes sense in a static environment where all alternatives are known *before execution* and remain valid *throughout the execution*, but those assumptions violate the assumed dynamism of a truly distributed, autonomous service network.

Of course, we hope to perform better than the baseline provided by the static scheduler. We establish an upper limit on expected performance by comparison to an *ideal*

scheduler. The ideal scheduler we construct sets our expectation for performance in terms of runtime, and also sets an upper limit on cost. We limit our Surety-Based Scheduler to spending no more than 10% more than the ideal scheduler on any given program schedule. Even with this restriction in place, our scheduler may pick alternative schedules that are cheaper, but run longer, or vice versa. This 10% ceiling is somewhat artificial, but it seems reasonable based on early experimentation. We will address the issue in more depth in section 6.4, but it turns out that the Surety-Based Scheduler does artificially well by restricting its budget to *no more than* the ideal scheduler, and thus we do not use this restriction here. (To assuage some cognitive dissonance from the statement that it can be easier to schedule with *fewer* available resources, here is the intuition: if we restrict available funds to *exactly* the amount used by the ideal scheduler, we are often restricted in choice to exactly the solution that the ideal scheduler chose!)

With a lower bound (static scheduler) and an upper bound (ideal scheduler) on performance, we would hope to fall somewhere in the middle with our scheduling techniques. Performing worse than the static scheduler would indicate complete failure, of course. We can further break down contributions to performance by testing our repair strategies (do nothing, service replacement, service duplication, and pushdown repair) in isolation against their full combination. Because consistently applying the "do nothing" repair strategy is equivalent to executing a static schedule, we do not differentiate these comparators. Scheduling alternatives to compare are:

1. Static scheduling (also equivalent to "do nothing" repairs)
2. Ideal scheduling
3. Only repair by service replacement
4. Only repair by service duplication
5. Only repair by pushdown
6. Full Surety-Based Scheduling

The most basic comparison among these 6 alternatives is to count successful schedule completions. How many programs finished within budget for each alternative? In the next section, we present additional metrics.

6.2.2 Judgments about Program Execution Cost

Simple successful program completion rates are one way to judge success; the more schedules completed within budget, the "better" the scheduler. However, completed schedules may also be judged by other qualities. For instance, some schedulers may have a tendency to complete schedules in a longer average time, but at a lower average cost. Because of these possible variations, we will also judge completed schedules by their relative cost and time to completion.

6.2.2.1 Accounting for Time

There is some inherent difficulty in comparing times across program executions. For instance, if one scheduler does not discover any a solution that finishes a program on time, how much time did that scheduler actually use? Is it correct to account for just time used for successfully completed schedules and to disregard incomplete executions? (For instance, this type of accounting could make a poor scheduler, one that only completes 10% of the programs it attempts within budget, look like it has a very low completion time and low cost because it never completes schedules that require repairs.) To counter this accounting problem, we score three separate values for execution time:

- Execution time for successful schedules,
- execution time for unsuccessful schedules, and
- execution time for *all* schedules, combined.

By presenting all three values, the reader can judge the effectiveness of a scheduler by any standard of time consumption. When accounting for the execution in time of a failed schedule, we cap that time at the maximum budgeted time; there is no notion of an additional penalty assessed against schedulers that do successfully complete a program.

6.2.2.2 Accounting for Cost

Unlike with time accounting, cost accounting is relatively straight-forward. Once currency is spent for a service, it is gone! That cost is real, and should be accounted for

even if an execution is not successful. However, in the spirit of completeness, we will present the breakdown of cost as well:

- Execution cost for successful schedules,
- execution cost for failed schedules, and
- execution cost for *all* schedules,.

6.2.2.3 User Preferences and Budgets

Because each sample program is different, we must agree on a basis for comparison between individual sample programs. To this end, we normalize both base cost and base execution time for each sample program, and allow a budget of time and cost that are also normalized. We represent the scaled cost of a sample program as $1000 (or ¥, or £, etc.), and the scaled runtime as 1000 hours (or minutes, or ticks, etc.).

We set the budgeted execution costs for each sample program dynamically, to some extent based on the difficulty of the repair: we set the limit on the time and cost of execution at *10% more than is required for an ideal scheduler to solve the problem.* (Again, this question of constraint is discussed further in the analysis section.)

Finally, we fix user preferences as equally favoring cost and time of execution, and user desired surety at 95%[4].

6.3 RESULTS

Consistent with the manner described above, 10,000 sample programs were generated, consisting of a total of 64,782 total services to execute. These sample programs and simulated service instances were each run to completion 6 times, within identical

---

[4] Alternatively, we can think of this 95% confidence value corresponding to about 1.65 standard deviations to one side of a normal distribution with *mean*=0 and *stdev*=1. We consider just the "late" tail of a schedule when determining that 95%~=1.65 standard deviations (services that finish early are not penalized in our model). A *z*-table can be used to find this value, or it can be found with numerous calculators, such as [115].

environments. They were run once each for the 6 scheduler options: static, push-down, replacement, duplication, surety-based, and ideal. This led to a total of 60,000 sample program executions (consisting of 388,692 service instances).

Each sample program execution forms a data point for analysis. The data points look like those shown in Table 4. Each data point describes the scheduler that was tested, the number of the sample program tested, the overall cost and time of the attempted execution, and finally the outcome of the run. The outcome consists of three parts, whether the scheduler could find *any* acceptable solution ("schedule found"), whether the execution would cost more than the available budget ("within cost budget"), and whether the execution finally took more time than budgeted ("within time budget"). Overall success is the logical AND of those three states: *finding* a solution, finding a solution within the cost budget, and finding a solution that completes in the available time.

We can take a closer look at each line in Table 4. The first line describes how the duplication-strategy scheduler performs on sample program #0. We see that the duplication-strategy finds at least one possible schedule that can complete within the allowable time, but that it cannot complete with the available funds. (Recall from earlier chapters that this is often the case when restricted to the duplication-strategy for handling service hazards; we achieve high levels of surety, but only by invoking duplicate services at substantial cost.) The second line describes how the ideal scheduler deals with sample program #0. It completes execution for $1061 in 1263 units of time. Because it is an *ideal* scheduler, each of the success test values will always be "true." The third line describes the outcome of the Surety-Based Scheduler (SBS) on the same case #0. In this instance, the SBS discovers the same solution as the ideal scheduler, and thus succeeds. The fourth line describes the static scheduler for sample program #0. The static scheduler always costs less than all alternatives, as it does not perform repairs, so it will show "true" for cost and time. In this instance, however, there was a hazard that required an active repair (a non-pseudo-hazard), thus the static scheduler did not successfully complete the program execution.

Table 4. Sample execution data points

Scheduler	Sample#	Execution Cost	Execution Time	Schedule found	Within cost budget	Within time budget	Overall success
duplicate	0	1167	1287	*true*	*false*	*true*	false
Ideal	0	1061	1263	*true*	*true*	*true*	true
SBS	0	1061	1263	*true*	*true*	*true*	true
Static	0	957	1149	*false*	*true*	*true*	false
…	**…**	**…**	**…**	**…**	**…**	**…**	**…**

These 60,000 data points are used to perform various types of analyses which we describe in balance of this section.

6.3.1 Scheduler Success Rates

The most basic comparison among the scheduling alternatives is what proportion of sample programs execute successfully (within budget). This gives the best first-approximation of the effectiveness of a scheduler: before considering the *quality* of a completed schedule (relative cost and time), a scheduler should have a high rate of successful schedule completion!

Table 5. Overall completion rates

	Static	PushDown	Duplicate	Replacement	SBS	Ideal
overall success	812	2555	4449	5821	9263	10000
overall fail	9188	7445	5551	4179	737	0

The data presented in Table 5 is also presented as a bar chart in Figure 24. The ideal scheduler successfully completes all 10,000 sample programs. The static scheduler is a poor performer, completing only 812 cases, or about 8%. This result may not seem to follow, given that P(pseudo hazard)=0.2, and that the static scheduler should be successful when faced with pseudo-hazards. Recall, however, that each program execution experiences 1 to 3 hazards, and that multiple hazards are not necessarily the same type. As such, we should expect the static scheduler to solve for the cases when

there is a single hazard *and* it is a pseudo-hazard, plus the occasions when there are two hazards and *both* are pseudo-hazards, and for cases when there are three hazards and *all three* are pseudo-hazards:

$$((0.33)*(0.2) + (0.33)*(0.2)^2 + (0.33)*(0.2)^3) *10000 \text{ cases} = 818 \text{ cases}$$

So, the performance of the static scheduler is about what we would expect for no repairs.

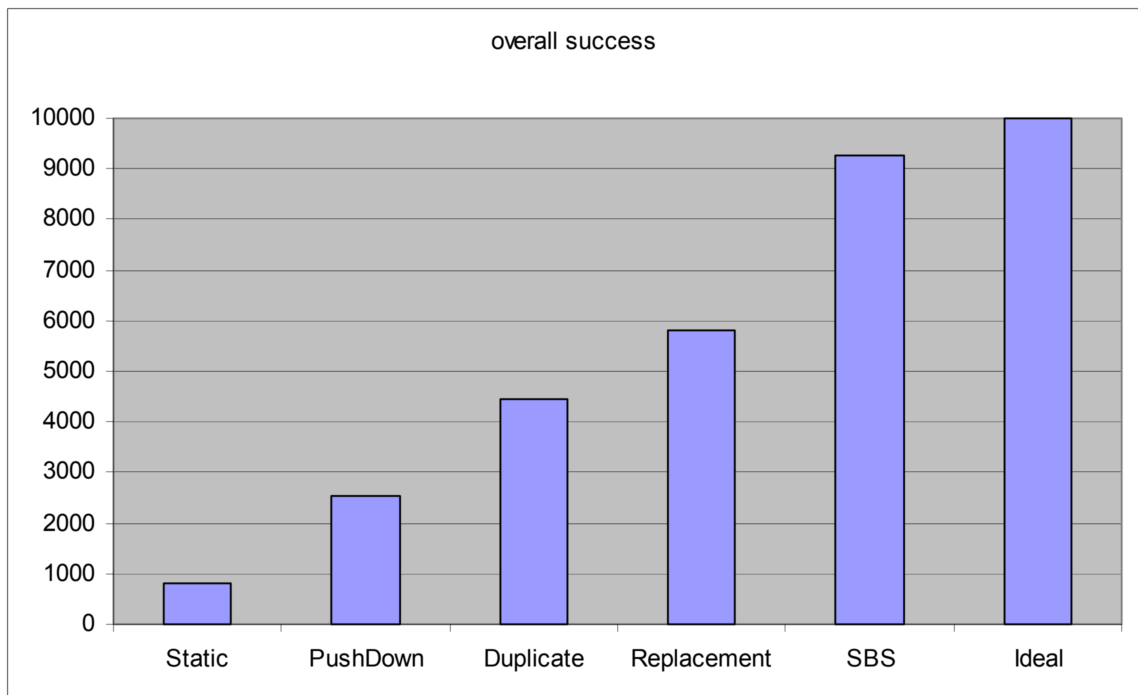


Figure 24 Sample programs successfully executed

The difference between the remaining options is fairly dramatic, with the SBS a clear winner over the more limited alternatives, but not quite as effective as the ideal scheduler. The differences in performance between the schedulers are statistically significant according to independent matched-pair *t* tests performed among all possible pairs of results [113]. (Rather than promote the discussion to a full appendix, we explain our specific *t* test parameters in this chapter, in section 6.3.4.) Because no rate of success between two schedulers is remotely similar, and all *t* tests indicate that the null hypothesis ($\mu_a = \mu_b$) is false, thus there is no need to analyze the result here. We take a

closer look at the *t* test results when considering just cost and/or time because interesting similarities do emerge from that data.

6.3.2 COMPARING SCHEDULE COSTS

There are three cost numbers to consider when comparing these scheduling strategies: average schedule cost, the cost of a successful schedule, and the cost of a failed schedule. It is interesting to note that the distinct scheduling strategies have very different behavior from each other, and that some cost more when successful than in failure, and that the reverse is true for other schedulers.

The data presented in Table 6 shows a few interesting pieces of information. One observation is that the average cost, cost of success, and cost of failure for the static scheduler are fairly consistent. This is because the scheduler does no repairs, and thus incurs no expense other that the initial cost of services. Clearly no other scheduler that engages in any *rescheduling* can compete with this floor value.

The average costs are not particularly interesting, in that they are *not* a simple average of "cost success" and "cost failure." They are weighted by the proportion of successes and

Note on Cost Averaging - Notice that pushdown strategy has a lower "cost average" than SBS, but that SBS costs less than pushdown in both "cost success" and "cost failure"! When the author first encountered this condition, it was quite confusing and difficult to rationalize. After a bit of analysis, the reasoning became clear. A short example may make it clearer for the reader, as it did for the author. *EXAMPLE: Scheduler A has the following results: {(fail, cost 4),(fail, cost 4),(succeed, cost 10)}; Scheduler B has the following results: {(fail, cost 3),(succeed, cost 9),(succeed, cost 9)}*. In this example, Scheduler A has an overall "average" cost of 6, an average cost of success of 10, and an average cost of failure of 4. Scheduler B has an overall "average" cost of 7, an average cost of success of 9, and an average cost of failure of 3. We have constructed a simple example where the overall average cost for a scheduler is higher than the overall average cost for a competing scheduler, despite the fact that both the cost of success and the cost of failure are each lower.

failures, and perhaps quite likely to lead readers astray (see *Note on Cost Averaging*). In examining the remaining 5 columns (pushdown, duplicate, replacement, SBS, ideal), we see that the SBS and ideal schedulers have the lowest costs for successful schedules, and that the SBS has the lowest average cost for failed schedules (by definition, the ideal

scheduler always finds a solution and thus never "fails"). For consistency, we present these average costs again in a bar chart (Figure 25)[5].

Table 6. Cost of generated schedules

	Static	PushDown	Duplicate	Replacement	SBS	Ideal
cost average	1000.7	1146.4	1243.8	1224.5	1158.2	1162.3
cost success	997.9	1172.6	1233.5	1221.6	1160.4	1162.3
cost failure	1001.0	1137.4	1252.1	1228.5	1130.0	-

This leaves the question of statistical significance. Are these lowest values for execution cost essentially the same? Does the SBS actually build schedules that cost no more than the ideal schedule?

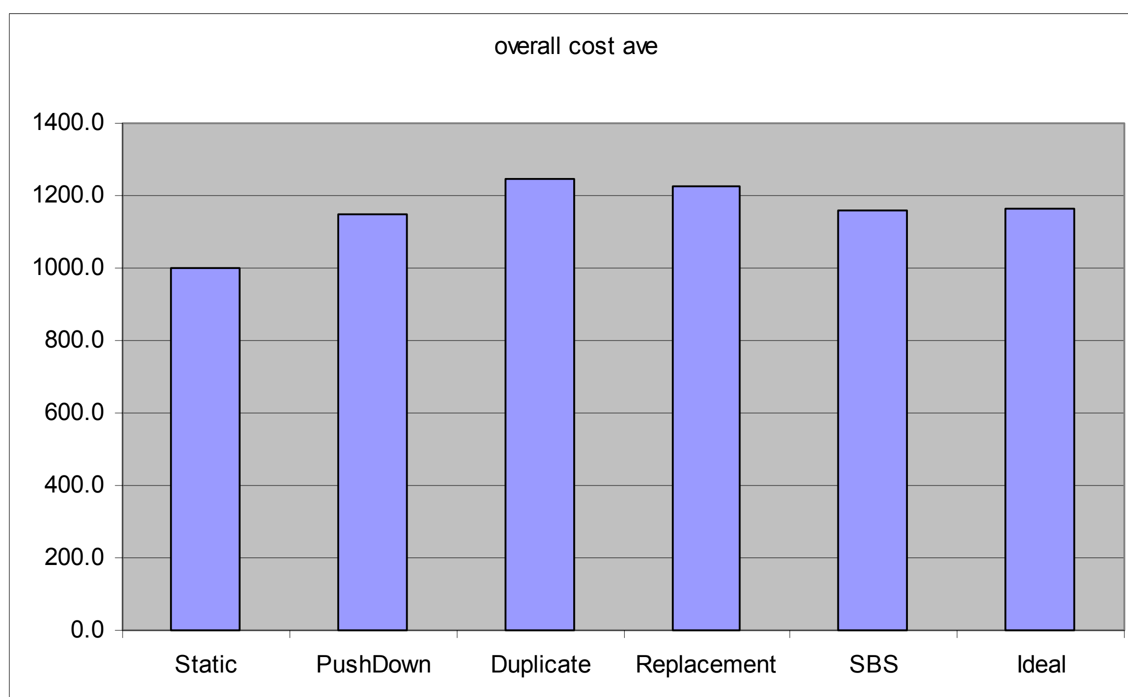


Figure 25 Overall cost average

[5] We again purposefully omit units of measure for the y-axis. Values are just a normalized number for purpose of comparison, and do not correspond to any real cost unit (e.g., dollars, pounds, yen).

To answer the question of cost similarity, we turn again to Student's matched-pair *t* test and compare the costs of each alternative scheduler to the cost of the ideal scheduler. We favor the matched-pair test because it most closely resembles our experiment. In this comparison, we have a set of sample program, each run by two schedulers (ideal and one other), and we test to see if the means of the result set are equivalent. We perform this test for the overall average cost, and for the average cost of success. We cannot perform this comparison for the cost of failure, as the ideal schedule does not fail! The test results are presented in Table 7.

Notice first that in the "ttest cost average" column, all tests yield false. We reject the hypothesis that any of the schedulers has a mean cost value equal to that of the ideal scheduler. They are all significantly different, even though this statement has very little explanatory power (as noted above in *Note on Cost Averaging*). What is more interesting is that "ttest cost success" shows "true" for two different schedulers, the Surety-Based Scheduler and the static scheduler. This means that when comparing the mean cost outcome for sample programs where both the ideal scheduler and the alternative scheduler were successful, the cost of the schedules generated are equal to the cost of the ideal schedule. This is good news for the SBS; it means that successful schedules generated by the SBS cost the same as the "ideal" successful schedules. By definition, the SBS could not be better than this!

Table 7. Comparing overall average cost and successful cost vs. cost of ideal scheduler

TTEST COST AVERAGE		**TTEST COST SUCCESS**	
ideal v duplicate		*ideal v duplicate*	
0.00	FALSE	0.00	FALSE
ideal v pushdown		*ideal v pushdown*	
0.00	FALSE	0.00	FALSE
ideal v replacement		*ideal v replacement*	
0.00	FALSE	0.00	FALSE
ideal v SBS		*ideal v SBS*	
0.00	FALSE	0.48	**TRUE**
ideal v static		*ideal v static*	
0.00	FALSE	0.80	**TRUE**

But how can the SBS scheduler *and* the static scheduler both have identical mean costs for successful schedules compared to the ideal scheduler when the SBS and static schedule have very different mean cost values for successful schedules (as seen Figure 26)? The key is in the phrase "for successful schedules." *The set of successful schedules for the SBS is very different from the set of successful schedules for the static scheduler* (recall that it was 812 versus 9263 schedules completed successfully). What the *t*-test helps to surface is the fact that during the 8% of sample programs when the static scheduler is effective, it is effective because *it engaged in no repair activities at no cost*. For that same set of cases, the ideal scheduler was also successful, and was successful for doing nothing.

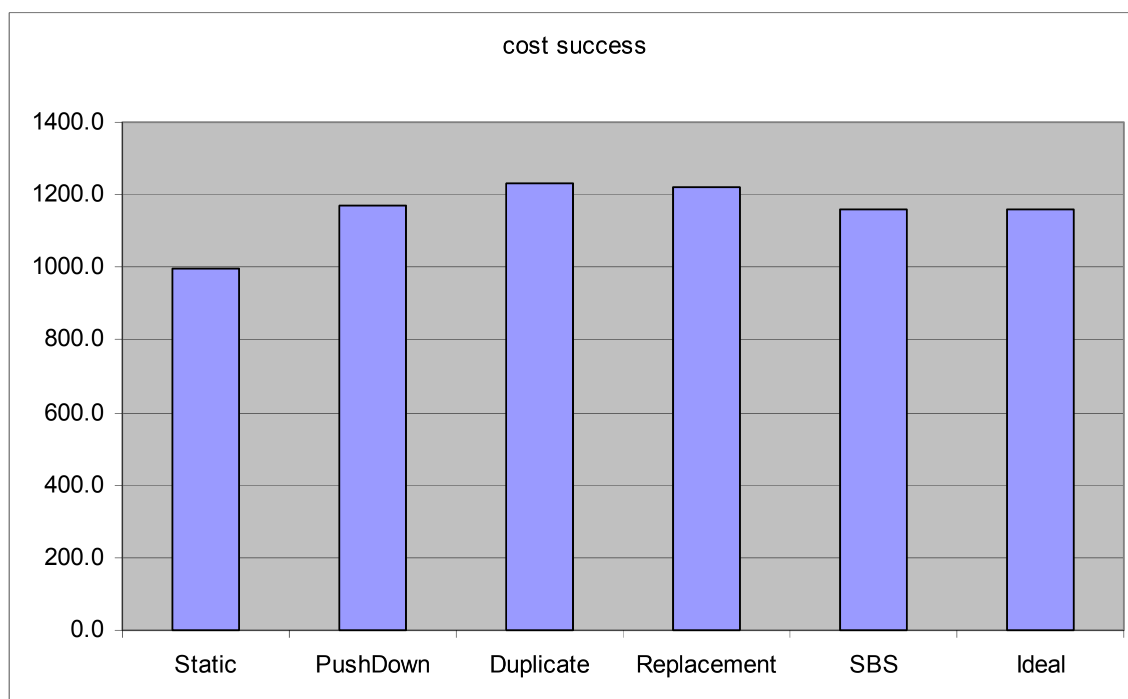


Figure 26 Relative cost of a successfully completed schedule

It is in this realization that the "cost of successful" schedules is insufficient for determining if a scheduler is effective. We have seen that the SBS and the static scheduler, *when successful*, have statistically the same cost as the ideal scheduler. However, the SBS generates successful schedules substantially more often than the static

scheduler. (There is 92.6% success rate for SBS versus the 8.1% success for the static scheduler.) Just because doing nothing works in some cases does not make it an effective general strategy!

6.3.3 Comparing Schedule Completion Times

Much like the comparison of cost among alternative schedulers, there are three time numbers to consider when comparing these scheduling strategies: average schedule time, the time to complete a successful schedule, and the time to complete a failed schedule. Once again, the distinct scheduling strategies have very different behavior from each other, and that some take more time when successful than in failure, and that the reverse is true for other schedulers.

The data presented in Table 8 shows a few interesting pieces of information. Once again, the average time, time of success, and time of failure for the static scheduler are fairly consistent. We advantage the static scheduler in allowing it to declare "failure" at the expiration of its initial schedule, rather than charging the static scheduler the fully allowable budgeted time. In a similar vein, we do *not* advantage the static scheduler by allowing it to declare failure early, and thus indicate a very short time to failure. We believe that this is a reasonable compromise, as it creates a similar distribution for time values as seen in the previous section on cost. It does not match our intuition to make the time of failure appear extremely low just because the static scheduler fails almost 92% of the time! Another interesting pattern is the time for successful schedule completion using the duplication strategy is *not* similar to the cost of success. When used, the duplication strategy costs much more, but also decreases overall time to completion. This is quite unique among the single-repair scheduling strategies.

Again, we see that the overall average times are not particularly interesting. As noted previously in the cost discussion, these values are perhaps quite likely to lead readers

astray[6]. In examining the 5 non-static columns (pushdown, duplicate, replacement, SBS, ideal), we see now that the duplication-strategy and ideal schedulers have the lowest times for successful schedules, and that these schedulers have similar average times for failed schedules (remember, the ideal scheduler always finds a solution and thus never "fails"). For consistency, we again present the overall average times in a bar chart (Figure 27)[7].

Table 8. Time of generated schedules

	Static	PushDown	Duplicate	Replacement	SBS	Ideal
time average	999.8	1145.2	1136.1	1223.5	1157.2	1161.6
time success	1001.3	1166.7	1138.0	1221.2	1158.9	1161.6
time failure	999.7	1137.8	1134.6	1226.8	1134.8	-

We now have 4 strategies that seemingly have lower average times than the ideal scheduler. How can this be the case, if the ideal schedule is truly *ideal*? (For the moment, we will defer the question of statistical significance.) The answer is that this is possible, that the average execution time can be lower that the ideal scheduler, but that the ideal scheduler still produces *better schedules*. The reason is that this advantage in lower runtimes comes first at substantially increased cost, and second, at the price of a substantially lower rate of discovering successful schedules[8].

---

[6] Again in this case, the pushdown strategy has a lower "time average" than SBS, but SBS takes less time than pushdown for both "time success" and "time failure."

[7] We again purposefully omit units of measure for the y-axis. Values are just a normalized number for purpose of comparison, and do not correspond to any real time units (e.g., hours, seconds, ticks).

[8] For example, consider the duplication-strategy scheduler. The only repair that scheduler can perform is to duplicate the service executing where a hazard was experienced. This leads to greatly increased costs in many cases, perhaps with the occasional decrease in overall runtime. When a user equally prefers cost and time, as is the case in this test, the average schedule generated will have lower utility. Also, the tradeoff made by the duplication-strategy scheduler to favor an occasional shortening of runtime is much less attractive considering that only 44% of schedules finish within budget!

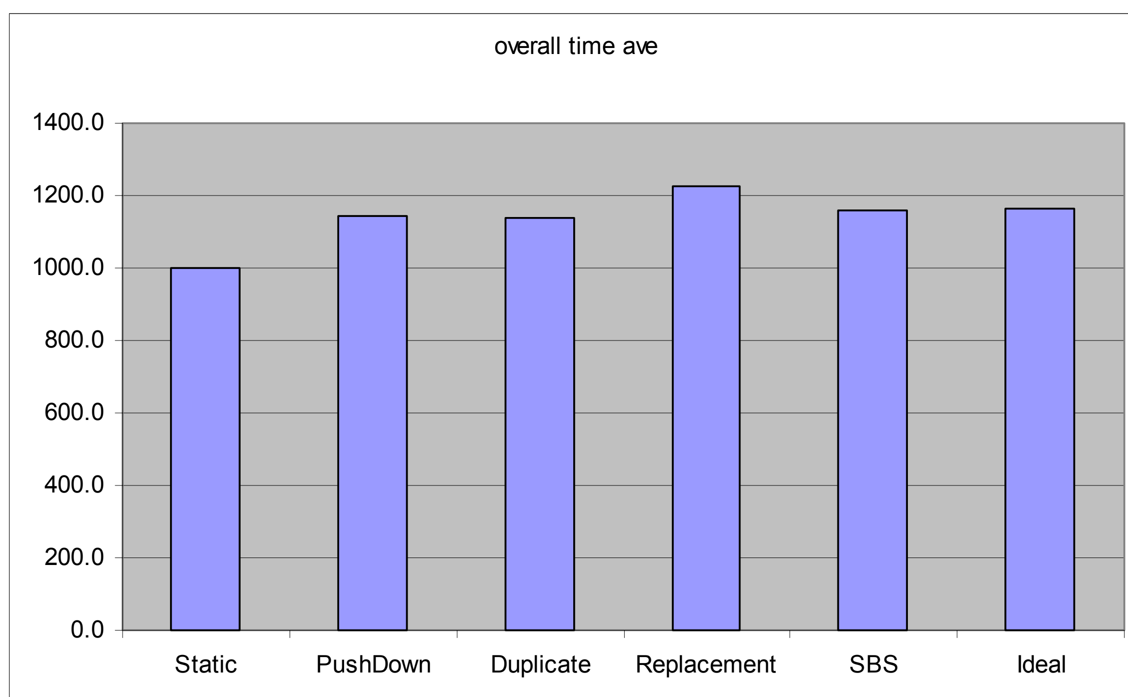


Figure 27 Overall time average

Are any of these run times similar to the runtime of the ideal scheduler? We turn again to Student's matched-pair *t* test and compare times of each of the alternative schedulers to the times of the ideal scheduler. We perform this test for the overall average time, and for the average time of successfully completed schedules. (We cannot perform this comparison for the time of failure, as the ideal schedule does not fail.) The results are presented in Table 9.

Much like the cost comparisons in the previous section, we see that for the "ttest time average," all tests yield false. We reject the hypothesis that any of the schedulers has a mean cost value equal to that of the ideal scheduler. They are all significantly different, even though this statement has very little explanatory power (as noted in *Note on Cost Averaging*). What is interesting is that "ttest time success" again shows "true" for two different schedulers, the Surety-Based Scheduler and the static scheduler. This means that when comparing the time for executing sample programs where both the ideal

scheduler and the alternative scheduler were successful, the running times of the schedules generated are equal to the time of the ideal schedule. This is good news again for the SBS; it means that successful schedules generated by the SBS have the same runtime as the "ideal" successful schedules. When this test result is coupled with the cost result, we find that *for successfully completed schedules, the SBS has equivalent costs and times to the ideal scheduler*! This is a strong result.

Table 9. Comparing overall average time and successful time vs. time of ideal scheduler

TTEST TIME AVERAGE		**TTEST TIME SUCCESS**	
ideal v duplicate		*ideal v duplicate*	
0.00	FALSE	0.00	FALSE
ideal v pushdown		*ideal v pushdown*	
0.00	FALSE	0.00	FALSE
ideal v replacement		*ideal v replacement*	
0.00	FALSE	0.00	FALSE
ideal v SBS		*ideal v SBS*	
0.00	FALSE	0.26	**TRUE**
ideal v static		*ideal v static*	
0.00	FALSE	0.25	**TRUE**

In a similar situation to cost comparisons, the SBS scheduler and the static scheduler again have identical mean times for successful schedules compared to the ideal scheduler, even though the SBS and static scheduler have radically different mean times values for successful schedules (as seen Figure 28). Our previous explanation for cost comparisons differences still holds: *the set of successful schedules for the SBS is very different from the set of successful static schedules*. The *t*-test helps to surface the fact that during the 8% of sample programs when the static scheduler is effective, it is effective because *it engaged in no repair activities that take no time*. For that same set of cases, the ideal scheduler was also successful, and was also very likely to have been successful by doing nothing.

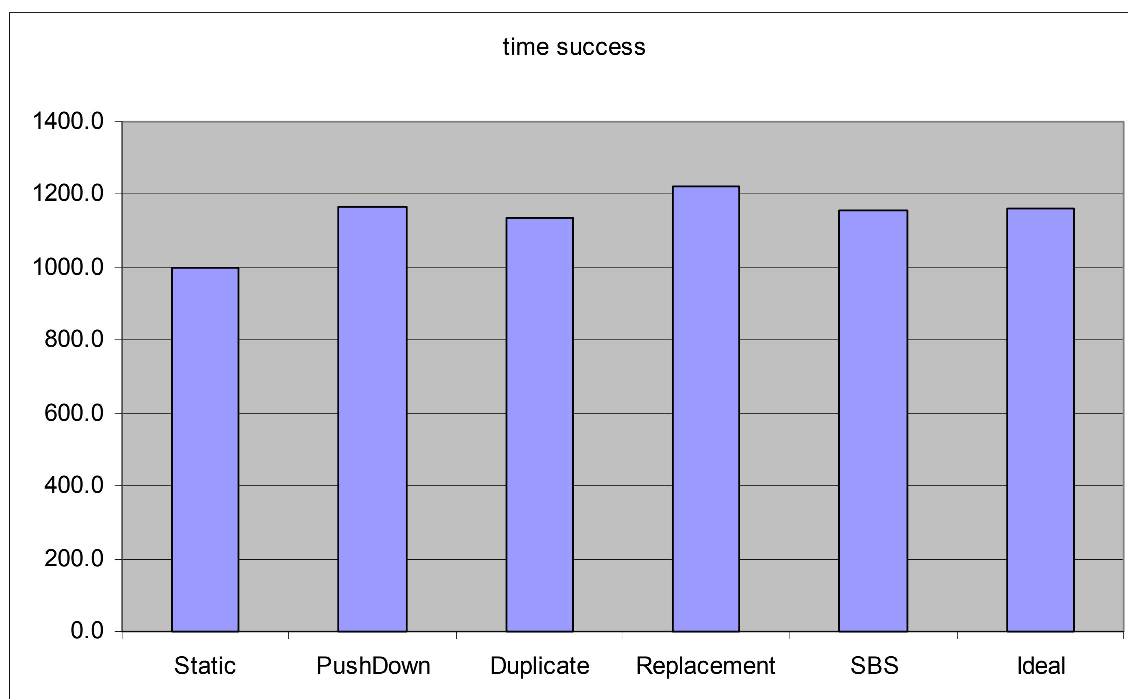


Figure 28 Relative time of a successfully completed schedule

The "times of successful" schedules are again insufficient for determining scheduler effectiveness. We have seen that the SBS and the static scheduler, *when successful*, have statistically the same runtime as the ideal scheduler, but the SBS still generates successful schedules substantially more often than the static scheduler. Just because doing nothing works in some cases does not make it an effective general strategy!

6.3.4 SAMPLE STUDENT'S MATCHED-PAIR T TEST

An in-depth explanation of Student's *t* test can be found in [113]. However, even if the reader has only a passing familiarity with the *t* test, it can be presented in short order. Also, because it is such a common statistical test, it is readily available as a basic function in most spreadsheet packages[9], and not just in dedicated statistical analysis packages like SAS and SPSS. The matched pair *t* test is used when dealing with two matched groups where the two sets of sample data contain corresponding members. In this case, there

[9] The author used the TTEST() function in Microsoft's Excel spreadsheet to validate all *t* test results in this document.

were 10,000 sample programs tested, and each scheduler considered each sample program under identical conditions. This makes a matched pair comparison between the ideal scheduler and any other scheduler the most appropriate test.

We present some sample data from the "overall average cost" comparison between the ideal scheduler and the SBS scheduler in Table 10.

Table 10. Sample *t* test data

sample program number	**overall cost for *ideal* scheduler**	**overall cost for *SBS* scheduler**	**Difference y_d**	**(Difference)^2 y_d^2**
0	1061	1052	9	81
1	1152	1155	-3	9
…	…	…	…	…
			$\sum y_d = 41791$	$\sum y_d^2 = 6665523$

The null hypothesis for this test is H_0: $\mu_d = 0$ and H_a: $\mu_d \neq 0$ in which μ_d is the population mean for the difference in cost for the two schedulers. If H_0 is true, then the schedulers have the same mean overall cost (because the mean of the *differences* in their overall costs is 0).

After finding y_d and y_d^2, we find the mean y_d:

$$\overline{y}_d = \frac{\sum y_d}{n} = \frac{41791}{10000} = 4.1791$$

We also find the variance:

$$s_d^2 = \frac{\sum y_d^2 - \left(\sum y_d\right)^2 / n}{n-1} = \frac{6665523 - (41791)^2 / 10000}{9999} = 649.15$$

The test statistic is:

$$t = \frac{\overline{y}_d - \mu_{d0}}{s_d / \sqrt{n}} = \frac{4.1791 - 0}{25.48/100} = 16.4$$

For all of our tests, we use $\alpha = 0.05$. Additionally, for all tests, the degrees of freedom (ν) is 9999 (number of samples minus 1), which is equivalent to ν = infinity for most critical value charts. Looking up the value for $t_{0.025,inf}$, we find 1.960 [113]. Since $t > 1.960$ ($16.4 > 1.96$), the test is significant and the two schedulers differ in the mean cost to execute the sample programs. Because $\overline{y}_d$ is positive, we can conclude that the SBS scheduler is less expensive on average than the ideal scheduler[10].

6.4 ANALYSIS

In the previous section, we discovered that the Surety-Based Scheduler has a significantly higher success rate than a static scheduler or schedulers featuring any single repair strategy. The SBS successfully scheduled almost 93% of the sample programs within budget, while the next best scheduling tactic could only complete about 58% of the sample programs within budget. More importantly, in assessing the SBS' performance, we find that it is statistically similar in both cost and run time to the ideal scheduler for *successfully completed schedules*. This is equivalent to saying that the SBS is indistinguishable from the ideal scheduler 93% of the time.

---

[10] We learned previously that this may be true, but it is likely meaningless information; less expensive *overall* has no correlation to performance in either successful or failed sample program executions. The real reason that the SBS seems less expensive *overall* when compared to the ideal scheduler is that the SBS quickly fails for low cost on some sample programs that the ideal scheduler spends more resources on, but successfully completes.

However, performance is not dependent only on the scheduler; performance depends on the environment that the scheduler works in. For example, given infinite time and money, *any* scheduling tactic should eventually stumble upon a successful schedule, if one exists. Alternatively, without *some* slack time and resources to purchase services to overcome hazards, almost no scheduler can be generally successful.

Until now, we have explained our performance in terms of completing sample programs within a specific fixed budget. We experimented with a set of 10,000 sample programs that are initially expected to execute in about 1000 units of time for 1000 units of cost. After throwing a series of hazards at the sample programs, we determined that an ideal scheduler could complete the programs in about 1160 units of time and for 1160 units of cost, on average. We set a budget boundary of *ideal scheduler cost plus ten percent*, or about 1275 units of time and 1275 units of cost. In this section, we investigate the impact of significantly raising or lowering the available budget.

6.4.1 Highly Constrained Budgets

Highly constrained budgets produce very different results from those presented so far. When the budget is limited to precisely the amount required by the ideal scheduler (no extra money or time), the only schedules that are successful for alternative schedulers are identical to those selected by the ideal scheduler[11]. When there is only one schedule that can possibly succeed, all schedulers suffer very poor performance. The success rates for the various schedulers when there is no extra money or time is shown in Table 11, and also as a bar chart in Figure 29.

Table 11. Sample program completion with no extra resources

	Static	**PushDown**	**Duplicate**	**Replacement**	**SBS**	**Ideal**
success	221	580	576	540	2530	10000
failure	9779	9420	9424	9460	7470	0

[11] That's a mouthful!

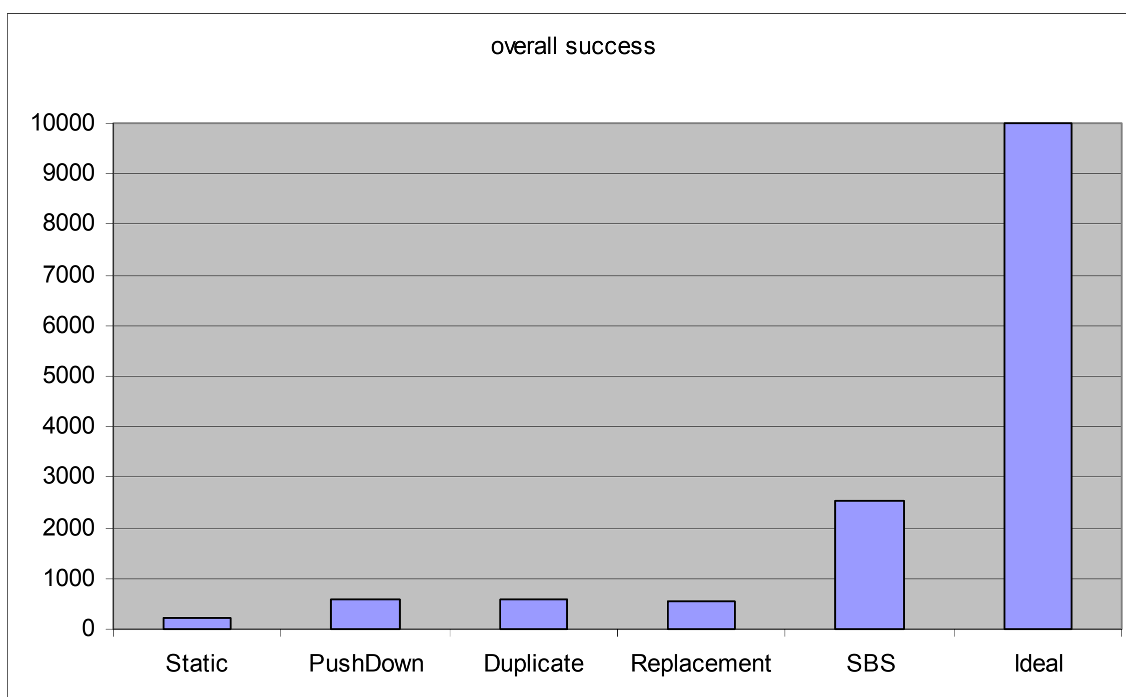


Figure 29 Sample programs successfully completed with no extra resources

Scheduling is quite difficult when there is exactly one acceptable schedule, and that schedule has been discovered only by an oracle. The SBS scheduler has a 25% success rate, about 5 times better than competitors. Perhaps the only silver lining here is that the SBS does find the single ideal schedule 25% of the time, establishing a performance floor for this type of execution environment. The 25% floor is certainly better than the naïve competition, and 12 times better than the alternative of static scheduling.

That the SBS finds the ideal solution in 25% of cases was initially quite surprising. Consider the average sample program; it experiences about two hazards. The hazard distributions are approximately likely to be as follows (bucketing them with consideration of order of appearance):

```
Pseudo-pseudo:               4%
Pseudo-catastrophic:         8%
Catastrophic-pseudo:         8%
Pseudo-progressive:          8%
Progressive-pseudo:          8%
Catastrophic-catastrophic:  16%
Catastrophic-progressive:   16%
Progressive-catastrophic:   16%
Progressive-progressive:    16%
```

Assuming that the SBS chooses solutions with a probability density precisely equal to that of the hazard distributions, we would expect the SBS to stumble upon a successful schedule less than 13% of the time. Where does the other 12% of successful schedules come from? Put another way, why does the SBS do almost twice as well as expected? The answer is within in the problem itself: *there is no extra time or money beyond that required by the ideal scheduler*. The impact of this situation may not be immediately obvious, but consider the "pseudo-pseudo" hazard pattern, when a program experiences two pseudo hazards. This occurs in 4% of cases, and we would expect the SBS to correctly generate the pseudo-pseudo solution about 4% of the time if it generates particular solutions with probabilities equal to the known distribution of hazards. Given that 0.04^2 is 0.0016, we would expect that around 0.16% of the time that we would encounter (and successfully solve) the pseudo-pseudo hazard pattern. But, if we indicate that the pseudo-pseudo hazard pattern has arrived to the SBS through budget restrictions (that this test case has *no additional money and no additional time available*), the SBS will do nothing… and solve the pseudo-pseudo hazard pattern the full 4% of the time!

In the above example, the test parameters themselves seemingly form an information conduit, leaking information from the oracle to the test subject. It does require the full spectrum of SBS scheduler capabilities to leverage this additional information, yet 25% success is still less-than-stellar performance. What overly constraining the budget tells us is nothing more than that it is difficult to reschedule around hazards when there are not enough resources to obtain meaningful repairs.

6.4.2 Loosely Constrained Budgets

There is little ground to be gained by the SBS with loosely constrained budgets, where user resources are plentiful. Given that the SBS initially started with a nearly 93% success rate, we cannot expect a dramatic improvement in performance as the budget increases. In this section, we consider the impact of increasing the available time and cost budget to 50% more than is required by the ideal scheduler. The success rates for the various schedulers when there is 50% extra money and time is shown in Table 12, and also as a bar chart in Figure 30. The static scheduler still solves the same 812 cases, and the push-down strategy, while significantly improved, is still a pretty dismal 36% success (up from about 26%).

Table 12. Sample program completion with 50% extra resources

	Static	PushDown	Duplicate	Replacement	SBS	Ideal
overall success	812	3602	9223	9191	9582	10000
overall failure	9188	6398	777	809	418	0

The first result of note is that the duplication-strategy and the replacement-strategy schedulers do substantially better, rising to about 92% success from about 45% and 58% success, respectively. They become quite similar in success rate to the SBS, though the replacement strategy still costs more and takes longer than SBS, and the duplication strategy costs *much* more than SBS, though is competitive in run time. In essence, the completion rates are much better for the two improved schedulers, but only become so by fully utilizing the newly available resources. Recall that the SBS was nearly this successful with *fewer* resources, yet it still consistently generates much *better schedules* (where "better" means "cheaper" and "faster"). The SBS scheduler and the static scheduler are still statistically identical in performance to the ideal scheduler for cost and time, for their respective successfully executed programs. As such, more completions for the alternative schedulers are not of that much value, as they come at substantially more cost than the SBS.

A second benefit accrues to SBS in environments that have a lot of extra resources, but wherein the cost of failure is high. Even if time and money are essentially free, a further distinguishing factor may appear when a particularly high cost of failure is considered. In those environments, we can consider the *failure rate* rather than the *success rate* to see that that the SBS has about half the gross failure rate of the most successful competitors. In situations where "it doesn't matter how much it costs, but it absolutely *has* to be done," the SBS remains an excellent option.

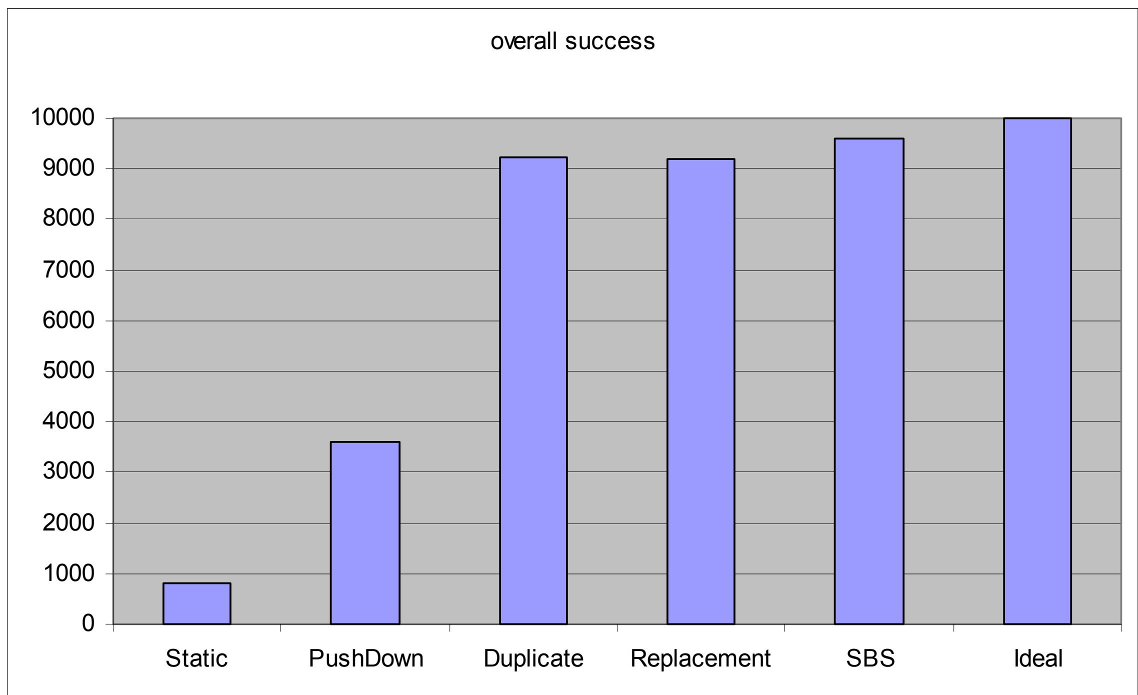


Figure 30 Sample program completion with 50% extra resources

6.5 SUMMARY

The SBS is successful more often than any alternative scheduler in normal, highly, and loosely constrained environments. While the SBS frequently finds at least *some* successful schedule, the successful schedules that it finds statistically have the same cost and execution time of ideal schedules. No other scheduling mechanism provides this level of *schedule quality*. (As noted in earlier chapters, this notion of scheduler quality is

novel to most systems.) Finally, even in resource rich environments, if the cost of failure is high, there is still reason to favor the SBS scheduler over alternatives.

7 FUTURE WORK

7.1 OPEN QUESTIONS

This project has produced tools that allow us to develop and test advanced scheduling techniques. However, we have discovered certain cases in which reasonable and rational schedulers have demonstrably poor performance compared to seemingly irrational schedulers. Some of these corner cases simply have no apparent, effective solutions. We have also identified points of tension between scheduling with global and local information. In this section, we illuminate some deeper questions about the impact of user preferences on the system, the expected grid environment in which these schedules will run, and the Surety-Based Scheduling model.

7.2 IMPACTS OF USER PREFERENCES

User preferences for {cost, time, surety} are used to select initial schedules, but user preferences are not relied upon for making repairs. When selecting repair strategies, the schedule repair process simply finds the repair strategy that gives the best surety increase per unit cost. In addition to not considering user preferences when making schedule repairs, we do not address the *economic* impact of myriad user preferences in shaping the execution environment.

7.2.1 USER PREFERENCES USED IN REPAIR SELECTION

An interesting and informative study would be to consider user preferences for time, cost, and surety when recovering from hazards. The current approach selects repairs that maximize surety while minimizing cost, resulting in the discovery of solutions to the largest number of scheduling hazards. In essence, we find the largest number of successful schedule repairs, without any consideration for the relative utility of those schedules. The consideration of user preferences in rescheduling does not find more successful repairs, but for the schedules to do complete within budget, it made find more *desirable* schedules. By introducing a notion of "repair quality" based on user

preferences, and using this measure to judge the outcome of repairs, we could perhaps better serve users on the edges: those with “deep pockets” or unlimited pools of time, at the time of repair. Such users would be very likely to find *some* repair, and with more consideration of their preferences, we could find them a more *desirable* repair.

7.2.2 Global Impact of Local Schedule Preferences

While it is more a question for economists, supply and demand can have a profound effect on the initial scheduling and schedule repair activity. A Pareto curve representing optimal service selections for a given schedule is valid at a given point in time, but what happens when other users enter the system, all demanding the same services? We would expect overall execution costs to go up, but we would not really expect the Pareto curve to change significantly. However, imagine that all the new users entering the system prefer execution time with a factor 100 times more than the importance of execution cost. In that situation, demand for very short running service instances will put price pressure on one factor (in this case, time), perhaps radically deforming the Pareto curve for other users. Price pressure in systems like this operates on both the simple demand model, where high-demand services would tend to be priced higher in the short run, but also where short-term user preferences can have a significant impact on pricing. A study of these effects would likely lead to a more informed scheduler, one capable of better utilizing execution options and service reservations.

7.3 QUESTIONING ASSUMPTIONS ABOUT THE EXECUTION ENVIRONMENT

There are issues with the execution environment and the availability of information that can have a substantial impact on scheduling. In some cases, we have made assumptions about the grid that may not hold universally. For instance, we can envision issues around service progress monitoring that could prove difficult to handle: imagine a case where the act of requesting progress information is extremely expensive, and adds an additional 10% to the overall execution time. This would make execution monitoring very

expensive, and could only occur very infrequently if a service was to ever finish execution. If the execution environment precludes using the tools that the CHAIMS system provides, meaningful scheduling and repair becomes extremely difficult.

7.3.1 Progress is a Secondary Indicator of Work Rate

One difficulty faced by the scheduler is that the progress values returned by the EXAMINE primitive are only secondary indicators of the underlying service's *work rate*. An example of the potential impact occurs when a service returns the same value for progress completed for two successive inquiries. Does seeing the same value for progress two times mean that the service's work rate has fallen to zero, and that no progress is being made? Does it mean that an earlier estimate was too optimistic, and the latest numbers reflect a correction? The EXAMINE primitive does not purport to expose anything "under the covers" at the service provider's site, and thus has to use certain assumptions to make projections about finish times. To probe those assumptions, imagine that we poll a running service 5 times, at 10 minute intervals, and get the following progress values in return:

0%, 10%, 20%, 30%, 30%

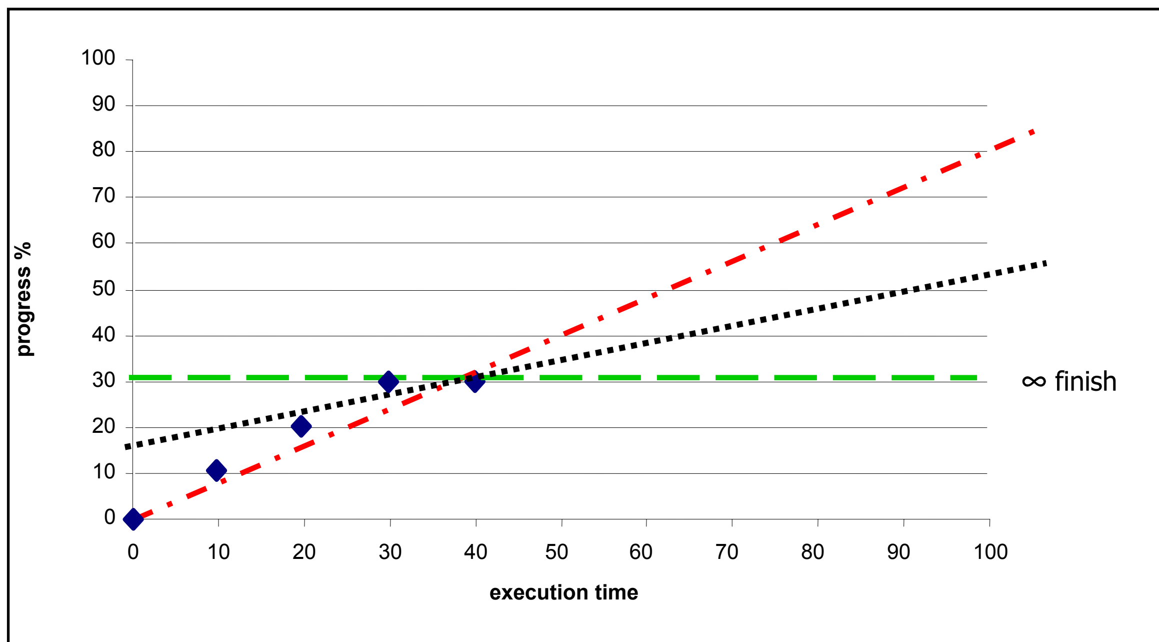


Figure 31 Progress versus finish time

At time 0, we observe 0% progress made; at 10 minutes we see 10% progress, and so on, up to the reading at 40 minutes, where we see 30% progress. (These measurement points are also represented by the diamonds in Figure 31.) How do these progress observations translate to work rate, and what do they indicate about the service's expected finishing time? If we consider just the first measurement of progress (0% at time 0 minutes), and the last (30% at time 40 minutes), we project that the finish time (100% progress) will be at time 133 minutes. If we consider just the last two measurements of progress (30% at time 30 and 40 minutes), then we will project a finish time of infinity, as the service seems to be making no progress. Currently, the Surety-Based Scheduler splits the difference between these two extremes. We cannot know if this model is informed enough to capture real behaviors until we test the scheduler on services outside the experimental sandbox.

7.3.2 Monitoring Resolution Adjustments

One level of flexibility that our scheduler does not provide is the ability to change the cost model for monitoring, during runtime. We assume that all services charge an

equivalent fee for monitoring, and have equal delay in responding to requests for progress information. In practice, there may be cases where some services cost much more than others to monitor, and thus the scheduler would have to operate with less progress information when making decisions. Similar situations can arise when some services can provide immediate feedback on progress, while other services have a significant delay between request and response. Neither of these problems is particularly difficult to handle at runtime, but lack a mechanism that accounts for monitoring idiosyncrasies when developing initial schedules. Our initial CLAM language specification and models for time and cost estimation of services did not include *estimations for the cost to monitor service executions*, and thus we were initially limited by these inflexibilities in our runtime system. For the future, it would be appropriate to build in the ability to initially favor certain services that provide rapid, low-cost progress information over those that do not.

7.3.3 Independence of Events

One of the primary assumptions of our Surety-Based Scheduler is that of service independence. We do not consider any interaction or interdependency among services or service providers (e.g., estimates and execution of one service do not affect estimates or execution of a second service.). If such interdependencies do arise (and they are detectable), strong statistical techniques, such as Bayes's Theorem, to reason about conditional probabilities do exist. It may be that conditional probability theorems do not lend themselves as well to Surety-Based Scheduling; however there is no strong evidence to support this supposition. The one complication we do foresee is computational: conditional reasoning via Bayes's Theorem is a more computationally intensive proposition than reasoning about independent actors. The reasons for this increase in computational complexity are many, and we present but one such reason here. Under the assumption of independence, we can eliminate services for which there is a competing service that has an absolute advantage in time, cost, and surety. We cannot make this obvious elimination with interdependent services because we cannot predict *a priori* if a given service's dependent counterpart will be part of any future reasoning or repair

strategy. Additionally, the PERT and CPM techniques assume event independence, though these techniques can be readily adapted. The hard problem occurs when independence of service providers is ambiguous: can we reliably predict anything when faced with hazards in such a system?

7.3.4 Gaussian or Power-Law Environment

Recent studies have shown that many complex systems, including many previously though to be Gaussian, are actually governed by power law event distributions [107]. For example, a study of recent natural disasters shows power law distributions of disaster severity, rather than Gaussian. Power law properties also seem to arise in many organically grown networks (P2P file distributions in a network, search engine query frequency distributions, etc.). Will the "fat tails" of power law curves in reliability substantially alter solution stability in recovery computing? The question itself leads us to two places:

- Knowing that systems follow power laws rather than Gaussian curves means that we can readily reason about them, but
- Because power law curves have generally higher probabilities for events at the tails, the effectiveness of *any* statistical techniques may be marginalized [107].

7.4 LIMITATIONS OF THE SURETY-BASED SCHEDULER IMPLEMENTATION

There is a set of open questions related to our implementation and approach to scheduling. For example, as mentioned in Section 5.4.4, we always begin recovering from a hazard by considering the space of single service repairs. If we find an adequate solution in this set, we utilize it. We could have chosen to compute the "ideal" repair, including consideration of multiple repairs, in all cases, but we did not. Decisions like these can impact the utility of our scheduler, and may affect its applicability in real-world environments.

7.4.1 SIMULTANEOUS RESCHEDULING

As noted above, the Surety-Based Scheduler may select non-ideal repairs that are quick to discover. It might be possible to constantly calculate possible repair strategies while the client program is not otherwise loaded, and cache the best known repair strategies *before* a hazard is encountered. In this way, the Surety-Based Schedule could build a tree of best repairs in the same manner that a chess-playing program precomputes possible responses to an opponent's pending move. In the chess problem, when an opponent's move is finally made, the chess program has at least a partial solution computed, and gets a head start on finding the best solution it can within a limited time period. Thinking of a hazard as an opponent's pending move, and thinking of known good repair strategies as possible board positions in the game tree, we can envision hiding the precomputation of hazard solutions behind client idle time. With this strategy, the Surety-Based Schedule may occasionally know the best possible repair strategy before a hazard occurs!

7.4.2 EXECUTION PENALTIES

One feature that constrains the algorithmic complexity of the Surety-Based Scheduler is the fact that programs are represented as directed, acyclic graphs with no negative weighted edges. In practice, however, we can readily imagine dealing with both semi-cyclical control structures (e.g., FOR loops) and negative edge weights. Cycles in the graph are often straight-forward to handle with well-known strategies, such as loop unrolling. The issue of negative weights in the graph is much more complex. Negative weight edges may arise when service providers miss deadlines.

Negative weights are only possible for cost, of course. Time is always moving forward, and thus accounted for by only non-negative values. Surety is represented by a point in the range 0 to 1. Interestingly, absolute cost for a service could go negative if a contract builds in severe penalties for service providers that miss deadlines. The current scheduler only considers the original cost of a service, and that some portion of the original cost may be recaptured after terminating an underperforming service, but does not consider diminishing fees due to execution contracts. A richer contract model, one that allows

cost to flow back to the client under certain circumstances, would cover more possible scheduling environments.

7.4.3 Second Order Cost Effects

As mentioned in Chapter 6, we have focused our experimental research on the issue of recovery and rescheduling in the face of hazards. The generation of initial schedules was not critically important, as it has been a well-studied problem. However, there is a consideration about the quality of initial schedules that we did not address: the value of the "left over" part of the budget power remaining after the initial scheduler is generated. What is the value of that purchasing power?

We know, intuitively, that having more slack time in the schedule, and more resources banked at the start of execution means that the schedule has a greater chance of recovery should a hazard arise. Can we use this information to generate initial schedules that are more likely to succeed, though perhaps less attractive to a user (according to their stated preferences)? We believe so. By sampling for cost estimates *during* runtime, we can essentially place a value on the time and cost reserves, knowing how much they are worth in their ability to purchase potential repairs. Surety may be better expressed as the combination of time and progress, with the addition of remaining budget divided by some current valuation drawn from the cost of possible repairs.

The preceding explanation may not be clear enough, so we ground it now in an example. Imagine that a user wants to execute a single service, "A." The user has a budget of $100, 10 hours, and wants a 95% likelihood of completion. The service instances available for execution are listed in Table 13.

Table 13. Reserving budget example

Service Provider	Time	Cost
VendorA	1 hour +/- 0 hrs	$100
VendorB	3 hours +/- 0 hrs	$80
VendorC	4 hours +/- 0 hrs	$40
VendorD	5 hours +/- 0 hrs	$50

Assume also that the user is *very* concerned about time, and that overall execution time is a far more important consideration than the cost of the service, though any execution of service A must still happen within budget. If the time and cost estimates in Table 13 are accurate, we can immediately see that VendorA is the best choice. VendorA can complete the job first (satisfying the user's desire for short execution), and within budget, and shows no variability in its estimate. VendorA must be the right choice.

But what happens if we know that the global likelihood of any single running service suffering a catastrophic failure is 10%? Which vendor is the right choice? Is it still VendorA? No. By selecting VendorA, the execution surety level falls to 90%, as any single service has a 10% chance of failing. Selecting VendorB is also incorrect: VendorB is also a 90% surety choice (compared to the user's specified surety level of 95%), and if VendorB fails, there is no chance of recovery because the budget is effectively depleted. The correct choice is VendorC. If VendorC fails, then there is still enough time and budget to repair with VendorD. The combined probability of failures by both VendorC and VendorD is acceptable at just 1%.

These considerations are fairly straightforward so far, but here is the kicker: what if we remove the worst performing vendor from the set of available vendors? If we remove

VendorD, one that is simultaneously slower *and* costs more than VendorC, is VendorC still the right choice? Absolutely not! Under that condition, if the user wishes to proceed with a compromised surety level, then the correct choice is once again VendorA for its performance characteristics. This example shows that services entering or exiting the available pool during scheduling or execution can radically alter the value of holding back budget for possible repairs. There is no obvious solution to account for this phenomenon, the variable value of budget holdbacks, though with improved knowledge of the global runtime environment, something could perhaps be done by adding another layer of statistical speculation. This is a problem best addressed outside the experimental sandbox, as stable solutions that may be found for experimental environments may have little or no bearing the behavior of real complex systems.

7.4.4 Is there a benefit for non-repairable schedules?

The Surety-Based Scheduler was designed to repair schedules so that they meet specified budget requirements. Some schedules, after encountering a hazard, cannot be recovered without violating budget constraints of cost, time, or surety. Do the repair strategies we offer have benefit for these schedules? We have the capacity to choose the *best* possible repair strategy, where *best* is determined by user preference. With metrics to measure the value of non-conforming repairs and the relative utility for a given schedule to a particular user, we could answer this question. Like measuring the quality of search engine results, the relative value of schedules that do not complete within a specified deadline is quite subjective.

8 APPENDIX A: CRITICAL PATH METHOD (CPM)

We use the Critical Path Method (CPM) to identify paths in a program that determine the expected running time of the overall program. CPM was developed in 1957 by DuPont, as a project management method designed to address the challenge of shutting down chemical plants for maintenance and then restarting the plants once the maintenance had been completed. They developed the Critical Path Method for managing the complexity of such projects [55, 96].

CPM provides the following benefits:

- Predicts the time required to complete the project.
- Shows which activities are critical to maintaining the schedule and which are not.
- Provides a graphical view of the project.

We leverage the first two benefits, without consideration of the graphical view.

CPM models the activities and events of a project as a network. Activities are depicted as nodes on the network and events that signify the beginning or ending of activities are depicted as arcs or lines between the nodes [96]. The following is an example of a CPM network diagram:

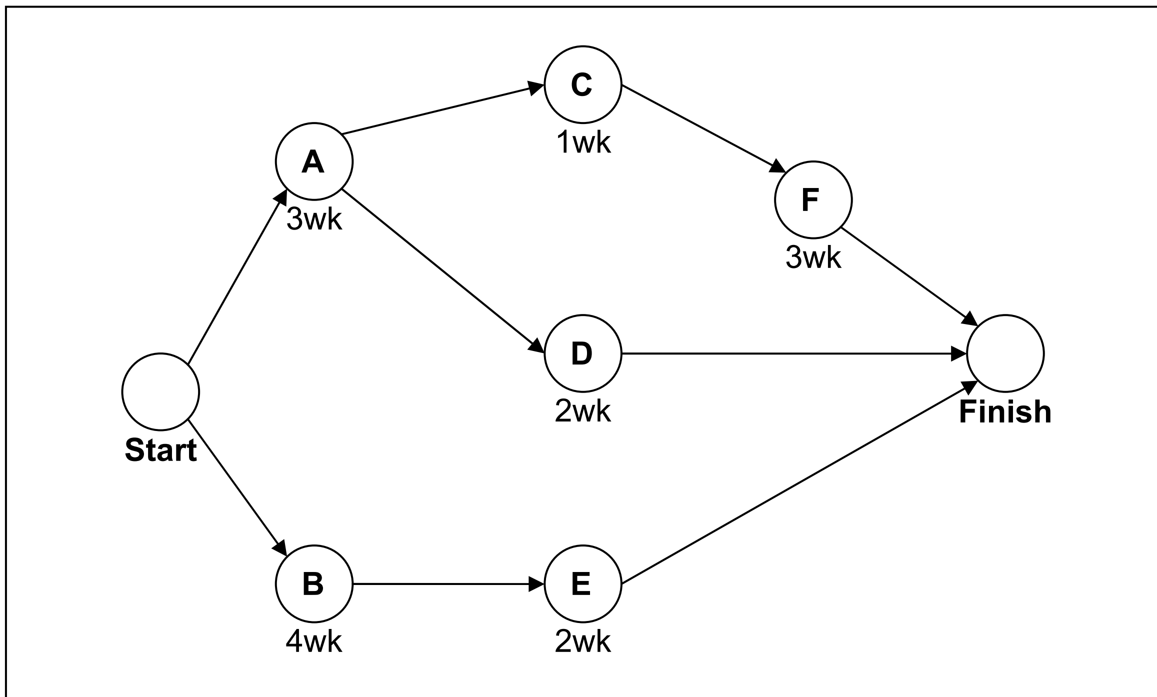


Figure 32. Sample CPM diagram

8.1 STEPS IN CPM PROJECT PLANNING

The traditional usage pattern for CPM involves the following steps.

1. Specify the individual activities.
2. Determine the sequence of those activities.
3. Draw a network diagram.
4. Estimate the completion time for each activity.
5. Identify the critical path (longest path through the network)
6. Update the CPM diagram as the project progresses.

We note our differences in each activity below. Because our scheduling task is automated, and derived from a CLAM program, some steps are less important or simply derivative from the CLAM program.

8.1.1 Specify the Individual Activities

Typically, from the work breakdown structure, a listing is made of all the activities in the project. This listing is used as the basis for adding sequence and duration information in later steps [96]. Our scheduler derives the activities list directly from the CLAM program.

8.1.2 Determine the Sequence of the Activities

Some activities depend on the completion of others. A listing of the immediate predecessors of each activity is useful for constructing the CPM network diagram [96]. Our schedule derives these dependencies from the CLAM program. In our current model, they are implicitly derived from the dataflows between services [87, 97].

8.1.3 Draw the Network Diagram

Once the activities and their sequencing have been defined, the CPM diagram can be drawn. CPM originally was developed as an *activity on node* (AON) network, but some project planners prefer to specify the activities on the arcs [96]. In the case of our scheduler, we translate the CLAM program into a Java-based graph data structure, using JDSL, a non-commercial use library from Brown University [98].

8.1.4 Estimate Activity Completion Time

In traditional CPM tasks, the time required to complete each activity can be estimated using past experience or the estimates of knowledgeable persons. CPM is a deterministic model that does not take into account variation in the completion time, so only one number is used for an activity's time estimate [96]. In our evaluation process, we gather estimated runtime information directly from service providers. We do not currently consider historical information gathered or maintained by single clients, but briefly address the issue of globally available information [33]. Also, we augment the CPM path identification process with PERT evaluation to get more accurate estimates and to do fuzzy evaluation of the execution time.

8.1.5 Identify the Critical Path

The critical path is the longest-duration path through the network. The significance of the critical path is that the activities that lie on it cannot be delayed without delaying the project. Because of its impact on the entire project, critical path analysis is an important aspect of project planning [96]. We use this information to build initial schedules and also to identify paths that are good targets for rescheduling activities to overcome runtime hazards [33].

The critical path can be identified by determining the following four parameters for each activity:

- ES - earliest start time: the earliest time at which the activity can start given that its precedent activities must be completed first.
- EF - earliest finish time, equal to the earliest start time for the activity plus the time required to complete the activity.
- LF - latest finish time: the latest time at which the activity can be completed without delaying the project.
- LS - latest start time, equal to the latest finish time minus the time required to complete the activity.

The *slack time* for an activity is the time between its earliest and latest start time, or between its earliest and latest finish time. Slack is the amount of time that an activity can be delayed past its earliest start or earliest finish without delaying the project.

The critical path is the path through the project network in which none of the activities have slack, that is, the path for which ES=LS and EF=LF for all activities in the path. A delay in the critical path delays the project. Similarly, to accelerate the project it is necessary to reduce the total time required for the activities in the critical path [96]. It is this property that allows us to quickly identify places to repair schedules faced with runtime hazards.

An interesting property of this path identification is that it is a necessary test, but not sufficient test to locate potentially interesting nodes to repair schedules. The reason is simple: repairing along one critical path may create a new, similar path in a DAG organized program! This brings up the necessity to consider multiple repairs to overcome individual hazards [33].

8.1.6 UPDATE CPM DIAGRAM

As the project progresses, the actual task completion times will be known and the network diagram can be updated to include this information. A new critical path may emerge, and structural changes may be made in the network if project requirements change [96]. We rely on this property and reevaluate the expected execution time whenever new status information becomes available. As mentioned previously, evaluating the runtime of critical paths identified by CPM is done using PERT [33].

8.2 CPM LIMITATIONS

CPM was developed for complex but fairly routine projects with minimal uncertainty in the project completion times. For less routine projects there is more uncertainty in the completion times, and this uncertainty limits the usefulness of the deterministic CPM model [96]. It is this primary limitation that prevents CPM from being independently useful as a scheduling tool when there is uncertainty involved, and the main reason we augment CPM with PERT and our own fuzzy evaluation techniques [33].

8.3 EXAMPLE CPM USAGE

To demonstrate CPM, we can consider the data displayed in the following table for a very small project. It is a small project consisting of six activities or jobs (“activities” are analogous to “services” in the CHAIMS model) [99]:

Table 14. CPM Example Job Table

Activity	Duration (days)	Immediate Predecessors
Training	6	--
Purchasing	9	--
Production	8	Training, Purchasing
Quality Control	7	Training, Purchasing
Assembling	10	Quality Control
Transporting	12	Assembling

For each activity, we must have three pieces of information: title, duration and set of immediate predecessors. The first two are obvious, so we shall examine only the third.

If an activity does not have any immediate predecessors (e.g., "Training" and "Purchasing"), then it can begin immediately. If an activity does have immediate predecessors, then it cannot begin before the completion of all its immediate predecessors. For example, "Production" cannot begin before "Training" and "Purchasing" have both been completed [99].

Recall that the mission of the critical path method is to determine how early the overall project can be completed and to identify the "critical activities," activities whose delays will cause a delay in the completion of the entire project. A dynamic programming approach to this problem is based on the following concepts:

- ET[A] := Earliest time that activity A can commence (given the precedence constraint and the duration of its predecessors)
- LT[A] := Latest time that activity A can be completed without causing a delay in the completion of the entire project.

1. TF[A] := (LT[A] - ET[A]) - t(A) ("total float" of activity A.)

The total float of an activity is then (by definition) the longest possible delay in the completion of this activity that will not cause a delay in the completion of the entire project. This suggests the following intuitive notion:

- Critical activity := An activity whose total float is equal to zero (TF[A]=0)

Thus, to identify the critical activities, we compute (as prescribed by step 1) the total floats of all the activities and identify those activities whose total floats are equal to zero. This means that we have to compute the ETs and LTs for all the activities. The following are typical dynamic programming observations. But first some notation and terminology:

- P[A] := Set of all immediate predecessors of activity A
- S[A] := Set of all immediate successors of activity A
- T := Completion time of the entire project, assuming no delays
- Initial activity := An activity with no predecessors (P[A] is empty)
- Terminal activity := An activity with no successors (S[A] is empty)

The first observation indicates an obvious state, namely it observes that the activities that do not have immediate predecessors can begin immediately, and that the last activity to be completed must be an activity with no immediate successors:

Theorem 1:

2. ET[A] = 0 , for all initial activities (i.e., P[A] is empty)
3. T = max { LT(A): A is a terminal activity (S[A] is empty)}
4. = max { ET(A) + t(A): all A }

The second observation is merely a formulation of the dynamic programming functional equations for ET and LT.

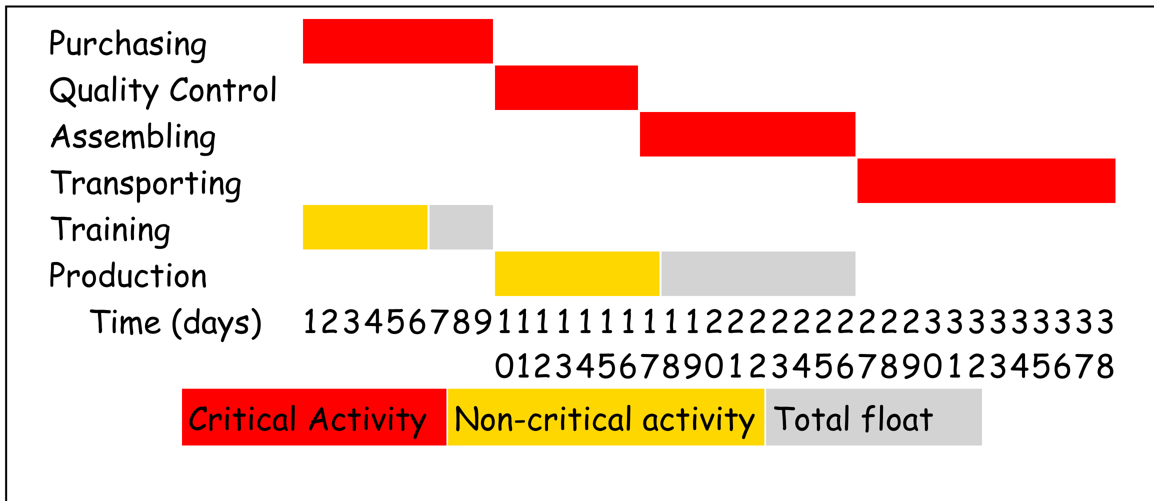
Theorem 2:

5. ET[A] = max { t(x) + ET(x): x in P[A] } , if P[A] is not empty.
6. LT[A] = min { LT(x) - t(x) : x in S[A] }, if S[A] is not empty.

The dynamic programming code used in the CHAIMS scheduler is based on these functional equations.

It is common practice to produce a Gantt chart for project management problems, especially when doing CPM manually. The following table (Table 15) is a Gantt chart for the problem specified by Table 14 above.

Table 15. Gantt Chart for Sample CPM Example Job Table



9 APPENDIX B: SERVICE MEDIATION

This section represents combined work by David Liu, Neal Sample, Jun Peng, Kincho Law, and Gio Wiederhold. It first appeared as "Active Mediation Technology for Service Composition" in the International Workshop on Component Based Business Information Systems Engineering, in 2003. We include it here because it presents an important piece to the GRID puzzle, it was collaborative work in the scope of the CHAIMS project, and this format properly credits the work substantially performed by Dr. David Liu. We have made updates to the paper, since the original presentation.

This chapter discusses how active mediation allows information clients to modify the processing behavior of source information services to facilitate service composition. A new computing paradigm is enabled by incorporating active mediators into autonomous services to allow execution of mobile classes – dynamic information processing modules. We describe two key components of active mediation in the FICAS service composition infrastructure: (1) the programming language support for specifying mobile classes and using mobile classes in a composed service; and (2) the runtime support that enables the execution of mobile classes by autonomous services. The power of active mediation is demonstrated by a few examples of mobile classes that conduct complex information processing for service composition. Finally, we discuss the performance issues in deploying active mediation. Specifically, we introduce an algorithm that determines the optimal placement of mobile classes on alternative autonomous services, and discuss the applicability of the algorithm using example mobile classes.

9.1 INTRODUCTION

9.1.1 BACKGROUND

We are seeing a continued increase in both the size and performance of computer networks. As networks become pervasive and ubiquitous, connectivity is assumed for all computing facilities, which can be accessed from any geographic location. Such

phenomenon can be observed not only with the development of the Internet and the paradigm of web services [81, 93] but also from the growth of intra- and inter-enterprise networks. This development in communication infrastructure has significant effects on the structure of current and future large-scale software services. Rather than being constructed from ground up, software applications are expected to utilize functionalities provided by existing service components. Commercial off-the-shelf (COTS) software applications and other information services are the building blocks that provide pieces of functionalities. This vision of software composition [88] is echoed in the megaprogramming framework [5, 71], which builds on software components called megamodules [91] that capture the functionality of autonomous services provided by large organizational units. Autonomous services are linked together according to composition specifications [19, 24] to form megaservices.

We use the term "autonomous services" to describe the cooperating service components and emphasize their common characteristics. First, autonomous services are usually computational or data intensive, involving long running processes that desire asynchronous invocations. Second, autonomous services run on distributed hosts connected via a communication network, potentially producing large amounts of communication traffic. Third, autonomous services may have heterogeneous access interfaces, introducing a variety of information formats and types. Finally, autonomous services are managed without central administrative control; this makes it difficult to tailor the functionalities and the interfaces of the services according to the needs of the service clients.

Figure 33 illustrates the architecture of a typical megaservice composed of autonomous information services. A communication network provides physical connectivity to the hosts on which the autonomous information services reside. The operating systems provide native support for running the information service applications. In general, the native operating systems are heterogeneous, so are the data produced and consumed by the information services. Through a uniformly defined access interface, the

functionalities of information services can be accessed with data mediated to share common formats and meanings among the services.

In our research of service composition infrastructures, we use the CHAIMS system (Compiling High-level Access Interfaces for Multi-site Software) [13, 24] as a point of departure. CHAIMS is a representative service composition system that is based on the concept of megaprogramming. It focuses on the composition of services that are provided by large, distributed components. CHAIMS provides a practical general-purpose compositional language. A purely compositional language CLAM has been defined in [19] to provide application programmers with the necessary abstractions to describe the behaviors of their megaprograms. CHAIMS also provides a runtime environment to hide heterogeneity in the underlying systems. The megaservice execution model is similar to that of Idealized Worker Idealized Manager (IWIM) model [11, 90], where a composed program selects appropriate processes to carry out sub-tasks. The composed program is known as the *manager*, and the processes are called *workers* for that manager. In the case of CHAIMS, megaservices are managers and autonomous services are workers. Autonomous services are entities with exposed methods. These methods can be accessed in a traditional way, invoked with input parameters then returning available values.

There are other compositional tools and frameworks that we could have chosen, such as Globus [38] or Ninja Paths [83]. The purely compositional nature of CHAIMS and CLAM allowed us to focus wholly on the mediation task without the distraction of the non-compositional (e.g., brokering, security) aspects of alternative frameworks.

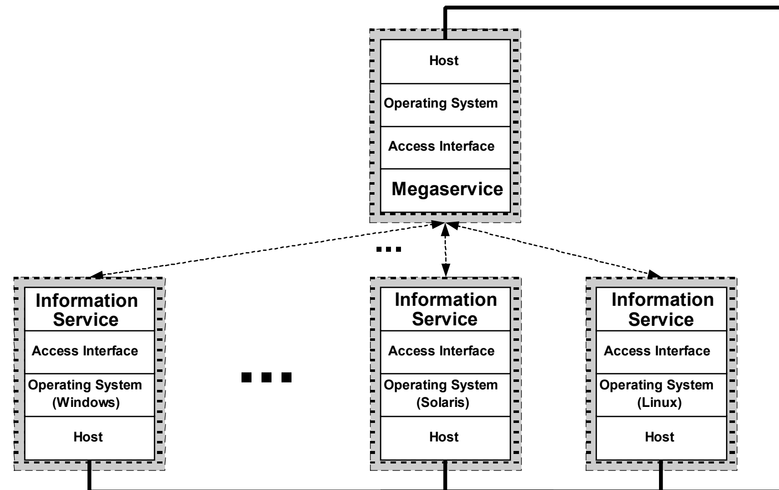


Figure 33 Composing autonomous services into megaservices

9.1.2 OBJECTIVES

This chapter discusses the use of active mediation in service composition infrastructures. Particularly, active mediation provides benefits in the following areas:

- Flexibility of autonomous services: Autonomous services are built and maintained by service providers who are under their own administrative control. It is difficult, if not impossible, for service providers to anticipate all current and potential information clients, and to provide them with information in a ready-to-use form. Furthermore, there are inevitable delays in modifying the functionalities and interfaces of autonomous services to satisfy the specific

requirements from information clients. Information clients therefore need to work around the differences between the information they require and the information provided by autonomous services. Clients usually have to write customized codes (wrappers, filters, etc.) to work around mismatches. On the other hand, service providers generally find it difficult to alter the existing autonomous services – a modification for one class of users can have unexpected effects on other classes of users. As the number of users of an autonomous service increases, the service provider becomes more reluctant to make significant changes to the service. Active mediation increases the customizability and flexibility of autonomous services. Through active mediation, information clients can send dynamic routines to autonomous services to expand the functionalities of existing services.

- Communication load reduction: In CHAIMS, autonomous service invocations are carried out in a fashion similar to that of remote procedure calls [66]. Parameters are sent to the autonomous services and the results are returned back to the megaservice, which processes and dispatches the data to other autonomous services. We have shown in [86] that an execution model that employs centralized data-flows incurs significant performance penalties compared to an execution model that allows distributed data-flows. Active mediation allows information processing to be distributed over autonomous services. It utilizes code transfer to reduce data traffic. Code segments are transferred to the most appropriate location for execution to reduce the amount of data communication between the autonomous services and the megaservice.

- Preserving the Power of a Compositional Language: It is important to have clear separation between computation and composition. CHAIMS enforces the separation by completely removing computational primitives from its compositional language, CLAM [19]. The drawback of this model is that it is difficult to specify application logic. Even for handling arithmetic operations, services need to be invoked. While keeping the design objective of CLAM to

provide a simple language that separates composition from computation, we use active mediation to enable complex logical computations in megaservice specification. Mobile routines can express complex application logic without the complexity of constructing heavy weight autonomous services. The mobile routines are executed on autonomous services that are enabled with active mediation.

Active mediation applies the notion of mobile code [82] to facilitate dynamic information processing in service composition. We introduce an innovative architecture that enables autonomous service to smoothly adapt to a service composition infrastructure that utilizes active mediation. The capabilities of active mediation are discussed through a spectrum of application scenarios, which demonstrate how active mediation removes some of the otherwise insurmountable blocks in conducting effective and efficient service composition. We use active mediation to perform relational operations on the site of information services, providing extra value mediation functionality as well as improved performance. Dynamic type conversion can be conducted efficiently via active mediation, addressing an important issue in collaborating heterogeneous services. Furthermore, active mediation provides a solution to extraction model incompatibilities among autonomous services.

A significant benefit of active mediation is the flexibility of the location on which information processing can be carried out. Mobile code may be sent to alternative autonomous services, which allows performance optimization. We introduce an algorithm that determines the optimal location where active mediation should be conducted. The range of applicability of the algorithm is also discussed.

This chapter is laid out in the following way. Section 9.2 introduces active mediation and describes how it is supported in FICAS (Flow-based Infrastructure for Composing Autonomous Services) by the compositional language and by the runtime environment. Section 9.3 illustrates three application scenarios of active mediation. Section 9.4

discusses performance optimization for active mediation. In section 9.5, we review the implications of active mediation, in FICAS and beyond.

9.2 ACTIVE MEDIATION IN FICAS

As software becomes more complex and services become more powerful, it is essential to define a framework by which software can be constructed to serve clients with a wide range of computation and communication power. The challenge mixes the need to lower the complexity of software design with an attempt to minimize software maintenance costs. To face certain challenges of dynamic collaborative computing environments, mediators were introduced.

9.2.1 MEDIATION

Mediators are intelligent middleware that reside between information clients and information sources [94, 95]. They provide integrated information, without the need to integrate the data resources. Mediators perform functions such as integrating domain-specific data from heterogeneous sources, restructuring the results into object-oriented structures, and reducing the integrated data by abstraction to increase the information density.

A mediation architecture conceptually consists of three layers, as shown in Figure 34. The mediation layer resides between the base resource access interfaces and the service interfaces, incorporating value-added processing by applying domain-specific knowledge processing.

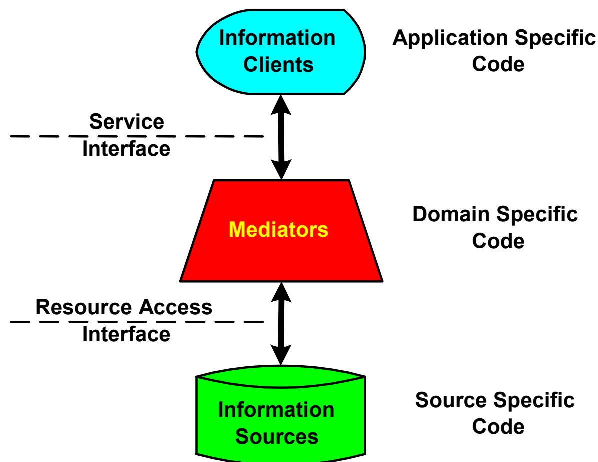


Figure 34 Mediation architecture

In traditional mediators, code is written to handle information processing at the time the mediators are built. We call these types of mediators *static mediators*. As an extension to the static mediators, *active mediators* are introduced to allow information clients to specify client-defined actions in conducting information processing [85]. Active mediators have the ability to dynamically adapt their behaviors to the users of information sources. Through active mediation, autonomous services can manipulate data according to the specification of the information clients before the data is returned.

9.2.2 MOBILE CLASSES

An information processing module dynamically loaded by the active mediators is called a *mobile class* (or *m-class*). Conceptually, a mobile class is a function that takes (multiple) input data elements, performs some operations on the input, then outputs a new data element. For instance, *Out = Func(param1, param2, param3)* represents a mobile class named *Func* that takes three data elements as input and produces an output data element *Out*.

In FICAS, mobile classes are specified as Java classes. Java [79] is chosen as the specification language because of Java's support for portability, its flexibility as a high-level language, and its support of dynamic linking/loading, multi-threading and standard

API libraries. All m-classes are derived from *MobileClass*, whose interface definition is shown below:

```
public interface MobileClass {
  public DataElement execute(Vector params);
}
```

The *execute* function takes a vector of data elements as input and generates a data element as output. M-classes overload the *execute* function to provide specific processing capability. The invoker of the mobile class fills in the input vector at runtime. Upon successful execution, the mobile class returns the output data element.

Mobile classes enable megaservices to include complex computational logic. For instance, m-classes are used for data compression, data expansion, aggregation, relational operations, etc. To invoke mobile classes with a megaservice program, we extend the compositional language CLAM with the *MCLASS* primitive:

```
variable = MCLASS (aclass, param1, param2, …)
```

The first argument *aclass* indicates the location from which the Java byte codes of the mobile class can be loaded. The location can be specified in two forms: (1) a fully qualified URL, or (2) a string, which is appended to a base repository URL to form the fully qualified URL. For example, if the base repository URL for the megaservice were http://mobile.class.repository, then the URL for loading the mobile class *aclass* would be http://mobile.class.repository/aclass.class.

The mobile class may be loaded and executed on any one of the potential autonomous services that support active mediation. For instance, the mobile class *aclass* can be loaded either onto the autonomous service that generates *param1* or onto the autonomous service that generates *param2*. As we will discuss in Section 9.4, the decision is made at runtime to optimize the performance of the megaservice.

Once the mobile class is dynamically loaded onto an active mediator, the parameters in the MCLASS statement are fetched from individual autonomous services and used to fill

the input vector *params* in the *execute* function of the mobile class. A data element will be returned as the result of the execution.

9.2.3 ACTIVE MEDIATION ARCHITECTURE

Figure 35 illustrates the architecture of an active mediator, which consists of multiple functional components. The **data mediator** provides information integration for the underlying information sources. It provides the functionalities that a static mediator would provide, incorporating information processing logic specific to the application domain. The other components are added to provide the dynamic processing capability:

1. The **m-class fetcher** is responsible for locating and fetching appropriate mobile classes for information processing.
2. The **m-class cache** is a temporary storage for the Java byte codes of mobile classes. The cache is used to avoid duplicate loading of the mobile classes. The byte codes of a mobile class are first looked up from the m-class cache. When a cache miss occurs, the m-class fetcher is used to load the byte codes.
3. The **m-class API library** stores utility classes that make the construction of mobile classes more convenient. For instance, the JDK library [79] is provided as part of the m-class API library.
4. The **m-class runtime** is the module where m-classes are executed. The runtime invokes appropriate mobile classes to conduct dynamic information processing.
5. The **exception handling** module provides a comprehensive set of policies to handle abnormalities in loading or processing mobile classes. Our current implementation prohibits any results from getting through the mediator in the case of an exception. In addition, the conditions are logged for future maintenance.

Information clients can access the information sources via the active mediator in two forms, either as a data query or as a mobile class query. Data queries are handled directly by the data mediator, which issues requests to the information sources and integrates data from multiple sources. The integrated results pass through the m-class runtime module and are returned to the information client.

Mobile class queries are first handled by the m-class fetcher, which locates and loads the related mobile classes into the m-class cache. Simultaneously, data queries are issued to the data mediator to obtain data from the information sources. The returned data are further processed by the m-class runtime by invoking the mobile classes to conduct information processing. The processed results are then returned to the information client.

We highlight the fact that active mediators can be used in place of static mediators without changing either the information client or the source information service. This enables a smooth transition to an infrastructure that uses active mediation. When received by an active mediator, a data query not in the form of a mobile class will simply flow through the system without the invocation of any mobile classes, producing behavior identical to a static mediator.

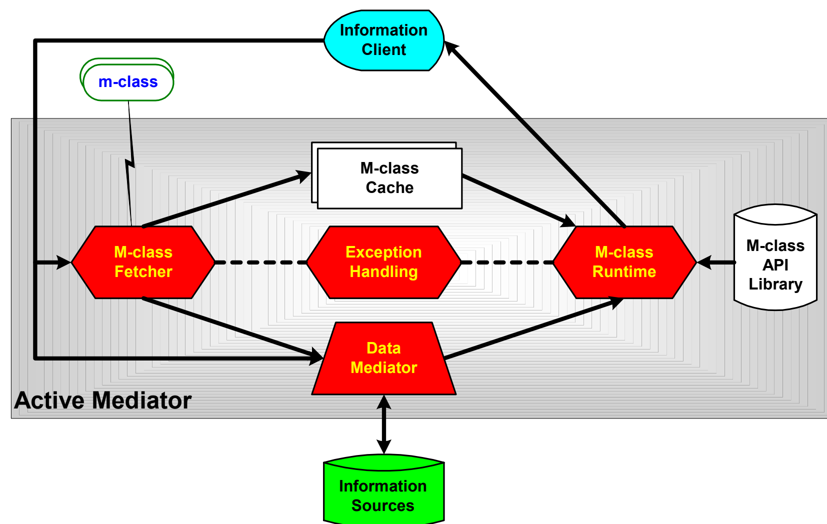


Figure 35 Active mediation architecture

9.2.4 Autonomous Service in FICAS

Autonomous services handle both immediate and deferred execution of mobile classes, which requires the separation of the control-flows from the data-flows. FICAS is an autonomous service metamodel (defined in [87]) that explicitly separates control-flows from data-flows. As shown in Figure 36, an autonomous service consists of an input

event queue, an output event queue, an input container, an output container, and a service core. The service core represents the functionality of the autonomous service that is normally delivered by a software application. The event queues are used to handle service requests and to negotiate control-flows decisions with the megaservice controllers. The containers are used to store data elements used (and produced) by the service core.

Each autonomous service contains two flows. For control-flow, the autonomous service is primarily concerned with the state management of an autonomous service, namely the completion of a task, the termination of the service, the invocation of a mobile class, etc. Events are processed from the input event queue and generated onto the output event queue. For data-flow, the autonomous service primarily deals with performing services with the input data elements in the input container. As the results of a service execution, output data elements are generated in the output container. Data containers enable autonomous services to look up generated data elements and to fetch data elements from other autonomous services. The existence of data containers is essential to enable the asynchronous invocation of mobile classes.

We observe that the autonomous services differ only in their service cores. Event queues, data containers, and their processing logic are identical for all autonomous services. By building these structures into a standard component called an *autonomous service mediator*, we simplify the process of constructing autonomous services. We have implemented autonomous service mediators as active mediators, so that autonomous services can handle not only data queries but also mobile class queries.

Given the autonomous service metamodel, we define an Autonomous Service Access Protocol (ASAP) by which the autonomous services are accessed. ASAP is a simple protocol for exchanging information among autonomous services in a distributed environment. The protocol removes the barriers imposed by different megaservice programming languages and distribution protocols.

ASAP manages control-flows and data- flows through a set of events. These events exist in the form of XML based messages that are used to interact with autonomous services. The protocol defines how the receiving autonomous services would act and respond to each event. ASAP is asynchronous and non-blocking. The sender of an event does not need to wait for the response to the event. Instead, the sender can continue executing other activities that are not dependent on the response to the event.

For simplicity, we represent the ASAP events here using their abbreviated functional representations instead of their full XML representations. The key ASAP events related to data-flow scheduling are listed below. More complete information on the ASAP protocol is given in [87].

- SETUP (Service) – initializing an autonomous service. The autonomous service is informed to prepare necessary system resources for an actual invocation. A reply event is issued after the initialization of the autonomous service.
- TERMINATE (Service) – terminating an autonomous service. Garbage collection is conducted during a termination process, so that the system resources involved with an autonomous service instance are released. A reply event is issued after the termination of the autonomous service.
- INVOKE (Service) – requesting service from an autonomous service. The service core of the autonomous service is started upon the processing of an INVOKE event. After the completion of the service invocation, output data elements generated by the service core are placed into the output container. In addition, a reply event is issued.
- MAPDATA (DataElement, SourceService, DestinationService) – exchanging data between two autonomous services. The event enables the distribution of data-flows within the service composition infrastructure. The MAPDATA event requests a data element to be moved from the output container of the

SourceService to the input container of the DestinationService. After completing the task, the SourceService issues a reply back to the originator of the MAPDATA event. It is important to note that the sender of the MAPDATA event does not need to be the recipient of the data element. The events can be sent from the megaservice controller that coordinates the autonomous service invocations, and the data elements are exchanged directly among the data containers of the autonomous services. While the support of the MAPDATA event makes it possible to have distributed data-flows, it is up to the megaservice controller to generate an execution plan that can take advantage this capability.

- INVOKEMCLASS (Service, AClass) – invoking a mobile class on an autonomous service. Input parameters of the mobile class are fetched from the input data container, and the result of the execution is put on the output container of the autonomous service. After the completion of the mobile class execution, a reply event is issued.

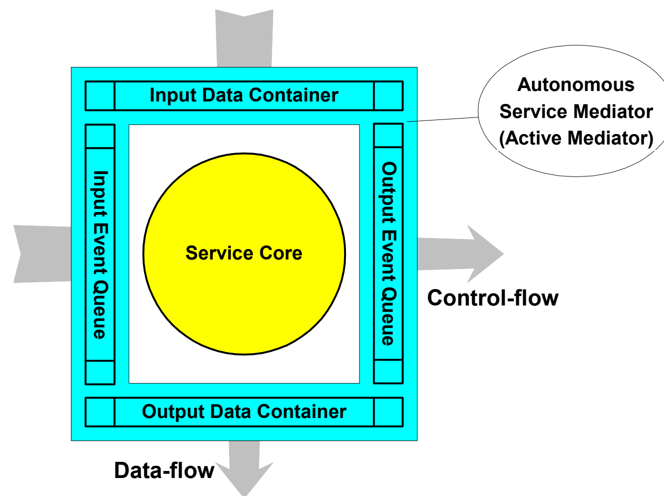


Figure 36 Autonomous service metamodel

9.3 APPLICATIONS OF MOBILE CLASSES

The key feature of the active mediators is their ability to dynamically configure the information processing behaviors. Information clients can send mobile classes to

autonomous services where the source information resides. This section presents several sample applications that can be addressed by an active mediator.

9.3.1 RELATIONAL M-CLASS

A large set of data operations can be represented as relational queries. We call this type of m-class a *relational m-class*. The relational m-class can be viewed as the implementation for its corresponding relational expression. The input and output data elements of a relational m-class are relations incorporated as objects. The input data elements are used as operands in the relational expression, and the output data element is the result of the relational operation. Complex expressions are built progressively by combining relational operators on sub-expressions. Table 16 lists the relational operators, their relational algebra representations, and their corresponding m-classes:

- Unary Operators (σ, π): The select operator σ selects tuples that satisfy a given predicate condition. The m-class implementation of a select operator takes a relation as the input data element, checks the condition on every tuple within the relation, and generates a result data element that contains all the satisfying tuples. The project operator π reduces the number of columns in a relation with only the desired attributes left. The m-class implementation of a project operator takes a relation as the input data element, truncates all the undesired attributes, and returns the resulting relation as the output.

- Set Operators ($\bigcup$, $\bigcap$, $-$): The union operator $\bigcup$ returns the tuples that appear in either or both of the relations. The intersection operator $\bigcap$ returns only the tuples that appear in both of the relations. The difference operator – returns the tuples that appear in the first relation that are not in the second relation. The m-class implementations of the set operators take two relations as the input data elements, perform the set operation on the relations, and return the resulting relation as the output.

- Combination Operators ($\times$, $\bowtie$): The Cartesian product operator associates every tuple of the first relation with every tuple of the second relation. The theta join operator combines a selection with Cartesian product, forcing the resulting tuples to satisfy the specific predicate condition. The m-class implementations of the combination operators take two relations as the input data elements, perform the combination operations on the relations, and return the resulting relation as the output.

Table 16. Relational operators and their corresponding M-classes

Operator	Relational Representation	M-class Interface
Select	$O = \sigma_{cond}(A)$	O = SELECT (A)
Project	$O = \pi_{attr}(A)$	O = PROJECT (A)
Union	$O = A \cup B$	O = UNION (A, B)
Intersect	$O = A \cap B$	O = INTERSECT (A, B)
Difference	$O = A - B$	O = DIFFERENCE (A, B)
Cartesian product	$O = A \times B$	O = CARTESIAN (A, B)
Theta join	$O = A \bowtie_{cond} B$	O = JOIN (A, B)

9.3.2 TYPE CONVERSION M-CLASS

Data generated by an autonomous service can be directly used by other autonomous services if the services share the same data types, formats, and granularities, etc. However, such homogeneity cannot be assumed within a large-scale service composition environment. Data enter with various types and will continue to appear in different types that are suited for different applications. The output data of one autonomous service needs to be converted to conform to the input data type of another autonomous service.

Traditionally, a type broker service or a distributed network of type brokers can be used to mediate the difference among data in various formats [89]. The type brokers can use data in unknown formats and convert them to known formats for the information client. The brokers serve as proxies connecting client requests with appropriate source services. A type graph is used to figure out the chain of conversions necessary. An example of

automating this process can be seen in [80]. The issue of using type brokers for type conversion in service composition is its inefficiency. Large amount of data are forwarded among the brokers, especially when a chain of conversions is involved. Figure 37(a) presents an example of data flow in type-broker architecture. Data from the source service are represented in type T1, and the information client consumes data in type T3. Two type brokers are employed to convert source data from type T1 to type T3. Data are passed back and forth among the type brokers and the information client in order to convert the data into consumable format.

Active mediation avoids data transmission among type brokers by performing type conversion at the source mediator. As shown in Figure 37(b), rather than data being forwarded among type brokers, type conversion functions specified in the form of mobile classes are forwarded to where the data is located. Similar to the network of type brokers, multiple type conversion m-classes can be chained together. Type conversion is conducted using mobile classes at the source mediator. Data in a consumable format is directly returned to the information client. Collocating the conversion mechanism with the data avoids the obvious cost of multiple interim data transfers, reducing data traffic to essential transmissions.

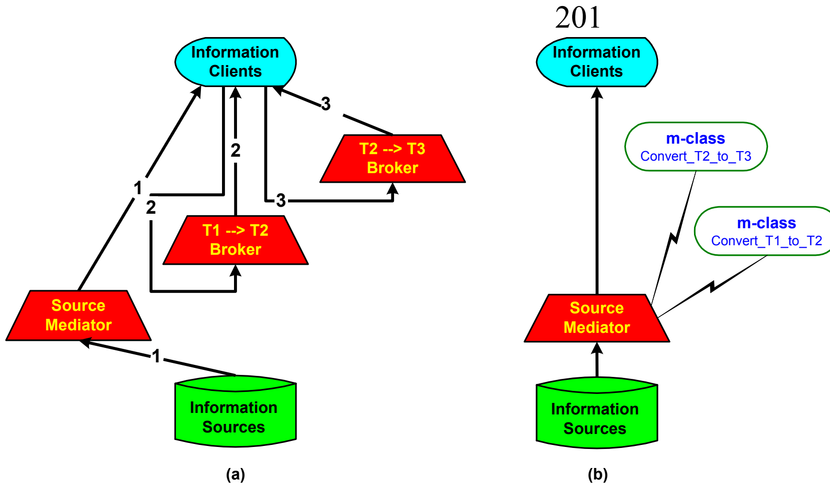


Figure 37 Type conversion for autonomous services

9.3.3 Extraction Model Mediation

Autonomous services can produce data with a wide variety extraction models [92]. Interesting megaservices will have a set of "upstream" autonomous services generating data that is consumed by a set of "downstream" autonomous service (the sets are generally not disjoint). When there is a mismatch between how data is produced and how that data is later consumed, extraction mediation can play an important role. For example, an upstream service might produce data progressively (as a sequence of tuples), while the downstream service requires that the data arrive as a whole relation. Active mediation can be used to prepare the data for different extraction models.

A taxonomy of extraction models for autonomous services that produce outputs based on specific inputs is presented in [92]. This taxonomy is based on three binary factors: partial extraction, runtime service status, and runtime result status. These three binary factors combine to form eight (2^3) basic types of data extraction methods, including familiar and obvious methods (e.g., SQL cursors and RPC), and some less obvious methods (e.g., semantic partial extraction and progressive extraction). These eight extraction formats have meaningful matches with respect to input formats, but do not fully describe all possible extraction models.

One extraction model not captured by the taxonomy in [92] is less interesting from the perspective of an autonomous service, but that nonetheless has utility, is the autonomous data stream. An example of an autonomous data stream is one that continually generates tuples, regardless of client input data. A stock ticker is this type of autonomous data stream. There is no input data to the stock ticker, yet it produces output data that may be used by other autonomous services and megaservices.

Active mediators play a critical role in extraction model mediation, ensuring that format inconsistencies do not prevent service composition. Some m-classes used for extraction model mediation are equivalent to type conversion or relational m-classes, others are externally indistinguishable (yet functionally distinct), while some are radically different from the m-classes that we have seen so far.

The simplest m-classes for extraction model mediation are functionally equivalent to relational m-classes used for projection operations. For instance, consider an autonomous service that produces three outputs, *A*, *B*, and *C*. The *partial extraction* model presented in [92] may not be fully implemented in the autonomous service, so *A*, *B*, and *C* are delivered as an opaque object *X*. By extracting only the components of *X* desired by downstream services, m-classes can mimic partial extraction, a behavior not directly supported by the service itself. This simple process can be coupled with selection and reordering (e.g., sorting) predicates for producing many different types of behaviors. With basic m-classes, opaque data can be filtered, set reordered, transmission delayed, projected, split, recombined, and/or extracted multiple times.

A more complex m-class for extraction model mediation is not externally distinguishable from simpler m-classes, but must be uniquely tailored to data extraction. For example, the stock ticker service is an instance of extraction model mediation that looks like a value-based filter, but is implemented in a very different way. A typical value-based filter (like a select) operates on an input set, and produces an output set of an equal or smaller size, after examining all the values in the set. However, how can an m-class with a selection predicate examine *all* the tuples in a continuous *stream* before returning any

data? With an understanding of the how the data is produced and how it is composed, streams become meaningful inputs to downstream services.

One downstream service might be interested in all elements coming from the ticker within a certain window of time; the corresponding m-class generates tuples until the data generated by the ticker falls outside that window, at which time the m-class has completed its task. (Simpler selection m-classes would not understand a termination condition beyond "there are no more tuples.") Another downstream service might be interested in a complete cycle of all elements coming from the ticker. An m-class can sample the stream at an arbitrary point, finding a first ticker symbol, say "QXYZ." The m-class can pass through all further symbols until seeing "QXYZ" again. Having observed an entire cycle, the data is then ready for the downstream consumer service.

Finally, certain extraction models are simply incompatible with other input models. These can be the most difficult to generate m-classes for, but are a place where m-classes are indispensable. An example of this incompatibility was mentioned earlier: an upstream service delivers an SQL cursor, but the downstream service expects a relation. The logic behind the m-class is transparent enough: scroll the cursor to fill a complete relation. However, such a fundamentally simple extraction model mismatch will stymie any upstream/downstream pair without mediation. The power of extraction model mediation does not end with upstream/downstream couplings. The separation of control-flow from data-flow can also be achieved through collocation of m-classes for extraction mediation, even for autonomous services that do not otherwise support such a separation. The models seen in [86] can all be realized with an intelligent application of the appropriate m-classes, even where legacy services do not have such support.

Autonomous services will invariably feature multiple dependencies on both the nature of the service and the expected audience. But not all producers and consumers are created equally. Even when there is a type and domain match between the upstream and downstream services, the extraction model (or the corresponding input model) may

provide and seemingly insurmountable block. Active mediation through the application of m-classes is a solution to extraction model incompatibilities.

9.4 PERFORMANCE OPTIMIZATION FOR ACTIVE MEDIATION

FICAS is a distributed data-flow infrastructure in that data can be passed directly between autonomous services without going through the megaservice. As shown in [86], distributed data-flow infrastructures offer significant performance improvement over centralized data-flow infrastructures in executing megaservices, in terms of both aggregated cost and response time. The improvement in aggregated cost comes from the savings in data passed between autonomous services and the megaservice in the distributed data-flow model. The improvement in response time comes from alleviating the communication bottleneck at the megaservices.

By enabling active mediation at autonomous services, information processing tasks may be dispatched to autonomous services, enabling further performance optimization.

9.4.1 PLACEMENT OF MOBILE CLASSES

The placement of a mobile class can greatly affect the performance of the megaservice it serves. We focus our effort on optimizing the megaservice performance on the communication aspects rather than the computational aspects. We assume that the cost of loading and executing an m-class remains the same regardless of which autonomous service the m-class is loaded onto. The megaservice controller makes the decision about the placement of the m-classes based on the cost that the resulting data-flows would have on the megaservice. For simplification, we assume a uniformly connected processor network, where the network bandwidths between any node pairs are the same.

Figure 38 shows a simple example of using m-classes. The result generated by *Invocation1* is a list of numbers. The m-class *sum* will add up the numbers in the list and pass the summation to *Service2* for further processing.

We can first analyze the example intuitively. Figure 39 illustrates three potential execution plans. The plans differ in the runtime placement of the m-class *Sum*. Consequently, these plans form different data-flow structures for the megaservice, as shown in Figure 40. The three plans are listed as follows:

- **Plan 1**: By placing *sum* at the megaservice controller, we can construct the execution plan as shown in Figure 39(a). *Service1* generates data element *A* and passes it to megaservice. The m-class *Sum* processes the data element *A* at the megaservice controller. The processed result *B* is then sent to *Service2* for further processing.
- **Plan 2**: By placing *sum* at the source autonomous service *Service1*, we can construct the execution plan as shown in Figure 39(b). *Service1* generates data element *A* and processes it locally using the m-class *Sum*. The processed result *B* is then sent to *Service2* for further processing.
- **Plan 3**: By placing *sum* at the destination autonomous service *Service2*, we can construct the execution plan as shown in Figure 39(c). *Service1* generates data element *A* and passes it to *Service2*. *Service2* processes the data locally using the m-class *Sum* and then uses the processed result *B* for further processing.

Our objective is to choose the plan with the best performance based on the cost criteria established in [86] – aggregated cost and response time. It is clear from the data-flow diagram that Plan 1 incurs the most communication traffic compared to the other two plans. Both the input data element *A* and output data element *B* are sent between the megaservice controller and the autonomous services. The plan has the worst performance in terms of both aggregated cost and response time.

Plan 2 and Plan 3 differ in the data content sent between the autonomous services. Plan 2 places the m-class on *Service1*. The data element *A* generated by *Service1* is processed locally by the m-class *Sum*. Thus, only data element *B* is sent from *Service1* to *Service2*. On the contrary, Plan 3 sends the data element *A* to the destination autonomous service.

The plan avoids the communication traffic of sending data element *B*, which is processed by the m-class *Sum* locally on *Service2*.

Evaluating Plan 2 against Plan 3 is a matter of comparing the sizes of data elements *A* and *B*. Due to the fact that the sample m-class *Sum* performs aggregation on the input data content, the output data *B* has smaller volume than the input data *A*. Therefore, Plan 2 incurs the least amount of communication traffic, and it would be chosen as the best execution plan.

```
/* ... */

Invocation1 = Service1.INVOKE()
A = Invocation1.EXTRACT();

B = MCLASS ("sum", A)

Invocation2 = Service2.INVOKE(B)
C = Invocation2.EXTRACT();

/* ... */
```

Figure 38 Sample megaservice program segment with M-classes

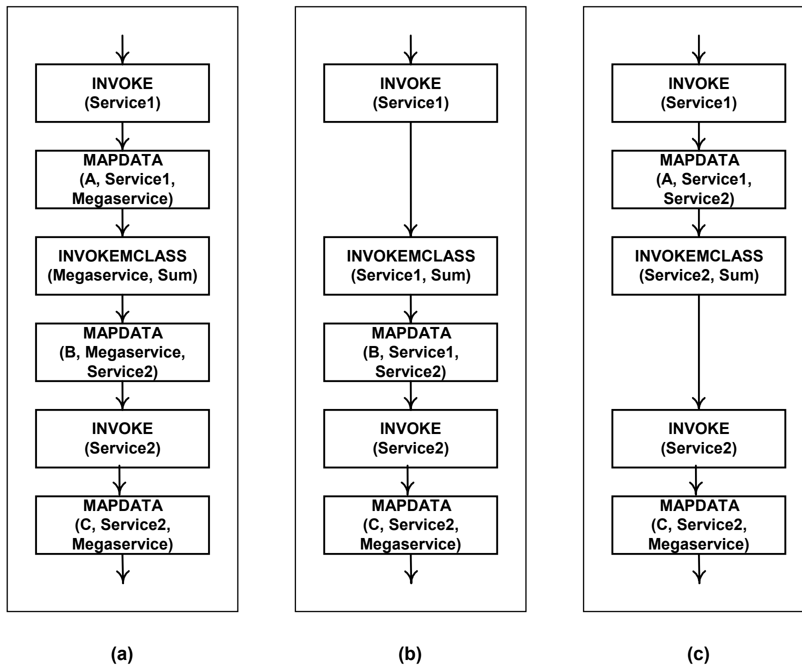


Figure 39 Execution plans for the sample megaservice using M-class *Sum*

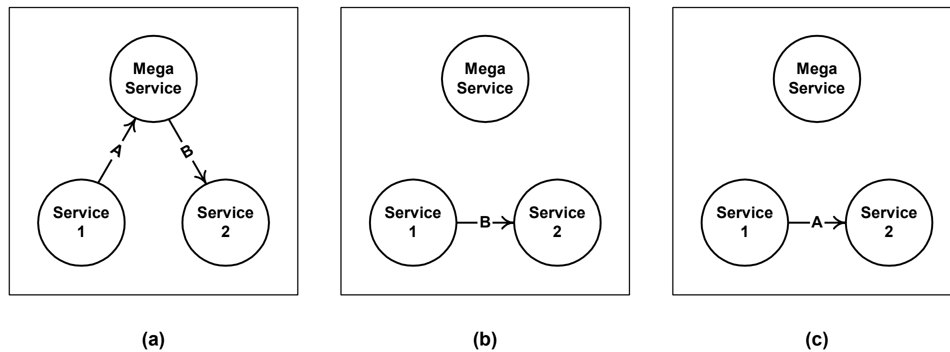


Figure 40 Data-flow diagrams for the sample megaservice using M-class *Sum*

9.4.2 LDS ALGORITHM

Now, we can construct execution plans for generic megaservices by placing m-classes at optimal locations to minimize communication traffic. For an m-class with known input and output data sizes, each input or output data element to the m-class can be characterized as a pair. Each input is a (S_i, V_i) pair, where S_i is the autonomous service that generates the ith input data element, and V_i is the volume of the data element. The

output is a (S_0, V_0) pair, where S_0 is the destination autonomous service to which the result of the m-class will be sent, and V_0 is the size of the data element.

```
INPUT: input pairs(S1, V1), …, (Sn,Vn)
       output pair (S0, V0)
OUTPUT: Smax
METHOD:
     Vmax=0
     for every unique S in input and output pairs
          V=0
          for i=0,…,n
           if Si==S
             V=V+Vi
          if V>Vmax
           Smax=S
           Vmax=V
```

Figure 41Figure 41 shows the LDS (Largest Data Size) algorithm, which targets the autonomous service that generates and consumes the largest volume of data for a given m-class. The algorithm first computes the total amount of data connected with the m-class for each unique autonomous service. Then, the autonomous service with the largest data volume is selected as S_{max}, which represents the optimal placement for the m-class. S_{max} is returned as the output of the algorithm.

Two special types of m-classes simplify the LDS algorithm. The first type of m-class is called *expansion m-class*, whose output data size is at least as large as the sizes of its input data. Based on the LDS algorithm, the optimal m-class placement would be the destination autonomous service to which the result of the m-class would be sent. The other special type of m-class is called a *compression m-class*, where the output data size is smaller than the sizes of its input data. The optimal m-class placement can be chosen as the input autonomous service that has the largest input data size.

The LDS algorithm is only applicable when the input and output data sizes are known for the m-classes. For a situation where the exact output data size of an m-class is unknown until after the execution of the m-class, we need a method to estimate the output data size.

We view the output data size of an m-class as a function on the input data sizes of the m-class: $S_O = f(S_A, S_B, \ldots)$. The function f is called the sizing function of the m-class, where S_O is the output data size and S_A, S_B are the input data sizes. The sizing function may be stored along with the Java byte codes in the m-class repository. The megaservice controller can then use the sizing function to estimate the m-class output data size for running the LDS algorithm.

Clearly, the effectiveness of the mediation based data-flow optimization technique depends on the accuracy of the characterization of m-classes using the sizing functions. In the following section, we discuss how to obtain the sizing functions for a special class of m-classes. In addition, we offer insights into how sizing functions may be formulated for general m-classes.

```
INPUT: input pairs(S1, V1), …, (Sn,Vn)
       output pair (S0, V0)
OUTPUT: Smax
METHOD:
    Vmax=0
    for every unique S in input and output pairs
         V=0
         for i=0,…,n
          if Si==S
            V=V+Vi
         if V>Vmax
          Smax=S
          Vmax=V
```

Figure 41 LDS algorithm for optimizing M-class placement

9.4.3 DATA-FLOW OPTIMIZATION FOR SAMPLE M-CLASSES

The relational operators have well defined relationships between the input and output data sizes. Their sizing functions can be mathematically formulated. Research results are borrowed from the field of query processing to estimate the result size of a relational query [84]. This section shows that the sizing functions do not need to be precise for the LDS algorithm to generate an optimal result. In many cases, sizing functions can be simplified to constant values, in which case the optimal m-class placement can be quickly determined.

Table 17 lists the simplified sizing functions for the relational m-classes. The unary operators *select* and *project* return only portions of the input relations. The m-classes implementing the unary operators are by definition compression m-classes. Their sizing functions return zero, guaranteeing that the output data size should be no larger than the input data size. Hence, the SELECT and PROJECT m-classes are always placed on the source autonomous service that generates the input data.

The *union* operator combines the two input relations. Hence, its m-class is an expansion m-class. The sizing function returns infinity, guaranteeing that the output data size should be greater than the input data sizes. The m-class is always placed on the destination autonomous service that uses the output data.

The *intersect* and *difference* operators return portions of the input relations. Their m-classes are compression m-classes, and thus the sizing function return zero. The m-class would be placed on one of the source autonomous services. The choice of the source autonomous services will made when the size of the input data elements are evaluated at runtime. The optimal placement is the input autonomous service with larger data size.

The result set of *Cartesian product* operator contains all possible combinations of one tuple from each input relations. The result relation is clearly larger than the input relations. Similar to the *union* operator, the sizing function of *Cartesian product* returns infinity. The m-class is placed on the destination autonomous service.

The sizing function for the *theta join* operator is more complex. The exact function usually depends on the characteristics of the input data and the types of predicate conditions. For instance, the sizing function may be set to $S_0=c\times S_A\times S_B$ if the *theta join* is an equality join with uniformly distributed values in input relations; if the result relation is expected to be rather small, the sizing function can be set to zero to enforce the LDS algorithm choose the input autonomous services. The sizing function may be set to infinity to have the LDS algorithm choose the output autonomous service if the result relation is expected to be larger than the input relations.

Relational expressions that are certain combinations of relational operators may have simple mathematical representations of their sizing functions. For instance, the O=PROJECT(INTERSECT(A,B)) has a simple sizing function of $S_O=0$.

As m-classes get more complex, the relationship between output data sizes and input data sizes becomes harder to represent with simple mathematical formulae. In some cases, the output data sizes may not be determined based on the input data sizes, as the content of the input data affects the output data size. We are exploring techniques to estimate the output size of arbitrary m-classes. As a future research direction, we are looking at statistical methods to adaptively estimate the correlations between the m-classes' input and output data sizes.

Table 17. Sizing functions for relational M-classes

M-class Interface	Sizing Function
O = SELECT (A)	$S_O = 0$
O = PROJECT (A)	$S_O = 0$
O = UNION (A, B)	$S_O = \infty$
O = INTERSECT (A, B)	$S_O = 0$
O = DIFFERENCE (A, B)	$S_O = 0$
O = CARTESIAN (A, B)	$S_O = \infty$
O = JOIN (A, B)	$S_O = f(S_A, S_B)$

9.5 CONCLUSIONS

Active mediation increases the customizability and flexibility of autonomous services. It utilizes code mobility to facilitate dynamic information processing in service composition. In this chapter, we offer a conceptual framework for active mediation. An innovative architecture is defined to enable smooth adoption of active mediation in autonomous services. Our evolutionary approach allows coexistence of active mediation and static mediation, making it feasible to build a service composition infrastructure that supports active mediation. We survey a spectrum of application scenarios that can be effectively addressed by active mediation. Specifically, valuable mediation functionalities such as relational operations, dynamic type conversion and extraction model mediation fit nicely into the active mediation framework. Through the discussion of the application scenarios, we reveal the importance of active mediation in conducting service composition.

We discuss the performance issues in deploying active mediation in the service composition runtime environment. The fact that mobile classes may be executed on alternative locations enables us to conduct performance optimization. We introduce an algorithm that determines the optimal placement of mobile classes, and discuss the applicability of the algorithm.

10 BIBLIOGRAPHY

[1] W. Tracz: “Confessions of a Used Program Salesman”; Addison-Wesley, 1995

[2] P. Naur and B. Randell, eds.: “Software Engineering”; Proc. of the NATO Science Committee, Garmisch, Germany, October 7-11, 1968. Proceedings published January 1969

[3] W. Rosenberry, D. Kenney and G. Fisher: “Understanding DCE”; O'Reilly, 1994

[4] C. Szyperski, “Component Software: Beyond Object-Oriented Programming”, Addison-Wesley and ACM-Press New York, 1997

[5] G. Wiederhold, P. Wegner, and S. Ceri: “Towards Megaprogramming: A Paradigm for Component-Based Programming”; Communications of the ACM, 1992(11): p.89-99

[6] G. Grafe and K. Karen: “Dynamic Query Evaluation Plans”; In James Clifford, Bruce Lindsay, and David Maier, eds., Proceedings of the 1989 ACM SIGMOD International Conference on Management of Data, Portland, Oregon, June 1989

[7] N. Sample and M. Haines: “MARS: Runtime Support for Coordinated Applications”; Proceedings of the 1999 ACM Symposium on Applied Computing, San Antonio, Texas, Feb-Mar 1999.

[8] C. Tornabene, D. Beringer, P. Jain and G. Wiederhold: “Composition on a Higher Level: CHAIMS,” *unpublished*, Dept. of Computer Science, Stanford University.

[9] H. Bal and M. Haines: “Approaches for Integrating Task and Data Parallelism,” Technical Report IR-415, Vrije Universiteit, Amsterdam, December 1996.

[10] M. Haines and P. Mehrotra: "Exploiting Parallelism in Multidisciplinary Applications Using Opus," Proceedings of the Seventh SIAM Conference on Parallel Processing for Scientific Computing, San Francisco, CA, February 1995.

[11] F. Arbab, I. Herman, and P. Spilling: "An Overview of MANIFOLD and its Implementation," *Concurrency: Practice and Experience*, Volume 5, issue 1, 1993.

[12] G. Papadopoulos and F. Arbab: "Modeling Electronic Commerce Activities Using Control Driven Coordination," Ninth International Workshop on Database And Expert Systems Applications (DEXA 98): Coordination Technologies for Information Systems, Vienna, August, 1998.

[13] D. Beringer, C. Tornabene, P. Jain, and G. Wiederhold: "A Language and System for Composing Autonomous, Heterogeneous and Distributed Megamodules," DEXA 98: Large-Scale Software Composition, Vienna, August 1998.

[14] B. Chapman, et al., "Opus: A Coordination Language for Multidisciplinary Applications," ICASE Tech Report 97-30, Jun 1997.

[15] J. Hanly, E. Koffman and J. Horvath, Porgram Design for Engineers, Addison-Wesley, Menlo Park, CA, 1995.

[16] ICASE Research Quarterly, Vol. 4, No. 1, March 1995.

[17] Workflow Management Facility, Revised Submission, OMG Document Number: bom/98-06-07, July 1998.

[18] L. Melloul, D. Beringer, N. Sample and G. Wiederhold, "CPAM, A Protocol for Software Composition," CAiSE'99, Heidelberg, Germany, June 1999 (Springer LNCS).

[19] N. Sample, et al., “CLAM: Composition Language for Autonomous Megamodules,” Third Int'l Conference on Coordination Models and Languages, COORD'99, Amsterdam, April 26-28, 1999 (Springer LNCS).

[20] Simple Workflow Access Protocol (SWAP), Keith Swenson, IETF internet draft, August 1998.

[21] Workflow Management Coalition, The Workflow Reference Model, Document Number TC00-1003, Nov 1994.

[22] C. Bartlett, N. Sample, and M. Haines "Pipeline Expansion in Coordinated Applications", 1999 International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA'99), Las Vegas, Nevada, June 28 - July 1, 1999.

[23] T. Pratt and M. Zelkowitz, Programming Languages, Design and Implementation, 1996, Prentice Hall, Inc.

[24] G. Wiederhold, D. Beringer, N. Sample, and L. Melloul, "Composition of Multi-site Services", Proceedings of IDPT'99, Kusadasi, Turkey, June 1999.

[25] T. Bray, J. Paoli and C. Sperberg-McQueen, “Extensible Markup Language (XML) 1.0,” W3C Recommendation, February 1998.

[26] A. Davidson, et al., “Schema for Object-Oriented XML 2.0,” W3C Note, July 1999.

[27] J. Robie, “XQL Tutorial,” March 1999 (http://metalab.unc.edu/xql/xql-tutorial.html).

[28] A. Deutsch, et al., “XML-QL: A Query Language for XML,” submission to the W3C, August 1998.

[29] J. Eder, E. Panagos, H. Pezewaunig, and M. Rabinovich, “Time Management in Workflow Systems,” In 3rd Int. Conf. on Business Information Systems, 1999.

[30] H. Pozewaunig, J. Eder, and W. Liebhart, "ePERT: Extending PERT for Workflow Management Systems," First EastEuropean Symposium on Advances in Database and Information Systems ADBIS ' 97, St. Petersburg, Russia, September 1997.

[31] F. Berman, High Performance Schedulers in Building a Computational Grid, I. Foster and C. Kesselman, editors, Morgan Kaufmann, 1998.

[32] R. Buyya, J. Giddy, D. Abramson, "A Case for Economy Grid Architecture for Service-Oriented Grid Computing," 10th IEEE International Heterogeneous Computing Workshop (HCW 2001), In conjunction with IPDPS 2001, April 2001.

[33] P. Keyani, N. Sample, and G. Wiederhold, "Scheduling Under Uncertainty: Planning for the Ubiquitous Grid," Fifth International Conference on Coordination Models and Languages (Coord2002).

[34] R. Buyya, J. Giddy, D. Abramson, "An Evaluation of Economy-based Resource Trading and Scheduling on Computational Power Grids for Parameter Sweep Applications," The Second Workshop on Active Middleware Services (AMS 2000), August 1, 2000.

[35] S. Lohr, "I.B.M. Making a Commitment to Next Phase of the Internet" New York Times 2001, http://www.nytimes.com/2001/08/02/technology/02BLUE.html.

[36] D. Marinescu, L. Bölöni, R. Hao, and K. Jun, "An Alternative Model for Scheduling on a Computational Grid," Proceedings of ISCIS'98, the Thirteenth International Symposium on Computer and Information Sciences, Antalya, pp. 473-480, IOP Press, 1998.

[37] N. Nisan, S. London, O. Regev, and N. Camiel, "Globally distributed computation over the internet-the popcorn project," In Proceedings for the 18th Int'l Conference on Distributed Computing Systems, 1998.

[38] I. Foster and C. Kesselman, “Globus: A Metacomputing Infrastructure Toolkit,” Proceedings of the Workshop on Environments and Tools for Parallel Scientific Computing, SIAM, Lyon, France, Aug. 1996.

[39] Juno, “Juno Announces Virtual Supercomputer Project,” Juno Press Release, February 1, 2001), http://www.juno.com/corp/news/supercomputer.html.

[40] United Devices, “Edge Distributed Computing with the MetaProcessor(TM) Platform,” White paper, 2001, https://www.ud.com/customers/met.pdf.asp.

[41] UDDI, Technical White Paper, September 6, 2000 http://www.uddi.org/pubs/Iru_UDDI_Technical_White_Paper.pdf.

[42] C. Kurt, “UDDI Version 2.0 Operator's Specification”, UDDI Open Draft Specification 8, June 2001 (Draft K) , http://www.uddi.org/pubs/Operators-V2.00-Open-20010608.pdf

[43] M. Stonebraker, R. Devine, M. Kornacker, W. Litwin, A. Pfeffer, A. Sah, and C. Staelin. “An economic paradigm for query processing and data migration in Mariposa,” In Proceedings of the Third International Conference on Parallel and Distributed Information Systems, Austin, TX, September 1994.

[44] J.Sidell, “Performance of Adaptive Query Processing in the Mariposa Distributed Database Management System,” unpublished manuscript, June 1997.

[45] W. K. Shih, J. W. S. Liu, and J. Y. Chung. “Algorithms for scheduling imprecise computations with timing constraints,” In Proc. IEEE Real-Time Systems Symposium, 1989.

[46] F. Seredynski, P. Bouvry, and F. Arbab, “Parallel and distributed evolutionary computation with Manifold,” In V. Malyshkin, editor, Proceedings of PaCT-97, volume 1277 of Lecture Notes in Computer Science, pages 94--108. Springer-Verlag, September 1997.

[47] F. Arbab, “The IWIM Model for Coordination of Concurrent Activities,” First International Conference on Coordination Models, Languages and Applications (Coordination'96), Cesena, Italy, April 15-17 1996. (Also appears in LNCS 1061, Springer-Verlag, pp. 3456.)

[48] A. Garvey, K. Decker, and V. Lesser, “A Negotiation-based Interface Between a Real-time Scheduler and a Decision-Maker,” Tech. Rep. 94-08, U. of Massachusetts Department of Computer Science, March 1994.

[49] A. Garvey and V. Lesser, “Design-to-time scheduling with uncertainty,” CS Technical Report 95--03, University of Massachusetts, 1995.

[50] F. Berman, “High-performance schedulers,” The Grid: Blueprint for a New Computing Infrastructure, 1999.

[51] A. Geppert, M. Kradolfer, and D. Tombros. “Market-Based Workflow Management,” Int'l Journal on Cooperative Information Systems (IJCIS), 7(4):297--314, December 1998.

[52] M.J. Atallah et al, "Models and Algorithms for Coscheduling Compute-Intensive Tasks on a Network of Workstations," Journal of Parallel and Distributed Computing, Vol. 16, 1992.

[53] High Performance Fortran Forum (HPFF), "HPF Language Specification", Version 2.0, January 31, 1997.

[54] A. Grimshaw and W. Wulf. “Legion - a View from 50,000 Feet,” Proc. 5th IEEE Symp. on High Performance Distributed Computing, pp. 89-99, IEEE Press, 1996.

[55] P. Lawrence, editor, Workflow handbook 1997, John Wiley 1997.

[56] J. Doyle, “Reasoned assumptions and Pareto optimality,” Proc. Ninth International Joint Conference on Artificial Intelligence, 1985.

[57] T. Anderson, D. Culler, and D. Patterson, “A Case for Networks of Workstations: NOW,” IEEE Micro, February 1995.

[58] R. G. Smith, “The CONTRACT NET: A formalism for the control of distributed problem solving,” In Proceedings of the 5th Intl. Joint Conference on Artificial Intelligence (IJCAI-77), Cambridge, MA, 1977.

[59] R. Balzer and K. Narayanaswamy, “Mechanisms for generic process support,” Proc. First ACM SIGSOFT Symp. Foundations Software Engineering, pages 21--32. ACM Software Engineering Notes, Vol. 18(5), December 1993

[60] J. Siegel, CORBA fundamentals and programming, Wiley, New York, 1996

[61] D. Platt, The Essence of COM and ActiveX, Prentice-Hall, 1997

[62] R. Van Renesse and K. Birman, "Protocol Composition in Horus," TR95-1505, 1995

[63] J. Jannink, S. Pichai, D. Verheijen and G. Wiederhold, "Encapsulation and Composition of Ontologies," AAAI'98 Workshop on AI and Information Integration, 1998

[64] "Information Processing -- Open Systems Interconnection -- Specification of Abstract Syntax Notation One" and "Specification of Basic Encoding Rules for Abstract Syntax Notation One", International Organization for Standardization and International Electrotechnical Committee, International Standards 8824 and 8825, 1987

[65] L. Perrochon, G. Wiederhold and R. Burback, "A compiler for Composition: CHAIMS," Fifth International Symposium on Assessment of Software Tools and Technologies (SAST `97), Pittsburgh, June 3-5, 1997

[66] A. Birrell and B Nelson, “Implementing Remote Procedure Calls,” ACM Transactions on Computer Systems, 1984. 2(1): p. 39-59

[67] ISO, "ISO Remote Procedure Call Specification," ISO/IEC CD 11578 N6561, 1991

[68] M.P. Atkinson, V. Benzaken, and D. Maier (eds.): Persistent Object Systems: *Springer-Verlag and British Computer Society,* 1995.

[69] Laszlo A. Belady: "From Software Engineering to Knowledge Engineering: The Shape of the Software Industry in the 1990's"; *International Journal of Software Engineering and Knowledge Engineering,* Vol.1 No.1, 1991.

[70] Dorothea Beringer, Gio Wiederhold, Laurence Melloul: "A Reuse and Composition Protocol for Services"; *Proc. Symp. on Software Reusability (SSR'99)*, ACM SIGSOFT, Los Angeles CA, May 1999.

[71] B. Boehm and B. Scherlis: "Megaprogramming"; *Proc. DARPA Software Technology Conference 1992*, Los Angeles CA, April 28-30, Meridien Corp., Arlington VA 1992, pp 68-82.

[72] G. Booch: *Object-Oriented Design with Applications*, 2nd Ed.; Benjamin-Cummins, 1994.

[73] A. Chavez, C. Tornabene, and G. Wiederhold, “Software Component Licensing Issues: A Primer,” *IEEE Software*, Vol.15 No.5, Sept-Oct 1998, pp.47-52.

[74] B. Cramsie et al, CIIMPLEX Reference Architecture, *Consortium for Integrated Manufacturing Protocols (CIIMPLEX),* Atlanta GA

[75] J. Gennari, H. Cheng, R. Altman, & M. Musen, “Reuse, CORBA, and Knowledge-Based Systems,” *Int. J. Human-Computer Sys*., Vol.49 No.4, pp.523-546, 1998

[76] “Oracle Business OnLine, Removing Barriers to Enterprise Applications Adoption,” http://www.oracle.com/businessonline/, Oracle Corporation, 1998.

[77] D. Searls, “Biowidgets,” Computational Methods in Molecular Biology, Elsevier Science, 1998.

[78] J. Ullman, Principles of Database and Knowledge-Base Systems, Volume 1: *Classical Database Systems, Computer Science Press*, 1988.

[79] K. Arnold, J. Gosling, and D. Holmes, "The Java Programming Language", Java Series, Addison-Wesley, 2000.

[80] S. Chandrasekaran, S. Madden, and M. Ionescu, "Ninja Paths: An Architecture for Composing Services over Wide Area Networks", UC Berkeley, Technical Report, 2000, http://ninja.cs.berkeley.edu/dist/papers/path.ps.gz.

[81] D. F. Ferguson, "Web Services Architecture: Direction and Position Paper", Proceedings of W3C Web Services Workshop, San Jose, CA, April 2001.

[82] A. Fuggetta, G. P. Picco, and G. Vigna, "Understanding Code Mobility", *IEEE Transactions on Software Engineering*, vol. 24(5), 1998, pp. 342-361.

[83] S. Gribble, M. Welsh, R. von-Behren, E. Brewer, D. Culler, N. Borisov, S. Czerwinski, R. Gummadi, J. Hill, A. Joseph, R. H. Katz, Z. M. Mao, S. Ross, and B. Zhao, "The Ninja Architecture for Robust Internet-Scale Systems and Services", U.C. Berkeley, To appear in a Special Issue of Computer Networks on Pervasive Computing, 2002, http://ninja.cs.berkeley.edu/dist/papers/ninja.ps.gz.

[84] H. F. Korth and A. Silberschatz, "Database System Concepts", 2nd ed, McGraw-Hill, 1991.

[85] D. Liu, K. Law, and G. Wiederhold, "CHAOS: An Active Security Mediation System", Proceedings of International Conference on Advanced Information Systems Engineering, LNCS, vol.1789, B. Wangler and L. Bergman (eds.), Springer-Verlag, 2000, pp. 232-246.

[86] D. Liu, K. Law, and G. Wiederhold, "Analysis of Integration Models for Service Composition", Proceedings of Third International Workshop on Software and Performance, Rome, Italy, July 2002.

[87] D. Liu, K. Law, and G. Wiederhold, "FICAS: A Distributed Data-Flow Service Composition Infrastructure", Stanford University, Unpublished Report, 2002, http://mediator.stanford.edu/papers/FICAS.pdf.

[88] M. D. McIlroy, "Mass Produced Software Components", *Software Engineering, NATO Science Committee*, January 1969, pp. 138-150.

[89] J. Ockerbloom, "Mediating Among Diverse Data Formats", Carnegie Mellon University, Pittsburgh, PA, PhD. Thesis, 1998.

[90] G. A. Papadopoulos and F. Arbab, "Coordination of Distributed Activities in the IWIM Model", *International Journal of High Speed Computing*, vol. 9(2), 1997, pp. 127-160.

[91] D. L. Parnas, "On the Criteria to be Used in Decomposing Systems into Modules", *P. Freeman and A. I. Wasserman*, Tutorial on Software Design Techniques, IEEE Computer Society Press, 1983.

[92] N. Sample, D. Beringer, and G. Wiederhold, "A Comprehensive Model for Arbitrary Result Extraction", Proceedings of ACM Symposium on Applied Computing, Madrid, Spain, March 2002

[93] W3C, "Simple Object Access Protocol (SOAP)", 2000, http://www.w3.org/TR/SOAP

[94] G. Wiederhold, "Mediators in the Architecture of Future Information Systems", *IEEE Computer*, March 1992, pp. 38-49.

[95] G. Wiederhold and M. Genesereth, "The Conceptual Basis for Mediation Services", *IEEE Expert, Intelligent Systems and Their Applications*, vol. 12(5), October 1997, pp. 38-47.

[96] NetMBA.com, "CPM – Critical Path Method," http://www.netmba.com/operations/project/cpm/, 2004

[97] D. Liu, N. Sample, J. Peng, K. Law, and G. Wiederhold, "Active Mediation Technology for Service Composition," International Workshop on Component Based Business Information Systems Engineering, 2003.

[98] The Data Structures Library in Java (JDSL), http://www.cs.brown.edu/cgc/jdsl/, 2004.

[99] "Critical Path Method," http://www.ifors.ms.unimelb.edu.au/tutorial/cpm/, 2004.

[100] C. H. Papadimitriou and M. Yannakakis, "Multiobjective query optimization," ACM SIGMOD/SIGACT Conference on Principles of Database Systems (PODS), 2001.

[101] V. Easton and J. McColl, "Statistics Glossary," http://www.stats.gla.ac.uk/steps/glossary/presenting_data.html#coeffvar, September, 1997.

[102] I. Foster, A. Roy, and V. Sander, "A Quality of Service Architecture that Combines Resource Reservation and Application Adaptation," 8th International Workshop on Quality of Service, 2000.

[103] I. Foster, A. Roy, V. Sander, and L. Winkler, "End-to-End Quality of Service for High-End Applications," Technical Report, 1999 (http://www.globus.org/documentation/incoming/end_to_end.pdf)

[104] C. Lee, C. Kesselman, J. Stepanek, R. Lindell, S. Hwang, B. Scott Michel, J. Bannister, I. Foster, and A. Roy, “The Quality of Service Component for the Globus Metacomputing System,” Proc. IWQoS '98, 1998.

[105] W. Smith, V. Taylor, and I. Foster, “Using Run-Time Predictions to Estimate Queue Wait Times and Improve Scheduler Performance,” Proceedings of the IPPS/SPDP '99 Workshop on Job Scheduling Strategies for Parallel Processing, 1999.

[106] L. Yang, J. Schopf, and I. Foster, “Conservative Scheduling: Using Predicted Variance to Improve Scheduling Decisions in Dynamic Environments,” Supercomputing 2003, November 2003.

[107] M. Buchanan, “Power Laws and the New Science of Complexity Management,” Strategy+Business, Spring 2004.

[108] D. Takahashi, “IBM to invest $10 billion in ‘on demand’ computing,” San Jose Mercury News, Wed, Oct. 30, 2002, story reposted to http://www.siliconvalley.com/mld/siliconvalley/4408211.htm

[109] I. Foster, “What is the Grid? A Three Point Checklist,” Grid Today, July 20, 2002.

[110] I. Foster and C. Kesselman, The Grid: Blueprint for a New Computing Infrastructure, Morgan Kaufmann Publishers, Inc., 1999.

[111] L. Kleinrock, “UCLA to be first station in nationwide computer network,” 1969, http://www.lk.cs.ucla.edu/LK/Bib/REPORT/press.html

[112] I. Foster, C. Kesselman, and S. Tuecke, “The Anatomy of the Grid: Enabling Scalable Virtual Organizations,” International Journal of Supercomputer Applications, 15(3), 2001.

[113] S. Dowdy and S. Wearden, Statistics for Research, John Wiley & Sons, 1991.

[114] P. Samuelson, Foundations of Economic Analysis, Harvard University Press, 1958, 1983.

[115] D. Lane, HyperStat Online Textbook, http://davidmlane.com/hyperstat/z_table.html, 2005.