Meaning for the Masses: Theory and Applications for Semantic Web and Semantic Email Systems

Luke K. McDowell

A dissertation submitted in partial fulfillment of the requirements for the degree of

Doctor of Philosophy

University of Washington

2004

Program Authorized to Offer Degree: Department of Computer Science & Engineering

University of Washington

Graduate School

This is to certify that I have examined this copy of a doctoral dissertation by

Luke K. McDowell

and have found that it is complete and satisfactory in all respects,

and that any and all revisions required by the final

examining committee have been made.

Co-Chairs of Supervisory Committee:

____

Oren Etzioni

____

Alon Halevy

Reading Committee:

____

Oren Etzioni

____

Alon Halevy

____

Henry Levy

Date: ____________________________

In presenting this dissertation in partial fulfillment of the requirements for the Doctoral degree at the University of Washington, I agree that the Library shall make its copies freely available for inspection. I further agree that extensive copying of this dissertation is allowable only for scholarly purposes, consistent with "fair use" as prescribed in the U.S. Copyright Law. Requests for copying or reproduction of this dissertation may be referred to Bell and Howell Information and Learning, 300 North Zeeb Road, Ann Arbor, MI 48106-1346, to whom the author has granted "the right to reproduce and sell (a) copies of the manuscript in microform and/or (b) printed copies of the manuscript made from microform."

Signature____________________________

Date______________________________

University of Washington

Abstract

Meaning for the Masses: Theory and Applications
for Semantic Web and Semantic Email Systems

by Luke K. McDowell

Co-Chairs of Supervisory Committee:

Professor Oren Etzioni
Computer Science & Engineering

Professor Alon Halevy
Computer Science & Engineering

The Semantic Web envisions a portion of the World-Wide Web in which the underlying data is machine understandable and can thus be exploited for improved querying, aggregation, and interaction. However, despite the great potential of this vision and numerous efforts, the growth of the Semantic Web has been stymied by the lack of incentive to create content, and the high cost of doing so.

The goal of this dissertation is to *enable and motivate non-technical people to both utilize and contribute content for the Semantic Web.* As the foundation for our work, we identify three design principles that are essential for producing a successful Semantic Web system:

1. **Instant Gratification** — provide an immediate, tangible benefit to users.
2. **Gradual Adoption** — offer such benefit even when the system has few users.
3. **Ease of Use** — be simple enough for a non-technical person to use.

We then design mechanisms and theory that support these principles in the construction of two novel systems: MANGROVE, a community Semantic Web system, and Semantic Email, a system for leveraging declarative content to automate email-mediated tasks.

First, we describe MANGROVE's architecture and explain how its explicit publish and feedback mechanisms can provide instant gratification to content authors. In addition, we describe several novel semantic services that motivate the annotation of HTML content by consuming semantic information. We show how these services can provide tangible benefit to authors even when pages are only sparsely annotated. Furthermore, we demonstrate how seeding and inline annotation with our lightweight annotation syntax can bolster gradual adoption in MANGROVE.

Second, we introduce a paradigm for Semantic Email and describe a broad class of semantic email processes (SEPs). In support of instant gratification, these automated processes offer tangible productivity gains on a wide variety of email-mediated activities. To manage these processes, we define two formal models for specifying the desired behavior of a SEP. We show that computing the optimal message handling policies for these models is intractable in general, but identify key restrictions that enable these problems to be solved in polynomial time while still enabling a range of useful functionality. We then address a number of significant problems related to SEP usage by non-technical people. In particular, we design a high-level language for SEP templates that greatly simplifies the process of specifying and invoking a new SEP. In addition, we show that it is possible to verify, in polynomial time, that a given template will always produce a valid instantiation, and demonstrate how to generate explanations for the SEP's behavior in polynomial time. Finally, we describe how to meet our principles of gradual adoption and ease of use via a template-based semantic email server that functions seamlessly for participants with any mail client and with no *a priori* knowledge of semantic email.

Both systems have been fully implemented and deployed in a real-world environment, allowing us to report on practical experience gained with actual users. Overall, this work produces two novel, usable systems, as well as insights and techniques that can direct future Semantic Web systems.

TABLE OF CONTENTS

List of Figures **iv**

List of Tables **viii**

Chapter 1: Introduction **1**

1.1 Desiderata and Challenges for a Successful Semantic Web 2

1.2 Overview of the Solution 4

1.3 Technical Contributions 11

1.4 Outline of the Dissertation 13

Chapter 2: Background **14**

2.1 Enabling Technologies for the Semantic Web 14

2.2 Semantic Web Applications 16

2.3 Content Provision for the Semantic Web 29

2.4 Cross-cutting Issues 34

2.5 Discussion 37

Chapter 3: Mangrove **39**

3.1 The Architecture of MANGROVE 39

3.2 Semantic Services in MANGROVE 52

3.3 Experience with MANGROVE 59

3.4 Related Work 66

3.5 Summary 70

Chapter 4: Semantic Email **71**

4.1 Introduction 71

4.2 Semantic Email Processes . . . 73
4.3 Logical Model of SEPs . . . 76
4.4 Decision-theoretic Model of SEPs . . . 82
4.5 Implementation and Usability . . . 91
4.6 Experience . . . 97
4.7 Related Work . . . 98
4.8 Summary . . . 102

Chapter 5: Specifying Semantic Email Processes **103**

5.1 Introduction . . . 103
5.2 Overview of SEP Creation . . . 105
5.3 Concise and Tractable Representation of Templates . . . 105
5.4 Template Instantiation and Verification . . . 113
5.5 Automatic Explanation Generation . . . 118
5.6 Related Work . . . 127
5.7 Summary and Implications for Agents . . . 128

Chapter 6: Conclusions **131**

6.1 Contributions . . . 131
6.2 Future Directions . . . 135

Bibliography **141**

Appendix A: Mangrove Schema **160**

Appendix B: Semantic Email Declarations and Templates **166**

B.1 Interpretation of SEP Declarations . . . 166
B.2 Ontology for Describing SEP Templates . . . 169
B.3 Ontology for Describing SEP Parameter Descriptions . . . 180

Appendix C: Proofs **183**

C.1 Proof of Theorem 4.3.1 183

C.2 Proof of Theorem 4.3.2 184

C.3 Proof of Theorem 4.4.1 – bounded suggestions 186

C.4 Proof of Theorem 4.4.1 – unlimited suggestions 189

C.5 Proof of Theorem 4.4.2 192

C.6 Proof of Theorem 5.4.1 194

C.7 Proof of Theorem 5.4.2 195

C.8 Proof of Theorems 5.5.1 and 5.5.2 199

C.9 Proof of Theorem 5.5.3 199

C.10 Proof of Theorem 5.5.4 200

C.11 Proof of Theorem 5.5.5 202

C.12 Proof of Theorem 5.5.6 202

LIST OF FIGURES

1.1 The MANGROVE architecture and sample services. Authors produce structured content using our annotation tool, then explicitly *publish* this content. Published data is immediately stored in the RDF database and available to a range of useful semantic services. In addition, registered services are notified about new data publications, enabling them to immediately update their outputs and provide feedback to the content author. 5

2.1 A taxonomy of Semantic Web applications. At the highest level, the applications are classified as either *information-providing* or *action-oriented*. While both categories have received significant research attention, significantly more information-providing applications have been deployed for actual usage. 16

2.2 Sample search applications: SHOE PIQ (left) and QuizRDF (right). PIQ "semantic-only" queries (shown in the top pane) are constructed using a complex graphical interface, while QuizRDF "semantic+text" queries use a combination of keywords and selection boxes. 17

2.3 Semantic browsers and portals: The Conzilla browser (left) and the KA2 community web portal (right). The former requires a special tool to navigate among the displayed concepts, while the latter produces HTML that can be viewed with a normal browser. 19

2.4 SHOE's TSE Path Analyzer (left) and the Snippet Manager (right), showing a collection of bookmarks. 22

2.5 The TRELLIS application, showing a user reviewing the justification for an earlier decision. 23

2.6 RCal (left), showing a schedule imported by the user, and ITtalks (right), showing a summary of relevant talks. 25

2.7 A screenshot from Sirin et al.'s program to compose multiple services, here constructing a foreign language translator. This program is freely downloadable (though not directly executable) from the web. 27

3.1 The MANGROVE architecture and sample services. Semantic Email (Chapters 4 and 5 is implemented as a MANGROVE service in order to facilitate interoperability with other MANGROVE services. 40

3.2 Example of HTML annotated with MTS tags. The `uw:` tags provide semantic information without disrupting normal HTML browsing. The <`reglist`> element specifies a regular expression where '`*`' indicates the text to be enclosed in MTS tags. 42

3.3 The MANGROVE graphical annotation tool. The pop-up box presents the set of tags that are valid for annotating the highlighted text. Items in gray have been tagged already, and their semantic interpretation is shown in the "Semantic Tree" pane on the lower left. The user can navigate the schema in the upper left pane. 44

3.4 Example output from the service feedback mechanism. Services that have registered interest in a property that is present at a published URL are sent relevant data from that URL. The services immediately return links to their resulting output. 47

3.5 The calendar service as deployed in our department. The popup box appears when the user mouses over a particular event, and displays additional information and its origin. For the live version, see `www.cs.washington.edu/research/semweb`. 52

3.6 The semantic search results page. The page reproduces the original query and reports the number of results returned at the top. Matching pages contain the phrase "assistant professor" and the properties <facultyMember> and <portrait>. The `?` in the query instructs the service to extract the <portrait> from each matching page. 55

3.7 The Who's Who service as deployed in our department. Notice how it allows users to provide as much information as they like, in whatever format is desired. 56

3.8 The number of distinct visits to the MANGROVE calendar during each month. These values exclude traffic from webcrawlers and MANGROVE team members. 64

4.1 The invocation and execution of a SEP. The originator is typically a person, but also could be an automated program. The originator invokes a SEP via a simple web interface, and thus need not be trained in the details of SEPs or even understand RDF. 74

4.2 A web form used to initiate a "balanced collection" process, such as our balanced potluck example. For convenience, clicking submit converts the form to text and sends the result to the server and a copy to the originator. The originator may later initiate a similar process by editing this copy and mailing it directly to the server. . 92

4.3 A message sent to participants in a "balanced potluck" process. The bold text in the middle is a form used for human recipients to respond, while the bold text at the bottom is a RDQL query that maps their textual response to RDF. 96

5.1 The creation of a Semantic Email Process (SEP). Initially, an "Author" *authors* a SEP template and this template is used to *generate* an associated web form. Later, this web form is used by the "Originator" to *instantiate* the template. Typically, a template is authored once and then instantiated many times. 105

5.2 SEP template for a "Balanced Potluck" process. The template is shown in N3 format [16], which is an alternative syntax for writing RDF. Variables in bold (e.g., **$Choices$**) are parameters provided by the originator when instantiating the template. Other variables are defined inside the declaration (e.g., `$x$, $TotalGuests$`) or are automatically computed by the system (e.g., `$Bringing.acceptable()$`). . 110

5.3 Part of a *parameter description* for the potluck template of Figure 5.2. Additional elements for variables such as `MaxImbalance` are not shown. 114

5.4 Examples of proof trees for rejecting response r. Each node is a possible state of the data set, and node labels are constraints that are *not* satisfied in that state. In both cases, response r must be rejected because every leaf node (shaded above) does not satisfy some constraint. 123

LIST OF TABLES

2.1 Different types of search. Textual inputs are generally based on keywords, while semantic inputs may be a combination of text and tags [41, 125], derived from a form [82, 121], or constructed graphically [83]. Either type of input may then be used to construct an output based on textual and/or semantic sources. 18

2.2 Types of semantic browsers and portals. Figure 2.3(left) demonstrates the "Object view" type of output. 20

2.3 The timeliness of benefit from authoring semantic content. 32

2.4 Techniques for making inference practical. 35

3.1 Comparison of Search Services. In each box, the first value is the *f-score* of the query, followed by the *precision* and *recall* in parentheses. Within each row, the values in bold represent the maximum value for that metric. 62

4.1 Summary of theoretical results for D-SEPs. The last two columns show the time complexity of finding the optimal policy for a D-SEP with N participants. In general, this problem is EXPTIME-hard but if the utility function is *K-partitionable* then the problem is polynomial time in N. (An MDP can be solved in time guaranteed to be polynomial in the number of states, though the polynomial has high degree.) Adding restrictions on how often the manager may send suggestions makes the problem even more tractable. Note that the size of the optimal policy is finite and must be computed only once, even though the execution of a SEP may be infinite (e.g., with "AnyUnlimited"). 88

5.1 Trigger conditions for a SEP notification. 108

5.2 Comparison of the size (in number of lines) of different ways of specifying a SEP. For the procedural prototype, the first numerical section displays the size of the Java code for encoding the SEP functionality, size of the HTML for acquiring parameters from the originator, and the total of these two. For the declarative approach, the second section displays the size of the template (OWL, in N3 format), size of the parameter description (see Section 5.4), and the total. The final column shows the percentage reduction in the size of a SEP when changing from the procedural approach to the declarative approach. 113

6.1 Summary of the roles of each person (or other agent) involved in the execution of an agent, and how they would benefit from a high-level, declarative template language with safety testing and explanation generation. This table shows that these types of features could benefit a broad range of agent systems, both email-based and otherwise. 134

ACKNOWLEDGMENTS

I am indebted to many people for their guidance and support along the path to this dissertation. First, I'd like to thank my two thesis advisors, Oren Etzioni and Alon Halevy. Both were invaluable in helping me to learn to research and to understand a new field. They joked that it was tough to have two opinionated advisors, but actually they complemented each other extremely well. Oren forced me to sharpen my ideas and always pushed me to say more with fewer words. His advice and direction taught me much about life, research, and the academic world. Alon helped me to navigate the realities of the Semantic Web and its relation to the database world. His guidance and willingness to suffer through the long details of the proofs were critical to the theoretical aspects of this dissertation. I also owe Alon a special thanks for challenging me to finish this dissertation far before I had originally thought possible.

I was fortunate enough to work with many other talented faculty at the University of Washington. Susan Eggers was my first advisor at UW. Susan's enthusiasm for the department and Seattle was a big factor in my decision to come to UW, and her keen attention to detail and vision for the big picture were indispensable in my early research in Computer Architecture. Steve Gribble and Hank Levy were involved in both my early research with Susan as well as with the later work that would become my dissertation. Steve's insight and explanations were always helpful, particularly in our many discussions with Susan about processor scheduling and performance effects. Hank's involvement ranged from my first work on speculation for SMT processors to the formation of Semantic Email. Hank was both a valuable critic and a great encourager of my ideas, and I am grateful for all his help.

Thanks to the many faculty from the Princeton University Electrical Engineering department, who encouraged my study of computers. Special thanks to Margaret Martonosi,

who advised my independent work and helped me to decide on graduate studies in Seattle. I'm also grateful to Bede Liu for his advice and for guiding me to that first job after graduation.

I've been blessed with many friends that made my time at the University of Washington so enjoyable. Donald Patterson and Gerome Miklau have been my closest companions since the first days of Automata, through all the ups and downs of research, life, and parenthood. Thanks for the many lunches, laughs, and not infrequent advice. Thanks also to the many other friends and officemates from the department who have given countless hours of help with courses, research, and practice talks. I owe a special debt to Doug Zongker for invaluable help with LaTeX, images, and all things Linux.

Family has always been a great help to me. On the east coast, thanks to my parents and siblings for all the love and support over the years. Farther west, I offer my thanks and much love to my wife Sophie, for the many wonderful years so far. We came to Seattle so that we could pursue further study together, and she has truly blessed me in this time with constant help, encouragement, and love. I also thank my son Ryan, who was born just as I began the research for this dissertation — on the very day, in fact, that the first MANGROVE poster submission was due. It's been a joy to watch Ryan grow up as my own research has matured. As we eagerly await the birth of our second child (but hopefully not before I get this dissertation to the graduate school!), I look forward to many new beginnings with my family by my side.

Most importantly, I thank and praise my Savior Jesus Christ, who has given me the ability to complete this task. May all the glory be given unto Him.

Not to us, O Lord, not to us,
but to your name be the glory,
because of your love and faithfulness.
(Psalm 115:1)

Chapter 1

INTRODUCTION

> The Semantic Web is an extension of the current web in which information is given well-defined meaning, better enabling computers and people to work in cooperation.[15]

In the Scientific American article quoted above, Berners-Lee et al. propose a future version of the World Wide Web (WWW) in which the underlying data is machine understandable and applications can exploit this data for improved querying, aggregation, and interaction. Despite significant amounts of effort, however, the "Semantic Web" has yet to achieve widespread impact. In particular, only a very small fraction of the people familiar with the WWW have ever used a Semantic Web application, and even fewer people have contributed any of the content that is needed to make the Semantic Web truly useful.

This dissertation examines how to make the Semantic Web a reality. More specifically, our goal is to *enable and motivate non-technical people to participate in the Semantic Web.* This chapter describes why this particular goal is both significant and challenging, and outlines our proposed solution. We begin in Section 1.1 by describing some desiderata for a successful Semantic Web, explaining why non-technical persons are so important to its success, and defining what we mean for such persons to *participate*. We also highlight a number of challenges that arise when trying to meet our goal. Next, Section 1.2 outlines our solution to these challenges. We first propose three design principles that a Semantic Web system must satisfy in order to achieve participation by non-technical people. We then introduce two novel, deployed Semantic Web systems: MANGROVE and Semantic Email. These systems are used to demonstrate the importance of the design principles and to explore the concrete mechanisms needed to support these principles in real systems, a task

that is continued in Chapters 3 to 5. Finally, we summarize the technical contributions of this work in Section 1.3 and give an outline of the rest of the dissertation in Section 1.4.

1.1 Desiderata and Challenges for a Successful Semantic Web

A successful Semantic Web must minimally consist of useful *applications*, sufficient *content*, and many *participants*. Clearly an information system without interesting applications is of little practical use, as is a system without enough content to support its applications. Likewise, to have noticeable impact a system must attract a significant number of participants, a requirement that precludes appealing only to technically-trained people.

In addition, in this work we will assume two additional desiderata that, while not strictly necessary for achieving a successful system, can greatly increase the applicability and usefulness of the system. First, the system should support *scalability*, both in terms of amounts of data and numbers of users. While practical, small-scale Semantic Web systems can be developed for local intranets or particular communities, scalability supports our aim of having many participants and enables the system to exploit new data sources as they arise. Second, we assume that a Semantic Web system will be based on *declarativism*. Representing data and services declaratively (i.e., as axioms or rules, rather than as procedures or data structures) can greatly simplify the design of the system. More importantly, declarativism may enable interoperability with other information systems (e.g., to enable data reuse in other contexts) and facilitate automated reasoning (e.g., to infer useful information that was not explicitly stated).

Thus, to succeed the Semantic Web should not only provide interesting, scalable, and declarative applications and content, but must readily accommodate participants with limited technical sophistication. Participation can of course take many forms. In the simplest case, non-technical people might just *utilize* the Semantic Web for specific purposes, much as many people today access powerful search services via Google or financial databases via an automated teller machine (ATM). On the other hand, such people might also *contribute* declarative content to the Semantic Web, just as millions have already created web pages. This represents a formidable goal, but the growth of the web demonstrates that, with

sufficient tools and motivation, average people can accomplish what would have seemed far-fetched just fifteen years ago. Furthermore, given the expense and challenge of obtaining semantic content in other ways, any system that does not actively include non-technical content contributors is certain to miss out on a large body of valuable information.

These desiderata are inter-related: more content makes applications more useful, and useful applications attract more participants. More participants may produce more content, both of which increase the need for scalability and increase the potential interoperability gains enabled by declarativism. We focus in this dissertation on appealing to the large number of non-technical participants who must be enticed to take part in the Semantic Web. Or, to restate our goal more precisely, we wish to *enable and motivate non-technical people to both utilize and contribute content for the Semantic Web.* Achieving this goal will necessarily entail significant work related to obtaining scalable, declarative applications and content, but a focus on enabling and motivating *participants* will be our central theme.

Unfortunately, achieving this goal presents a number of challenges. Most significantly, there are currently very few applications and very little content for the Semantic Web, and hence little motivation to explore Semantic Web applications or to author declarative content. In addition, even if authors were so motivated, authoring such content can be very challenging for non-technical people because of the complex languages involved, the lack of simple, ubiquitous authoring tools, and the need to understand how their data relates to an existing ontology. In essence, authoring is complicated by a fundamental "chasm" [76] between the structured content required by declarative systems and the unstructured content (e.g., text files, spreadsheets) that is familiar to most users. Moreover, data often evolves over time, and this evolution imposes an ongoing maintenance cost on its authors to ensure that the declarative data remains consistent with its original form. In sum, the cost is too high and the benefit is too low to persuade non-technical people to participate in the Semantic Web. The next section will introduce our solution to these challenges.

1.2 Overview of the Solution

We posit that a successful Semantic Web system must address the above challenges by adhering to the following three design principles:

- **Instant Gratification**: Most importantly, the system must provide an immediate, tangible benefit to users for both utilizing its applications and for contributing content. Applications must have a clear, useful purpose and sufficient content. Likewise, content creation should be motivated by an application that immediately consumes the content and yields satisfaction to the author, not by some potential future benefit — as when the consuming application must await a web crawl [44, 83] or may not exist at all (e.g., MailSMORE [98]).

- **Gradual Adoption**: The system must be highly useful even when only a small number of people have adopted the technology. In particular, applications must provide enough data to make them initially valuable, and must not rely exclusively on network effects that are not initially present. Likewise, individual users should not have to fully commit to the system before incurring any benefits from the technology. Instead, systems should allow users to begin without any software installation and permit content to be structured and contributed incrementally over time.

- **Ease of use:** The entire system must be simple enough for a non-technical person to use. It should not expect such users to understand declarative languages (e.g., RDF [107]), require the use of complex tools, or insist that all data obey a set of integrity constraints.

We have applied these principles to the design, implementation, deployment, and evaluation of two distinct systems: MANGROVE, a community Semantic Web system, and Semantic Email, a system for leveraging declarative content to automate email-mediated tasks. Below we summarize these systems and elaborate on their application of the design principles.

1.2.1 MANGROVE

MANGROVE is a system that provides novel semantic services that are intended to motivate specific communities of people to annotate their existing content from the web. For instance,

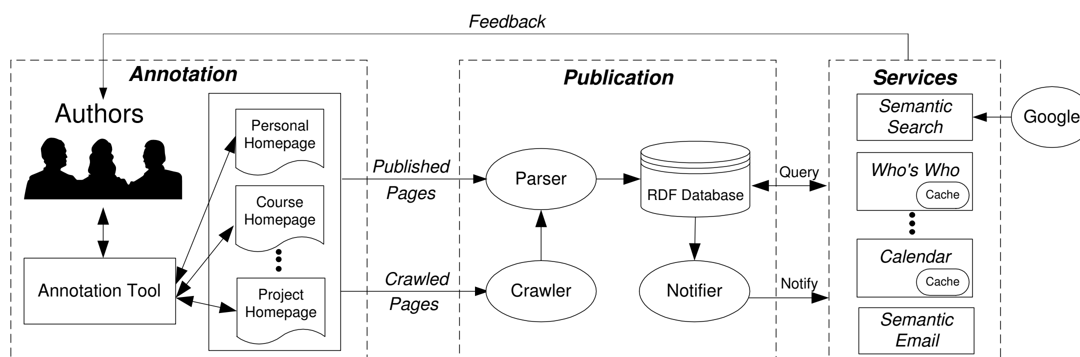


Figure 1.1: The MANGROVE architecture and sample services. Authors produce structured content using our annotation tool, then explicitly *publish* this content. Published data is immediately stored in the RDF database and available to a range of useful semantic services. In addition, registered services are notified about new data publications, enabling them to immediately update their outputs and provide feedback to the content author.

consider the web site of our computer science department. The web pages at this site contain numerous facts including contact information, locations, schedules, publications, and relationships to other information. If users were enabled and motivated to semantically annotate these pages, then the pages and annotations could be used to support both standard HTML-based browsing as well as novel semantic services. For example, we have created a departmental calendar that draws on annotated information found on existing web pages, which describe courses, seminars, and other events. Because the calendar is authoritative and prominently placed in the department's web, events that appear in it are more likely to receive the attention of the department's community. As a result, people seeking to advertise events (e.g., seminars) are motivated to annotate their pages, which leads to their automatic inclusion in the department's calendar and in the Semantic Web.

Figure 1.1 shows the architecture of MANGROVE organized around the following three phases of operation. First, in the annotation phase, authors use our *graphical annotation tool* or an editor to insert declarative annotations into existing HTML documents. Second, in the publication phase, authors can explicitly *publish* annotated content, causing the *parser* to immediately parse and store the contents in an *RDF database*. The *notifier* then passes information about relevant updates to registered services, who may send *feedback* that informs authors of how their data was used and of any errors that were encountered. Finally, in the services phase, newly published content is immediately available to a range

of *services* that access the content via RDF queries. Because the data and queries are represented declaratively, content may be authored with one service in mind but utilized by a range of different services. For example, a page may be annotated for MANGROVE's Who's Who service but then provide personal information that is usable by the departmental calendar.

This architecture and our novel semantic services support the design principles as follows:

- **Instant Gratification:** In the HTML world, a newly authored page is immediately accessible through a browser. We mimic this feature in MANGROVE by making annotated content instantly available to services via the explicit publish mechanism. We posit that semantic annotation will be motivated by services that consume the annotations and result in immediate, tangible benefit to authors. MANGROVE provides a number of such services, including the department calendar, a publications database, and a novel search service that combines declarative and non-declarative information. In addition, the service feedback mechanism ensures that authors can immediately locate the results of their work and correct errors as necessary. Note that our explicit publish mechanisms and registration of services for such feedback enables MANGROVE to provide this immediate response in a much more scalable manner than would be possible in systems based on periodic web crawls (e.g., [44, 83]).

- **Gradual Adoption:** MANGROVE provides significant "seeding" of its services to make them highly useful even when very few users have contributed content to the system. In addition, MANGROVE services are all accessible via an unmodified commodity web browser. For content creation, we designed MTS (the MANGROVE Tagging Syntax), an "inline" annotation syntax that allows existing content to be annotated incrementally and in a way that is resilient to changes on the underlying data. This is in contrast to languages such as RDF that force existing content to be duplicated in order to be annotated, creating a future maintenance burden [125, 81].

- **Ease of Use:** MANGROVE services use simple abstractions like calendars, lists, and text queries that are already familiar to web users. For content creation, our graphical

web-page annotation tool enables users to easily annotate existing HTML content. In addition, to ease semantic authoring MANGROVE does not require authors to obey integrity constraints, such as data uniqueness or consistency. Data cleaning is deferred to the services that consume the data. Furthermore, MANGROVE maintains and displays the data provenance (i.e., source URL) of every piece of data that is used in a service. This provides a simple, lightweight method for dealing with issues of trust, similar to how users ascertain the reliability of information on the web today.

MANGROVE has been deployed in our department for almost two years, permitting us to make a number of observations about its impact. First, we found that simple services such as the calendar can offer substantial added value compared to other methods of accessing the same information. For instance, the MANGROVE calendar quickly became a popular service after it was introduced, despite the fact that identical information was almost always available elsewhere on the web. Second, we found that users may be willing to annotate their existing documents if the process is easy and interesting services exist to use the annotations. For instance, in less than two weeks thirty graduate students made the effort to structure and submit their personal information so that it could be used in MANGROVE's version of a new departmental Who's Who service.

Chapter 3 describes MANGROVE's architecture more completely and examines the mechanisms that support our three design principles. We also discuss MANGROVE's initial semantic services and report in more detail on experience gained with this system.

1.2.2 Semantic Email

Semantic Email is a system that motivates people to add basic declarative content to some of their email messages in order to obtain automated processing of and reasoning with their mail. Just as MANGROVE seeks to alter the cost/benefit equation related to the structuring of web data, Semantic Email identifies a particular pain point where the addition of some structured content can have a large impact. Like the WWW, email is a vast information space where people spend significant amounts of time, yet that typically has no semantic features (aside from generic header fields). While the majority of email will remain this

way, we argue that adding semantic features to email offers opportunities for improved productivity while performing some very common tasks.

Consider several examples:

- **Information Dissemination:** Sending a talk announcement via email could also result in posting the announcement to a talks web site and sending a reminder the day before the talk.

- **Event Planning:** Imagine sending mail asking the members of a program committee their preferences for the PC dinner, and having semantic email automatically tabulate the responses, periodically reminding those that have not responded.

- **Report Generation:** Consider asking a set of managers for projected budgets and having the email system automatically tabulate the responses, possibly requiring the values to satisfy certain individual or aggregate constraints.

- **Auction/Giveaway:** Imagine sending an email announcement offering to give away (or auction) some tickets that you cannot use. The semantic email system could give out the tickets to the first respondents, then politely respond to subsequent requests when all tickets are claimed.

Because email is not set up to handle these types of tasks effectively, accomplishing them manually can be tedious, time-consuming, and error-prone. Consequently, we designed a novel, general model of *semantic email processes* (SEPs). These processes support the common task where an *originator* wants to (1) ask a set of *participants* some questions, (2) collect their responses, and (3) ensure that the results satisfy some set of *goals*. In order to satisfy these goals, the SEP *manager* may utilize a number of *interventions* such as rejecting a participant's response or suggesting an alternative response.

Our aim in this work is to sketch a general infrastructure for SEPs and to analyze the inference problems that semantic email needs to solve to manage processes effectively and guarantee their outcome. This leads to two primary challenges.

First, how can the manager automatically pursue a wide variety of goals on the originator's behalf, and do so in a way that scales to support a large number of originators,

participants, and goals? To address this challenge, Chapter 4 defines and explores two useful models for specifying the goals of a process and formalizing when and how the manager of the process should intervene. In the logical model, the originator specifies a set of *constraints* over the data set that should be satisfied by any process outcome. This model is intuitive, but suffers from an inability to strive for partially satisfied goals and a disregard for the costs of its interventions. In the decision-theoretic model, we address these shortcomings via a probabilistic framework where the goal of a SEP is a function representing the *utility* of possible process outcomes. Both models are useful in different situations; Chapter 4 considers their relative strengths in more detail.

For both models we analyze several important and practical reasoning problems. In particular, we show that, for the logical model, the problem of determining if a response is acceptable with respect to the constraints is NP-complete in the number of participants (Theorem 4.3.1), and that, for the decision-theoretic model, the corresponding problem of determining the optimal message handling policy is PSPACE-hard or worse (Theorem 4.4.1). These are significant limitations, since the manager must solve one of these problems to decide when to intervene in a SEP, and for many SEPs it is natural to wish to scale to large numbers of participants. Consequently, we identify suitable restrictions on SEPs that retain enough power to express all the examples given previously, but that enable tractable reasoning. Intuitively, these restrictions — bounded constraints (Definition 4.3.5) and K-partitionable utilities (Definition 4.4.1) — capture the notion that for many SEPs what matters is the number of people whose responses belong to a fixed number of groups (e.g., how many responded `Yes`?), rather than the specific responses of each participant. We show that these restrictions enable reasoning that is polynomial time in the number of participants, both for the logical (Theorem 4.3.2) and decision-theoretic models (Theorem 4.4.2). Collectively, this analysis gives a solid theoretical footing to SEPs and enables a host of practical reasoning.

Second, leveraging the theory above requires a formal, declarative SEP specification to execute. How can a non-technical user easily create such a specification that corresponds to his goals? Chapter 5 tackles this challenge with a solution based on SEP *templates*. This approach shifts most of the complexity of SEP specification from untrained originators onto

a much smaller set of trained authors, but also raises the problems of generality, safety, and understandability. To address *generality*, we present a high-level, declarative language for specifying SEP templates. This language vastly simplifies the task of creating new SEPs, shrinking the size of such specifications by 80%-90% compared to an original procedural prototype. In addition, this language provides a number of features such as quantification, guards, and set manipulation that make it easier for a single SEP template to be applicable in many different situations. This flexibility, however, can lead to potential *safety* problems. In particular, to avoid frustrating the originator we must ensure that every possible instantiation of a template is executable, yet checking this property manually can be very difficult for the SEP author. We formally define this problem of verifying *instantiation safety* and show that it is NP-complete (Theorem 5.4.1). However, we also show that by applying a few reasonable restrictions (e.g., to restrict the number and type of quantifications), this problem can be solved in time polynomial in the size of the template (Theorem 5.4.2). Finally, for *understandability*, we examine how to automatically generate explanations for the manager's interventions in terms of *why* a particular response could not be accepted and *what* responses would be more acceptable. We show that, while the general case is NP-hard or worse (Theorems 5.5.1 and 5.5.4), if we restrict the goals to be bounded/K-partitionable and the explanations to be of bounded size, then these computations can be solved in polynomial time (Theorems 5.5.2, 5.5.3, 5.5.5, and 5.5.6). Together, our declarative template language and associated theory vastly simplifies the specification of SEPs and enables their safe, explainable execution.

Semantic Email is another instance of a system that demonstrates the usefulness of our three design principles:

- **Instant Gratification**: Users are not expected to annotate outgoing or incoming mail for some vague future benefit. Instead, we provide untrained users with existing, useful SEPs that can be immediately invoked and yield a tangible output in the form of messages sent and processed on the their behalf. Our formal analysis permits this processing to scale to complex goals involving many participants.

- **Gradual Adoption**: At first, semantic email will be initiated by only a small number of "early adopters." If semantic email could be profitably exchanged only among these users, it would have very limited applicability. Instead, we allow the originator to include any person in the execution of a SEP, regardless of whether they have ever even heard of semantic email. In addition, a SEP can be originated by anyone via a simple web interface, without the need to install any software.

- **Ease of use**: We provide SEP templates that encapsulate common email tasks, making it easy for anyone to originate a new SEP without any programming or understanding of the system's internals. In addition, email messages that are exchanged contain simple text forms that can be handled by any email client and without any knowledge of RDF. Finally, our formal analysis allows the system to automatically perform useful inferences, such as calculating the set of responses which are currently acceptable with respect to the process's goals, in order to aid participants in deciding upon their responses.

Note that SEPs effectively integrate the processes of application usage and content creation. In particular, the originator creates declarative content (in terms of participants, choices, and goals) while making a request for an application service (e.g., a SEP execution). Likewise, participants respond to SEP requests in a fashion similar to how they would respond to any email request, but in such a way that their responses become interpretable in RDF. This RDF content may be helpful for future reasoning tasks (see Chapter 4). Thus, any person involved in the execution of a SEP automatically becomes a content contributor to some degree, accomplishing a significant part of our goal.

1.3 Technical Contributions

Despite significant effort, the Semantic Web has yet to achieve widespread participation by non-technical people. This dissertation makes an important step towards meeting that goal. As the foundation for our work, we identify three design principles that are essential for producing a successful Semantic Web system: instant gratification, gradual adoption, and ease of use. Satisfying these principles ensures that the system's usage and content creation are well motivated, that the system is beneficial and poised to grow even in its infancy, and

that the entire system is accessible to non-technical people. We then apply these principles in two fully-deployed systems to make the following contributions:

- **Mangrove:** We describe the MANGROVE architecture that supports the complete Semantic Web "life-cycle" from content authoring to Semantic Web services. We demonstrate how elements of the architecture support each of our three design principles. In particular, we demonstrate how explicit publish and feedback mechanisms can provide instant gratification, how seeding and inline annotation with MTS can bolster gradual adoption, and how simple interfaces for content creation and service invocation can support ease of use. Furthermore, we describe several novel semantic services that motivate the annotation of HTML content by consuming semantic information. We show how these services can provide tangible benefit to authors even when pages are only sparsely annotated. These are some of the first "semantic services" that are invoked by ordinary users as part of their daily routine.

- **Semantic Email:** We introduce a paradigm for semantic email and describe a broad class of semantic email processes. To support these SEPs, we introduce two formal models for specifying the goals of a process. In both cases we show that the core reasoning problems of determining when to intervene in a process is NP-hard or worse, but that reasonable restrictions can enable each problem to be solved in polynomial time. We also describe how to automatically generate explanations for these interventions and identify cases where such explanations can be computed in polynomial time. In addition, we define a declarative language for specifying SEPs that vastly simplifies the task of creating general SEP templates. For these templates, we show the computational complexity of the important problem of instantiation safety, and demonstrate conditions under which this problem can be solved in polynomial time. These capabilities all support instant gratification by enabling automated, goal-directed, explainable, and scalable processing of messages on the originator's behalf. Finally, we describe how to meet our principles of gradual adoption and ease of use via a template-based semantic email server that functions seamlessly for participants with any mail client and with no *a priori* knowledge of semantic email.

In addition, our work on the specification of general, safe, and explainable SEP templates contributes to the field of intelligent agents. For instance, almost any agent needs some capability to explain its behavior, and many agents react to the world based on constraints or expected utilities. We show that generating explanations can be NP-hard in general, but that the combination of simple explanations and modest goal restrictions may enable explanation generation in polynomial time. Likewise, an agent template should support a wide range of functionality, yet ensure the *safety* of of each possible use. We motivate the need for one such type of safety (instantiation safety), show how to verify it efficiently, and highlight the need to carefully balance flexibility in the template language with the tractability of such verification.

1.4 Outline of the Dissertation

The next chapter provides some background on the Semantic Web, focusing particularly on applications that have been previously described in the literature and different methods for obtaining semantic content to support these applications. Chapter 3 describes the MANGROVE system and its application of our design principles, while Chapter 4 serves the same purpose for Semantic Email. Chapter 5 describes our language for specifying semantic email processes and analyzes the three challenges of generality, safety, and understandability that arise in this context. Related work is discussed within each chapter as appropriate. Finally, Chapter 6 concludes and considers directions for future work. The appendices contain proofs for all of the theorems given in the body of the dissertation and some additional technical details on MANGROVE and Semantic Email.

Parts of this dissertation have been previously published. MANGROVE (Chapter 3) is described in a ISWC-2003 paper [126] and more briefly in a CIDR-2003 paper [76]. Semantic Email 4 (Chapter 4) has been described in a WebDB workshop paper [56] and a WWW-2004 paper [127]. Finally, parts of our language for specifying semantic email processes and our analysis of automatic explanation generation (Chapter 5) were described in a workshop paper [129] and will appear as a ISWC-2004 paper [128].

Chapter 2

BACKGROUND

This chapter provides some background on the Semantic Web and its supporting technologies, focusing particularly on proposed applications and content generation methods. Where appropriate, we describe how existing systems have or have not complied with the three design principles introduced in Chapter 1. First, Section 2.1 provides a very brief overview of technology and standards that could underly the Semantic Web. In Section 2.2, we introduce a taxonomy of Semantic Web applications, describe existing implementations of these ideas, and examine important factors in making these applications usable by non-technical people. Section 2.3 then examines content creation. We first discuss creation via automated techniques and technical users, because this can be an important part of making applications initially useful, then examine creation by non-technical users in more detail. Next, Section 2.4 discusses cross-cutting issues such as inference and trust management that impact both application usage and content creation. Finally, Section 2.5 summarizes the implications of this survey and its relation to the remainder of the dissertation.

2.1 Enabling Technologies for the Semantic Web

Recall the definition of the Semantic Web from the previous chapter:

> The Semantic Web is an extension of the current web in which information is given well-defined meaning, better enabling computers and people to work in cooperation. [15]

The remainder of this chapter discusses a number of applications of such cooperation and examines how to produce the needed supporting information. This section briefly describes how to express and manipulate information with such "well-defined" meaning.

Recent Semantic Web systems are almost always based on the standard language RDF [107]. RDF knowledge can be written down in many ways, but the most common uses a XML serialization. For instance, the following RDF example states that there is a `Person` whose `name` is "John Meyers" and whose `mbox` (personal mailbox) is "myers@tennis.org."

```
<rdf:RDF xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
  xmlns:foaf="http://xmlns.com/foaf/0.1/">
  <foaf:Person>
     <foaf:name>John Myers</foaf:name>
     <foaf:mbox>myers@tennis.org</foaf:mbox>
  </foaf:Person>
</rdf:RDF>
```

In this dissertation, we refer to data expressed in RDF as *semantic* data (though occasionally we will refer to its *declarative* approach). Such data has two key properties. First, it represents the data in a logical manner. For instance, the above snippet makes a number of abstract statements with a subject, predicate, and object, rather than specifying particular data structures or files. Second, RDF data is (at least potentially) ontology-based. Specifically, RDF permits users to refer to known schemas or ontologies that can define shared terminology such as `name` and `mbox`. Such ontologies are referred to via a unique namespace, e.g., `http://xmlns.com/foaf/0.1/`. Ontologies provide a shared understanding that, together with the declarative data representation, can potentially enable different systems to utilize and understand data written by different users who did not communicate but who both chose to use the same ontology. Of course, dealing with problems that arise when terms are used inconsistently or different terms refer to the same concept is a significant challenge [48, 77].

A number of additional points regarding RDF are relevant. First, the above snippet demonstrates that RDF syntax is somewhat verbose and difficult to write by hand. Second, RDF can be embedded inside HTML documents, but representing parts of existing HTML content in RDF requires that such content be replicated in separate HTML and RDF sections [125, 81]. Third, RDF schemas permit the description of only fairly basic ontological relationships, e.g., subclass, subproperty, domain, and range restrictions. Languages such as

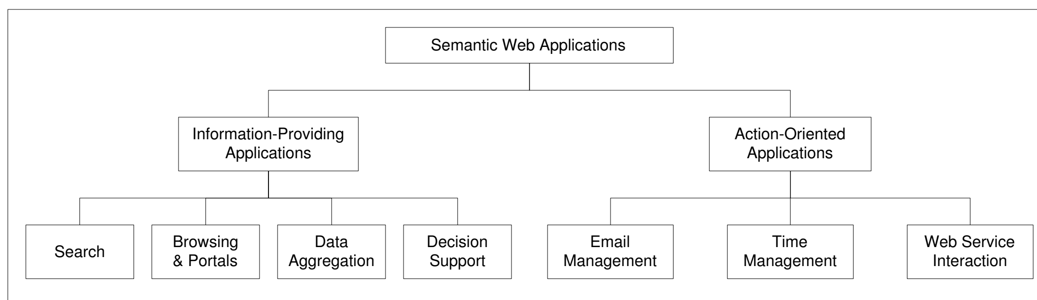


Figure 2.1: A taxonomy of Semantic Web applications. At the highest level, the applications are classified as either *information-providing* or *action-oriented*. While both categories have received significant research attention, significantly more information-providing applications have been deployed for actual usage.

DAML+OIL [86] and OWL [178] build on top of RDF to allow more expressiveness. Finally, because RDF and OWL are widely accepted standards, there is the potential for significant reuse of data and tools based on these languages. Indeed, freely available tools such as Jena [123] and Sesame [27] vastly simplify the task of creating Semantic Web applications.

2.2 Semantic Web Applications

Figure 2.1 presents a taxonomy of Semantic Web applications, divided broadly into *information-providing* applications and *action-oriented* applications. Each category is further broken down into three or four subcategories. While not covering all possible or proposed applications, this taxonomy does include the vast majority of applications that have been discussed in the literature. Below we elaborate on each category in turn.

2.2.1 Information-providing Applications

This section describes applications that present information but do not perform world-altering actions on behalf of the user.

Semantic Search

Most Semantic Web systems have provided some sort of search capability. Initially, such applications were "semantic-only," e.g., they searched based only on

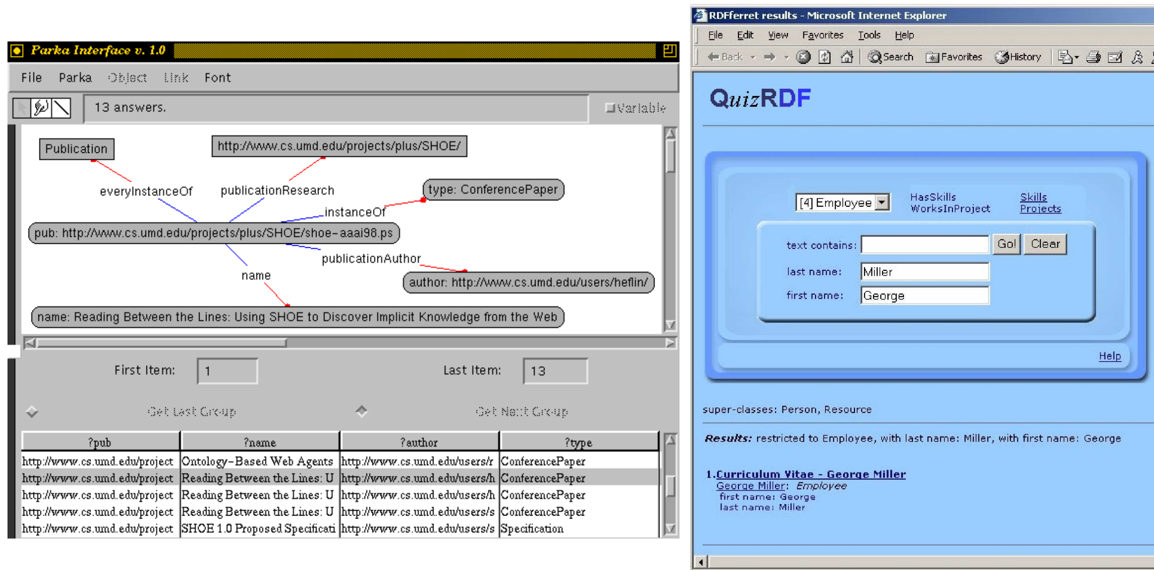


Figure 2.2: Sample search applications: SHOE PIQ (left) and QuizRDF (right). PIQ "semantic-only" queries (shown in the top pane) are constructed using a complex graphical interface, while QuizRDF "semantic+text" queries use a combination of keywords and selection boxes.

fully structured queries and data [83, 44, 120]. For instance, the SHOE PIQ query in Figure 2.2(left) requests every `Publication` whose `publicationResearch` is `http://www.cs.umd.edu/projects/plus/SHOE/`. A key problem, however, was that given the very low coverage of existing semantic knowledge bases, there was likely to be no semantic content related to a given query. In response, many applications have sought to augment semantic search with some type of text-based search such as Google:

1. **Semantic+text backup:** Several systems added the feature to either automatically [142] or easily [82] reformulate the semantic (e.g., structured) query and send it to a standard text-based search engine.

2. **Semantic+text union:** Recently, TAP Search [74] proposed the inverse. Instead of starting with a semantic query, TAP accepts standard textual queries from the user and attempts to automatically construct a reasonable semantic query. The search results then consist of the union of independently-executed textual and semantic search results.

Table 2.1: Different types of search. Textual inputs are generally based on keywords, while semantic inputs may be a combination of text and tags [41, 125], derived from a form [82, 121], or constructed graphically [83]. Either type of input may then be used to construct an output based on textual and/or semantic sources.

Search type	Input type	Output based on	Examples
Textual	Textual	Textual query	Google
Semantic-only	Semantic	Semantic query	SHOE PIQ [83], WebKB-2 [121]
Semantic+text backup	Semantic	Sem. query; textual if fails	SHOE [82], SoccerSearch [142]
Semantic+text union	Textual	Independent sem. and text. queries	TAP Search [74]
Semantic+text synthesis	Semantic	Dependent sem. and text. queries	QuizRDF [41]

3. **Semantic+text synthesis:** Finally, QuizRDF [41] (see Figure 2.2(right)) performs both a textual and a semantic search based on an initial semantic query, but uses information from the textual search to guide the semantic search. This feature can enable more useful searches when web content is only partially annotated, and is also the basis for the MANGROVE search service described in Chapter 3.

Table 2.1 summarizes these different approaches. While the "semantic+text backup" technique avoids the embarrassment of returning no answers to a query, in the common case it provides no improvement over existing textual search engines and thus is unlikely to be used. The "semantic+text union" technique, employed by TAP, is interesting because it enables untrained users to search simply by entering keywords. This search, however, is less flexible (since the user cannot provide any guidance on what semantic information is being sought) and cannot profitably combine information from the textual and semantic worlds. Finally, the "semantic+text synthesis" technique, employed by QuizRDF, eliminates these shortcomings of TAP, but requires the user to construct more complex semantic queries (though QuizRDF's user interface does provide some assistance in this process). A promising direction for future work is to combine these latter two approaches: permit users to enter plain textual queries (like TAP), have the system deduce an appropriate query that is split into textual and semantic portions (like those used by QuizRDF), then display results to the user and guide her in query refinement as necessary. This technique would provide more benefit to users while remaining consistent with our ease of use design principle.

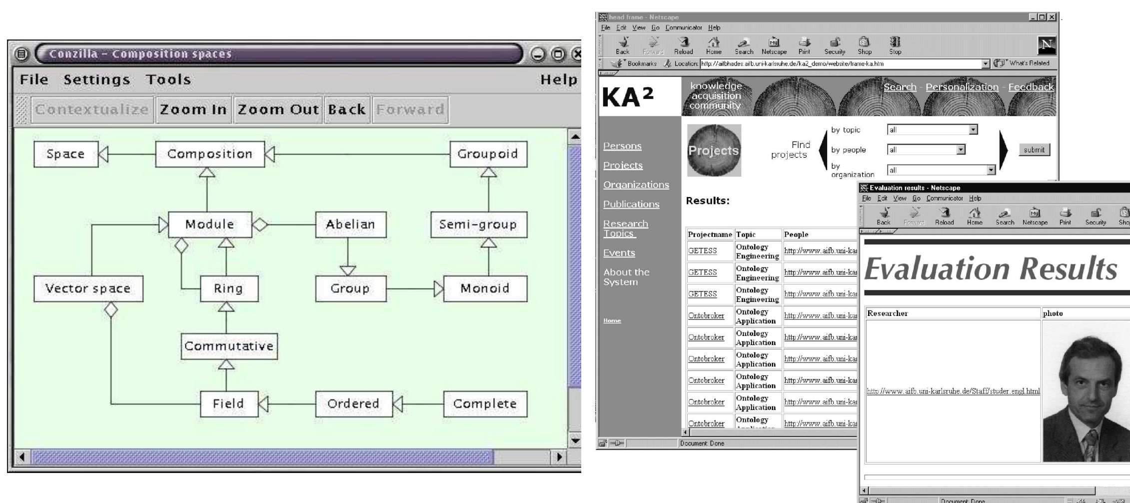


Figure 2.3: Semantic browsers and portals: The Conzilla browser (left) and the KA2 community web portal (right). The former requires a special tool to navigate among the displayed concepts, while the latter produces HTML that can be viewed with a normal browser.

Semantic Browsers and Portals

Aside from improving search, semantic data has also been applied to enable direct semantic browsing or to improve traditional HTML browsing:

- **Semantics-only browsing:** In the simplest case, systems may provide a *semantic-only* browser (e.g., Conzilla [138], Boeing's browser [10]). These applications permit users to graphically navigate through a knowledge base, following semantic connections to look for information of interest (see Figure 2.3(left)). The utility of these applications obviously depends on the content of the knowledge base and the user's ability to understand the connections and find relevant data. Recently, Haystack [152] presented a more sophisticated browser that can exploit domain-specific presentation information and that allows users to easily create their own associations and collections, expanding upon the bookmark features of traditional web browsers.

- **Semantics-boosted browsing:** Alternatively, COHSE [12], Annotea [97], and Magpie [55] start with a base of HTML content and modify this content based on associated semantics. For instance, COHSE uses semantic annotations to add or suppress links in existing HTML documents.

Table 2.2: Types of semantic browsers and portals. Figure 2.3(left) demonstrates the "Object view" type of output.

Type	Base data	Supplementary Data	Output type	Examples
Traditional	HTML	None	HTML	Netscape, Internet Explorer
Semantics-only	Semantic	None	Object view	Boeing [10], Conzilla [138], Haystack [152]
Semantics-boosted	HTML	Semantic	HTML	COHSE [12], Magpie [55]
Semantic portals	Semantic	None	HTML	InfoLayer [81], KA2 [14], SEAL [116, 88]

- **Semantic Portals:** Finally, systems such as InfoLayer [81], KA2 [14], and SEAL [116, 88] generate entire websites based on underlying semantic data. These systems build on the key concept of separating the logical view of a web site from its graphical presentation, as embodied in earlier database research (e.g., Strudel [62]). In particular, the portal specifies a view over the semantic data via a template for each type of desired output page [81] or concept (e.g., person) [14], possibly along with an ontology describing navigation options for the user [116]. For instance, in Figure 2.3(right), the left window shows a list of projects known by the KA2 portal, along with various navigational links. Some portals also permit users to create and issue arbitrary queries; the right portion of that figure contains the result for one such query requesting the person "Struder" and his photo.

Table 2.2 summarizes these different browsing and portal applications. For average users, basic semantics-only browsers are likely the least useful, due to the need to learn a new tool, the challenges of finding desired information, and the uniform display of all types of content. However, when the presentation can be specialized for particular data-intensive domains (as Haystack has demonstrated in the case of bioinformatics [152]), the ability to explore related concepts and find associations can be very useful. Semantics-boosted browsing has the advantage that it can integrate seamlessly with existing web browsing, adding value where possible without requiring any extra effort from users. In this sense it is similar to the TAP search feature discussed previously. To do better than just "doing no harm," however, still requires the existence of sufficient semantic content.

Semantic portals are also trivial to use with normal web browsers, and have the advantage of completely producing an attractive web page, eliminating the need to maintain

separate semantic and textual versions of data. In addition, they have the potential to achieve common usage, either because they replace existing, hand-maintained pages (e.g., a personal or project home page), or because they present a useful view of the knowledge of a community. The challenges for this approach are ensuring that data is easy to create and maintain, and that a "community" portal attracts enough content to make it initially useful [167], as required by our gradual adoption design principle.

Data Aggregation

A third category of information-providing applications are those that *aggregate* data from a number of sources for some specialized processing. Unlike generic search or browsing applications, these applications base their presentation and features on a particular domain of interest. These applications are similar to data integration systems, which also integrate data from multiple distributed sources [67, 75, 4, 109, 54, 106, 118]. Note, however, that data integration systems handle more heterogenous sources by mapping data to a common mediated schema, whereas the applications below typically assume that all data is stored using a single schema (though wrappers may have been used to obtain data from a variety of sources). Of course, developing and applying similar schema mappings would enable Semantic Web systems to leverage substantially more data [48, 77, 80, 181].

For instance, the SHOE "Path Analyzer" (Figure 2.4(left)) graphically displays possible pathways between animal sources and end products to support understanding of disease transmission [83]. Likewise, the Snippet Manager (Figure 2.4(right)) assists users with viewing, organizing, and sharing personal collections of information such as pictures and bookmarks [9]. Other examples include CS AKTive Space [122], which summarizes U.K. computer science research, Bibserv [1], which collects bibliographic information, and Elena [163, 141], which proposes to collect metadata about educational materials.

Often, aggregation applications provide exactly the same data that could have been obtained with an appropriate query. For instance, the results in Figure 2.4(left) could have been obtained with a (complex) query for properties connecting sources to processes, and processes to end products. In fact, several early systems (e.g., Ontobroker [44], Web-

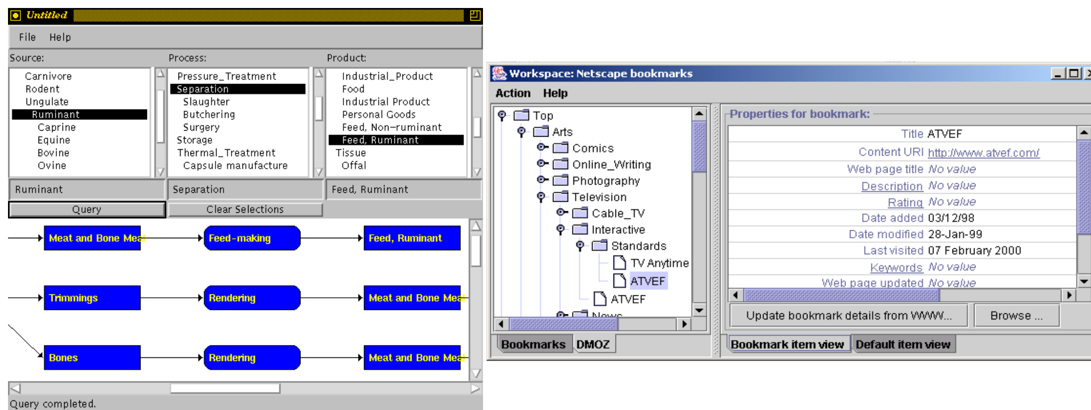


Figure 2.4: SHOE's TSE Path Analyzer (left) and the Snippet Manager (right), showing a collection of bookmarks.

KB [120, 121]) appear to have originally assumed that such queries (perhaps pre-constructed for convenience) would be the principal, if not only, way that users interacted with the system. In addition, aggregation applications introduce a number of restrictions on how data may be processed, such as limiting querying to only those properties hard-coded into the application. Thus, in a sense aggregation applications add little new functionality and may in fact constrain what is possible.

In practice, however, aggregation applications provide several important benefits compared to more generic applications. First, domain-specific applications can often transform results into a form that is much more understandable for typical users [83]. Second, these applications can apply customized data cleaning or processing (e.g., to integrate syntactically different references to the same person [50, 122] or to classify bookmarks based on a web directory [9]), enabling higher precision display of data than would be obtained from a generic query service. Finally, these applications may implement important performance optimizations, such as optimized server requests [9, 83] or application-specific caching. The combination of these features enable aggregation applications to present higher quality, highly responsive services that can be utilized by even average users — as required by our instant gratification and ease of use design principles. MANGROVE provides several such aggregation applications (e.g., the Calendar and WHO'S WHO services); these are described in Chapter 3.

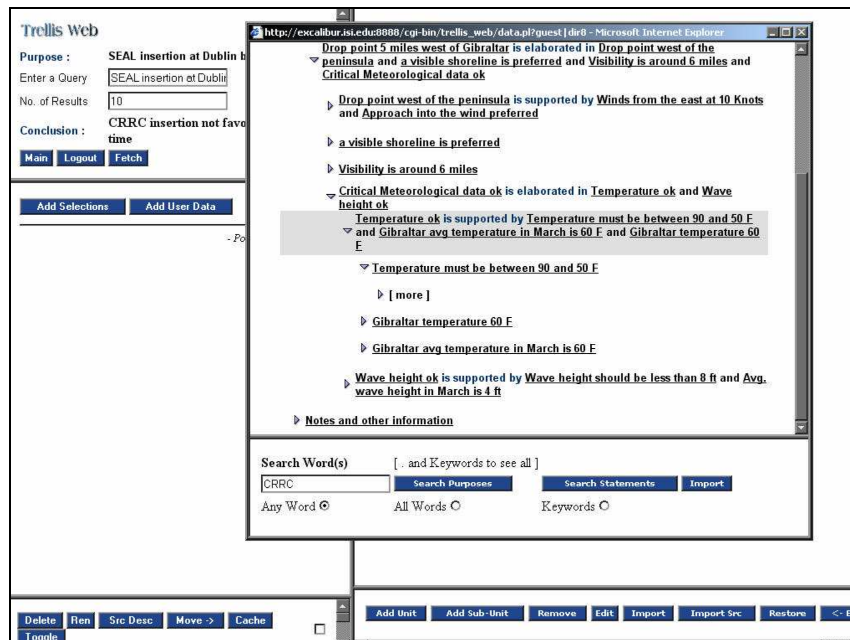


Figure 2.5: The TRELLIS application, showing a user reviewing the justification for an earlier decision.

Decision Support

The final category of information-providing applications are *decision support* applications. Like aggregation applications, these applications typically collect data from multiple sources and display it in some convenient form. Decision support applications, however, are distinguished by their inclusion of additional features and analysis intended to help users reach some conclusion from the data. In this sense they are similar to earlier database research, which augmented traditional transaction processing systems with additional features for more complex querying [69, 100, 32, 30], data cleaning [66, 153], and view selection [36, 5] to support analysis of current and historical data.

Examples of decision support applications are ClaiMaker [110], which uses human annotations to assist users with analyzing the competing claims of different research papers, and TRELLIS [68] (see Figure 2.5), which uses information from previous sessions and users to aid the planning or analysis of military operations. Other such applications include those focused on financial analysis (e.g., Analyst Workbench [162], Ontoprise's Corporate His-

tory Analyzer [142]), and project planning [159]. Hyperclip [157] provides an application designed to help office workers understand the context of documents created or referenced by other workers, though this system suffers from a lack of clear motivation for producing the needed annotations of these documents.

While almost all aggregation applications are focused on a particular domain, some decision support applications add extra functionality while remaining domain-independent. For example, while demonstrated in particular domains, ClaiMaker's ability to reason about relationships and TRELLIS's construction of aggregate user ratings permit reuse in many other contexts. Other applications such as the Analyst Workbench and the Corporate History Analyzer are more restricted to a particular domain.

2.2.2 Action-oriented Applications

This section describes applications that may take actions on behalf of the user.

Email Management

As discussed in Chapter 1, email represents a rich information space where many people spend significant amounts of time. However, little prior work has considered how adding semantics to email might increase productivity.

A few researchers have proposed systems to filter or search email based upon turning the generic header tags into RDF [39] or based on annotations manually added by the sender or recipient (e.g., MailSMORE [98]). The problem with these systems is that the former offers little improvement over the search and filter features of existing mail clients, while the latter requires substantial manual effort for a questionable future benefit (or for the benefit of someone else, e.g., the recipient).

Thus, while the above approaches may enable the integration of some email-based data with other web-based sources, they offer little benefit for improving email management. Chapter 4 describes our system for Semantic Email and how it provides more instant gratification in this realm.

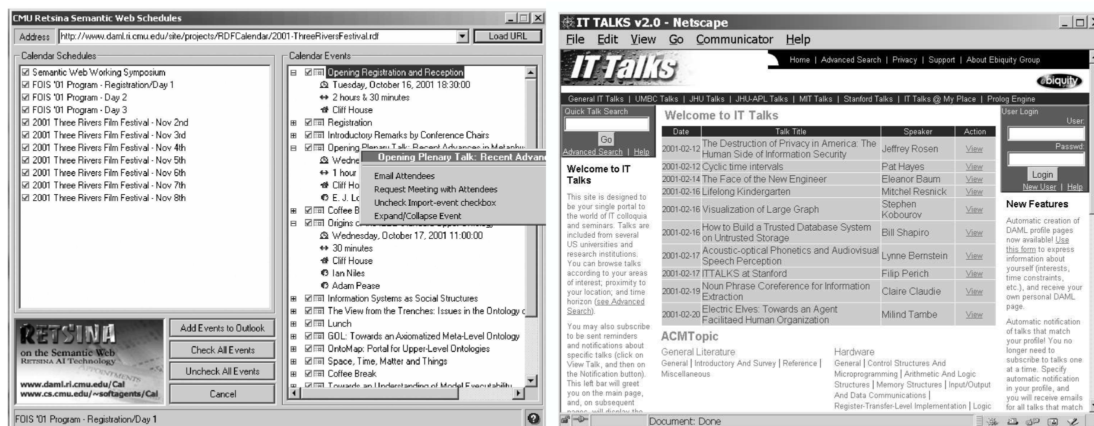


Figure 2.6: RCal (left), showing a schedule imported by the user, and ITtalks (right), showing a summary of relevant talks.

Time Management

Several systems have explored ways of improving time management with semantics, and an entire W3C task force has been devoted to considering RDF calendaring issues [176]. For instance, RCal [147] (see Figure 2.6(left)) can import schedule information from an annotated conference web page into Outlook, and can use calendar info to automatically schedule meetings involving several RCal-enabled participants. A second example is ITtalks [148] (see Figure 2.6(right)), which presents information about talks in the information technology domain. Users may view relevant talks based on a profile registered with the ITtalks website, or may install a personal agent that receives talk announcements and automatically decides whether to notify the user based upon the attendance of friends and expected distance to the talk. Other systems that can assist with time management include GraniteNights [70], SKICal [2], and our Semantic Email system described in Chapter 4.

These prototypes demonstrate the potential of automated applications to assist with common time management tasks. However, present technology remains far from the popular example given by Berners-Lee et al. [15] where the personal agents of several family members coordinate to automatically schedule and assign responsibility for a set of appointments subject to various bureaucratic, medical, and availability constraints. Achieving this visionary goal will require much more sophisticated profiles and ontologies as well as the ability to compose multiple semantic services as described in the next section.

Web Service Interaction

The web today offers a great variety of consumer products and services online. However, actually purchasing these products or services can be tedious (e.g., due to repeated personal data entry at different sites) and time-consuming (especially if multiple services are required to accomplish some goal, as is common with travel). Likewise, the use of web services for direct B2B supplier discovery, customization, and ordering offers the promise of greatly increased efficiency, but in practice has been limited to facilitating repeated transactions between known business partners. The development of standards for web service description, discovery, and execution such as UDDI, WSDL, SOAP, and BPEL4SW is a first step towards automating this process. These standards, however, are not sufficient for enabling automated composition of such services without human involvement. For instance, UDDI can describe that a service accepts two integers as inputs and produces a text string as output, but the crucial information (that this service results in the user purchasing a book identified by ISBN) must be specified as a human-readable text string.

In response, several groups have developed methodologies for semantically describing web services (e.g., DAML-S/OWL-S [7], METEOR-S [145], Bernstein and Klein [17], CAW-ICOMS [58], WSMF [28]) and some have explored algorithms and query languages for discovering suitable services to achieve some goal (e.g., Woogle [51], PQL [17], CAW-ICOMS [58], DAML-S Matchmaker [144], Trastour et al. [171], Benatallah et al. [13]). For instance, DAML-S [7] provides an ontology for describing the capabilities, requirements, and effects of a service. Applications such as the DAML-S Matchmaker [144] use this description to flexibly match service advertisements and requests (e.g., to match requests for services selling "sedans" with a more general service offering "vehicles").

Based on these service descriptions, some prototypes to compose multiple services have been constructed. For example, McIlraith et al. [133] and Liebig [111] both describe applications that utilize multiple suppliers to make travel arrangements based on user preferences. Likewise, Sirin et al. [164] (see Figure 2.7) implement a system that assists users in identifying and composing suitable web services to achieve some goal. More advanced systems have also been proposed [28, 166, 35]. For instance, Bussler et al. [28] focuses on tight integra-

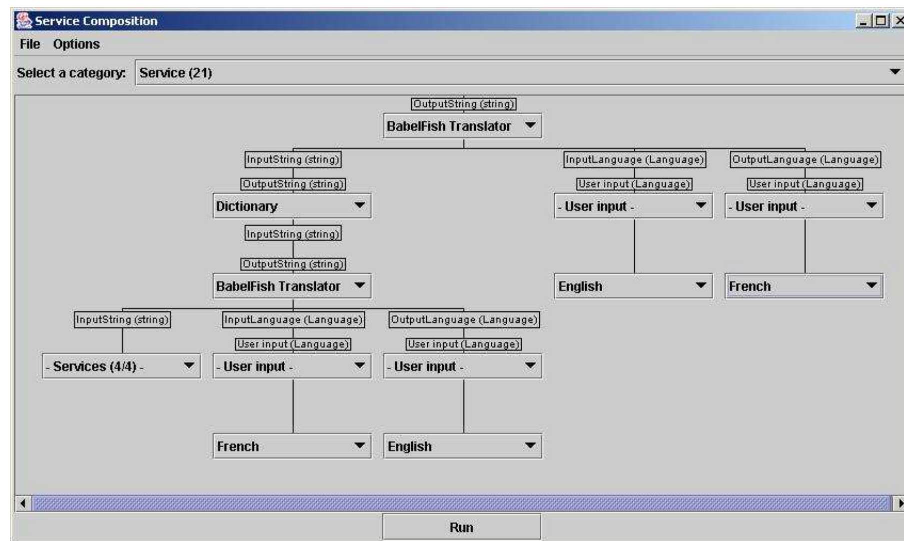


Figure 2.7: A screenshot from Sirin et al.'s program to compose multiple services, here constructing a foreign language translator. This program is freely downloadable (though not directly executable) from the web.

tion with business environments, for instance to automatically identify a desired product, receive approval for a purchase order, transmit the order to the supplier, and propagate the resulting receipt back to all needed departments in the purchaser's company.

These applications offer the potential to profitably identify and compose not just web data, but web services as well, both for consumer and for e-commerce applications. However, unlike the information-providing applications and the email/time management applications discussed above, none of these systems appear to be sufficiently developed for practical usage today. A key limitation is the need to create many detailed, comprehensive semantic descriptions. This task is unlikely to be undertaken by casual users, though some research has explored semi-automatic approaches for generating these descriptions [85] or for searching based on existing WSDL/UDDI files [51].

2.2.3 Discussion

The examples above highlight a wide range of possible Semantic Web applications. These applications also reveal multiple features that impact how amenable their usage is by non-

technical persons. Below we examine these features in terms of our three proposed design principles. Later, the final section of this chapter will discuss the overall implications for the adoption of Semantic Web systems.

First, consistent with our instant gratification principle, many applications did offer a tangible benefit to usage. For instance, TAP Search [74] and SEAL [116, 88] provide potentially useful information browsing and retrieval capabilities for the real-world domains of entertainment and researchers, respectively. Likewise, RCal [147] schedules meetings and McIlraith et al.'s system [133] offers potentially useful (though not yet practical) travel scheduling. These plausible uses are a critical advantage compared to systems such as MailSMORE [98] and HyperClip [157] that don't offer a compelling usage scenario.

Second, these applications showed mixed results in terms of gradual adoption. Regarding content, applications that can include non-semantic data in their outputs (e.g., QuizRDF [41], TAP Search [74]) are more likely to find relevant results for their users given the initial lack of semantic data. Regarding users, the ability to interact with anyone else is better than than the ability to interact only with other users that have installed specific software (e.g., as in RCal [147]). Unfortunately, none of the systems we surveyed provided such flexible interaction capabilities, but Chapter 4 describes our solution for Semantic Email.

Finally, these applications revealed a number of features that support our ease of use principle. For instance, applications that are web-accessible and have simple keyword (e.g., TAP Search [74]) or browsing (e.g., SEAL [116, 88]) interfaces are more likely to succeed than applications requiring local installation (e.g., TRELLIS [68]) or complex query construction (e.g., SHOE PIQ [83]). Furthermore, domain-customized applications such as Snippet Manager [9] and ITtalks [148] are often easier for non-technical persons to use than more generic browsers or search engines. An interesting research challenge is to develop applications that are broadly applicable (simplifying development and usage training), but that maintain the helpful features of more customized applications. Possible approaches to this problem include the use of predefined queries [167] and the encoding of navigation and presentation information in an ontology [116, 152].

2.3 Content Provision for the Semantic Web

The previous section described a range of possible Semantic Web applications that could potentially be useful for and usable by ordinary people. In order to actually make these applications viable, however, they must be supplied with data of adequate quantity and quality.

In this section we consider three possible sources for such data: automatic generation, and manual generation by either technical users or non-technical users. While our focus in this dissertation is on application usage and content creation by non-technical users, we first discuss content creation via automated techniques and technical users, because these can be important additional sources of data, particularly for making applications initially useful. We then consider how to both motivate and enable non-technical users to contribute content. Obviously these two issues are interrelated — if the authoring task is too onerous, practically no motivation may be sufficient to entice participation. Nonetheless, we consider motivation first because our instant gratification principle implies that it is of prime importance — without motivation, users are unlikely to perform any authoring task regardless of its ease.

2.3.1 Automatic Content Creation

Ideally, humans would not need to create semantic content at all, but it could simply be derived (semi)automatically from existing data sources. Indeed, there are several possible ways to automatically acquire semantic data: by leveraging existing structured data from databases [172, 80] or XML sources [44, 77], or by utilizing wrappers or information extraction techniques to extract data from semi-structured or unstructured sources (e.g., HTML, text, publications, emails) [88, 82, 60, 103, 156, 114, 11, 50]. All of these techniques enable the automatic leveraging of existing data in the semantic world. Note, however, that they still require some human effort to configure the translations, wrappers, etc. for each source. In addition, these configurations may be very sensitive to changes in the structure of the underlying data, which is often under the control of some other person. Finally, while they

apply to a significant amount of data, they cannot be used everywhere, either because the needed data does not exist or because it is not amenable to such automation.

Nonetheless, automatic content generation remains a key part of many systems. One of the most important uses for these techniques is for *seeding* a knowledge base for a particular application. Several systems have used such seeding, e.g., TAP Search [74] and Bibserv [1]. These efforts help to make an application immediately useful even before any manual content creation has been performed, thus supporting our gradual adoption principle. Chapter 3 provides a number of examples of how seeding is used for this purpose in MANGROVE.

2.3.2 Content Creation by Technical Users

Some systems specifically assume that technically-sophisticated people will produce all of the content necessary for the system to be successful. For instance, WebKB's content representation is most appropriate for "knowledge engineers," [119], and the InfoLayer system assumes that users are familiar with UML modeling [81]. Likewise, constructing web service descriptions in DAML-S is difficult, requiring navigation of a complex ontology and specification of pre- and post- conditions.

For web services, the number of descriptions needed is relatively small, and thus relying on technical users is plausible. Exploiting content from the huge number of existing web pages, however, presents more of a challenge. While technical users may produce significant amounts of content, it makes sense to focus instead on the vast numbers of users who lack technical sophistication yet have a web page and could be enticed to contribute content to the Semantic Web. The next section describes techniques to encourage content creation by this group. Of course, these techniques can also be beneficial for technical users as well.

2.3.3 Motivating Content Creation by Non-Technical Authors

For a few kind souls, the simple joy of adding content that might benefit others is sufficient motivation for semantic authoring. In general, however, more tangible benefit is necessary:

> People tend to use systems on a tit-for-tat basis. They tend not to invest work when they cannot recognize *immediate benefit* from it. ([88], emphasis added)

There are two primary factors influencing the degree of "immediate benefit" received by authors. First, what type of benefit is received? Second, how immediate is the benefit? Below we consider each in turn.

What type of benefit is received?

Roughly speaking, benefits from semantic authoring fall into two categories: personal benefits and social benefits. First, *personal* benefits cover those that apply even if no one else ever directly makes use of the semantic content. For instance, semantic authoring may enable convenient *organization* of bookmarks and notes [9] or analyses of research papers [110]. Alternatively, content creation may enable useful *automation* of meeting scheduling [147] or repetitive email tasks (as with our Semantic Email system). As a final example, semantic authoring may simplify future *maintenance* of data, for instance to detect inconsistencies in the data or to ensure that a publication list remains up to date [81, 61]. In each case, users are motivated to author semantic content because of some intrinsic benefit to their own work or life, independent of anyone else.

Second, *social* benefits describe those where the author's benefit depends heavily on the extent to which others will encounter the author's semantic data. For instance, an application might encourage semantic authoring by offering a more attractive [81, 88] or informative [12] *presentation* of the content (e.g., a personal or department home page) than could be easily achieved via other means. Alternatively, authors may be motivated by the promise of more extensive and accurate *proliferation* of data that is semantically annotated. For instance, calendar applications bring annotated events to the attention of more potential attendees [148], while publication databases [1, 110] lead to more numerous (and factual) citations for submitted papers. Finally, authors may be motivated by the potential for *profit*. For instance, Elena [163] assumes that contributors will collect registration fees when users discover their annotated learning services, just as travel services may benefit from annotating their service descriptions for automated planners [133, 111]. This final motivation is similar to the "proliferation" incentive, but the addition of direct financial incentives may significantly alter both application usage and the impartiality of contributed data.

Table 2.3: The timeliness of benefit from authoring semantic content.

Type	Typical speed	Reasons for the speed	Examples
Instant	Few seconds	New content is immediately collected, efficient access mechanisms.	Snippet Manager [9], Bibserv [1], InfoLayer [81]
Delayed	Minutes to hours	New content collected right away, but requires processing/cache flush before usable.	QuizRDF [41], TAP Search [74]
Eventual	Few days or more	New content available only after (re)loaded by crawler, or approval is needed.	SHOE [83], Ontobroker [44], SEAL [116]
Uncertain	Some future time or never	Data may never be used, or only used by another person with no tangible benefit to author.	MailSMORE [98], Hyperclip [157]

How immediate is the benefit?

To some extent, people tend to be motivated by benefit that is more immediately available. For the Semantic Web, providing timely benefit depends both upon the general system (how quickly is data collected and available to applications?) and upon the speed of the target application (which may require substantial time to initialize or execute for some inputs [41, 60, 9]).

Consequently, benefit to semantic authors may occur across roughly four time scales. Table 2.3 describes these different time scales and gives examples of systems in which they occur. For instance, the InfoLayer system [81] generates its portal directly from a database that may be efficiently updated, and thus content authors may potentially see the effects of changes *instantly.* The current configuration of QuizRDF [41], however, requires index regeneration before searching a new or modified RDF data set, thus *delaying* any use of the data for about a minute. Many early systems (e.g., SHOE [83], OntoBroker [44]) relied upon periodic web crawls to obtain semantic content, resulting in content that is *eventually* available to applications. Finally, some applications may provide benefit only at some *uncertain* future point, if at all (e.g., when annotating incoming email messages to enhance future searches [98] or when marking up the relationships among documents for the potential future use of others [157]).

The ultimate timeliness of benefit depends on multiple factors. For instance, some systems (e.g., Ontobroker [44]) with *eventual* benefit also offered alternative ways of collecting data that were more immediate, but that were less convenient for typical web authors. On the other hand, if an *instant* application has only a small number of users, authors may conclude that there is only *uncertain* benefit to contributing content.

Thus, aside from applications offering *personal* benefits, providing timely benefit to authors depends just as much upon the existence of application users as it does upon the system and application infrastructure. Our proposed design principle of instant gratification addresses these issues by requiring that applications have whatever system architecture, usage scenarios, content base, and attracted user population are necessary to provide an immediate, tangible benefit to authoring.

2.3.4 Enabling Content Creation by Non-Technical Authors

Once users are motivated to create semantic content, they must be enabled either to annotate existing content (usually HTML) or to generate entirely new content. While early Semantic Web systems such as SHOE and Ontobroker required users to author such semantic context with a text editor, essentially all complete systems today provide some form of support for this task.

Possible techniques for simplifying content creation and maintenance include the provision of graphical annotation tools [78, 83, 98], enabling non-redundant markup of HTML [60], permitting the annotation of documents not under the annotator's control (e.g., external annotation [12, 97]), supporting semi-automatic annotation [79], and supporting web-based, installation-free content creation [116, 121, 148]. An additional interesting feature is the ability to create new content in the midst of viewing existing data. For instance the decision-support applications ClaiMaker, TRELLIS, and Hyperclip [110, 68, 157] all enable the user to add new statements about documents or objects while examining existing connections. This functionality may eliminate the need for a separate annotation tool and makes it easier to entice casual users to contribute data to the system. Our Semantic Email system provides a related technique, where users naturally create semantic content in the course of requesting a service from the system. Chapters 3 and 4 provide more details on the approaches we have taken with MANGROVE and Semantic Email to enable users to easily author such content.

2.3.5 Summary

This section, while focused on content creation, has highlighted the importance of our instant gratification design principle — both application usage and content creation must provide immediate benefit. When applied correctly, these factors complement each other, with more application usage increasing the "social" benefits of content creation, yielding more content and hence more useful applications. Techniques for making application usage immediately beneficial include seeding with wrapper-based data (Section 2.3.1), customizing presentation based on the domain (Section 2.2.1), and providing zero-installation applications (Section 2.2.3). Likewise, content creation can be improved by providing immediate "personal" benefits to authors (Section 2.3.3), supporting semi-automatic annotation (Section 2.3.4), enabling content creation within consuming applications (Section 2.3.4), and making new content immediately available to applications that consume the data (Section 2.3.3). In the next section, we examine a few issues that can affect both of these factors.

2.4 Cross-cutting Issues

This section considers a number of issues that impact the design and execution of the entire system, from content creation, to query execution, to application usage. We focus specifically on inference and trust management. Additional discussion of other relevant issues such as application construction, semantic interoperability, and ontology engineering is beyond the scope of this work.

2.4.1 Inference

Inference (i.e., deducing additional facts beyond those explicitly stated) can significantly enhance Semantic Web systems in two ways. First, inference improves the output of applications by augmenting incomplete knowledge [44, 9]. Second, inference reduces the burden of annotation by making it unnecessary to explicitly specify every fact [61]. Examples of practical inference include deducing types from subclass relations (e.g., `BookmarkItem` is also an `Item` [9]), handling symmetric and transitive properties (e.g., `cooperatesWith` is

Table 2.4: Techniques for making inference practical.

Type	Description	Example
Manually-inserted	Manually constructed code inserts derived facts into knowledge base ahead of time.	`BookmarkItem's` are also of type `Item` [9].
Automatically-inserted	An inference engine inserts derived facts into the knowledge base ahead of time.	If X `cooperatesWith` Y, then Y is of type `Researcher` [60].
Application-based	Applications derive needed facts at run-time.	`events` inherit location from parent `course` (MANGROVE Calendar); suggested by [9].
Restricted	Query engine directly supports limited types of inferencing.	Multiple backends with varying capabilities [82], OWL-Lite vs. OWL-Full [43].

symmetric [44]), and processing assertions that two resources (such as topics for a talk [148]) are equivalent or distinct.

Supporting inference, however, can also cause a serious scalability problem, without necessarily improving results (e.g., the experience of OntoBroker in [60]). To ameliorate this problem, Table 2.4 lists a number of techniques that systems have employed in an attempt to support inference practically. In the first two cases, inferencing occurs offline, before a query is generated. In this case, inferred facts are inserted directly into the knowledge base, based either on manually constructed code (for a few key inferences) or a generic inference engine. The next two cases perform inferencing at query time, either within the application (with no system support), or within the query processor itself, but for a restricted set of axioms.

None of these approaches is ideal. Manually-inserted and application-based inferencing can provide inferencing targeted at the needs of specific applications, but are error-prone and do not generalize well. Automatically-inserted inferencing is more general while still having little impact on query execution time, but may produce large amounts of useless inferences unless carefully focused [60]. Restricted inferencing seems the most promising since it is general, focuses directly on inferences needed for a specific query, and can be efficient for appropriate choices of the restrictions. However, many common tools have yet to support this technique in a scalable manner [123], and more work is required to allow designers to easily select the complexity needed for an application.

In our systems, MANGROVE makes use of application-based inferencing to support a small number of key inferences. The current Semantic Email system performs no reasoning about the ontology (though a number of uses for interpreting responses are suggested). This

system does, however, make heavy use of reasoning about instances of the ontology in order to determine if a response is consistent with the originator's goals. Chapters 3 and 4 explain these issues in more detail.

2.4.2 Trust Management

Another critical issue for Semantic Web systems is dealing with the trustworthiness and reliability of data sources. Existing schemes for managing trust can be divided into three general types:

- **Authority-based filtering:** Authority-based systems either have a central moderator approve or modify content to ensure reliability (e.g., KA2 [167], SemTalk [64]) or only accept content from approved, trusted sources (e.g. SHOE's Path Analyzer [83], Ontobroker's "Ontogroup" [14]). More recently, TAP [73] has proposed using a "Web of Trust" among centralized registries to provide reliable data.

- **User-based filtering:** Alternatively, systems may permit the end users of the system to specify what data sources to trust. For instance, WebKB-2 [121] permits users to include or exclude the data from specified users when querying, while ITtalks [148] proposes using a set of reliability claims from users or their agents to determine what sources to trust.

- **User-based verification:** Instead of filtering, systems may assist users in *verifying* the reliability of data returned by applications. For instance, Annotea [97] and ClaiMaker [110] present the user with data from all relevant sources, but also reveal the source of each output fact. These systems assume that the (human) users can generally ascertain the reliability of the data by examining the source content or characteristics of its author. TRELLIS [68] seeks to aid users in this process by aggregating reliability estimates from multiple users that can then be exploited by individual users to select appropriate sources for their analysis.

Ultimately, the most appropriate trust policy depends upon the target application. For instance, authority-based filtering may help provide reliability, but can conflict with our

instant gratification principle by adding delay before new data is available to applications or new users can contribute. Some argue that we must have fully automated schemes for deciding what information is reliable, since programs will be processing the data and cannot employ human judgment [74, 15]. While this may be true for most of the action-oriented applications described in Section 2.2, it is not necessarily true for the information-providing applications, as demonstrated by the practical systems that utilize user-based verification for trust.

In our systems, MANGROVE utilizes user-based verification that may be supplemented by an initial authority-based filtering step that utilizes local domain knowledge about reliable sources. The Semantic Email system performs user-based filtering by allowing originators to restrict the acceptable responses to those from the set of invited participants. As before, Chapters 3 and 4 provide more details on these techniques.

2.5 Discussion

This chapter has presented a wide variety of proposed Semantic Web applications and explored issues relevant to content creation for these applications. While prior to our work with Semantic Email there were no compelling, feasible usage scenarios for email management, our summary of existing work has identified a number of potentially practical and useful applications for other domains. A few such applications (e.g., Bibserv [1], TAP Search [74]) are deployed and seem highly usable already. In addition, a number of other applications offer very useful ideas that simply need more focus on the key issues that will drive adoption. For instance, systems such as SEAL [116] and InfoLayer [81] create attractive portals based on semantic data, but would benefit from being able to more easily collect new data without crawling (SEAL) and from providing additional applications beyond just rendering HTML pages (InfoLayer). Likewise, RCal [147] provides useful calendaring services but needs to be able to communicate more easily with non-Rcal users, while the Snippet Manager [9] needs to integrate more closely with existing tools (e.g., bookmark collections inside browsers).

This chapter described a number of such features that are intended to help drive system adoption. Sections 2.2.3 and 2.3.5 summarized these features directed towards application

usage and content creation, respectively. However, a number of significant challenges have not been addressed. For instance, once a user provides content to the system, how can she immediately and easily discover how that content is being used? Without such a mechanism, errors processing the data or difficulty finding the results could easily prevent her from obtaining any benefit from the contribution. Likewise, given the redundancy requirements of RDF for annotating existing HTML, how can authors annotate content in a way that will not become a future maintenance problem? In addition, existing action-oriented applications require that all participants be knowledgeable about the system and have appropriate software installed (e.g., as described above for RCal). How can such an application encourage adoption by instead enabling interaction with naive participants? Even more importantly, the vision of the Semantic Web promised benefits based on logical reasoning — but what kinds of practical inference can benefit these applications? And how can naive participants easily express their goals to such a system?

Thus, the key deficit in existing Semantic Web research has not been a lack of application ideas, but a failure to identify and implement a complete set of features needed to drive adoption by actual users. This dissertation addresses this problem in two ways. First, we have already proposed three key design principles needed to drive system adoption. These principles can focus the design of a Semantic Web system or be used as a metric to gauge the likely success of an existing system. Second, we introduce a number of novel mechanisms that are inspired by these principles and that address previously unsolved problems related to adoption. The next three chapters explain these mechanisms in the context of MANGROVE and Semantic Email and elaborate on how these systems embody the key design principles.

Chapter 3

MANGROVE

This chapter presents MANGROVE, a system designed to enable and entice "ordinary users" to annotate and contribute their existing HTML content to the Semantic Web. The next section introduces MANGROVE's architecture and explains how it supports the three key principles of instant gratification, gradual adoption, and ease of use. Section 3.2 describes our first semantic services while Section 3.3 discusses our initial experience from deploying MANGROVE. Finally, Section 3.4 discusses related work, and Section 3.5 concludes.

3.1 The Architecture of MANGROVE

This section presents the high-level architecture of MANGROVE, details some of the key components, and relates them to our design principles.

3.1.1 Architecture Overview

Figure 3.1 shows the architecture of MANGROVE organized around the following three phases of operation:

- **Annotation:** Authors use our *graphical annotation tool* or an editor to insert annotations into existing HTML documents. The annotation tool provides users with a list of possible properties from a local schema based on the annotation context (e.g., describing a person or course), and stores the semantic data using the MTS syntax that is described in Section 3.1.2.

- **Publication:** Authors can explicitly *publish* annotated content, causing the *parser* to immediately parse and store the contents in an *RDF database*. The *notifier* then notifies registered services about relevant updates to this database. Services can then send *feedback* to the authors in the form of links to updated content (or diagnostic

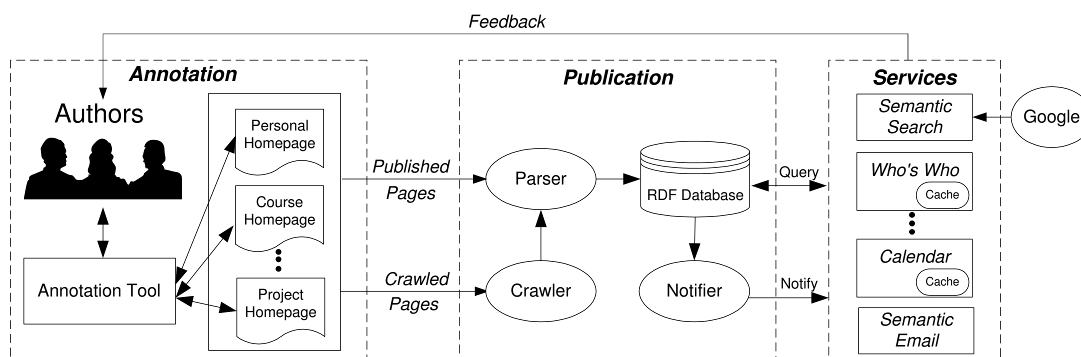


Figure 3.1: The MANGROVE architecture and sample services. Semantic Email (Chapters 4 and 5 is implemented as a MANGROVE service in order to facilitate interoperability with other MANGROVE services.

messages in case of errors). In addition, MANGROVE's *crawler* supplies data to the parser periodically, updating the database when authors forego explicit publishing.

- **Service Execution:** Newly published content is immediately available to a range of *services* that access the content via database queries. For example, we support semantic search and more complex services such as the automatically-generated department calendar.

These three phases are overlapping and iterative. For instance, after annotation, publication, and service execution, an author may refine her documents to add additional annotations or to improve data usage by the service. Supporting this complete life-cycle of content creation and consumption is important to fueling the Semantic Web development process.

Below, we describe the components of MANGROVE in more detail. Section 3.2 then describes the semantic services that make use of MANGROVE to provide instant gratification to content authors.

3.1.2 Annotation in MANGROVE

Semantic annotation of HTML content is central to MANGROVE. This section describes how MANGROVE enables such annotation in a manner consistent with our principles of gradual adoption and ease of use. In particular, we describe the MTS syntax and our graphical annotation tool, and explain how they enable authors to annotate existing content in a way

that deviates very little from existing practices, does not require that a page be annotated all at once, and that is robust to future factual changes in the data.

Instead of manual annotation, we considered the use of wrapper technology for automatically extracting structured data from HTML, as described in Chapter 2. However, such technology relies on heavily regular structure; it is appropriate for *recovering* database structure that is obscured by HTML presentation (e.g., Amazon.com product descriptions that are automatically generated from a database), but not for analyzing pages authored by hand. Thus, MANGROVE utilizes a small number of wrappers to generate "seed" data to initially populate semantic services, but this is not sufficient for solving the general problem. We also considered natural language processing and, specifically, information extraction techniques (e.g., [165, 104, 103, 156, 114, 11]). While such approaches have been very successful in some contexts, they require unambiguous inputs, are often domain-specific, and may produce unreliable results. As a result, we formulated the more pragmatic approach described below.

The MTS *Syntax*

We developed MTS (the MANGROVE Tagging Syntax) to enable easy annotation (or *tagging*) of existing HTML. Ideally, we would have liked to use RDF for this annotation. However, as previously mentioned, RDF's syntax is currently inadequate for our purposes, since it requires that existing HTML data be *replicated* in a separate RDF section [81]. Since HTML documents are frequently updated by their authors, this data replication can easily lead to inconsistency between the RDF and its data source, particularly if "semantically-unaware" tools (e.g., Microsoft FrontPage) are used for editing.

In essence, MTS is a syntax that supports gradual adoption by enabling authors to *embed* declarative RDF statements within their HTML documents without disrupting traditional browsing of those documents. As a consequence, MTS does not have the aforementioned redundancy problem, because HTML and MTS tags may interleave in any fashion, permitting "inline" annotation of the original data. Therefore, factual changes to the annotated data using any tool will result in a seamless update to the semantic content on the page.

```
<html xmlns:uw="http://wash.edu/schema/example">
<uw:course about="http://wash.edu/courses/692">
<h1><uw:name>Networking Seminar
    </uw:name></h1>

<p>Office hours for additional assistance:
<uw:instructor>
    <uw:name><b>Prof.</b> John Fitz</uw:name>
    <uw:workPhone>543-6158</uw:workPhone>
</uw:instructor>
<uw:instructor>
    <uw:name><b>Prof.</b> Helen Long</uw:name>
    <uw:workPhone>543-8312</uw:workPhone>
</uw:instructor>

<table> <tr><th>2003 Schedule</tr>
   <uw:reglist=
      '<tr><uw:event>
         <td><uw:date>*</uw:date>
         <td><uw:topic>*</uw:topic>
      </uw:event></tr>'>

      <tr> <td>Jan 11 <td>Packet loss</tr>
      <tr> <td>Jan 18 <td>TCP theory</tr>
  </uw:reglist>
</table>

</uw:course> </html>
```

Figure 3.2: Example of HTML annotated with MTS tags. The `uw:` tags provide semantic information without disrupting normal HTML browsing. The `<reglist>` element specifies a regular expression where '`*`' indicates the text to be enclosed in MTS tags.

MTS consists of a set of XML tags chosen from a simple local schema. The tags enclose HTML or plain text. For instance, a phone number that appears as "`543-6158`" on a web page would become "`<uw:workPhone>543-6158</uw:workPhone>`". Here "`uw`" is the namespace prefix for our local domain and "`workPhone`" is the *tag name*. Nested tags convey property information. For instance, Figure 3.2 shows a sample tagged document where the `<workPhone>` tag is a property of the `<instructor>` named "Prof. John Fitz."

MTS contains a number of additional features to support ease of use. For instance, in order to enable users to annotate existing data regardless of its HTML presentation, the MTS parser disregards HTML tags when parsing (though images and links are reproduced in the parser output to permit their use by services). In addition, in order to reduce the annotation burden for items such as lists and tables that exist inside HTML documents, Mangrove provides a simple regular expression syntax. For instance, the `<reglist>` element in Figure 3.2 enables the table to be automatically tagged based on existing HTML patterns. This enables new rows that are added to the table to be automatically tagged without any effort on the part of the author. Finally, MTS supports RDF-like `about` attributes (e.g., of the `<course>` element in Figure 3.2) that enable information about an

object to appear in multiple documents and be fused later. For ease of use, however, we do not require or typically expect users to provide such unique identifiers. Section 3.2.4 discusses some of the resultant referential integrity challenges.

MTS's expressive power is equivalent to that of basic RDF. For simplicity, we omitted advanced RDF features such as containers and reification. Note that the goal of MTS is to express base data rather than models or ontologies of the domain (as in RDF Schema, DAML+OIL [86], OWL [43]).

Finally, MTS syntax is not based on XHTML because most existing pages are in HTML and hence would require reformatting to become valid XHTML, and many users are averse to any such enforced reformatting. Furthermore, a large fraction of HTML documents contain HTML syntax errors (generally masked by browsers) and thus would require manual intervention to reliably convert to legal XHTML. However, existing XHTML (or XML) documents that are annotated with MTS will still be valid XHTML (XML) documents, and thus tools that produce or manipulate these formats may still be freely used with MTS-annotated documents.

The Graphical Annotation Tool

The MTS syntax enables a small number of tags to concisely annotate a document in a way that is robust to future document changes. However, for most users, direct use of this syntax is still too complex and error-prone.

Thus, to facilitate semantic authoring, we developed the simple graphical annotation tool shown in Figure 3.3. The tool displays a rendered version of the HTML document (right pane) alongside a tree view of the relevant schema (upper left pane). Users highlight portions of the HTML document, and the tool automatically pops up a list of MTS tags that may be selected, based on the current annotation context. The tool also displays a simplified tree version of the tagged portion of the document (lower left pane), showing the value of each property. This enables the author to easily verify the semantic interpretation of the document or, by clicking on a node in the tree, to browse through the document based on its semantics.

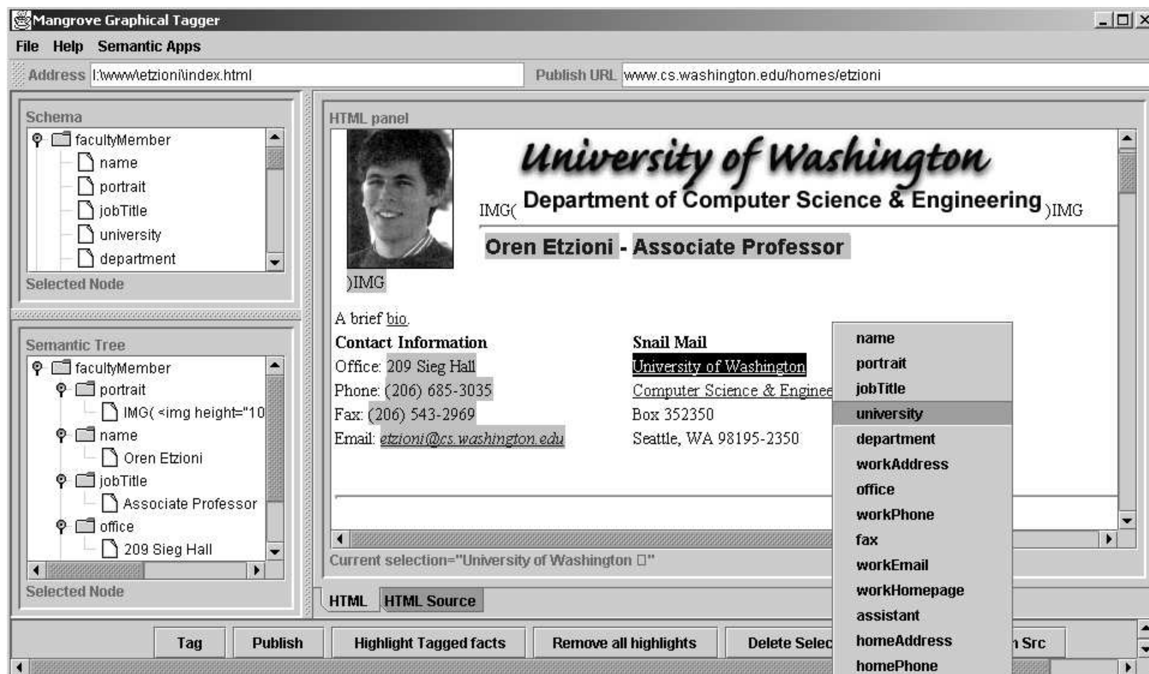


Figure 3.3: The MANGROVE graphical annotation tool. The pop-up box presents the set of tags that are valid for annotating the highlighted text. Items in gray have been tagged already, and their semantic interpretation is shown in the "Semantic Tree" pane on the lower left. The user can navigate the schema in the upper left pane.

The annotation tool is freely downloadable and is constructed in Java, enabling it to run on any platform. In addition, the annotation tool allows authors to easily *publish* newly annotated content (see the "Publish" button in the lower left of Figure 3.3), as described in Section 3.1.3.

Schema in MANGROVE

Currently, MANGROVE provides a single, predefined XML schema to support the annotation process (see Appendix A). Providing a schema is a crucial, as we can't expect casual users to design their own (and that would certainly not entice people to use the system). The intent of the schema is to capture most aspects of the domain of interest. Consistent with our gradual adoption principle, the pages being annotated do not have to contain *all* the details of a certain schema. Instead, authors map the data on their page to the appropriate

schema tags. In the future, we anticipate a process by which users can collectively evolve the schema as necessary.

3.1.3 Document Publication

In today's web, changes to a web page are immediately visible through a browser. We create the analogous experience in MANGROVE by enabling authors to *publish* semantically annotated content, which instantly transmits that content to MANGROVE's database and from there to services that consume the content.

MANGROVE authors have two simple interfaces for publishing their pages. They can publish by pressing a button in MANGROVE's graphical annotation tool, or they can enter the URL of an annotated page into a web form. Both interfaces send the URL to MANGROVE's parser, which fetches the document, parses it for semantic content, and stores that content in the RDF database. This mechanism ensures that users can immediately view the output of relevant services, updated with their newly published data, and then iterate either to achieve different results or to further annotate their data. In addition, before adding new content, the database purges any previously published information from the corresponding URL, allowing users to retract previously published information (e.g., if an event is canceled).

Crawling or polling all potentially relevant pages is an obvious alternative to explicit publication. While MANGROVE does utilize a crawler, it seems clear that crawling is insufficient given a reasonable crawling schedule. This is an important difference between MANGROVE and previous systems (e.g., [44, 83]) that do not attempt to support instant gratification and so can afford to rely exclusively on crawlers. MANGROVE's web crawler regularly revisits all pages that have been previously published, as well as all pages in a circumscribed domain (e.g., `cs.washington.edu`). The crawler enables MANGROVE to find semantic information that a user neglected to publish. Thus, publication supports instant gratification as desired, while web crawls provide a convenient backup in case of errors or when timeliness is less important.

Notification: Services specify data of interest by providing a query to the MANGROVE notifier.[1] When the database is updated by a new data publication or a web crawl, the notifier forwards data matching that query to the corresponding services for processing. For instance, the calendar service registers its interest in all pages that contain <event> properties (or that had such properties deleted). When it receives notification of relevant new data, the calendar processes that data and updates its internal data structures, ensuring that content authors see their new data on the calendar with minimal delay.[2]

Service feedback: MANGROVE also provides a *service feedback* mechanism that is a key element of its architectural support for instant gratification. As noted earlier, services can register their interest in arbitrary RDF properties (e.g., `event`). Then, when a URL that contains such a property is published by an author, the services are automatically notified about the new information. Each notified service can return feedback to the author as shown in Figure 3.4. The feedback can identify problems encountered (e.g., a date was ambiguous or missing) or can confirm that the information was successfully "consumed" by the service.

The feedback mechanism supports instant gratification by making it easier for authors to immediately see the tangible output resulting from their new semantic data. Authors can click on any of the links shown in Figure 3.4 and they will be directed to a web page that shows how the information they *just* annotated is being used by a semantic service. For example, as soon as an event page is annotated and published, the organizer can click on a link and see her event appearing in the department's calendar. To be true to the 'instant' in 'instant gratification', publishing a page returns feedback to authors in about two seconds.

An important advantage of MANGROVE's declarative data representation is that it enables content that was designed for one particular service to also be exploited by other services. For instance, content intended for the Who's Who service described later may also improve the output of MANGROVE's calendar. Because MANGROVE services and infor-

[1] Currently, we assume that such a query consists of just a set of "relevant" RDF properties. More complex queries can be also supported efficiently [154].

[2] Note that while only registered services receive such notifications, any service that follows a simple API (of Mangrove or Jena[123]) may query the database for content.

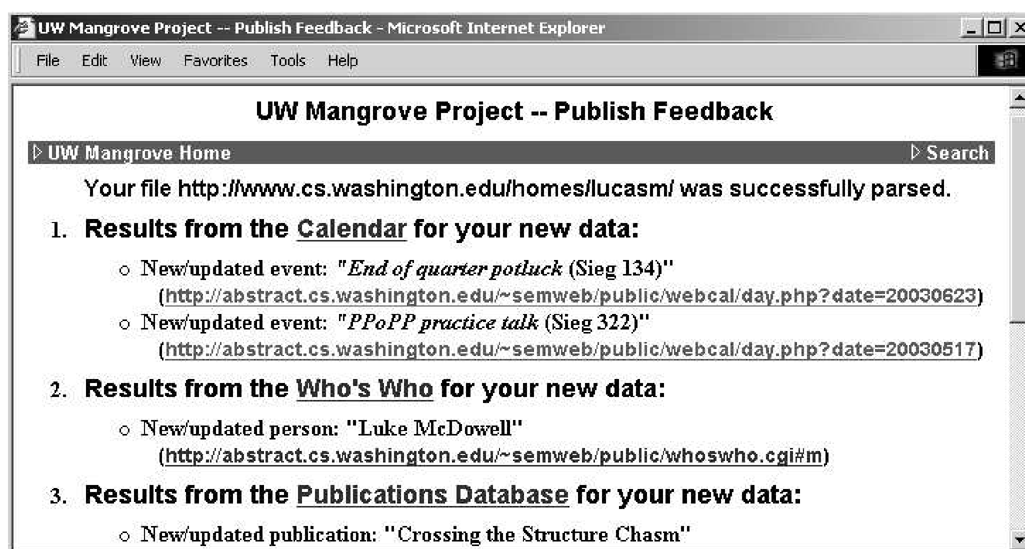


Figure 3.4: Example output from the service feedback mechanism. Services that have registered interest in a property that is present at a published URL are sent relevant data from that URL. The services immediately return links to their resulting output.

mation are created independently by different sets of people, there is thus the potential for authors to be unaware of additional services that consume their information and that would provide further motivation for them to author more semantic information. The service feedback mechanism acts as a service discovery mechanism that addresses this problem. Once a service registers its interest in a particular property, an author that publishes relevant information will be notified about that service's interest in the property.[3] We expect that users will typically publish content with a particular service in mind, and then decide whether or not to investigate and possibly annotate additional content for the services that they learn of from this feedback. As the number of services grows, an author can avoid "feedback spam" by explicitly selecting the services that send her feedback, by limiting their number, or by filtering them according to the criteria of her choice (e.g., by domain or category). Additional techniques for supporting useful feedback across very large numbers of services, content providers, and distinct ontologies is an interesting area for future work. Note that since the author is publishing information with the hope of making it broadly available, privacy does not seem to be a concern in this context.

[3]Very loosely speaking, this is analogous to checking which web pages link to your page — a service that is offered through search engines such as Google.

The service feedback mechanism also supports ease of use by helping authors to produce well-formed data. This addresses an important problem we experienced with MANGROVE prior to the development of feedback, where data would be published, but because of some error would not appear in the output of the expected service. Service feedback directly informs content authors of such problems, allowing them to easily produce usable data. We discuss further support for producing well-formed data for services below.

3.1.4 Practical Data Maintenance

Database and knowledge base systems have a set of mechanisms that ensure that the contents of a database are clean and correct. For example, database systems enforce integrity constraints on data entry, thereby eliminating many opportunities for entering "dirty" data. In addition, database applications control carefully who is allowed to enter data, and therefore malicious data entry is rarely an issue. On the Semantic Web, however, there is no single authority that everyone agrees upon, and hence such integrity constraints are difficult if not impossible to define. In addition, there is no central administration of the data that could enforce such constraints. Even more importantly, applying such constraints would be inconsistent with our ease of use design principle for two reasons. First, enforcing integrity constraints would create another hurdle preventing people from joining the Semantic Web, rather than enticing them. Second, on the Semantic Web authors who enter data may not even be aware of which services consume their data and what is required in order for their data to be well formed. Thus, enforcing such constraints would be very frustrating for such authors.

Hence, in MANGROVE our goal is for authors to be able to add content without considering constraints, and for services to be able to consume data that is cleaned and consistent as appropriate for their needs. Furthermore, when users do intend their data to be consumed by certain services, there should be a feedback loop that ensures that their data was in a form that the service could consume. Below we describe how MANGROVE supports such flexibility in a large-scale data sharing environment.

Deferring integrity constraints: On the HTML web, a user can put his phone number on a web page without considering whether it already appears anywhere else (e.g., in an employer's directory), or how others have formatted or structured that information. Despite that, users can effectively assess the correctness of the information they find (e.g., by inspecting the URL of the page) and interpret the data according to domain-specific conventions. In contrast, existing systems often restrict the way information may be expressed. For instance, in WebKB-2 [121], a user may not add information that contradicts another user unless the contradictions are explicitly identified first. Likewise, in SHOE [83], all data must conform to a specified type (for instance, dates must conform to RFC 1123).

MANGROVE purposefully does not enforce any integrity constraints on annotated data or restrict what claims a user can make. With the calendar, for instance, annotated events may be missing a name (or have more than one), dates may be ambiguous, and some data may even be intentionally misleading. Instead, MANGROVE *defers* all such integrity constraints to allow users to say anything they want, in any format. Furthermore, MANGROVE allows users to decide how extensively to annotate their data. For instance, the `instructor` property may refer to a resource with further properties such as `name` and `workPhone`, or simply to a string literal (e.g., "`John Fitz`"). Permitting such "light" annotations simplifies the annotation of existing HTML and allows authors to provide more detail over time, consistent with our gradual adoption principle.

To complement the deferral of integrity constraints, MANGROVE provides three mechanisms that facilitate the creation of appropriate data for services: service feedback (discussed earlier), data cleaning, and inspection of malicious information.

Data cleaning: The primary burden of *cleaning* the data is passed to the service consuming the data, based on the observation that different services will have varying requirements for data integrity. In some services, clean data may not be as important because users can tell easily whether the answers they are receiving are correct (possibly by following a hyperlink). For other services, it may be important that data be consistent (e.g., that an event have the correct location), and there may be some obvious heuristics on how to resolve conflicts. The

source URL of the data is stored in the database and can serve as an important resource for cleaning up the data.

To assist with this process, MANGROVE provides a *service construction template* that enables services to apply a simple rule-based *cleaning policy* to the raw results obtained from the RDF database. For instance, for course events, our calendar specifies a simple policy that prefers data from pages specific to a particular course over data from general university-provided pages. Thus, factual conflicts (e.g., a location change not registered with the university) are resolved in the course-specific page's favor. The cleaning policy also helps the calendar to deal with different degrees of annotation. For instance, to identify the instructor for a course lecture, the calendar simply requests the value of the `<instructor>` property, and the template library automatically returns the `<name>` sub-property of the instructor if it exists, or the complete value of that property if sub-properties are not specified.

Even when data is consistent and reliable, services still face the problem of *interpreting* the semantic data. For instance, dates found on existing web pages are expressed in natural language and vary widely in format. We note that while this problem of data interpretation is difficult in general, once users have explicitly identified different semantic components (e.g., with a `<date>` property), simple heuristics are sufficient to enable useful services for many cases. For instance, MANGROVE's service template provides a simple date and time parser that we have found very effective for the calendar service. In addition, semantic context can assist the cleaning process, e.g., to provide a missing year for an event specified as part of a course description. To utilize these features, services may create their own cleaning policy or use a default from the service template.

Inspection of malicious information: Another reason that we store the source URL with every fact in the database is that it provides a mechanism for partially dealing with malicious information. The highly distributed nature of the web can lead to abuse, which popular services such as search engines have to grapple with on a regular basis. Potential abuse is an issue for semantic services as well. What is to prevent a user from maliciously publishing misleading information? Imagine, for example, that a nefarious AI professor

purposefully publishes a misleading location for the highly popular database seminar in an attempt to "hijack" students and send them to the location of the AI seminar.

We have considered several approaches to combating this kind of "semantic spoofing." We could have an administrator verify information before it is published, creating a "moderated" semantic web. However, this non-automated approach prevents instant gratification and does not scale. Alternatively, we could enable automated publishing for password-authenticated users, but investigate complaints of abuse and respond by disabling an abuser's publishing privileges. This approach, however, violates our ease of use principle and prevents the same data from being easily shared by more than one semantic domain. Instead, we chose a fully automated system that mirrors the solution adopted by search engines — associating a URL with every search result and leaving decisions about trust to the user's discretion.

Thus, MANGROVE services associate an easily-accessible source (i.e., a URL) with each fact made visible to the user. For example, as shown in Figure 3.5, a user can "mouse over" any event in the calendar and see additional facts including one or more originating URLs. The user can click on these URLs to visit these pages and see the original context. Naturally, service writers are free to implement more sophisticated policies for identifying malicious information, based on freshness, URL, or further authentication. For instance, in case of conflict, our department calendar uses its previously mentioned cleaning policy to enable facts published from pages whose URL starts with `www.cs.washington.edu/education/` to override facts originating elsewhere.

3.1.5 Scaling MANGROVE

Scalability is an important design consideration for MANGROVE, and it has influenced several aspects of MANGROVE's architecture, such as our explicit publish/notification mechanisms. Nevertheless, the scalability of our current system is limited in two respects. First, at the logical level, the system does not currently provide mechanisms for composing or translating between multiple schemas or ontologies (all users annotate data with a common local

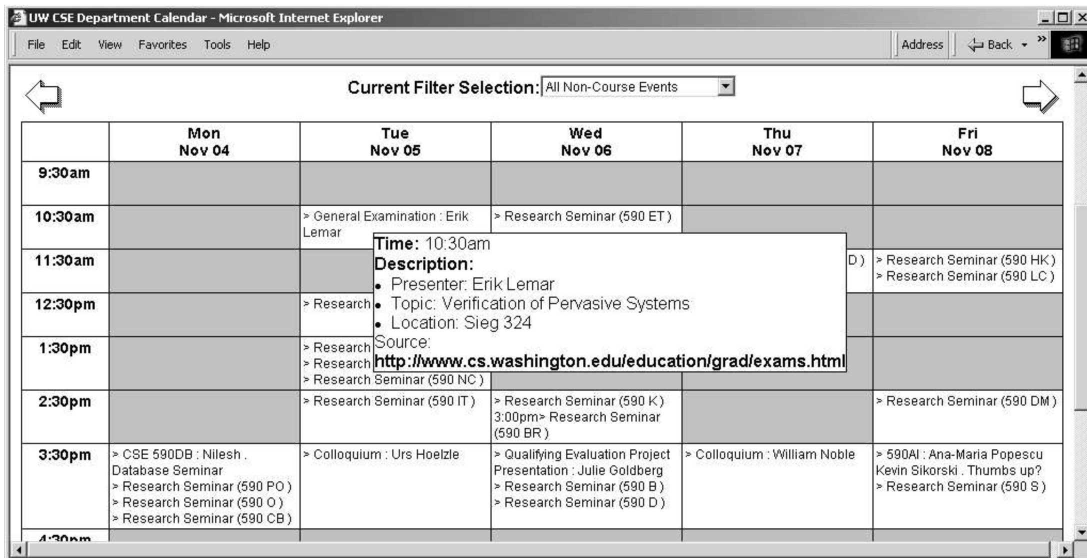


Figure 3.5: The calendar service as deployed in our department. The popup box appears when the user mouses over a particular event, and displays additional information and its origin. For the live version, see `www.cs.washington.edu/research/semweb`.

schema). Second, at the physical level, the central database in which we store our data could become a bottleneck.

We address both scalability issues as part of a broader project described in [76]. Specifically, once a department has annotated its data according to a local schema, it can collaborate with other structured data sources using a *peer-data management system* (PDMS) [77]. In a PDMS, semantic relationships between data sources are provided using *schema mappings*, which enable the translation of queries posed on one source to the schema of the other. Our group has developed tools that assist in the construction of schema mappings [47, 48], though these tools are not yet integrated into MANGROVE. Relying on a PDMS also distributes querying across a network of peers, eliminating the bottleneck associated with a central database.

3.2 Semantic Services in MANGROVE

One of the goals of MANGROVE is to demonstrate that even modest amounts of annotation can significantly boost the utility of the web today. To illustrate this, MANGROVE supports

a range of semantic services that represent several different web-interaction paradigms, including Google-style search and novel services that aggregate semantically annotated information. Below, we briefly discuss service construction and describe MANGROVE's initial services.

3.2.1 Constructing MANGROVE *Services*

Services are written in Java and built on top of the MANGROVE service construction template that provides the basic infrastructure needed for service creation. Currently, we use the Jena [123] RDF-based storage and querying system, which enables our services to pose RDF-style queries to extract basic semantic information from the database via a JDBC connection. Alternatively, services may use higher-level methods provided by the template. For instance, the template contains methods to retrieve all relevant information about a given resource from the RDF database, augmented with a summary of the sources of that information. The template also provides methods to assist with data cleaning and interpretation, as explained in Section 3.1.4.

The MANGROVE service template also aids service construction with support for incrementally computing and caching results. First, the template provides a standard `runUpdate()` method; this method is invoked by the MANGROVE notifier, which passes in a handle to the complete RDF dataset as well as a handle to the new RDF data for which the notification has occurred. Upon notification, invoked services rely primarily on the new data and local cached state, but an application can consult the complete dataset as necessary. Second, the template also provides a simple caching mechanism that maintains pre-computed information (e.g., processed event descriptions) and a mapping between each piece of information and its source page(s). For instance, when the calendar is invoked by the notifier, it uses those source mappings to determine what events may have changed, then updates the cache with the new information. The calendar viewer then uses this cache to quickly access the information requested by users.

Overall, MANGROVE makes services substantially easier to write by encapsulating commonly-used functionality in this service template. Moreover, MANGROVE's caching

features help to ensure that the performance is sufficient to provide instant gratification to both content authors and service users. To further minimize response time, MANGROVE's services are executed by a Jakarta Tomcat servlet engine. This provides substantially faster performance than the original CGI-based implementation, since services and their RDF data may remain memory resident in between service invocations.

3.2.2 Semantic Search

Consistent with our gradual adoption principle, we believe that annotation will be an incremental process starting with "light" annotation of pages and gradually increasing in scope and sophistication as more services are developed to consume an increasing number of annotations. It is important for this "chicken and egg" cycle that even light annotation yield tangible benefit to users. One important source of benefit is a Google-style search service that responds appropriately to search queries that freely mix semantic properties and text. The service returns the set of web pages in our domain that contain the text and properties in the query.

The interface to the service is a web form that accepts standard textual search queries. The service also accepts queries such as "assistant professor" <facultyMember> <portrait>?, which combines the phrase "assistant professor" with properties. Like Google, the query has an implicit AND semantics and returns exactly the set of pages in our domain containing the phrase "associate professor" and the specified properties. The `?` after the `<portrait>` property instructs the service to extract and return the HTML inside that property (as with the SELECT clause of a SQL query). Users select appropriate properties for the search from the simple schema available on the search page; an interesting area for future work is considering ways to make this selection even easier.

The service is implemented by sending the textual portion of the query (if any) to Google along with instructions to restrict the results to the local domain (`cs.washington.edu`). The MANGROVE database is queried to return the set of pages containing all the properties in the query (if any). The two result sets are then intersected to identify the relevant set of pages. When multiple relevant pages are present, their order in the Google results is

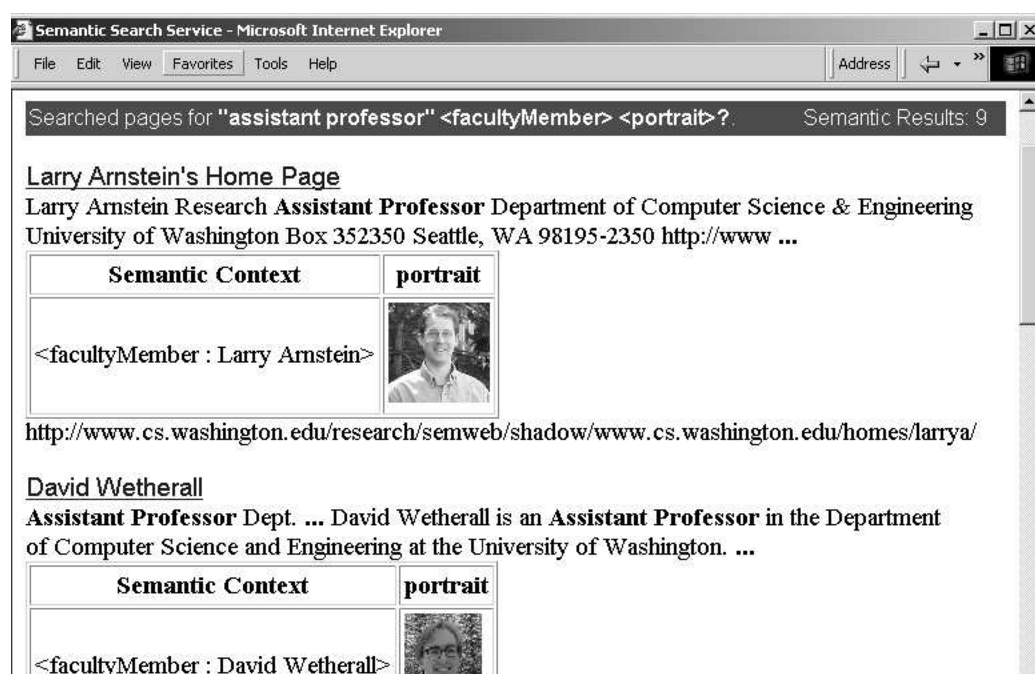


Figure 3.6: The semantic search results page. The page reproduces the original query and reports the number of results returned at the top. Matching pages contain the phrase "assistant professor" and the properties <facultyMember> and <portrait>. The `?` in the query instructs the service to extract the <portrait> from each matching page.

preserved to enable more prominent pages to appear first in the list. Finally, any extraction operations indicated by one or more question marks in the query are performed and included in the result (see Figure 3.6). Like Google, not every result provides what the user was seeking; the search service includes *semantic context* with each result — a snippet that assists the user in understanding the context of the extracted information. The snippet is the `name` property of the extracted property's subject. For instance, when extracting the <`portrait`> information as shown in Figure 3.6, the snippet is the name of the faculty member whose portrait is shown.

With its ability to mix text and properties, this kind of search is different from the standard querying capability supported by MANGROVE's underlying RDF database and other Semantic Web systems such as SHOE [83] and WebKB [121]. Our search service has value to users even when pages are only lightly annotated, supporting our goal of gradually enticing users onto the Semantic Web.

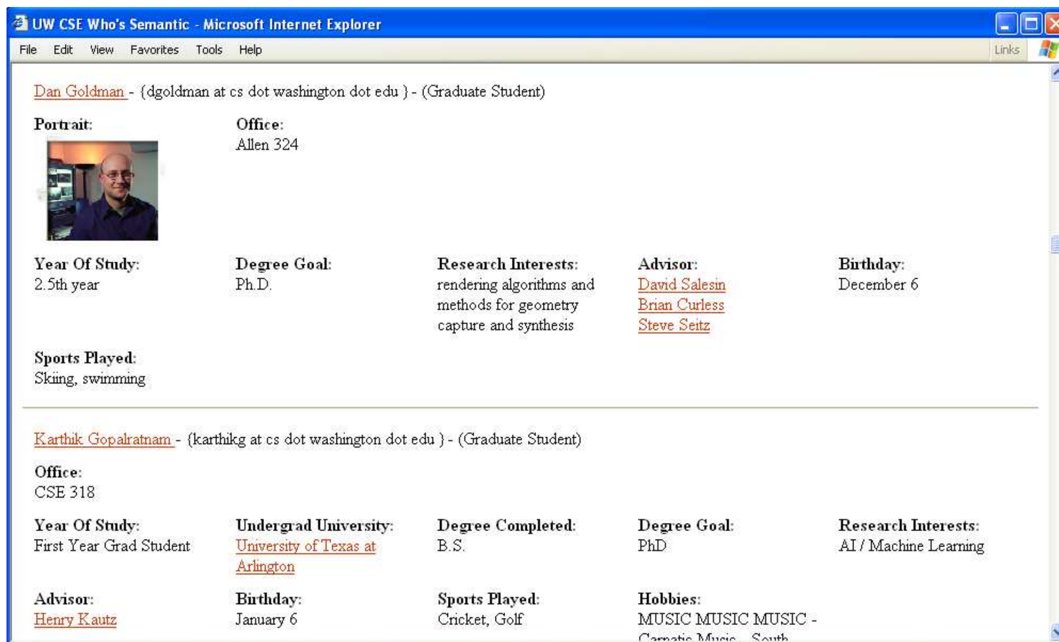


Figure 3.7: The Who's Who service as deployed in our department. Notice how it allows users to provide as much information as they like, in whatever format is desired.

3.2.3 Aggregation Services

Aggregation services provide useful views on data from the Semantic Web. We describe the aggregation services we implemented with MANGROVE below.

First, our Who's Who service compiles pictures, contact information, and personal data about people within an organization. In our department, a static Who's Who had existed for years, but was rarely updated (and was woefully out-of-date) because of the manual creation process required. Our dynamic Who's Who (see Figure 3.7) directly uses more up-to-date information from users' home pages, enabling users to update their own data at any time to reflect their changing interests.

Our experience with the Who's Who service illustrates an important advantage of the MANGROVE annotation approach over other approaches such as asking users to enter information into databases via web forms. A large amount of useful data already exists in hand-crafted personal and organizational web pages, and the active viewing of this data over the web motivates users to keep this information up-to-date. Once these pages are tagged,

MANGROVE automatically *leverages* the author's HTML updates to keep the information in its database up-to-date without additional effort on the author's part. Thus, manually maintained databases often become stale over time whereas MANGROVE's information is as fresh as HTML.

Whereas Who's Who merely collects information from a set of web pages, our Research Publication Database compiles a searchable database of publications produced by members of our department based on the information in home pages and project pages. To support ease of use, this service applies simple heuristics to avoid repeated entries by detecting duplicate publications. To support gradual adoption, only a single `<publication>` property enclosing a description of the publication is required in order to add an entry to the database, which facilitates light, incremental annotation. However, users may improve the quality of the output and the duplicate removal by specifying additional properties such as `<author>` and `<title>`.

Our most sophisticated service, the department calendar (shown in Figure 3.5), automatically constructs and updates a unified view of departmental events and displays them graphically. As with our other services, the calendar requires only a date and name to include an event in its output, but will make use of as much other information as is available (such as time, location, presenter, etc.).

Department members are motivated to annotate their events' home pages in order to publicize their events. We initially seeded the calendar with date, time, and location information for courses and seminars by running a single wrapper on a university course summary page. Users then provide more detail by annotating a page about one of these events (e.g., users have annotated pre-existing HTML pages to identify the weekly topics for seminars). Alternatively, users may annotate pages to add new events to the calendar (e.g., an administrator has annotated a web page listing qualifying exams). Typically, users annotate and publish their modified pages, the calendar is immediately updated, and users then view the calendar to verify that their events are included. For changes (e.g., when an exam is re-scheduled), users may re-publish their pages or rely on the MANGROVE web crawler to capture such updates later.

It is easy to conceive of other services as well. We view the services described above as

informal evidence that even light annotation can facilitate a host of useful services, which motivates further annotation, etc.

3.2.4 Inference and Referential Integrity Issues

Mangrove services currently perform very limited, though practical, inferencing. For instance, the publications database can infer missing information (e.g., the author of a paper) from context (e.g., the paper was found on the author's home page). Likewise, the calendar can infer the instructor associated with a lecture `event` when that event is embedded inside a `course` element. These inferences are implemented directly by the application service, aided by some specific methods from the service construction template, rather than by a general mechanism associated with the RDF database. As discussed in Chapter 2 this application-based inferencing is well-targeted for specific inferencing needs, but is error-prone and difficult to modify. Extending Mangrove to use more general reasoners is an interesting opportunity for future work. For instance, we could utilize a RDF Schema reasoner to simplify Mangrove's applications with suitable subclass reasoning (e.g., to infer that all subclasses of `Person` should have a `name`). Note, however, that doing the slightly more complex reasoning described above (for publications and events) requires capabilities that are not supported by either RDF Schema or OWL [178]. Hence, currently there appears to be little benefit to using OWL in Mangrove, though Mangrove could benefit from an appropriate rules language combined with an efficient reasoner.

Another challenging issue for Mangrove services is dealing with multiple references to the same person or object. For instance, an annotated personal web page may be published via slightly different URLs, causing Mangrove to treat the content as two distinct people that happen to have identical descriptive information. Likewise, if a person annotates personal content on multiple web pages, Mangrove may believe that the information is about more than one person. These are problems for RDF content in general and can be solved by the content author (in either RDF or MTS) by providing a unique `about` attribute for the person. However, this is more complex for typical authors and does not address the problem of the same person being referred to by different `about` attributes.

Currently, MANGROVE offers a very primitive solution to this problem by constructing a canonical URL for a published document based on local knowledge of the departmental webspace, then using this URL as a default `about` attribute for the first object described on a page. For instance, a `graduateStudent` described with annotations on the page `www.cs.washington.edu/homes/lucasm/contact.html` is assigned an `about` attribute of `www.cs.washington.edu/homes/lucasm/`. This approach effectively coalesces all the information about a person in one user's webspace into a single entity unless explicit `about` attributes are used. In most cases this technique has been sufficient for our purposes, but more sophisticated techniques based on identifying related entries [19, 33, 38, 59, 124, 50, 122] or isomorphic RDF graphs [31] could be useful.

3.3 Experience with MANGROVE

This section presents our initial experience using and evaluating MANGROVE. Our goal is to answer some basic questions about the MANGROVE approach:

1. **Feasibility:** Can MANGROVE be used to successfully tag and extract the factual information found in existing HTML pages?

2. **Benefit:** Can MANGROVE services actually benefit users when compared with popular commercial services? Specifically, we attempt to quantify the potential benefit of MANGROVE's semantic search service as compared with Google.

3. **Practice:** Given the actual costs and benefits in a deployment, will MANGROVE services be invoked by actual users? Are these users willing to contribute content to MANGROVE?

These questions are mostly qualitative, however we develop some simple measures in an attempt to quantify the feasibility and benefits of our approach.

3.3.1 Information Capture

To test the extent to which (1) our system can successfully extract a range of information from existing HTML pages, and (2) our existing web actually contains the information of interest, we created a copy of our department's web space for experimentation. The department web consists of about 68,000 pages whose HTML content is about 480 MB. We then tagged the home pages of all 44 faculty members using the graphical tagger, focusing particularly on adding 10 common tags such as `<name>`, `<portrait>`, and `<advisedStudent>`. Four graduate students were instructed to tag and publish each document, but not to make *any* other changes to the original HTML. The students were familiar with MANGROVE, though some had previously tagged only their own home page.

We evaluate tagging success by examining the output of our Who's Who service and comparing it with the original HTML documents. Of the 440 possible data items (e.g., a faculty member's name or picture), 96 were not present in the original HTML. For instance, only half of the professors had their office location on their home page. Of the remaining 344 facts, the vast majority (318, or 92.4%) were correctly displayed by Who's Who, while 26 had some sort of problem. Nine of these problems were due to simple oversight (i.e., the data was present but simply not tagged), while eight items had tagging errors (e.g., using an incorrect tag name). For six data items, it was not possible to succinctly tag the data with MTS. For instance, MTS presently cannot tag a single string as both a home and office phone number. Finally, three tagged items revealed minor bugs with the Who's Who service itself.

Thus, despite the variety of HTML pages (we have no standard format for personal home pages) and the presence of some inevitable annotation errors, we successfully extracted a large amount of relevant information and constructed a useful service with the data. This simple measurement suggests that while additional MANGROVE features may improve the tagging process, the overall annotation and extraction approach is feasible in practice.

3.3.2 Benefits of MANGROVE *Search*

Using the tagged data discussed above, we examined the effectiveness of MANGROVE's search service (described in Section 3.2.2). As a simple search exercise, we issued a small set of queries to retrieve the picture and phone number of all assistant and associate professors in our department. For example, we issued the query:

<facultyMember> <jobTitle="assistant professor"> <portrait>? For comparison, we sent comparable search queries to Google and to MANGROVE's tag-only search, which accepts sets of tags (with no text terms) as queries.

Obviously, Google has different goals than our search service, so the results are not necessarily comparable, however the exercise helps to illustrate the potential benefit from even a modest amount of tagging. When sending queries to Google, we included `site:cs.washington.edu` to restrict the query to our site. We tried several variants of each query (e.g., `"assistant professor,"` `"assistant professor" phone`, `"assistant professor" phone OR voice`, etc.). For finding photos, we also queried the Google image search directly. In each case, we inspected all results returned by Google for the desired photo or phone number.

In addition, we wanted to assess how robust MANGROVE is to tagging omissions or errors. What happens, for example, when the `<jobTitle>` is omitted or applied incorrectly? Users can fall back on MANGROVE's tag+text search, which filters Google results using tag information as explained in Section 3.2.2. In our tag+text queries, we combined a phrase (e.g., `"assistant professor"`) with the tag `<facultyMember>` and the tag to be extracted (`<workPhone>` or `<portrait>`). Figure 3.6 shows the results of one such query.

Table 3.1 summarizes the results for our three experimental conditions: Google, tag-only search, and tag+text search. We use the standard information retrieval metrics of precision (p) and recall (r), and combine them into an f-score (f) as follows:

$$f = \frac{(\beta + 1)pr}{(\beta p + r)}$$

The f-score is a standard method of combining recall and precision to facilitate compari-

Table 3.1: Comparison of Search Services. In each box, the first value is the *f-score* of the query, followed by the *precision* and *recall* in parentheses. Within each row, the values in bold represent the maximum value for that metric.

Search Objective	Google f (Prec.,Rec.)	Tag-only Search f (Prec.,Rec.)	Tag+Text Search f (Prec., Rec.)
Assistant Professor photos	0.75 (**100%**,60%)	0.82 (**100%**,70%)	**0.84** (89%, **80%**)
Associate Professor photos	0.52 (75%,40%)	0.89 (**100%**,80%)	**0.91** (83%,**100%**)
Assistant Professor phone numbers	0.64 (58%,70%)	0.89 (**100%**,80%)	**0.95** (91%,**100%**)
Associate Professor phone numbers	0.29 (19%,60%)	0.67 (75%,60%)	**0.80** (**80%**, **80%**)

son [161].[4] As is commonly done, we set the parameter β to 1 in order to weight precision and recall equally. In this experiment, precision is the percentage of the results, for each engine, that were correct; recall is the percentage of the total correct answers returned by each engine.

The table supports some tentative observations. First, tags can substantially improve precision over Google's results. Second, and more surprising, tags often result in improved recall over Google as well. The reason is that querying Google with a query such as `"assistant professor"`, restricted to our site, returns 176 results with very low precision. A query that yields much better precision and a much higher f-score for Google is `"assistant professor" phone OR voice`; however, this longer query yields lower recall than the tag-based searches, because it only returns pages that contain the word `phone` or the word `voice`. Since the tag-based searches yield both better precision and better recall, it is not surprising that their f-score is substantially better than Google's. Of course, far more extensive experiments are needed to see if the above observations are broadly applicable.

The table also enables us to compare the two variants of tag-based search. We see that tag-only search tends to have very high precision, but lower recall when compared to tag+text search. Tag+text has higher recall because it is more robust to omitted or incorrect tags. For example, in some cases a professor's rank was not tagged properly due to human error. Tag+text search also offers the ability to search based on data that was not tagged because a suitable tag did not exist, as would be the case if we had omitted `<jobTitle>` from the schema.

[4] To be fair to Google, we tried multiple formulations of each query as mentioned above. The results reported for Google in each row are the ones whose f-score was maximal.

Both of our tag-based searches have the further benefit that they can extract the tagged information from web pages (see Figure 3.6), whereas Google only sometimes manages to extract this information with its image search or in its result snippets. This feature of our service makes it much simpler to quickly view the results and facilitates the use of search queries as building blocks for more sophisticated services.

These measurements are from a single limited domain and the results, while thought-provoking, are far from definitive. Nevertheless, the measurements do provide evidence for the feasibility of MANGROVE and its potential for supporting value-added services for users. One might argue that the comparison of MANGROVE's search with Google is not fair because the semantic search makes use of additional information in the form of tags that have to be inserted into HTML pages manually. However, our goal is not to argue that MANGROVE has a better search algorithm, but rather to quantify the benefit that can result from this annotation effort.

3.3.3 Deployment Experience

The results above highlight some of the potential benefits of MANGROVE on real data, but were carried out in a controlled setting. The ultimate question, however, is whether MANGROVE is usable and useful for ordinary people in an actual deployment. Below, we make a number of observations regarding this question gleaned from almost two years of MANGROVE's deployment in our department.

First, simple services such as the calendar can offer substantial added value over other forms of accessing the same information. For instance, in the 21 months the online calendar has been operational, it has received more than 7600 distinct visits, with an average of about two page views per visit. Figure 3.8 plots this activity over time. After an initial burst of activity, this graph shows that calendar usage has remained fairly constant around 300-400 distinct visits per month, with a noticeable drop in activity over the summer. These measurements show that community members have continued to find the calendar service useful, even though the same raw data is available elsewhere on the web, validating our claim that there is value in extracting existing information for novel presentations based on associated semantics.

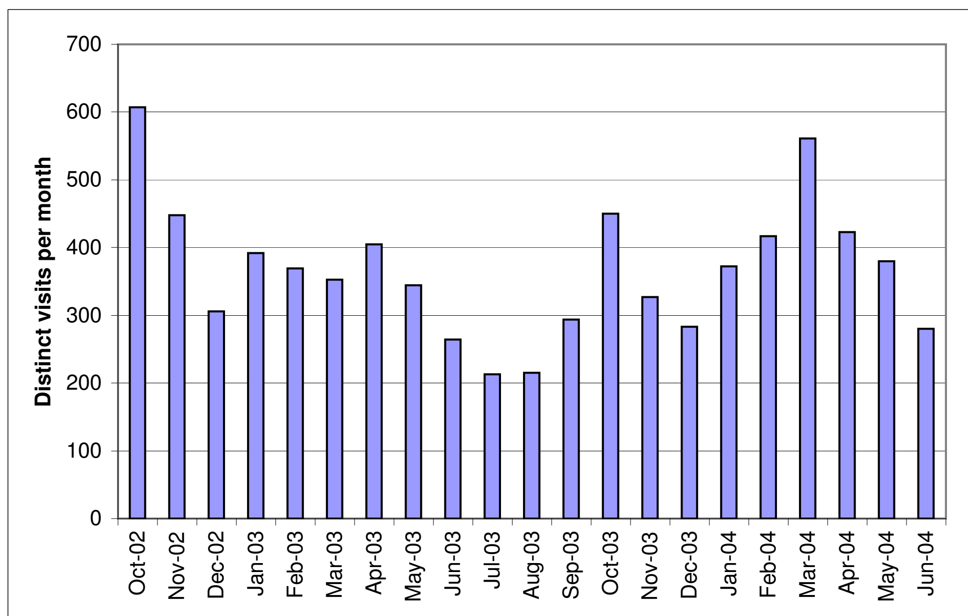


Figure 3.8: The number of distinct visits to the MANGROVE calendar during each month. These values exclude traffic from webcrawlers and MANGROVE team members.

Second, users are willing to annotate their documents if the process is easy and interesting services exist to use the annotations. For instance, administrators, students, and faculty have all utilized annotation to promote a wide range of events, ranging from official departmental events to visitor schedules to informal events at a local pub. Our experience also highlights the importance of service popularity in providing instant gratification to content authors. For example, during the first year of the Who's Who operation, only eight graduate students annotated content in order to be included. Then, however, the official departmental Who's Who was changed to link to the MANGROVE version. In the two weeks after this change was announced, thirty additional graduate students performed some annotation in order to add themselves to the service. Since then, new contributions have continued, though more slowly, increasing the total number to forty-six participants. In contrast, the lack of an existing user base and prominent link to the Who's Who for undergraduates means that a total of just four such students have bothered to annotate content for this service, though our department has many more undergraduate students than graduate students.

Third, MANGROVE is focused on enabling the annotation of *existing* content. However, in our deployment users also contributed a significant amount of *original* content designed for specific services. For instance, a previously mentioned departmental administrator provides content for the calendar about graduate student exams. Typically, she does not annotate existing data, but rather creates annotated descriptions for newly scheduled exams by copying and modifying the entry for a previous exam. Likewise, a significant number of graduate students (20 out of 46) chose to contribute at least some of their content to the Who's Who by filling out and making web-accessible a provided MTS template. Modifying this template was in some ways easier for a student than changing his existing web page, and made it easy to publish information (e.g., his birthday) that he may not have had on his existing web page but that was traditionally a part of Who's Who entries. Thus, MANGROVE's ease of use and focus on instant gratification can motivate the contribution of both existing and original content to the Semantic Web. However, more work is needed to better support this creation of original content. For instance, if some information provided in these alternative manners duplicates content already present in HTML, we must address the consistency problems that will inevitably arise.

Finally, an important aspect of MANGROVE is how annotated pages evolve over time. In particular, if pages are modified with annotation-unaware editors such as Microsoft FrontPage, it is possible for the annotations to be lost or corrupted. In this area we had mixed results. For personal home pages, we found that annotations seemed to endure fairly well. For instance, of the graduate students that contributed HTML content usable by the Who's Who, only one student's web page is currently lacking well-formed semantic content. To some extent, this may reflect the usage of simpler text editors that make the semantic tags more obvious when editing.

On the other hand, our experience with the evolution of course and seminar web pages has been more disappointing. For instance, over the past two years a number of seminar web pages have been annotated, either by the seminar organizer or by a MANGROVE team member on their behalf. We had hoped that once annotated, future versions of a seminar's page would continue to benefit from the annotations, and that the number of annotated seminars would thus grow over time. Unfortunately, this has not occurred, for perhaps

two reasons. First, one potential explanation is that MANGROVE and its calendar failed to reach "critical mass" in this context. While the calendar provides a useful summary of the times of all seminars, it never reached the point where a majority of seminars had associated annotations, and hence provided more details on seminar topics, presenters, etc. To some extent, this may reflect the fact that we performed relatively little advertising for MANGROVE — a few newsgroup announcements and some small talks describing the research — and substantially more publicity is required to launch a new system. Second, we found that many seminar pages appear to be frequently recreated "from scratch" each academic quarter, often by administrators who were not aware of the old annotations. Hence, the new pages lacked annotations and there was not sufficient demand and awareness to insist that they be added. Instead, we've had greater success with more permanent pages (e.g., those announcing graduate student exams or events of the local ACM chapter). These sources, with more stability in their content and in the human editors responsible for them, have continued to contribute a significant amount of new semantic content, even though doing so requires a small amount of work to maintain the annotations.

Clearly, additional deployments in different universities, organizations, and countries are necessary to gain additional insight and further refine MANGROVE's design. Nonetheless, our experience strongly suggests that the MANGROVE system and services are both feasible and beneficial.

3.4 Related Work

MANGROVE is the first system to articulate and focus on *instant gratification* as a central design principle for a Semantic Web system. Many of the key differences between MANGROVE's architecture and that of related Semantic Web systems follow from this distinct design goal. We discuss these differences in more detail below.

Haustein and Pleumann [81] note the importance of semantic data being "immediately visible" in a way that yields benefit to content authors. Their system, however, primarily provides this benefit by eliminating redundancy between HTML and semantic data, and then using this data and templates to dynamically generate attractive HTML or RDF

content. While these features potentially make maintaining interrelated HTML and RDF data more convenient, their system is very different from MANGROVE. Specifically, they have a different architecture that doesn't support explicit publication, notification, or service feedback. In addition, we have identified and deployed a set of instant gratification services as an essential part of MANGROVE, which are absent from their system.

Two other projects most closely related to our work are OntoBroker [44] and SHOE [83], both of which make use of annotations inside HTML documents. Although SHOE's language did permit users to inject annotations into HTML pages, their annotations do not actually use the existing HTML content. Thus, both with SHOE and with OntoBroker's RDF-based annotations, all factual HTML data must be repeated in the semantic annotation, leading to the redundancy and maintenance problems discussed earlier.[5]

SHOE's services, like those of many other systems, primarily consisted of tools to simply search or view semantic data, although their "Path Analyzer" [84] provided a convenient interface for exploring relationships among concepts. OntoBroker did implement a number of services, such as a Community Web Portal [167] and services intended to assist business processes [142]. SHOE and OntoBroker, however, primarily rely upon periodic web crawls to obtain new information from annotated HTML, thus preventing instant gratification and content creation feedback. In addition, MANGROVE has the advantage of enabling useful services even when content is only lightly annotated. For instance, while OntoBroker's "SoccerSearch" service [142] tries a semantic search and then a textual search if the former fails, MANGROVE's semantic+text search service can profitably combine both types of information.

As an alternative to crawling, some systems provide a web interface for users to directly enter semantic knowledge [121, 44] or to instruct the system to immediately process the content of some URL [121]. However, we are aware of no existing systems that support this feature in a manner that provides instant gratification for typical web authors. For instance, the WebKB-2 system supports a command to load a URL, but this command

[5]Early work with OntoBroker's HTML-A annotation language and OntoPad [167] originally permitted annotations to reuse existing data; however, later work with CREAM [78] lost this very useful feature as the group focused on an RDF encoding.

must be embedded within a script, and existing data must be manually deleted from the repository before a (modified) document can be reprocessed.

WebKB-1 and WebKB-2 [120, 121] also provide a way to embed semantic knowledge in HTML documents, this time using expressive conceptual graphs and an extensive ontology, but generally require the duplication of information in those documents. In addition, their services are currently limited to either information browsing or a semantic-only search. The OntoSeek project [72] addresses goals of information retrieval similar to WebKB but does not support the encoding of information in HTML pages and does not provide any services beyond search.

Conceivably, we could leave the data in the HTML files and access them only at query time. In fact, several data integration systems (e.g., [34, 4, 91]) do exactly this type of polling. The difference between MANGROVE and such systems is that in the latter, the system is given *descriptions* of the contents of every data source. At query time, a data integration system can therefore prune the sources examined to only the relevant ones (typically a small number). In MANGROVE we cannot anticipate *a priori* which data will be on a particular web page, and hence we would have to access every page for any given query – clearly not a scalable solution. An additional reason why we chose publishing to a database over query-time access is that the number of queries is typically much higher than the number of publication actions. For example, people consult event information in the department calendar much more frequently than announcing new events or changing the events' time or location.

Other systems that have permitted the annotation of HTML documents include the "lightweight databases" of Dobson and Burrill [49] and the annotation tool of Vargas-Vera et al. [174], but both systems are merely modules for complete Semantic Web systems. CREAM [78] provides a sophisticated graphical tool that allows annotation of HTML documents similar to our graphical tagger, but it must replicate data due to its use of RDF. Some systems (i.e., Annotea [97]) avoid redundancy by using XPointers to attempt to track which part of a document an annotation refers to, but this approach may fail after document revisions and only applies to XHTML documents, which makes it incompatible with the majority of information on the web.

In Mangrove we chose to store annotations within the original HTML pages, for simplicity and to enable easy updates of the annotations when the source data changes. However, the overall architecture is also consistent with external annotation, where a user may annotate any page and the annotations are transmitted directly to a semantic database, as possible with CREAM [78], Annotea [97], or COHSE [12]. A side effect of these tools is that they automatically aggregate data as with our explicit publish operation; Mangrove completes the necessary features for instant gratification by providing service notification, feedback, and a host of useful services.

Recall that the TAP semantic search [74] executes independent textual and semantic searches based on traditional text queries. This service is easy to use but cannot currently exploit information from one search in the other, nor can the user specify the type of semantic information that is desired. QuizRDF [41] searches combine textual and semantic content, but are more restricted than those provided by Mangrove's search service, making them more difficult to use as a building block for other services. However, QuizRDF has an elegant user interface that more readily assists users in identifying relevant properties.

For storing and accessing RDF data, we utilize the Jena toolkit [123]. Other systems that also offer centralized RDF storage include Kaon [136] and Sesame [27]. Edutella [141] extends these approaches to provide RDF annotation, storage, and querying in a distributed peer-to-peer environment, and proposes some services, but primarily assumes the pre-existence of RDF data sources rather than considering the necessary architectures and services to motivate Semantic Web adoption. We view these systems as valuable modules for complete Semantic Web systems such as Mangrove. In contrast, Mangrove supports the complete process of content creation, real-time content aggregation, and execution of services that provide instant gratification to content authors.

The W3C and many others have advocated the use of digitally signed RDF to ensure the reliability of RDF data [177, 179, 90]. This approach may be logical in cases where data integrity is essential, but is too heavyweight for average users, and is not necessary for most services (where usability and freshness are more important). Furthermore, signed RDF only solves half of the data authentication problem — the part Mangrove solves by simply maintaining the source URL for all content. The more difficult problem is, given

the known source of all data, how should services decide what data sources and data are reliable, and how should this information be presented to the user? This chapter highlights how some simple policies can work well for common services and argues for revealing the source of the data to the end user, similar to the way users ascertain the validity of web content today.

3.5 Summary

This chapter introduced the MANGROVE architecture and described how it supports the complete Semantic Web "life-cycle" from content authoring to practical services. We demonstrated how elements of the architecture support each of our three design principles. Specifically, MANGROVE supports instant gratification with a loop that takes freshly published semantic content to semantic services, and then back to the user through the service feedback mechanism. Next, MANGROVE provides gradual adoption by seeding its services with a variety of useful data and by providing the MTS syntax, which allows content to be annotated incrementally in a way that makes future maintenance trivial. Finally, MANGROVE supports ease of use by reusing familiar application interfaces, providing a simple graphical annotation tool, deferring integrity constraints to the services, and associating a source URL with every fact in the database for lightweight trust management.

Overall, MANGROVE provides a range of semantic services, combined with a system designed to drive adoption by applying our three key design principles. These services have been deployed and actively used in our department for almost two years, and we provided evidence supporting our claim that applying the design principles both enables and motivates non-technical people to participate in the Semantic Web. The next chapter will consider the application of these ideas to the domain of email.

Chapter 4

SEMANTIC EMAIL

This chapter describes a general notion of *semantic email*, focusing particularly on the theory and application of *semantic email processes* (SEPs). We explain how these processes can provide instant gratification to the user via practical reasoning that can scale to support SEPs with many participants. In addition, we examine how our design principles of gradual adoption and ease of use can address usability challenges that arise in this context. The next chapter then introduces a language for specifying such processes and examines some challenges that such a language raises.

4.1 Introduction

Email offers a particular opportunity where the cost/benefit equation associated with structuring data can be changed dramatically, thus potentially extending the impact of the Semantic Web far beyond its current reach. Like the WWW, email is a vast information space where people spend significant amounts of time, yet that typically has no semantic features (aside from generic header fields). While the majority of email will remain this way, Chapter 1 listed a number of common examples where adding semantic features to email offers opportunities for improved productivity. In general, there are at least three ways in which semantics can be used to streamline aspects of our email habitat:

1. **Update:** We can use an email message to add data to some source (e.g., to add an event announcement to a web calendar)
2. **Query:** Email messages can be used to *query* other users for information. Semantics associated with such queries can then be used to automatically answer common questions (e.g., seeking my phone number or directions to my office).

3. **Process:** We can use semantic email to manage simple but tedious processes that we currently handle manually (e.g., to organize meetings or give away/auction items).

Because email is not set up to handle these tasks effectively, accomplishing them by hand can be tedious, time-consuming, and error-prone. The techniques needed to support the first two uses of semantic email depend on whether the message is written in text by the user or formally generated by a program on the sender's end. In the user-generated case, we would need sophisticated methods for extracting the precise update or query from the text (e.g., [52, 103]). In both cases, we require some methods to ensure that the sender and receiver share terminologies in a consistent fashion.

This chapter focuses on the third use of semantic email to streamline processes, as we believe it has the greatest promise for increasing productivity and is where users currently feel the most pain. These processes support the common task where an *originator* wants to (1) ask a set of *participants* some questions, (2) collect their responses, and (3) ensure that the results satisfy some set of *goals*. In order to satisfy these goals, the SEP *manager* may utilize a number of *interventions* such as rejecting a participant's response or suggesting an alternative response.

The remainder of this chapter is organized as follows. We first examine how to provide instant gratification to originators by formally defining practical SEPs and solving relevant inference problems for them. Specifically, Section 4.2 introduces a formalization for SEPs that exposes several fundamental reasoning problems that can be used by the semantic email manager to facilitate SEP creation and execution. In particular, a key challenge is to decide when and how the manager should intervene to direct the process toward an outcome that meets the originator's goals. We address this challenge with two different formal models. First, Section 4.3 describes a model of *logical* SEPs (L-SEPs) and demonstrates that it is possible to automatically infer which email responses are acceptable with respect to a set of ultimately desired constraints in polynomial time. Second, Section 4.4 describes a model of *decision-theoretic* SEPs (D-SEPs) that alleviates several shortcomings of the logical model, and presents results for the complexity of computing optimal policies for D-SEPs. These capabilities all support instant gratification by enabling automated, goal-directed processing

of messages on the originator's behalf. Next, Section 4.5 discusses implementation issues related to gradual adoption and ease of use that arise for semantic email and how we have addressed these in our system. Finally, Section 4.6 describes our experience with the deployed system, Section 4.7 contrasts our approach with related work, and Section 4.8 concludes.

4.2 Semantic Email Processes

Our formalization of SEPs serves several goals. First, the formalization captures the exact meaning of semantic email and the processes that it defines. Second, it clarifies the limitations of SEPs, thereby providing the basis for the study of variations with different expressive powers. Finally, given the formalization, we can pose several reasoning problems that can help guide the creation of semantic email processes as well as manage their life cycle. We emphasize that the *users* of SEPs are not expected to understand a formalization or write specifications using it. Generic SEPs are written by trained *authors* (who create simple constraints or utility functions to represent the goal of a process) and *invoked* by untrained users. The semantic email system then coordinates the process to provide the formal guarantees we describe later.

Figure 4.1 illustrates the three primary components of a SEP:

- **Originator:** A SEP is initiated by the *originator*, who is typically a person, but could be an automated program or agent.

- **Manager:** The originator invokes a new SEP by sending a message to the semantic email *manager*. The manager sends email messages to the *participants*, handles responses, and requests changes (i.e., intervenes) as necessary to meet the originator's goals. The manager stores all data related to the process in an RDF supporting data set, which may be configured to allow queries by external services (or other managers). To accomplish its tasks, the manager may also utilize external services such as inference engines, ontology matchers, and other Semantic Web applications, as described further below. The manager may be a shared server or a program run directly by the originator.

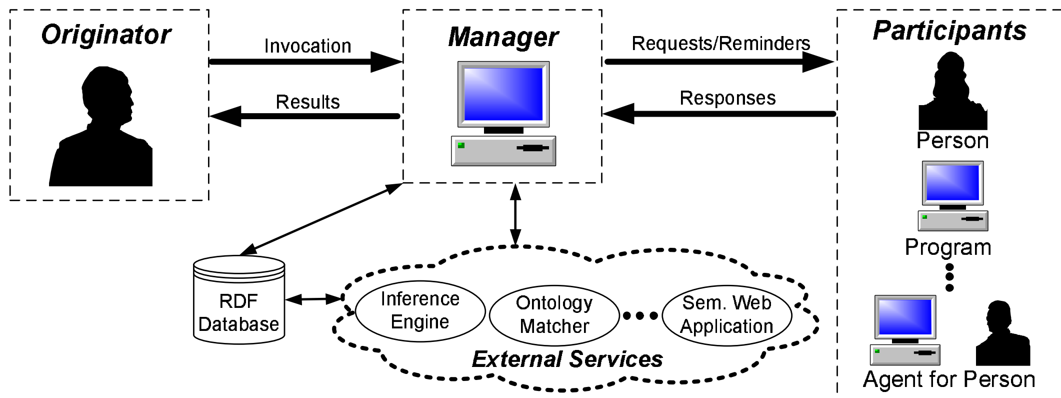


Figure 4.1: The invocation and execution of a SEP. The originator is typically a person, but also could be an automated program. The originator invokes a SEP via a simple web interface, and thus need not be trained in the details of SEPs or even understand RDF.

- **Participants:** The participants respond to messages received about the process. A participant may be a person, a standalone program (e.g., to represent a resource such as a conference room), or a software agent that acts on behalf of a person (e.g., to automatically respond to requests where possible, deferring others to the person). We assume that email addresses uniquely determine individuals or sets of potential participants in the process.

Informally speaking, the execution of a process is achieved by the supporting data set and the set of data updates that email recipients make as they respond. Logically, we describe our data in the model below as a set of relations (i.e., the relational database model). However, as the application domains get more complex, we expect to use a richer representation language. To enable these future extensions as well as interactions with other Semantic Web applications, our system implements the data set in RDF, using the Jena storage system [123].

We illustrate our formalization with the running example of a "balanced potluck." The originator invokes a process to announce the potluck and ask everyone whether they are bringing an appetizer, entree, or dessert. The originator also expresses a set of goals for the potluck. For example, he may specify that the number of appetizers, entrees, or desserts

should differ by at most two. Note that this particular problem, while it has a number of other uses (e.g., distributing N persons evenly among K committees or time slots), is just an example. Both our formalization and implementation of SEPs support a much broader range of uses.

The manager seeks to expedite the execution of this process and to achieve the originator's goals. Because all SEP data is represented declaratively, there are a number of ways in which reasoning over this data can enhance the manager's operation:

- **Predicting responses:** The manager may be able to infer the likely response of some participants even before sending any requests. For instance, the manager could employ another Semantic Web application or data source to detect that a suggested meeting time is unacceptable for a certain participant, based on information from calendars, course schedules, or other processes. The manager could use this information either to warn the originator as the process is being created, or to serve as a surrogate response until a definitive answer is received. Also, the manager could add a helpful annotation to the request sent to the participant, indicating what time is likely to be a conflict and why. As suggested above, this same reasoning could also be profitably employed on the participant's end, where an agent may have additional information about the participant's schedule.

- **Interpreting responses:** Typically, the originator will provide the participants with a finite set of choices (e.g., `Appetizer, Entree, Dessert`). However, suitable reasoning could enable substantially more flexibility. For instance, we could allow a potluck participant to respond with any value (either in plain text or in some formal language). Then, the manager could use a combination of information extraction or wrapper techniques (e.g., [52, 103]) and/or ontology matching algorithms (e.g., [48, 47]) to map the participant's response into the potluck's ontology. There are several interesting outcomes to this mapping. First, the response may directly map to one of the original potluck choices (e.g., "Cake" is an instance of `Dessert`). Second, the response may map to multiple choices in our ontology (e.g., "Jello salad" may be both an `Appetizer` and a `Dessert`). In this case, the manager might consider the response to be half of

an appetizer and a dessert, or postpone the decision to a later time and classify it as is most convenient.[1] Third, the response may not map to any given choice, but may still be a subclass of `Food` (e.g., a "Sorbet" is a `Palette Cleanser`); here the manager might accept the response but exclude it from the goal calculations. Fourth, the response may map to a known ontology element that is not `Food` (e.g., "A hat"). Finally, the response may not map to any known element. In these latter two cases, the manager may either reject the response or notify the originator.

- **Recommending interventions:** Reasoning can also assist the manager with directing the process towards outcomes consistent with the originator's goals. For instance, if the manager detects that a potluck process is becoming unbalanced, it could refuse to accept certain responses, request changes from some participants, or warn the originator that further action is needed. In this case reasoning is needed to deduce the likely outcome of a process from the current state, and the likely effects of possible interventions.

In this work we focus on using reasoning for recommending interventions, leaving the other two items for future work. Specifically, we provide two different approaches for modeling the originator's goals and when to intervene. In the logical model (Section 4.3), the originator specifies a set of *constraints* over the data set that should be satisfied by any process outcome, while in the decision-theoretic model (Section 4.4) the originator provides a function representing the *utility* of possible process outcomes. Below we consider each in turn, discuss possible variants, and present results for fundamental reasoning tasks that can determine how and when the manager should intervene.

4.3 Logical Model of SEPs

We now introduce our model of a logical semantic email process (L-SEP) and analyze important inference problems for this model.

---

[1] This a very simple form of semantic negotiation; more complex techniques could also be useful [170].

4.3.1 Definition of L-SEPs

A L-SEP is a 5-tuple $\Lambda(P, D, R, M, C_D)$ with parts as follows:

Participants P**:** the set of participants in the process. Note that P may include the originator.

Supporting data set D**:** the set of relations that holds all data related to the process. The initial contents of D are specified by the originator (usually to be a set of default values for the columns). With each relation in D we associate a schema that includes:

- a relation name and names, data types, and range constraints for the attributes. A special data type is emailAddress, whose values are elements of the set P. Attributes may have default values.

- possibly a distinguished from attribute, of type emailAddress, which means that rows in the relation whose from value is p can only result from messages from the participant p. The from attribute may be declared unique, in which case every participant can only affect a single row in the table.

Responses R**:** the set of possible responses to the originator's email. R is specified as follows:

- Attributes: the set of attributes in D that are affected by responses from participants. This set of attributes *cannot* include any from attributes.

- Insert or Update: a parameter specifying whether participants can only add tuples, only modify tuples, or both. Recall that if there is a from field then all changes from p pertain only to a particular set of tuples.

- Single or Many: a parameter specifying whether participants can send a single response or more than one. As we explain in the next section, some responses may be *rejected* by the system. By single, we mean one non-rejected message.

Messages M**:** the set of messages that the manager may use to direct the process, e.g., to remind the participants to respond or to *reject* a participant's response.

Constraints C_D: the set of constraints for every relation in D. We use the following definitions to specify the constraint language C:

Definition 4.3.1 (**variable**) A variable v is defined by a SQL query over D. v may be either an attribute variable (the value of a specific attribute in a row), or an aggregate variable. An aggregate variable may select a group of rows in an attribute A by specifying an equality/inequality predicate, and aggregate the corresponding values in an attribute B. We allow the aggregation functions COUNT, MIN, MAX, SUM, and AVERAGE. □

Definition 4.3.2 (**term**) A term may be

- a constant,
- a variable as defined above, or
- an expression combining two terms with any arithmetic operator □

Definition 4.3.3 (**atomic predicate**) An atomic predicate compares two terms, or a term with an enumerated set. We allow comparison predicates ($=, \neq, <, \leq$), LIKE, and $\in, \notin$. □

A set of constraints C_D then consists of atomic predicates combined in any manner with conjunction and disjunction.

Example: In our example, D contains one table named Potluck with two columns: email, a from attribute of type `emailAddress` and declared to be unique, and bringing, with the range constraint Potluck.bringing $\in$ {`not-coming, appetizer, entree, dessert, NULL`}. The set of possible responses R is { `not-coming, appetizer, entree, dessert` }. In addition, C_D contains a few constraint formulas similar to the abstract one below, specifying that the potluck should be balanced:

(SELECT COUNT(*) WHERE bringing = 'dessert') $\leq$
(SELECT COUNT(*) WHERE bringing = 'appetizer') + 2

Finally, the set of messages in our example includes (1) the initial message announcing the potluck and asking what each person is bringing, (2) messages informing each responder

whether their response was accepted or not, (3) a reminder to those who have not responded 2 days before the potluck, (4) regular messages to the originator reporting the status of the RSVPs, and (5) a message to the originator in the event that everyone has responded.

4.3.2 Inference for L-SEPs

Given the formal model for an L-SEP we can now pose a wide variety of inference problems whose results can serve to assist in the manager's operation. This section describes the first such inference problem, which has different variations.

The core problem we want to address is determining whether to *accept* a new response r, given the current state of D and the constraints C_D. The output of the inference problem is a condition that we will check on D and r to determine whether to accept r. The condition will decide whether to accept r by considering the impact of this acceptance on whether the L-SEP will terminate in an *acceptable* state, i.e., a state that satisfies C_D. In our discussion, we assume that r is a legal response, i.e., the values it inserts into D satisfy the range constraints on the columns of D; if not, the manager can respond with error messages until a legal response is received.

The space of possible inference problems is defined by several dimensions:

- **Necessity vs. possibility:** As in modal logics for reasoning about future states of a system [149, 57], one can either look for conditions that guarantee that *any* sequence of responses ends in a desired state (the $\Box$ operator), or that it is possible that *some* sequence ends in a desired state (the $\Diamond$ operator).

- **Assumptions about the participants:** In addition to assuming that all responses are legal, we can consider other assumptions, such as: (1) *all* the participants will respond to the message or (2) the participants are flexible, i.e., if asked to change their response, they will cooperate.

- **The type of output condition:** At one extreme, we may want a constraint C_r that the manager checks on D when a response r arrives, where C_r is specified in the same language used to specify C_D. At another extreme, we may produce an arbitrary

> procedure with inputs D and r that determines whether to accept r. We note that a constraint C_r will inevitably be weaker than an arbitrary algorithm, because it can only inspect the state of D in very particular ways. As intermediate points, we may consider constraints C_r in more expressive constraint languages. Note that in cases where we can successfully derive C_r, we can use database triggers to implement modifications to D or to indicate that r should be rejected.

As a very simple example, consider the case where we want *all* response sequences to end in an acceptable state, we make no assumptions on the participants except that we can elicit a legal response from them, and we are interested in deriving a constraint C_r that will be checked when a response arrives. If the initial state of D is an acceptable state, then simply setting C_r to be C_D provides a sufficient condition; i.e., we only let the data set D be in states that satisfy C_D. In the example of the balanced potluck, we would not accept a response with a dessert if that would lead to having 3 more desserts than entrees or appetizers. As another example, we would not accept a request for a giveaway process that caused the total number of tickets claimed to be more than the number that is available.

In many cases, such a conservative strategy will be overly restrictive. For example, we may want to continue accepting desserts so long as it is still *possible* to achieve a balanced potluck. Furthermore, this approach is usable only when the constraints are initially satisfied, even before any responses are received, and thus greatly limits the types of goals that can be expressed. This leads us to the following inference problem.

4.3.3 Ultimate Satisfiability

Our goal is to find necessary and sufficient conditions for accepting a response from a participant. To do that, we cut across the above dimensions as follows. Suppose we are given the data set D after 0 or more responses have been accepted, and a new response r. Note that D does not necessarily satisfy C_D, either before or after accepting r. The manager will accept r if it is *possible* that it will lead to a state satisfying C_D (i.e., considering the $\Diamond$ temporal operator). We do not require that the acceptance condition be expressed in our constraint language, but we are concerned about whether it can be efficiently verified on D

and r. We assume that D defines some constant number of attributes (e.g., `emailAddress`, `bringing`). Furthermore, we assume that participants can only update their (single) row, and only do so once.

Definition 4.3.4 (**ultimate satisfiability**) Given a data set D, a set of constraints C_D on D, and a response $r \in R$, we say that D is ultimately satisfiable w.r.t. r if there exists a sequence of responses from the participants, beginning with r, that will put D in a state that satisfies C_D. □

Unfortunately, ultimate satisfiability is difficult in general (proved by a reduction from 3-SAT):

Theorem 4.3.1 *Let Λ be an* L-SEP *with N participants and constraints C_D. If C_D may be any set of constraints permitted by the language C, then ultimate satisfiability is NP-complete in N.*

Note that this is a significant limitation, since for many SEPs it is natural to wish to scale to large numbers of participants (e.g., for large meetings or company-wide surveys). To address this problem, we begin with the following definition:

Definition 4.3.5 (**bounded constraints**) Given a data set D and a set of constraints C_D on D, we say that C_D is bounded iff one of the following holds:

- **Domain-bounded:** the predicates of C_D only refer to attributes whose domain size is at most some constant L.
- **Constant-bounded:** the predicates of C_D refer to at most K distinct constants, and the only aggregate used by C_D is COUNT. □

All of the examples in this paper may be described by constraints that satisfy the constant-bounded (COUNT-only) condition above, while the domain-bounded case may be useful for SEPs that require more complex interactions (e.g., where the AVERAGE number of guests must be less than J). Using this definition, we can show that ultimate satisfiability is much more tractable if the constraints are bounded:

Theorem 4.3.2 *Let Λ be an* L-SEP *with N participants and constraints C_D. If C_D is bounded, then determining ultimate satisfiability is polynomial time in N and $|C_D|$.*

As an example of applying this theorem to the balanced potluck, suppose a new dessert response arrives. At that point, the inference procedure needs to verify that, even if the dessert response is accepted, there are still enough people who have not yet answered such that the ultimate set of dishes could be balanced. To check this condition, the procedure must determine if there is any possible final state of the database, consistent with the responses received so far plus this new response, that satisfies the constraints. Naively considering every possibility is infeasible, since there are $O(4^N)$ possible final states for our example potluck with N participants. The key to checking this condition more efficiently is to express the database states in terms of variables representing *aggregates* on the number of participants with each response. For instance, one possible (unbalanced) final state of the potluck might be {12 `appetizers`, 23 `entrees`, 31 `desserts`, 4 `not-coming`}. Using these aggregates, only $O(N^4)$ states need to be considered, which is typically a much smaller number. The proof in Appendix C shows how such compact representations are always possible when the constraints are bounded.

4.4 Decision-theoretic Model of SEPs

The logical model of SEPs described above supports a number of useful inferences that have both theoretical and practical applications. This model, however, has a number of shortcomings. First, L-SEPs, like logical theories in general, make no distinctions among unsatisfied outcomes. In our example, there is no way for L-SEPs to strive for a "nearly-balanced" potluck, since all unbalanced potlucks are equivalently undesirable. Second, an L-SEP ignores the cost of the actions taken in pursuit of its goals. For instance, a potluck L-SEP will always reject a response that results in unsatisfiable constraints, even if rejecting that response (e.g., from an important official) may produce far worse effects than a slightly unbalanced potluck. Finally, L-SEPs make a very strong assumption that participants are always willing to change their responses if rejected. For instance, participants in a meeting

scheduling process may be somewhat accommodating, but may refuse to modify a rejected response due to other commitments.

To address these limitations, we offer a decision-theoretic approach. We describe the goal of a decision-theoretic SEP (D-SEP) by a utility function over the outcome of the process that takes into consideration the cost of all actions required to achieve that outcome. In addition, instead of *rejecting* responses, the decision-theoretic model sometimes *suggests* that participants modify their choices. For instance, the balanced potluck uses a utility function that measures the extent to which the final meal selection is balanced, minus the costs (social or otherwise) of asking some participants to switch their responses. Below we formalize this model and then examine the tractability of finding optimal policies for it.

4.4.1 Definition of D-SEPs

A decision-theoretic SEP is a 6-tuple, $\delta(P, S, V, A, U, T)$. Below we explain each component and contrast it to the corresponding component of the logical model.

- **Participants** P**:** the set of participants, of size N, as in the logical model.

- **States** S**:** the set of possible states of the system. A state s describes both the responses that have been received (just like the supporting data set D does in the logical model) as well as information about outgoing change requests that have been sent by the system.

- **Values** V**:** the set of possible values for participants to choose from (e.g., $V = \{appetizer, entree, dessert\}$). This set is equivalent to R, the set of possible responses in the logical model.

- **Actions** A**:** the set of actions available to the system after sending out the initial message. Actions we consider are $NoOp$ (do nothing until the next message arrives), SW_v (ask a participant to switch their response from v to something else), or $Halt$ (enter a terminal state, typically only permitted when a message has been received from every participant). Other variants of actions are also useful (e.g., ask a participant to switch from v to a particular value w); such additions do not fundamentally change the model or our complexity results. The set of actions corresponds roughly to the set

of messages M that may be sent in the logical model, though the logical model sends rejections instead of suggestions.

- **Utilities** $U(s,a)$**:** the utility from executing action a in state s. For the potluck example, $U(s,SW_v)$ is the (negative) utility from making a change suggestion, while $U(s,Halt)$ is the utility based on the final potluck balance. The utility function corresponds to the set of constraints C_D for the logical model.

- **Transitions** $T(s,a,s')$**:** the probability that the system will transition to state s' after performing action a in state s. However, rather than having to specify a probability for each transition, these are computed from a smaller set of building blocks. For instance, ρ_v is the probability that a participant will originally respond with the value v; ρ_{vw} is the probability that, when asked to switch from the choice v, a participant will change their response to w (ρ_{vv} is the probability that a participant refuses to switch). This component has no analogue in the logical model.

The execution of the process proceeds in discrete steps, where at each step the manager decides upon an action to take (possibly $NoOp$). The outcome of this action, however, is uncertain since the manager is never sure of how participants will respond. The transition function $T()$ models this uncertainty.

A *policy* π describes what action the manager will take in any state, while $\pi(s)$ denotes the action that the manager will take in a particular state s. An *optimal policy* $\pi^\star$ is a policy that maximizes the expected utility $U(\delta)$ of the process, where

$$U(\delta) = U(s_1,a_1) + U(s_2,a_2) + ... + U(s_j,a_j)$$

for the sequence of states and actions $(s_1,a_1),...,(s_j,Halt)$.

D-SEPs are a special case of *Markov Decision Processes* (MDPs), a well-studied formalism for situations where the outcome of performing an action is governed by a stochastic function and costs are associated with state transitions [150]. Consequently, we could find the optimal policy for a D-SEP by converting it to an MDP and using known MDP policy

solvers.[2] However, this would not exploit the special characteristics of D-SEPs that permit more efficient solutions, which we consider below.

4.4.2 Variations of D-SEPs

As with our logical model, the space of possible D-SEPs is defined by several dimensions:

- **Restrictions on making suggestions:** Most generally, the manager may be allowed to suggest changes to the participants at *any time*, and to do so repeatedly. To be more user-friendly, we may allow the manager to make suggestions anytime, but *only once* per participant. Alternatively, if users may be expected to make additional commitments soon after sending their response (e.g., purchasing ingredients for their selected dish), then we may require the manager to respond with any suggestion *immediately* after receiving a message, before any additional messages are processed.

- **Assumptions about the participants:** In addition to the assumed probabilities governing participant behavior, we may also wish to assume that all participants will eventually respond to each message they receive. Furthermore, we might assume that participants will respond *immediately* to any suggestions that they receive (particularly if the manager also responds immediately to their original message), or instead that they can respond to suggestions *anytime*.

- **The type of utility functions:** At one extreme, we might allow complex utility functions based upon the individual responses of the participants (e.g., "+97 if Jay is bringing dessert"). Often, however, such precision is unnecessary. For instance, all potluck outcomes with 8 desserts and 1 entree have the same low utility, regardless of who is bringing what dish.

Below we consider the impact of these variations on the complexity of computing the optimal policy.

---

[2] Specifically, D-SEPs are "Stochastic Shortest-Path" MDPs where the terminal state is reachable from every state, so an optimal policy is guaranteed to exist [18]. Incorporating additional features from *temporal MDPs* [26] would enable a richer model for D-SEPs (e.g., scheduling a meeting should be completed before the day of the meeting). However, existing solution techniques for TMDPs do not scale to the number of participants required for semantic email.

4.4.3 Computing the Optimal Policy

In this section we examine the time complexity of computing the optimal policy $\pi^\star$ for a D-SEP. We begin by considering a D-SEP with an arbitrary utility function and then examine how restrictions to the utility function and the permitted quantity and timing of suggestions make computing $\pi^\star$ more tractable. In all cases we assume that the participants will eventually respond to each message and suggestion that they receive. (We can relax this assumption by representing in the model the probability that a participant will *not* respond to a message.) The following theorem is proved by reduction from QBF (quantified boolean formula) and the EXPTIME-hard game G_4 [168, 112]:

Theorem 4.4.1 *Let δ be a* D-SEP *with N participants where the utility $U(s, a)$ is any deterministic function over the state s and the current action a. If the manager can send only a bounded number of suggestions to each participant, then determining $\pi^\star$ is PSPACE-hard in N. If the manager can send an unlimited number of suggestions, then this problem is EXPTIME-hard in N. The corresponding problems of determining if the expected utility of $\pi^\star$ for δ exceeds some constant θ are PSPACE-complete and EXPTIME-complete, respectively.*

Thus, for the case of arbitrary utility functions determining $\pi^\star$ for a D-SEP is impractical for large values of N. Note that conversion to an MDP offers little help, since the MDP would require a number of states exponential in N. As with L-SEPs, this represents a significant problem, since we would like SEPs to scale to many participants. Below, we begin to make the calculation of $\pi^\star$ more tractable by restricting the type of utility function:

Definition 4.4.1 (**K-Partitionable**) The utility function $U(s, a)$ of a D-SEP is *K-partitionable* if it can be expressed solely in terms of the variables $a, C_1, ..., C_K$ where a is the current action chosen by the manager and each C_i is the number of participants who have responded with value V_i in state s. □

Intuitively, a utility function is K-partitionable if what matters is the number of participants that belong to each of a fixed number of K groups, rather than the specific participants in each of these groups. For instance, the utility function of our example potluck

is *4-partitionable*, because all that matters for evaluating current and future utilities is the current number of participants that have responded `Appetizer`, `Entree`, `Dessert`, and `Not-Coming`. In this case a simple utility function might be:

$$\begin{aligned} U(s, Halt) &= -\alpha(|C_A - C_E|^2 + |C_A - C_D|^2 + |C_E - C_D|^2) \\ U(s, SW_v) &= -1 \end{aligned}$$

where α is a scaling constant and C_A, C_E, and C_D are the numbers of appetizers, entrees, and desserts, respectively. Note that the maximum utility here is zero.

A K-partitionable utility function is analogous to the COUNT-only constraint language of Theorem 4.3.2. As with Theorem 4.3.2, we could allow more complex utility functions (e.g., with variables representing the MAX, SUM, etc. of the underlying responses); with suitable restrictions, such functions yield polynomial time results similar to those described below. Note, however, that the simpler K-partitionable definition is still flexible enough to support all of the SEPs discussed in this paper. In particular, a K-partitionable utility function may still distinguish among different types of people by counting responses differently based on some division of the participants. This technique increases the effective value of K, but only by a constant factor. For instance, the utility function for a meeting scheduling process that desires to have the number of *faculty* members attending ($C_{yes,F}$) be at least three and the number of *students* attending ($C_{yes,S}$) be as close as possible to five, while strongly avoiding asking faculty members to switch, might be:

$$\begin{aligned} U(s, Halt) &= -\alpha[max(3 - C_{yes,F}, 0)]^2 - \beta|C_{yes,S} - 5|^2 \\ U(s, SW_{no,F}) &= -10 \\ U(s, SW_{no,S}) &= -1 \end{aligned}$$

A D-SEP that may make an unlimited number of suggestions but that has a *K-partitionable* utility function can be represented as an "infinite-horizon" MDP with just $O(N^{2K})$ reachable states. Consequently, the D-SEP may be solved in time polynomial

Table 4.1: Summary of theoretical results for D-SEPs. The last two columns show the time complexity of finding the optimal policy for a D-SEP with N participants. In general, this problem is EXPTIME-hard but if the utility function is *K-partitionable* then the problem is polynomial time in N. (An MDP can be solved in time guaranteed to be polynomial in the number of states, though the polynomial has high degree.) Adding restrictions on how often the manager may send suggestions makes the problem even more tractable. Note that the size of the optimal policy is finite and must be computed only once, even though the execution of a SEP may be infinite (e.g., with "AnyUnlimited").

Restrictions	**Description of Restrictions**	**Complexity with arbitrary utility function**	**Complexity when *K-partitionable***
AnyUnlimited	Manager may suggest changes at any time, and may send an unlimited number of suggestions to any participant.	EXPTIME-hard	MDP with $O(N^{2K})$ *states*
AnyOnce	Manager may suggest changes at any time, but only once per participant.	PSPACE-hard	$O(N^{3K})$ time
Immediate	Manager may suggest changes only immediately after receiving a response, once per participant.	PSPACE-hard	$O(N^{2K})$ time
Synchronous	Same as "Immediate", but each participant is assumed to respond to any suggestion before the manager receives any other message.	PSPACE-hard	$O(N^{K})$ time

in N with the use of linear programming (LP), though alternative methods (e.g., policy iteration, simplex-based LP solvers) that do not guarantee polynomial time may actually be faster in practice due to the large polynomial degree of the former approach [113].

Furthermore, if we also restrict the system to send only one suggestion to any participant (likely a desirable property in any case), then computing the optimal policy becomes even more tractable:

Theorem 4.4.2 *Let δ be a* D-SEP *with N participants where $U(s,a)$ is K-partitionable for some constant K and where the system is permitted to send at most one suggestion to any participant. Then $\pi^\star$ for δ can be determined in $O(N^{3K})$ time. (If the system can send at most L suggestions to any participant, then the total time needed is $O(N^{(2L+1)K})$.)*

Table 4.1 summarizes the results presented above as well as a few other interesting cases ("Immediate" and "Synchronous"). These results rely on two key optimizations. First, we can dramatically reduce the number of distinct states via K-partitioning; this permits $\pi^\star$ to be found in polynomial time. Second, we can ensure that the state transition graph is acyclic (a useful property for MDPs also noted in other contexts [21]) by bounding the number of suggestions sent to each participant; this enables us to find $\pi^\star$ with simple graph search algorithms instead of with policy iteration or linear programming. Furthermore, this

approach enables the use of existing heuristic search algorithms where an exact computation remains infeasible. Consequently, with appropriate restrictions many useful D-SEPs can be efficiently solved in polynomial time.

4.4.4 Discussion

Compared to L-SEPs, the primary advantages of D-SEPs are their ability to balance the utility of the process's goals vs. the cost of additional communication with the participants, and their graceful degradation when goals cannot be completely satisfied. On the other hand, the need to determine suitable utilities and probabilities is an inherent drawback of any decision-theoretic framework. Below we consider techniques to approximate these parameters.

First, the $\pi^\star$ for a D-SEP depends upon the relative value of positive utilities (e.g., having a well-balanced potluck) vs. negative utilities (e.g., the cost of making a suggestion). Our discussion above exhibited a number of simple but reasonable utility functions. In practice, we expect that D-SEPs will provide default utility functions based on their functionality, but would allow users to modify these functions by adjusting parameters or by answering a series of utility elicitation questions [23].

Second, D-SEPs also require probabilistic information about how participants are likely to respond to original requests and suggestions. This information can be determined in a number of ways:

- **User-provided:** The process originator may be able to provide reliable estimates of what responses are likely, based on some outside information or past experience.

- **History-based:** Alternatively, the system itself can estimate probabilities by examining the history of past processes.

- **Dynamically-adjusted:** Instead of or in addition to the above methods, the system could dynamically adjust its probability estimates based on the actual responses received. If the number of participants is large relative to the number of choices, then the

system should be able to stabilize its probability estimates well before the majority of responses are received.

Finally, some versions of D-SEPs require calculating the probability of the next message received being an original message ($\rho_o(S)$) or a response to a suggestion ($\rho_s(S)$). One reasonable approximation is as follows:

$$\begin{aligned} \rho_o(S) &= \frac{\beta o}{\beta o + (1-\beta)s} \\ \rho_{s,v}(S) &= \frac{(1-\beta) \cdot s_v}{\beta o + (1-\beta)s} \end{aligned}$$

where o is the number of participants who have yet to make an original response, $s = s_1 + ... + s_k$ (s_v is the number of participants that have yet to respond to a suggestion SW_v), and β is a parameter in the range (0,1). As β approaches 0, responses to suggestions become more and more likely to arrive before any additional original responses (as in the "Synchronous" case), while setting β to 0.5 assumes that the relative likelihood of original responses vs. suggestion responses depends only on the number of pending messages of each type. The most appropriate choice of β can be determined by any of the probability estimation techniques discussed above. Of course more sophisticated models based on the specific response that a participant was asked to change or the different types of participants (e.g., student, faculty, etc.) are also possible.

Thus, although the need to provide utility and probability estimates is a drawback of D-SEPs compared to L-SEPs, simple techniques can produce reasonable approximations for both. In practice, the choice of whether to use a D-SEP or L-SEP will depend on the target usage and the feasibility of parameter estimation. In our implementation, we allow the originator to make this choice. For D-SEPs, we currently elicit some very basic utility information from the originator (e.g., see Figure 4.2), and use some probabilities provided by the SEP author for expected participant behavior. Extending our implementation to support history-based and dynamically-adjusted probabilities is future work.

4.5 *Implementation and Usability*

We have implemented a complete semantic email system and deployed it in several applications. In doing so, we faced several challenges. Our design principles from Chapter 1, particularly those of gradual adoption and ease of use, provide a framework for tackling these challenges. Below we elaborate on these challenges and describe how we have addressed them in our system.

4.5.1 Process Creation and Execution

Translating SEP theory to real problems: Applying our SEP theory to real problems requires enabling an originator to easily create an L-SEP or D-SEP model that corresponds to his goals. One option is to build a GUI tool that guides the originator through constructing the appropriate choices, messages, and constraints or utilities for the process. Practically, however, a tool that is general enough to build an arbitrary process is likely to be too complex for untrained users.

Instead, our system is based on the construction of reusable *templates* for specific classes of SEPs. Facilitating the authoring of general, widely-applicable templates that can be safely instantiated even by naive originators is an important challenge that is the focus of the next chapter. For our current discussion, we briefly describe the use of templates from the point of view of the originator. An untrained originator finds a SEP from a public library of SEP templates and instantiates the template by filling out a corresponding web form, yielding a SEP *declaration*. For instance, Figure 4.2 shows such a form for the balanced potluck. Note that the bottom of this form allows users to choose between executing an L-SEP (the "strictly" and "flexibly" options) or a D-SEP (the "tradeoff-based" option). In addition, originators may specify either individuals or mailing lists as participants; for the latter case, the form also asks the originator for an estimate of the total number of people that will respond (not shown in Figure 4.2).

The originator then invokes the process by forwarding the declaration to the manager. Given the formal declaration, the manager then executes the process, using appropriate

Semantic Email -- Balanced Collection Process - Microsoft Internet Explorer

File Edit View Favorites Tools Help

- **Who should receive email about this process?**
  uw-db@cs.washington.edu, faculty@cs.washington.edu
- **The following greeting will be sent on your behalf.** You can edit it you wish.
  You have been invited to the potluck described below. Please use the form below to indicate what you are bringing. To ensure that our meal selection is balanced, you may be asked to change your selection.
- **Enter details about the event:**
  Description: Database Annual Potluck *(Optional)*
  Location: Dan Suciu's house, 5320 Northwest Way *(Optional)*
  Start time: Oct 5, 2003 **at** 5 : 00 p.m. PDT
- **Enter the choices for recipients to choose from,** separated by commas.
  Not Coming, Appetizer, Entree, Dessert
- This process will count the number of responses for each choice given above. To automatically "balance" the results, specify any desired restrictions to enforce on these responses: (Explain this)
  Minimum count of each choice: (enter a positive integer or leave blank)
  Maximum count of each choice: (enter a positive integer or leave blank)
  Maximum discrepancy in counts: 2 (enter a positive integer or leave blank)
  Choices to exclude from these restrictions: Not Coming (comma-separated list)
- **How should these restrictions be enforced?** (Explain this)
  - **Enforce strictly**, so that the collection result is guaranteed to be balanced.
  - **Enforce flexibly**, to give responders maximum choice so long as balance is still possible.
  - **Tradeoff-based**: accept all responses, but request a change from someone if the imbalance in any restriction is projected to be more than 1

Submit

Figure 4.2: A web form used to initiate a "balanced collection" process, such as our balanced potluck example. For convenience, clicking submit converts the form to text and sends the result to the server and a copy to the originator. The originator may later initiate a similar process by editing this copy and mailing it directly to the server.

L-SEP and D-SEP algorithms to decide how to direct the process via appropriate message rejections and suggestions.

Facilitating responses: Another key challenge is enabling participants to respond to messages in a way that is convenient but that can be automatically interpreted by the manager. A number of different solutions are possible:

- **Client software:** We could provide a custom email client that would present the participant with an interface for constructing legal responses or automatically respond to messages it knows how to handle (e.g., "Decline all invitations for Friday evenings"). This client-based approach, however, requires all participants in a process to install

additional software (conflicting with our gradual adoption goal) and is complicated by the variety of mail clients currently in use.

- **Information extraction:** We could allow participants to respond in a natural language (e.g., "I'll bring a dessert"). We could then use wrappers or information extraction techniques to attempt to convert this response to one of the offered choices. This approach is promising but risks having the wrapper fail to extract the correct information.

- **Email or web forms:** We could provide participants with a text-encoded form to fill out, or we could send them a link to a suitable web-based form to use for their response. Embedded HTML forms are also attractive, but unfortunately are not handled uniformly by existing email clients.

While web forms have some advantages, we chose to use email text forms instead because they fit more naturally with how people typically handle incoming messages. In addition, text forms offer a simple solution that works for any participant. Participants respond by replying to the process message and editing the original form.

Our earlier discussion generally assumed that participants would send a single acceptable response. However, our implementation does permit participants to "change their mind" by sending additional responses. For the logical model, this response is accepted if changing the participant's original response to the new value still permits the constraints to be satisfied (or if the response must always be accepted, e.g., for `Not-Coming`). For the decision-theoretic model, the new response is always accepted but may lead to a change suggestion based on the modified state of the process.

Manager deployment: Potentially, the manager could be a program run on the originator's personal computer, perhaps as part of his mail client. This permits an easy transition between authoring traditional mails and invoking SEPs, and can also benefit from direct access to the originator's personal information (e.g., calendar, contacts). However, as with providing client software for participants, this approach requires software installation and must deal with the wide variety of existing mail clients.

Our implementation instead deploys the manager as a shared server. The server receives

invocations from the originator and sends out an initial message to the participants. Participants reply via mail directly to the server, rather than to the originator, and the originator receives status and summary messages from the server when appropriate. The originator can query or alter the process via additional messages or a web interface.

Discussion: Our server-based approach is easy to implement and satisfies our gradual adoption and ease of use principles, since it requires no software installation, works for all email clients, and does not require users (as originators) to read or write RDF. In addition, we believe that divorcing the processing of semantic email (in the server) from the standard email flow (in the client) will facilitate gradual adoption by ameliorating user concerns about privacy[3] and about placing potentially buggy code in their email client. Furthermore, this approach supports our instant gratification principle by providing untrained users with existing, useful SEPs that can be immediately invoked and yield a tangible output (in the form of messages sent and processed on the users' behalf).

Note also that this approach effectively integrates content creation into the act of invoking a SEP. In particular, simply by filling out a web form with the participants, choices, and goals for a SEP, the originator creates semantic content. Likewise, participants respond to requests in a way that is easy to use but that makes it easy to interpret their responses declaratively. This declarative data enables a range of useful reasoning that may benefit both the current SEP and future interactions, as described in Section 4.2. Thus, any person involved in the execution of a SEP automatically contributes declarative content to some extent, accomplishing a significant part of our overall goal.

4.5.2 Human/Machine Interoperability

The previous section highlighted how semantic email messages can be handled by either a human or by a program operating on their behalf. Thus, an important requirement is that every message must contain both a human-understandable portion (e.g., "You're invited to the potluck on Oct 5...") and a corresponding machine-understandable portion.

---

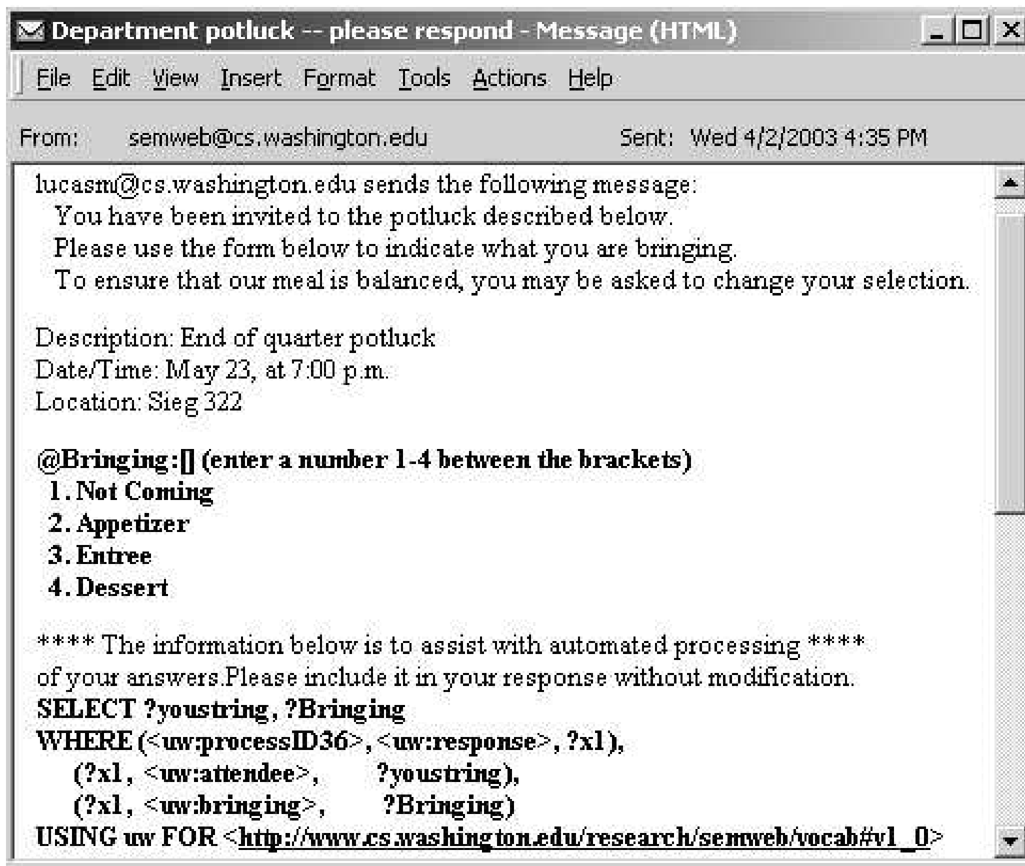
[3] Only semantic email goes through the server, personal email is untouched. Of course, when the semantic email also contains sensitive information, the security of the server becomes significant.

For messages sent to a participant, this approach supports gradual adoption by permitting the originator to send the same message to all participants without any knowledge of their capabilities. For responses, a machine-understandable portion enables the manager to evaluate the message against the process constraints/utilities and take further action. The human-readable component provides a simple record of the response if needed for later review.

In our implementation, we meet this interoperability requirement with a combination of techniques. For responses, a human can fill out the included text form (see Figure 4.3), which is then converted into RDF at the server with a simple mapping from each field to an unbound variable in a RDQL query associated with the message. Alternatively, a machine can respond to the message simply by answering the query in RDF, then applying the inverse mapping in order to correctly fill out the human-readable text form.

For messages to the participants, the challenge is to enable the manager to construct these textual and RDF/RDQL portions directly from the SEP declaration. Here there is a tension between the amount of RDF content that must be provided by the SEP author (in the template) vs. that provided by the SEP originator (when instantiating the template). Very specific SEP templates (e.g., to balance N people among appetizer, entree, and dessert choices) are the easiest to instantiate, because the author can specify the RDF terms needed in advance. General SEP templates (e.g., to balance N people among K arbitrary choices) are much more reusable, but require substantially more work to instantiate (and may require understanding RDF). Alternatively, authors may provide very general templates but make the specification of RDF terms for the choices optional; this enables easy template reuse but fails to provide semantic content for automated processing by the participants.

In our current system, we offer both highly specialized SEPs (e.g., for meeting scheduling) and more general SEPs (e.g., to give away some type of item). Enabling originators to easily customize general SEPs with semantic terms, perhaps from a set of offered ontologies, is an important area of future work.


Department potluck -- please respond - Message (HTML)

File Edit View Insert Format Tools Actions Help

From: semweb@cs.washington.edu Sent: Wed 4/2/2003 4:35 PM

lucasm@cs.washington.edu sends the following message:
You have been invited to the potluck described below.
Please use the form below to indicate what you are bringing.
To ensure that our meal is balanced, you may be asked to change your selection.

Description: End of quarter potluck
Date/Time: May 23, at 7:00 p.m.
Location: Sieg 322

@Bringing:[] (enter a number 1-4 between the brackets)
1. Not Coming
2. Appetizer
3. Entree
4. Dessert

**** The information below is to assist with automated processing ****
of your answers.Please include it in your response without modification.
SELECT ?youstring, ?Bringing
WHERE (<uw:processID36>, <uw:response>, ?x1),
(?x1, <uw:attendee>, ?youstring),
(?x1, <uw:bringing>, ?Bringing)
USING uw FOR <http://www.cs.washington.edu/research/semweb/vocab#v1_0>

Figure 4.3: A message sent to participants in a "balanced potluck" process. The bold text in the middle is a form used for human recipients to respond, while the bold text at the bottom is a RDQL query that maps their textual response to RDF.

4.5.3 Integrating with Non-Semantic Messages

Despite the advantages of semantic email, we do not want to create a strict dichotomy in our email habitat. In our potluck example, suppose that one of the participants wants to know whether there is organized transportation to the potluck (and this information affects his decision on what to bring). What should he do? Compose a separate non-semantic email to the originator and respond to the semantic one only later? A better (and easier to use) solution would be to treat both kinds of emails uniformly, and enable the participant to ask the question in replying to the semantic email, ultimately providing the *semantic* response later on in the thread.

Our implementation supports this behavior by supplying an additional *Remarks* field in each response form, where a participant may include a question or comment to be forwarded to the originator. For a question, the originator can reply, enabling the participant to respond to the original semantic question with the included form or pose another question.

4.6 Experience

Our semantic email system is deployed and may be freely used by anyone without any software installation; the source code for deploying other instances of the server is also available. So far we have developed simple processes for functions like collecting RSVPs, giving tickets away, scheduling meetings, and balancing a potluck. Our system uses standard ontologies where possible (e.g., RDF Calendar [176]), augmented as needed with a local semantic email schema.

Our semantic email server has seen growing interest over the fifteen months that it has been available. For instance, a DARPA working group has adopted semantic email for all of its meeting scheduling and RSVP needs, students have used semantic email to schedule seminars and Ph. D. exams, and semantic email has been used to organize our annual database group and departmental-wide potlucks. Furthermore, a number of other institutions have expressed interest in deploying copies of semantic email locally at their sites. These are merely anecdotes but lend credence to our claim that semantic email is both useful and practical.

Despite the usage we have seen, however, the group of people instantiating new SEPs as originators seems to be much smaller than the group of people who have learned about and shown enthusiasm for the system. While some of this effect is to be expected, we also believe that a significant reason is the amount of *initial* work required to instantiate a SEP. In spite of SEPs' advantages, when faced with a particular data-collection task it is easier in the short-term to just send a non-semantic email message and deal with the consequences later. In short, the instant gratification from using SEPs is not instant enough. Chapter 6 proposes some future work to make the invocation a SEP even easier.

Our experience with untrained participants has been mixed. On one hand, partici-

pants have generally demonstrated an ability and willingness to respond to semantic email messages and requests. However, our use of plain text messages, while allowing anyone to participate, has sometimes been problematic. For instance, some fraction of users inevitably returns forms that are ambiguous or invalid in some way. These problems have highlighted the importance of both careful instructions and simple, understandable error messages. In addition, while our use of an RDQL query embedded in the participant's response has some intuitive appeal (the message is entirely understandable on its own, less state in the server, etc.), in practice this method sometimes fails because the participant neglects to return the entire message or because a mail client performs some unusual reformatting. Consequently, we modified our server to use a cached version of the RDQL query in cases where the query is lost or distorted.

4.7 Related Work

Some hardcoded email processes, such as the meeting request feature in Outlook, invitation management via *Evite*, and contact management via *GoodContacts*, have made it into popular use. Each of these commercial applications is limited in its scope, but validates our claim about user pain. Our goal in this work is to sketch a *general* infrastructure for semantic email processes, and to analyze the inference problems it needs to solve to manage processes effectively and guarantee their outcome.

Collaboration systems such as Lotus Notes/Domino and Zaplets offer scripting capabilities and some graphical tools that could be used to implement sophisticated email processes. However, these systems (as with the workflow systems discussed later) lack support for reasoning about data collected from a number of participants (e.g., as required to balance a potluck or ensure that a collected budget satisfies aggregate constraints). In addition, such processes are constructed from arbitrary pieces of code, and thus lack the formal properties that our declarative model provides. Finally, messages in such systems lack the RDF content of semantic email, precluding automated processing by the recipient (e.g., to decline invitations for unavailable times).

Information Lens [117] used forms to enable a user to generate a single email mes-

sage with semi-structured content that might assist recipients with filtering and prioritizing that message. Our SEPs generalize this earlier work by enabling users to create an email *process* consisting of a set of interrelated messages, and by extending Information Lens's rule-based message processing to support more complex constraint and utility reasoning based on information from the entire set of messages. Consequently, SEPs support a much broader range of possible applications. More recently, Kalyanpur et al. [98] proposed having users semantically annotate messages to improve mail search, sorting, and filtering. This approach can potentially result in rich semantic content, but requires users to invest significant annotation effort for some *potential* future benefit (e.g., in improved searching for an old email) or primarily for the benefit of the recipient. SEPs instead generate both the semantic content and the text of the email message directly from simple forms, and provide instant gratification by immediately utilizing this content for simple but time-saving email processes.

Possible uses of semantic email are similar to those of some existing Semantic Web systems (e.g., [146, 102, 133], cf., RDF Calendar group discussions [176]). The key differentiating aspects of our work are its generality to many different tasks, its ability to interoperate freely with naive participants, and its polynomial time reasoning for recommending interventions. For instance, RCal [146] uses messages between participants to agree upon meeting times and McIlraith et al. [133] describe an agent that makes travel arrangements by invoking various web services (which could be modeled as participants in a SEP). These systems, however, enable full interaction only between two parties that are both executing domain-specific software. For instance, though RCal provides a web interface to let anyone schedule an appointment with an installed RCal user, an RCal user cannot use the system to request an appointment with a non-"RCal-enabled" person. Likewise, McIlraith et al.'s agent is designed only to communicate with specific web services, not with humans (such as human travel agents) that could offer the same functionality. Our system instead permits processes to include any user, regardless of their capabilities. An additional, though less critical, distinction is our use of email instead of HTTP or a custom protocol (cf., Everyware [63]). Email provides a convenient transport mechanism because the vast majority of users already have well-known addresses (no additional directories are

needed), messages can be sent regardless of whether the recipient has performed any configuration, and existing email clients provide a useful record of messages exchanged. Finally, our framework enables the automated pursuit of a wide variety of goals through reasoning in guaranteed polynomial time, a result not provided by the other systems discussed above. The combination of these factors makes semantic email a lightweight, general approach for automating many tasks that would be impractical with other systems.

4.7.1 Efficient Reasoning With Aggregation

A significant challenge for the SEP theory that we describe is reasoning about the possible relationships between *aggregate* values (current and future), given a particular state of the SEP database (i.e., the messages received so far). Reasoning about aggregation has received significant attention in the query optimization literature [155, 108, 37, 71] and some in the description logic literature (e.g., [8]). This body of work considered the problem of optimizing queries with aggregation by moving predicates across query blocks, and reasoning about query containment and satisfiability for queries involving grouping and aggregation. In contrast, our L-SEP results involve considering the current state of the database to determine whether it can be brought into a state that satisfies a set of constraints. Furthermore, since C_D may involve several grouping columns and aggregations, they cannot be translated into single-block SQL queries, and hence the containment algorithms will not carry over to our context.

Workflow systems [137, 135, 173, 105] could also be used to represent some SEPs, and many such systems have a solid formal foundation based on Petri Nets [92] or the Pi calculus [134]. In addition, languages for workflow typically permit much more complex control flow than allowed by our current SEP framework (which involves asking a single set of questions from a single set of participants). Workflow systems, however, typically have very weak support for reasoning about values and aggregations of data, instead restricting their attention to reasoning about temporal and causality constraints. Such formalisms could potentially convert aggregation constraints to temporal constraints by enumerating all possible data combinations, but this may result in an exponential number of states. One

exception is the recent work of Senkul et al. [160], who extend workflows to include *resource constraints* based on aggregation. Each such constraint, however, is restricted to performing a single aggregation with no grouping (and thus could not express the potluck constraint given in the earlier example). In addition, their solution is based on general constraint solving and thus will take exponential time in the worst case. We have shown, however, that in our domain L-SEPs can easily express more complex aggregation constraints while maintaining polynomial-time inference complexity for bounded constraints.

Other more database-focused work (e.g., Abiteboul et al. [3], Bonner [22], Deutsch et al. [46]) has defined formalisms that could potentially be applied to representing SEPs, at least at a high level. For instance, Deutsch et al. [46] define a language for specifying and verifying data-driven web services, while Abiteboul et al. [3] investigate specifying and verifying *relational transducers* for business processes. These formalisms offer more support for reasoning about data than directly possible with workflow systems, but still lack support for reasoning about aggregation. For instance, the notion of goal reachability for relational transducers [3] is similar to our definition of ultimate satisfiability. Various restrictions on the model allow decidability of goal reachability in P, NP, or NEXPTIME, but none of these restrictions (nor the extensions described by Hull [89]) permit goals involving aggregation.

Essentially, efficient reasoning about aggregation requires the ability to abstract away from the details of the data to concentrate only on important summary information (e.g., the number of `desserts` so far), the details of which depend upon the structure of the goal. To some extent, this same idea of exploiting structural information has been studied in the field of Markov Decision Processes (e.g., [24, 25]). For instance, Boutilier et al. [25] describe an improved method of policy iteration for solving MDPs that represents the optimal policy as a structured decision tree, rather than explicitly representing the optimal action for each possible state. This approach offers a helpful (though not guaranteed) technique for reducing the effective state space that must be considered. This work is complementary to our analysis of D-SEPs — we apply *K-partitioning* to drastically reduce the number of states for a D-SEP, which may then be used as the input to an improved policy solver that efficiently represents the optimal policy over these states.

4.8 Summary

This chapter generalizes the original vision of the Semantic Web to also encompass email. We have introduced a paradigm for semantic email and described a broad class of semantic email processes. These automated processes offer tangible productivity gains on email-mediated tasks that are currently performed manually in a tedious, time-consuming, and error-prone manner. Moreover, semantic email opens the way to scaling similar tasks to large numbers of people in a manner that is infeasible today. For example, large organizations could carry out surveys, auctions, and complex meeting coordination via semantic email with guarantees on the behavior of these processes.

Semantic Email is a second example of how a successful Semantic Web system should exhibit our three proposed design principles. First, SEPs offers instant gratification to originators in the form of messages sent and processed on their behalf. Our declarative data representation enables a range of helpful reasoning to support this processing. In particular, we define two formal models for specifying the desired behavior of a SEP and identify key restrictions that enable tractable reasoning over these models. Second, our system provides gradual adoption by enabling anyone to launch and participate in a SEP without needing to understand RDF or install any software. Finally, our system supports ease of use with its template-based SEP instantiations and its use of simple text forms for responses. The next chapter examines ways to make Semantic Email even more useful and practical by simplifying the process of authoring a SEP.

Chapter 5

SPECIFYING SEMANTIC EMAIL PROCESSES

The previous chapter described the theory of SEPs and how they can automatically pursue goals on behalf of an originator, but how can a non-technical originator tell a SEP what to do? One approach to this problem is to use templates that are authored once but then instantiated many times by ordinary users. This approach, however, raises a number of challenges. For instance, how can templates concisely represent a broad range of potential uses, yet ensure that each possible instantiation will function properly? And how does the SEP explain its actions to the humans involved? This chapter describes the three challenges of generality, safety, and understandability that arise in this context. We then describe how we address each challenge via a declarative, high-level template language, instantiation safety testing, and automatic explanation generation, and relate these solutions to our three design principles.

5.1 Introduction

Chapter 4 demonstrated that SEPs can be used for a wide range of useful interactions and that important reasoning problems for SEPs are computationally tractable in many common cases. Applying this theory to real problems, however, requires the ability to create a SEP specification that corresponds to an originator's goals.

Our approach to this problem is to encapsulate classes of common behaviors into reusable *templates* (cf., program schemas [45, 65] and generic procedures [132]). Templates address the specification problem by allowing a domain-specific template to be *authored* once but then *instantiated* many times by untrained users. In addition, specifying such templates declaratively opens the door to automated reasoning to verify important properties and to compose templates for more complex interactions.

However, specifying SEP behavior via templates presents a number of challenges:

- **Generality:** How can a template concisely represent a broad range of potential uses?

- **Safety:** Templates are written with a certain set of assumptions — how can we ensure that any (perhaps unexpected) instantiation of that template by a naive originator will function properly (e.g., do no harm [180], generate no errors)?

- **Understandability:** When executing a template, how can a SEP explain its actions to the humans (or other agents) that are involved?

This chapter addresses each of these challenges. For *generality*, we describe the essential features of our template language that enable authors to easily express complex goals without compromising the tractability of SEP reasoning. The sufficiency of these features is demonstrated by our implementation of a small but diverse set of SEPs. For *safety*, we show how to verify, in polynomial time, that a given template will always produce a valid instantiation. Finally, for *understandability*, we examine how to automatically generate explanations of *why* a particular response could not be accepted and *what* responses would be more acceptable. We also identify suitable restrictions where such explanations can be generated in polynomial time. Collectively, these results greatly increase the usefulness of semantic email. In addition, they highlight important issues that may be relevant to other Semantic Web agents, because many such agents face the same general challenges of generality, safety, and understandability in interacting with non-technical people.

In this chapter, our discussion is motivated primarily by our design principles of gradual adoption and ease of use. For instance, originators should be able to easily and safely specify a new SEP without needing any specialized training or software, and participants should receive understandable requests from the SEP manager. In addition, the template language that we discuss also helps to support instant gratification, by facilitating the development of a number of pre-existing SEPs with a wide range of functionality.

The next section gives a brief overview of SEP creation, while Section 5.3 describes our template language and a complete example. Sections 5.4 and 5.5 examine the problems of instantiation safety and explanation generation that were discussed above. Finally, Sec-

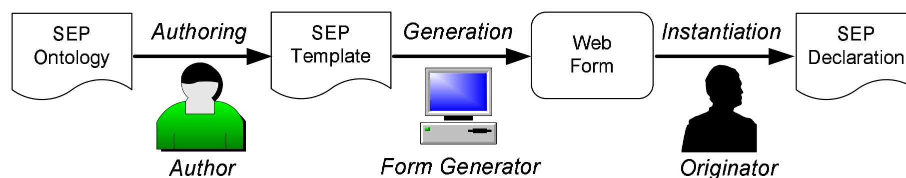


Figure 5.1: The creation of a Semantic Email Process (SEP). Initially, an "Author" *authors* a SEP template and this template is used to *generate* an associated web form. Later, this web form is used by the "Originator" to *instantiate* the template. Typically, a template is authored once and then instantiated many times.

tion 5.6 considers related work and Section 5.7 concludes with implications of our results for both SEPs and other types of agents.

5.2 Overview of SEP Creation

Figure 5.1 demonstrates how a template is used to create a new SEP. Initially, someone who is assumed to have some knowledge of RDF/OWL and semantic email authors a new template using an editor (most likely by modifying an existing template). We call this person the SEP *author*. The template is written in OWL based on an ontology that describes the possible questions, goals, and notifications for a SEP; Section 5.3 describes this in more detail. For instance, a balanced potluck template defines some general balance constraints, but has placeholders for *parameters* such as the participants' addresses, the specific choices to offer, and how much imbalance to permit. Associated with each template is a simple *web form* that describes each needed parameter; Section 5.4 describes a tool to automatically generate such forms. An untrained originator finds an appropriate web form from a public library and fills it out with values for each parameter, causing the corresponding template to be instantiated into a SEP *declaration*. The semantic email server executes the declaration directly, using appropriate algorithms to direct the SEP outcome via message rejections and suggestions, as explained in Chapter 4.

5.3 Concise and Tractable Representation of Templates

Our first challenge is to ensure that a template can concisely represent a broad range of possible uses while still ensuring the tractability of SEP reasoning (e.g., for checking the

acceptability of a participant's response). This section describes our language for specifying templates, presents a complete example for the balanced potluck, and discusses how the language meets this challenge.

5.3.1 Components of a SEP *template*

A SEP *template* is a (parameterized) OWL document that includes:

- a **preamble** that identifies the participants,
- **questions** to ask the participants,
- **goals** to pursue over the participants' responses, and
- **notifications** to send to the originator and/or participants at appropriate times.

Below we describe each of these components in more detail, and relate them to the logical and decision-theoretic models of Chapter 4.

Preamble: the set of participants and a prompt to be sent with the initial request (e.g., "You have been invited to the following potluck..."). This corresponds to the participant set P for the L-SEP and D-SEP models.

Questions: the set of questions to ask each participant. For instance, a potluck SEP might ask each participant for the food item and the number of guests that they are bringing. Each question defines a variable name for later use and the type of valid responses to that question (e.g., integer, boolean, etc.). Questions may also specify further restrictions on what responses are considered valid (e.g., `NumGuests` must be non-negative). Finally, each question item provides an RDQL query that specifies the semantic meaning of the requested information and is used to map the participant's textual response to RDF (see Section 4.5.2).

The questions effectively define the possible responses of the participants, and thus correspond to the responses R for L-SEPs and the values V for D-SEPs. In addition, the RDQL query defines a mapping of responses to RDF that corresponds to the mapping of responses to relations of the supporting data set D for L-SEPs or the specific states S of a D-SEP.

Goals: the originator's goals for the process. For the logical model, the goals correspond to the constraints C_D of an L-SEP. For the necessity case, we define a `MustConstraint`, which is a constraint that must be satisfied at every point in time. For the possibly case, we define a `PossiblyConstraint`, which is a constraints that should, if possible, be ultimately satisfied by the final process outcome. Alternatively, for a D-SEP, the goals can be expressed via a utility function over the eventual process outcome. We refer to this type of goal as a `TradeoffGoal` because it strives to balance the utility U of the expected process outcome against the costs of actions taken to achieve that outcome. Currently, the SEP author must also encode information about the expected participant behavior (e.g., the transition function T of the D-SEP model) when specifying a `TradeoffGoal`, but future work could use the techniques described in Section 4.4.4 to compute this automatically.

The manager uses the process's goals to decide when to make a rejection or suggestion. Goals may also specify some text to explain these interventions to the participants. This text may be static or dynamically generated based on the current state (e.g., "Sorry, we already have 5 more Appetizers than Desserts"). Providing enough detail in the messages so that they are understandable to the participants (supporting ease of use and also helping to produce the desired cooperation) can be a challenge for the SEP author. Section 5.5 discusses techniques for automatically constructing these explanations.

The constraints or utility functions are written as expressions involving arbitrary arithmetic functions over constants and variables. There are three classes of variables:

- **Parameters:** a value provided by the originator when instantiating the template (e.g., `Choices`, the options to offer the participants).
- **Author-defined:** any variable explicitly defined by the SEP author. These variables may represent common subexpressions or may be used as quantification variables (e.g., to consider each value of `Choices`, verifying that none violate the constraints). In addition, these variables may be queries over a supporting data set that contains the responses of each participant to the originator's request. For convenience, these RDF responses are mapped to a virtual relational table that may be queried via SQL.

Table 5.1: Trigger conditions for a SEP notification.

Trigger name	Description
OnAllResponsesReceived	Fires once when the expected number of responses have been received.
OnMessageReceived	Fires every time a message is received.
OnMessageAccepted	Fires every time a message is accepted.
OnMessageRejected	Fires every time a message is rejected.
OnDateTime	Fires once when the current time equals the specifed time. Used primarily for reminders.
OnConditionSatisfied	Fires when the given condition (usually based on a query of the current state) is satisfied, but was *not* true in the previous state. Useful for sending a message such as "Enough participants have RVSP'd for the game" that should be sent only once unless the truth-value of the condition changes.
OnConditionSatisfiedFirstTime	Fires when the given condition is satisifed, but only if this is the first such occurrence in the life of the process. Useful for sending a message such as "All the tickets have been claimed" that should be sent to everyone exactly once.
OnConditionSatisfiedAnyTime	Fires when the given condition is satisfied, after any change in the state of the process. Because this generates very frequent messages, it is useful primarily for updating the process summary.

- **Pre-defined:** variables automatically computed by the manager (e.g., `NumResponses`, the total number of responses received so far). The system also provides a few common queries over the supporting data set (e.g., `Bringing.Entree.count()` is the number of "Entree" responses received).

Notifications: a set of email messages to send when some condition is satisfied. The target of a notification may be an arbitrary list of email recipients (possibly from a query over the underlying data set). Alternatively, the target may be the `Originator`, `Responders`, `NonResponders` (particularly useful for sending a reminder to respond after a few days), or `AllParticipants`. Finally, sending a notification to the virtual target `ProcessSummary` adds the notification text to a process-specific web page. This web page contains a table with the response of each participant; adding `ProcessSummary` notifications is useful for displaying further summary information over this data (e.g., to show the most popular response to a vote).

A notification may be triggered by the conditions listed in Table 5.1. For instance, `OnAllResponsesReceived` may be used to notify the originator or the participants of the final process outcome, or `OnConditionSatisfied` may be used to trigger a notification instead when the number of guests reaches a certain level. The `OnDateTime` condition is useful for sending a reminder to the participants that have not yet responded after a few days. Reminder notifications may be set to automatically repeat after every T seconds until a designated point in time. In addition to the notifications specified in the template, our

implementation allows the originator to easily create an arbitrary number of `OnDateTime` "reminders" when instantiating the process or viewing its results later.

Notifications have some overlap with the set of messages M (for L-SEPs) and the set of actions A (for D-SEPs) that the manager may use to direct the outcome of a SEP. For instance, reminder notifications can be also be considered to be in these sets. Other notifications (e.g., detecting when the number of guests reaches some value) do not have a direct analogue in the L-SEP or D-SEP model, but are introduced to make SEPs more practical.

5.3.2 Template Example

Figure 5.2 shows a complete SEP template for our example balanced potluck. Parameters that must be instantiated by the originator are shown in bold; other variables such as `TotalGuests` will be evaluated as the SEP is executed. The declaration follows the four main parts described above. First, the template specifies the participants and a suitable prompt for the initial message. Second, the template defines two questions. The `Bringing` question indicates that a valid response to this question must be in the (originator-provided) set `Choices`. Here the `query` property provides the aforementioned RDQL query. The `NumGuests` question is "guarded" so that it applies only if the parameter `AskForNumGuests` is true; if so, this question will accept only non-negative integers. Because a question defines data that may be accessed in multiple other locations in the template, it is important to be able to reason about whether its guard might evaluate to false. Section 5.4 considers this issue in more detail.

Third, the template specifies one `MustConstraint` goal. The constraint is evaluated over every possible (`x,y`) where `x` and `y` are in the set `(Choices - OptOut)`; `OptOut` is for choices such as "Not Coming" that should be excluded from the constraints. The constraint requires that the number of responses `x` (e.g., Appetizer) must differ from the number of responses `y` (e.g., Dessert) by no more than `MaxImbalance`. The `message` property provides an explanation to send to a participant if their response is rejected because of this constraint.

```
:participants   "$ParticipantsList$";
:prompt "You're invited to the following potluck. Please use the form below to indicate what you are
        bringing. To ensure that our meal selection is balanced, you may be asked to modify your choice.
        Description:   $PromptDescription$
        Date and Time: $PromptDateTime.toUserFriendly()$";
:questions (
  [a              :StringQuestion;
   :name          "Bringing";
   :query         "WHERE (?process, <rdfcal:attendee>, ?x1),
                   (?x1,  <rdfcal:calAddress>,        ?EMAIL),
                   (?x1,  <uw:bringing>,              ?Bringing)
                   USING rdfcal FOR <http://www.w3.org/2002/12/cal/ical#>,
                   uw FOR <http://www.cs.washington.edu/research/semweb/vocab#v1_0>";
   :enumeration   "$Choices$" ]
  [a              :IntegerQuestion;
   :guard         "$AskForNumGuests$";
   :name          "NumGuests";
   :query         "WHERE (?process ....";
   :minInclusive  "0"; ]  );
:goals (
  # Reject the message if it results in too much inbalance between any two pairs
  [a              :MustConstraint;
   :forAll        ([:name   "x";   :range "$Choices$-$OptOut$"]
                   [:name   "y";   :range "$Choices$-$OptOut$"]);
   :suchThat      "$x$ != $y$";
   :enforce       "abs($Bringing.{$x$}.count()$ - $Bringing.{$y$}.count()$) <= $MaxImbalance$";
   :message       "Your request to bring a $Bringing.last()$ could not be accepted.
                   Choices that could be accepted right now are $Bringing.acceptable()$.";]  );
:notifications (
  # Notify the owner if the number of guests crosses a threshold (ignore if $GuestThreshold$ is zero)
  [a              :OnConditionSatisfied;
  :guard          "$GuestThreshold$ != 0";
  :define         ([:name "TotalGuests";   :value "[SELECT SUM(NumGuests) FROM CURR_STATE]"]);
  :condition      "$TotalGuests$ >= $GuestThreshold$";
  :notify         :Originator;
  :message        "Currently, $TotalGuests$ guests are expected.";]
  # Update the process summary
  [a              :OnMessageReceived;
  :notify         :ProcessSummary;
  :message        (  "Here's how many of each choice confirmed so far:"
                     [ :forAll   ([:name   "x";  :range "$Choices$"]);
                       :evaluate "$x$: $Bringing.{$x$}.count()$"; ] ) ] )
```

Figure 5.2: SEP template for a "Balanced Potluck" process. The template is shown in N3 format [16], which is an alternative syntax for writing RDF. Variables in bold (e.g., **$Choices$**) are parameters provided by the originator when instantiating the template. Other variables are defined inside the declaration (e.g., `$x$, $TotalGuests$`) or are automatically computed by the system (e.g., `$Bringing.acceptable()$`).

This message utilizes the predefined variable `Bringing.acceptable()`, which is explained in Section 5.5.

Finally, the template specifies two notifications. The first notifies the originator as soon as the total number of expected guests (computed via a SQL query over the supporting data set) reaches `GuestThreshold`. The other notification updates the process summary to include counts of each type of response received. Notice the use of the `forAll` property to iterate over the possible responses, similar to its use in the `MustConstraint`.

The above example demonstrated the use of a `MustConstraint` goal; the same properties may be used to define a `PossiblyConstraint` instead. A `TradeoffGoal` follows the same

general form but instead of an `enforce` property it provides a utility expression via an `optimize` property, along with additional properties to describe the associated costs and probabilities. Appendix B provides a complete description of the allowable properties in a SEP template and explains their interpretations once instantiated as a declaration.

5.3.3 Discussion

The example above illustrates two different ways for accessing the data collected by the SEP: via a pre-defined variable (e.g., `Bringing.last(),Bringing.$x$.count()`) or, less commonly, by utilizing an explicit SQL query over the RDF data (e.g., as with `TotalGuests`). The former method is more convenient and allows the author to easily specify decisions based on a variety of views of the underlying data. More importantly, if the goals refer to response data only through such pre-defined variables, then they are guaranteed to be *constant-bounded* (for L-SEPs) or *K-partitionable* (for D-SEPs), because they enable the SEP to summarize all of the responses that have been received with a set of counters, where the number of counters is independent of the number of participants.[1] Recall that for these types of goals (which still enable many useful SEPs), the optimal message handling policy can be computed in polynomial time (Theorems 4.3.2 and 4.4.2). Thus, the language enables more complex data access mechanisms as necessary but helps authors to write SEPs that are computationally tractable.

This example also highlights additional key features of our language, including:

- **Guards** extend the L-SEP and D-SEP model to enable optional functionality, e.g., to ask a question only if the parameter `$AskForNumGuests$` is true.

- **Sets and universal quantification,** together with set manipulation, make it possible to expand single template elements into multiple elements of the L-SEP or D-SEP

[1] This restriction effectively limits a SEP to counting responses according to a bounded number of equality predicates (e.g., how many have $Bringing = Dessert$). Thus, Definition 4.4.1 directly implies that a D-SEP utility function will be K-partitionable. Likewise, this restriction ensures that L-SEP constraints will be constant-bounded, though for this case Definition 4.3.5 also permits counting responses via inequality predicates (e.g., how many have $NumGuests > 3$). Note also that this restriction is necessary only for goals, not for notifications (which require only current evaluation, not reasoning over future states).

model. For instance, the example quantified `MustConstraint` automatically expands to produce an L-SEP constraint for each possibility in the set `$Choices$-$OptOut$`.

- **Question types and restrictions** enable the author to limit the valid type (e.g., `IntegerQuestion`) and range of responses (e.g., with `minInclusive`).

- **Multiple goal types** provide access to both L-SEP (e.g., `MustConstraint, PossiblyConstraint`) and D-SEP (e.g., `TradeoffGoal`) functionality.

- **Mathematical functions and comparisons** enable the expression of complex goals and conditions (e.g., `abs($x$-$y$) <= $MaxImbalance$`).

- **Pre-defined queries over the supporting data set** make it convenient to access the data and use it in helpful ways (e.g., the use of `$Bringing.last()$` in the explanatory `message`). In addition, these queries facilitate the specification of goals in terms of aggregates of the data (e.g., `$Bringing.$x$.count()$`).

Among other advantages, guards, sets, and universal quantification enable a single, concise SEP template to be instantiated with many different choices and configurations. Likewise, question types and restrictions reduce template complexity by ensuring that responses are well-formed. Finally, multiple goal/notification types, mathematical functions, and pre-defined queries simplify the process of making decisions based on the responses that are received. Overall, these features make it substantially easier to author useful SEPs with potentially complex functionality.

Using this template language, we have authored and deployed a number of SEPs for simple tasks such as collecting RSVPs, giving tickets away (first-come, first-served), scheduling meetings, and balancing a potluck. This experience has demonstrated that the language is sufficient for specifying a wide range of useful SEPs.

In addition, we benefited from an unintended experiment that highlights the advantages of specifying SEP templates and declarations declaratively. As a proof-of-concept, we originally implemented SEPs procedurally, using Java functions and manually constructed HTML forms for each type of SEP. Later, we re-implemented our SEPs using the declara-

Table 5.2: Comparison of the size (in number of lines) of different ways of specifying a SEP. For the procedural prototype, the first numerical section displays the size of the Java code for encoding the SEP functionality, size of the HTML for acquiring parameters from the originator, and the total of these two. For the declarative approach, the second section displays the size of the template (OWL, in N3 format), size of the parameter description (see Section 5.4), and the total. The final column shows the percentage reduction in the size of a SEP when changing from the procedural approach to the declarative approach.

	Procedural approach			**Declarative approach**			**Size Reduction**
SEP name	**Java code**	**Forms**	**Total**	**Template**	**Forms**	**Total**	**for Declarative**
Balanced Potluck	1283	397	1680	113	57	170	90%
First-come, First-served	301	235	536	66	33	99	82%
Meeting Coordination	471	272	743	60	22	82	89%
Request Approval	772	286	1058	80	29	109	90%
Auction	392	111	503	55	43	98	81%

tive language described above, producing much simpler and more concise specifications. In particular, Table 5.2 displays the number of lines of OWL needed for a number of sample SEP templates vs. the number of lines of Java/HTML needed in our original prototype. Overall, the declarative approach requires about 80-90% fewer lines than the procedural approach. There are also additional advantages of declarativism. For instance, a declarative template greatly simplifies the deployment of a new SEP, both because no programming is required and because authors need not run their own server (since shared servers can accept and execute OWL declarations from anyone, something they are unlikely to do for arbitrary code). An additional advantage of declarative specifications is that they could enable future work that automatically composes several SEPs to accomplish more complex goals. Finally, this approach enables the use of a variety of automated reasoning procedures to ensure that a SEP declaration is valid. After introducing template instantiation, the next section will describe one important instance of such reasoning.

5.4 Template Instantiation and Verification

To provide ease of use, the second major challenge for template-based specifications is to ensure that originators can easily and safely instantiate a template into a SEP declaration that will accomplish their goals. This section first briefly describes how to acquire and validate instantiation parameters from the originator. We then examine in more detail the problem of ensuring that a template cannot be instantiated into an invalid declaration.

```
:parameters (
  [a          :TypeStringSet;
   :name      "Choices";
   :prompt    "Choices for the recipients to choose from" ]
  [a          :TypeStringSet;
   :name      "OptOut";
   :prompt    "Choices to exclude from these restrictions";
   :subsetOf  "$Choices$" ]
  [a          :TypeBoolean;
   :name      "AskForNumGuests";
   :choices (
    [:value :True;  :prompt "Yes, ask how many guests each person is bringing"  ]
    [:value :False; :prompt "No, don't ask about guests"] ) ]
  [a          :TypeInteger;
   :name      "GuestThreshold";
   :prompt    "Notify me when the number of guests reaches (enter 0 to ignore):";
   :minInclusive "0" ]
)
```

Figure 5.3: Part of a *parameter description* for the potluck template of Figure 5.2. Additional elements for variables such as `MaxImbalance` are not shown.

5.4.1 Parameter Descriptions

Each SEP template must be accompanied by a web form that enables originators to provide the parameters needed to instantiate the template into a declaration. To automate this process, our implementation provides a tool that generates such a web form from a simple OWL *parameter description*:

Definition 5.4.1 (**parameter description**) A parameter description ϕ for a template τ is a set $\{R_1, ..., R_M\}$ where each R_i provides, for each parameter P_i in τ, a name, prompt, type, and any restrictions on the legal values of P_i. Parameters may have a simple type (Boolean, Integer, Double, String, Email address) or a set type (i.e., a set of simple types). Possible restrictions are: (for simple types) enumeration, minimal or maximal value, and (for sets) non-empty, or a subset relationship to another set parameter. □

Figure 5.3 shows a partial example for our example balanced potluck. For instance, the first parameter block specifies that `Choices` is a set of strings, while the second parameter indicates that `OptOut` is a set of strings that must be a subset of `Choices`. The last two parameters relate to asking participants about the number of guests that they will bring to the potluck. The ontology for parameter descriptions also contains elements for adding descriptive text, and for specifying layout information (e.g., to group similar items together). Appendix B presents the complete ontology.

The form generator tool takes a parameter description and template as input and outputs a form for the originator to fill out and submit. If the submitted variables comply with all parameter restrictions, the template is instantiated with the corresponding values and the resulting declaration is forwarded to the manager for execution. Otherwise, the tool redisplays the form with errors indicated and asks the originator to try again.

5.4.2 Instantiation Safety

Unfortunately, not every instantiated template is guaranteed to be executable. For instance, consider instantiating the potluck template of Section 5.3 with the following (partial list of) parameters:

```
AskForNumGuests = False
GuestThreshold  = 50
```

In this case the notification given in Section 5.3 is invalid, since it refers to a question symbol `NumGuests` that does not exist because the parameter `AskForNumGuests` is false. Thus, the declaration is not executable and must be refused by the server. This particular problem could be addressed either in the template (by adding an additional `guard` on the notification) or in the parameter description (by adding a parameter restriction on `GuestThreshold`). However, this leaves open the general problem of ensuring that *every* instantiation results in a *valid declaration*:

Definition 5.4.2 (**valid declaration**) An instantiated template δ is a valid declaration if:

1. **Basic checks:** δ must validate without errors against the SEP ontology, and every expression $e \in \delta$ must evaluate to a valid numerical or set result.

2. **Enabled symbols:** For every expression $e \in \delta$ that is *enabled* (i.e., does not have an unsatisfied guard), every symbol in e is defined once by some enabled node.

3. **Non-empty enumerations:** For every enabled `enumeration` property $p \in \delta$, the object of p must evaluate to a non-empty set. □

Definition 5.4.3 (**instantiation safety**) Let τ be a template and ϕ a parameter description for τ. τ is instantiation safe w.r.t. ϕ if, for all parameter sets ξ that satisfy the restrictions in ϕ, instantiating τ with ξ yields a valid declaration δ. □

Instantiation safety is of significant practical interest for two reasons. First, if errors are detected in the declaration, any error message is likely to be very confusing to the originator (who knows only of the web form, not the declaration). Thus, to ensure ease of use an automated tool is desirable to verify that a deployed template is instantiation safe. Second, constructing instantiation safe templates can be very onerous for authors, since it may require considering a large number of possibilities. Even when this is not too difficult, having an automated tool to ensure that a template remains instantiation safe after a modification would be very useful.

Some parts of verifying instantiation safety are easy to perform. For instance, checking that every *declaration* will validate against the SEP ontology can be performed by checking the *template* against the ontology, and other checks (e.g., for valid numerical results) are similar to static compiler analyses. However, other parts (e.g., ensuring that a symbol will always be enabled when it is used) are substantially more complex because of the need to consider all possible instantiations permitted by the parameter description ϕ. Consequently, in general verifying instantiation safety is difficult:

Theorem 5.4.1 *Given τ, an arbitrary* SEP *template, and ϕ, a parameter description for τ, then determining* instantiation safety *is co-NP-complete in the size of ϕ.*

This theorem is proved by a reduction from $\overline{SAT}$. Intuitively, given a specific counter-example it is easy to demonstrate that a template is *not* instantiation safe, but proving that a template is safe potentially requires considering an exponential number of parameter combinations. In practice, ϕ may be small enough that the problem is feasible. Furthermore, in certain cases this problem is computationally tractable:

Theorem 5.4.2 *Let τ be a* SEP *template and ϕ a parameter description for τ. Determining* instantiation safety *is polynomial time in the size of τ and ϕ if:*

- *each* `forAll` *and* `enumeration` *statement in τ consists of a bounded number of set parameters combined with any set operator, and*
- *each* `guard` *consists of conjunctions and disjunctions of a bounded number of terms (which are boolean parameters, or compare a non-set parameter with a constant/parameter).*

These restrictions are quite reasonable and still enable us to specify all of the SEPs described in this work. Note that they do not restrict the total number of parameters, but rather bound the number that may appear in any one of the identified statements. The restrictions ensure that only a polynomial number of cases need to be considered for each goal/notification item, and the proof relies on a careful analysis to show that each such item can be checked independently while considering at most one question at a time. See Appendix C for details on the proof.

5.4.3 Discussion

In our implementation, we provide a tool that approximates instantiation safety testing via limited model checking. The tool operates by instantiating τ with all possible parameters in ϕ that are boolean or enumerated (these most often correspond to general configuration parameters). For each possibility, the tool chooses random values that satisfy ϕ for the remaining parameters. If any instantiation is found to be invalid, then τ is known to be not instantiation safe. Extending this approximate algorithm to perform the exact, polynomial-time (but more complex) testing of Theorem 5.4.2 is future work.

Clearly nothing in our analysis relied upon the fact that our SEPs are email-based. Instead, similar issues will arise whenever 1.) an author is creating a template that is designed to be used by other people (especially untrained people), and 2.) for flexibility, this template may contain a variety of configuration options. A large number of agents, such as the RCal meeting scheduler [146], Berners-Lee et al.'s appointment coordinator [15], and McIlraith et al.'s travel planner [132], have the need for such flexibility and could be

profitably implemented with templates. This flexibility, however, can lead to unexpected or invalid agents, and thus produces the need to verify various safety properties such as "doing no harm" [180] or the instantiation safety discussed above. Our results highlight the need to carefully design the template language and appropriate restrictions so that such safety properties can be verified in polynomial time.

5.5 Automatic Explanation Generation

While executing, the manager utilizes rejections or suggestions to influence the eventual SEP outcome. However, the success of these interventions depends on the extent to which they are understood by the participants. For instance, the rejection "Sorry, the only dates left are May 7 and May 14" is much more likely to elicit cooperation from a participant in a seminar scheduling SEP than the simpler rejection "Sorry, try again." For a particular set of goals, the author of a SEP could manually specify how to create such explanations, but this task can be very difficult when goals interact or depend on considering possible future responses. Thus, below we consider techniques for automatically generating explanations based on *what* responses are acceptable now and *why* the participant's original response was not acceptable.

We begin by defining more precisely a number of relevant terms. For a SEP, the *current state* D is the state of the supporting data set given all of the responses that have been received so far. We assume that the number of participants is known and that each will eventually respond to the initial request and to any interventions. Recall that in our implementation the manager intervenes only with rejections in the logical case (L-SEPs), and only with suggestions in the decision-theoretic case (D-SEPs).

For D-SEP goals, our template language utilizes `TradeoffGoals`. For L-SEPs, recall that we allow both `MustConstraints` and `PossiblyConstraints`, corresponding to the necessity and possibly conditions discussed in Chapter 4. We now define the difference between these two more precisely:

Definition 5.5.1 (**MustConstraint**) A `MustConstraint` C is a constraint that is satisfiable in state D iff evaluating C over D yields `True`. □

Definition 5.5.2 (**PossiblyConstraint**) A `PossiblyConstraint` C is a constraint that is *ultimately satisfiable* in state D if there exists a sequence of responses from the remaining participants that leads to a state D' so that evaluating C over D' yields `True`. □

For simplicity, we assume that the constraints C_D are either all `MustConstraints` or all `PossiblyConstraints`, though our results for `PossiblyConstraints` also hold when C_D contains both types.

5.5.1 Acceptable Responses

Often the most practical information to provide to a participant whose response led to an intervention is the set of responses that would be "acceptable" (e.g., "An Appetizer or Dessert would be welcome" or "Sorry, I can only accept requests for 2 tickets or fewer now"). This section briefly considers how to calculate this *acceptable set* for L-SEPs, then extends this notion to D-SEPs.

Definition 5.5.3 (**L-SEP acceptable set**) Let Λ be an L-SEP with current state D and constraints C_D on D. Then, the *acceptable set* A of Λ is the set of legal responses r such that D would still be satisfiable (for `MustConstraints`) or ultimately satisfiable (for `PossiblyConstraints`) w.r.t. C_D after accepting r. □

For a `MustConstraint`, this satisfiability testing is easy to do and we can compute the acceptable set by testing some small set of carefully chosen responses. For a `PossiblyConstraint`, the situation is more complex:

Theorem 5.5.1 *Let Λ be an* L-SEP *with N participants and current state D. If the constraints C_D may be any set of* `PossiblyConstraints` *permitted by the language C, then computing the acceptable set A of Λ is NP-hard in N.*

Theorem 5.5.2 *Let Λ be an* L-SEP *with N participants and current state D. If C_D consists of bounded* `PossiblyConstraints`*, then this problem is polynomial time in N, $|A|$, and $|C_D|$.*

In this case we can again compute the acceptable set by testing satisfiability over some small set of carefully chosen values; this testing is polynomial if C_D is bounded (Theorem 4.3.2). In addition, if we represent A via a set of ranges of acceptable values, instead of explicitly listing every acceptable value, then the total time is polynomial in only N and $|C_D|$.

For a D-SEP (i.e., a `TradeoffGoal`), we define an "acceptable" response as one that is "good enough" so that the manager will not respond with a change suggestion. We might also be interested in computing responses that are "better" than others, e.g., those which result in an expected utility in the top 25% compared to other possible responses. This information could be used when making a suggestion ("Please consider one of these values..."), or could be displayed as part of the process summary to assist participants that have yet to respond.

Both of these problems can be solved by comparing the expected utility of a small number of states of the D-SEP (e.g., considering all possible responses from a given initial state). Computing these utilities is intractable in general (Theorem 4.4.1), but in many cases can be computed efficiently:

Theorem 5.5.3 *Let δ be a* D-SEP *with N participants where the utility function $U(s, a)$ is K-partitionable for some constant K and where the system is permitted to send at most one suggestion to any participant. Then the expected utility of δ for every possible state of the* D-SEP *can be computed in time polynomial in N.*

This result follows from the proof of Theorem 4.4.2, since computing the optimal policy in this case involves computing and comparing the expected utility of all possible states.

Our implementation currently computes and makes available the acceptable set for `MustConstraints` and `TradeoffGoals` (see the use of `Bringing.acceptable()` in Figure 5.2). Extending this computation to support `PossiblyConstraints` is future work.

5.5.2 Explaining L-SEP *Interventions*

In some cases, the acceptable set alone may not be enough to construct a useful explanation:

Example 5.5.1 Suppose a SEP invites 4 professors and 20 students to a meeting, with the following (informally specified) constraints:

- C_P: At least three professors must attend.

- C_Q: For quorum, at least 10 persons (students or professors) must attend.

Imagine that, in the current state, one professor has said `Yes`, one professor has said `No`, three students have said `Yes`, and one student has said `No`. Now suppose that a new `No` response from a professor arrives. Since one professor has already said `No`, the SEP should ask the latest respondent to change their answer. However, when requesting this change, explaining *why* the change is needed (e.g., "We need you to reach the required 3 professors") is much more effective than simply informing them of *what* response is desired (e.g., "Please change to Yes"). A clear explanation both motivates and rules out alternative reasons for the request (e.g., "We need your help reaching quorum") that may be less persuasive (e.g., because many students could also help reach quorum). More detailed responses might also be helpful (e.g., "We need one more professor to attend, and the other professors have already responded."). □

For L-SEPs, this section discusses how to generate such explanations for an intervention based on identifying the constraint(s) that led to the intervention (e.g., "C_P"); the next section considers the corresponding problem for D-SEPs. We do not discuss the additional problem of translating these constraints/utilities into a natural language suitable for sending to a participant, but note that even fairly simple explanations (e.g., "Too many Appetizers (10) vs. Desserts (3)") are much better than no explanation.

Conceptually, the manager decides to reject a response for an L-SEP based on constructing a *proof tree* that shows that some response r would prevent constraint satisfaction. However, this proof tree may be much too large and complex to serve as an explanation for a participant. This problem has been investigated before for expert systems [140, 169], constraint programming [96], description logic reasoning [130], and more recently in the context of the Semantic Web [131]. These systems assumed proof trees of arbitrary complexity and handled a wide variety of possible deduction steps. To generate useful explanations, key techniques included abstracting multiple steps into one using rewrite rules [130, 131],

describing how general principles were applied in specific situations [169], and customizing explanations based on previous utterances [29].

In our context, the proof trees have a much simpler structure that we can exploit. In particular, proofs are based only on constraint satisfiability (over one state or all possible future states), and each child node adds one additional response to the parent's state in a very regular way. Consequently, we will be able to summarize the proof tree with a very simple type of explanation. These proof trees are defined as follows:

Definition 5.5.4 (**L-SEP proof tree**) Given an L-SEP Λ, current state D, constraints C_D, and a response r, we say that P is a *proof tree* for rejecting r on D iff:

- P is a tree where the root is the initial state D.
- The root has exactly one child D_r, representing the state of D after adding r.
- If C_D is all `MustConstraints`, then D_r is the only non-root node.
- If C_D is all `PossiblyConstraints`, then for every node n that is D_r or one of its descendants, n has all children that can be formed by adding a single additional response to the state of n. Thus, the leaf nodes are only and all those possible final states (e.g., where every participant has responded) reachable from D_r.
- For every leaf node l, evaluating C_D over the state of l yields `False`. □

Figure 5.4A illustrates a proof tree for `MustConstraints`. Because accepting r leads to a state where some constraint (e.g., C_T) is not satisfied, r must be rejected. Likewise, Figure 5.4B shows a proof tree for `PossiblyConstraints`, where C_P and C_Q represent the professor and quorum constraints from Example 5.5.1. Since we are trying to prove that there is no way for the constraints to be ultimately satisfied (by any outcome), this tree must be fully expanded. For this tree, every leaf (final outcome) does not satisfy some constraint, so r must be rejected.

We now define a simpler explanation based upon the proof tree:

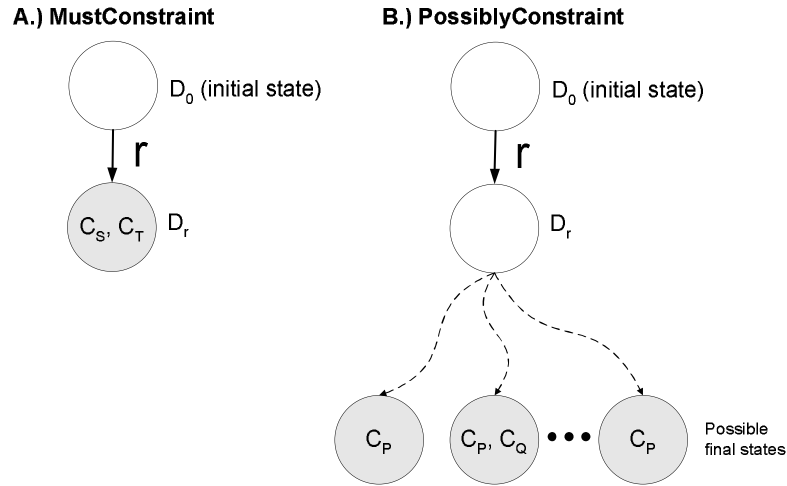


Figure 5.4: Examples of proof trees for rejecting response r. Each node is a possible state of the data set, and node labels are constraints that are *not* satisfied in that state. In both cases, response r must be rejected because every leaf node (shaded above) does not satisfy some constraint.

Definition 5.5.5 (**L-SEP sufficient explanation**) Given an L-SEP Λ, current state D, constraints C_D, and a response r such that a proof tree P exists for rejecting r on D, we say that E is a *sufficient explanation* for rejecting r iff,

- E is a conjunction of constraints that appear in C_D, and
- for every leaf node l in P, evaluating E over the state of l yields `False`. □

Intuitively, a sufficient explanation E justifies rejecting r because E covers every leaf node in the proof tree, and thus precludes ever satisfying C_D. Note that while the proof tree for rejecting r is unique (modulo the ordering of child nodes), an explanation is not. For instance, an explanation based on Figure 5.4A could be C_S, C_T, or $C_S \wedge C_T$. Likewise, a valid explanation for Figure 5.4B is $C_P \wedge C_Q$ (e.g., no way satisfy both the professor and quorum constraints) but a more precise explanation is just C_P (e.g., no way to satisfy the professor constraint). The smaller explanation is often more compelling, as we argued for the meeting example, and thus to be preferred [42]. In general, we wish to find an explanation of minimum size (i.e., with the fewest conjuncts):

Theorem 5.5.4 *Given an* L-SEP Λ *with* N *participants, current state* D*, constraints* C_D*, and a response* r*, if* C_D *consists of* `MustConstraints`*, then finding a minimum sufficient explanation* E *for rejecting* r *is polynomial time in* N *and* $|C_D|$*. If* C_D *consists of* `PosiblyConstraints`*, then this problem is NP-hard in* N *and NP-hard in* $|C_D|$*.*

Thus, computing a minimum explanation is feasible for `MustConstraints` but likely to be intractable for `PossiblyConstraints`. For the latter, the difficulty arises from two sources. First, checking if any particular E is a sufficient explanation is NP-hard in N (based on a reduction from ultimate satisfiability); this makes scaling SEPs to large numbers of participants difficult. Second, finding a minimum such explanation is NP-hard in the number of constraints (by reduction from SET-COVER [93]). Note that this number can be significant because we treat each `forAll` quantification as a separate constraint; otherwise, the sample potluck described in Section 5.3 would always produce the same (complex) constraint for an explanation. Fortunately, in many common cases we can simplify this problem to permit a polynomial time solution:

Theorem 5.5.5 *Given an* L-SEP Λ *with* N *participants, current state* D*, constraints* C_D*, and a response* r*, if* C_D *is bounded and the size of a minimum explanation is no more than some constant* J*, then computing a minimum explanation* E *is polynomial time in* N *and* $|C_D|$*.*

This theorem holds because a candidate explanation E can be checked in polynomial time when the constraints are bounded (Theorem 4.3.2), and restricting E to at most size J means that the total number of explanations that must be considered is polynomial in the number of constraints. Both of these restrictions are quite reasonable. As previously mentioned, bounded constraints permit a wide range of functionality. Likewise, SEP explanations are most useful to the participants when they contain only a small number of constraints, and this is adequate for many SEPs (as in the meeting example above). If no sufficient explanation of size J exists, the system could either choose the best explanation of size J (to maintain a simple explanation), approximate the minimum explanation with a greedy

algorithm, or fall back on just providing the participant with the acceptable set described in the previous section.

Many different types of agents can describe their goals in terms of a set of constraints [115, 139], and often need to explain their actions to users. Our results show that while generating such explanations can be intractable in general, the combination of simple explanations and modest restrictions on the constraint system can enable explanation generation in polynomial time.

5.5.3 *Explaining* D-SEP *Interventions*

As with L-SEPs, we would like to be able to automatically generate explanations for the manager's interventions. Below we briefly consider this problem in the context of D-SEPs.

Compared to L-SEPs, it is more difficult for a D-SEP to single out specific terms that are responsible for a manager's suggestion, because every term contributes to the process utility to some extent, either positively or negatively. Note, though, that if the manager decides to make a suggestion, then the expected improvement must outweigh the certain cost of this action. Thus, for non-zero costs, there must be a significant difference in the utility of the state where the manager requested a switch (S_{sw}) vs. where the manager did not (S_0).

We seek to identify the terms that explain most of this difference. In particular, given a M-term additive utility function

$$U(s) = u_1(s) + ... + u_M(s)$$

we define the change δ_u in each utility term as

$$\delta_u = u(S_{sw}) - u(S_0).$$

We wish to identify an explanation as follows:

Definition 5.5.6 (**D-SEP sufficient explanation**) Given a D-SEP δ with an M-term additive utility function U, a constant β ($0 \leq \beta \leq 1$), and two states S_{sw} and S_0, a sufficient explanation is a set $E \subseteq \{u_1, .., ..u_M\}$ such that

$$\sum_{u \in E} \delta_u \geq \beta[U(S_{sw}) - U(S_0)]$$

(i.e., so that the terms in E explain at least β of the change). □

As before, we are interested in finding the explanation of minimal size (i.e., the smallest such set):

Theorem 5.5.6 *Let δ be a* D-SEP *with N participants and M-term additive utility function U. If U is K-partitionable and the manager is permitted to make only a bounded number of suggestions to each participant, then computing the minimal sufficient explanation between two states S_{sw} and S_0 is polynomial time in N and M.*

This theorem can be proved in two steps. First, if the utility function is K-partitionable and the number of suggestions is bounded (and hence we can compute the optimal policy in polynomial time), then we can also compute each δ_u in time polynomial in N by applying Theorem 5.5.3. Second, given the values for δ_u, a greedy algorithm can find the explanation E of guaranteed minimal size: set E to $\emptyset$, then incrementally add to E the term with the largest (most positive) δ_u until E explains at least β of the total change. Sorting the M δ_u terms can be done in time polynomial in M, so the total algorithm runs in time polynomial in N and M.

Note that this procedure will never produce an explanation with a term δ_u where $\delta_u \leq 0$, since such a term could always be removed and still yield a sufficient explanation. This makes sense so long as we are primarily interested in explanations that justify why a switch is beneficial, i.e., where $\delta_u > 0$. If we wish to consider utility terms with both positive and negative changes, then this problem becomes more challenging (cf., Klein and Shortliffe [101]).

5.6 Related Work

Other projects have considered how to simplify the authoring of Semantic Web applications. For instance, Jena [123] and Kaon [175] offer programmers standard APIs for manipulating RDF, whereas Haystack provides the Adenine programming language to simplify these tasks [151]. Adenine resembles our template language in that it can be compiled into RDF for portability and contains a number of high-level primitives, though Adenine incorporates many more imperative features and does not support the types of declarative reasoning that we describe. Finally, languages such as DAML-S and OWL-S [40] enable the description of an application as a Semantic Web *service*. These languages, however, focus on providing details needed to *discover* and *invoke* a relevant service, and model every participant as another web service. Our work instead concisely specifies a SEP in enough detail so that it can be directly *executed* in contexts involving untrained end users.

More generally, SEP templates could be viewed as an instance of *program schemas* [45, 65] that encapsulate a general class of behavior, e.g., for automated program synthesis [65] or software reuse [45, 6]. Similarly, McIlraith et al. [132] propose the use of *generic procedures* that can be instantiated to produce different compositions of web services. Concepts similar to our definition of instantiation safety naturally arise in this setting; proposals for ensuring this safety have included manually-generated proofs [45], automatically-generated proofs [65], and language modification [6]. Our work focuses on the need for such schemas to be safely usable by ordinary people and demonstrates that the required safety properties can be verified in polynomial time.

Recent work on the *Inference Web* [131] has focused on the need to explain a Semantic Web system's *conclusions* in terms of base data and reasoning procedures. In contrast, we deal with explaining the SEP's *actions* in terms of existing responses and the expected impact on the goals. In this sense our work is similar to prior research that sought to explain decision-theoretic advice (cf., Horvitz et al. [87]). For instance, Klein and Shortliffe [101] describe the VIRTUS system that can present users with an explanation for why one action is provided over another. Note that this work focuses on explaining the relative impact of multiple factors on the choice of some action, whereas we seek the simplest

possible reason why some action could *not* be chosen (i.e., accepted). Other relevant work includes Druzdzel [53], which addresses the problem of translating uncertain reasoning into qualitative verbal explanations.

For constraint satisfaction problems (CSPs), a *nogood* [158] is a reason that no *current* variable assignment can satisfy all constraints. In contrast, our explanation for a `PossiblyConstraint` is a reason that no *future* assignment can satisfy the constraints, given the set of possible future responses. Potentially, our problem could be reduced to nogood calculation, though a direct conversion would produce a problem that might take time that is exponential in N, the number of participants. However, for bounded constraints, we could create a CSP with variables based on the *aggregates* of the responses, rather than their specific values, as described in Chapter 4. Using this simpler CSP, we could then exploit existing, efficient nogood-based solvers (e.g., [95, 99, 94]) to find candidate explanations in time polynomial in N. Note though that most applications of nogoods have focused on their use for developing improved constraint solving algorithms [158, 99] or for debugging constraint programs [143], rather than on creating explanations for average users. One exception is Jussien and Ouis [96], who describe how to generate user-friendly `nogood` explanations, though they require that a designer explicitly model a user's perception of the problem as nodes in some constraint hierarchy.

Finally, we assumed that users were best served by a *single*, minimal explanation, which we defined to be the explanation with the fewest conjuncts. In some cases, however, it may make sense to provide users with more information, e.g., all reasons why a response could not be accepted, or a more sophisticated summary of these reasons. For this task, related work on the problem of computing *all* minimal explanations may be useful [42].

5.7 Summary and Implications for Agents

This chapter examined how to specify SEPs that are usable by ordinary people. We adopted a template-based approach that shifts most of the complexity of SEP specification from untrained originators onto a much smaller set of trained authors. We then examined the three key challenges of generality, safety, and understandability that arise in this approach.

In particular, we discussed how high-level features of our template language enable the concise specification of complex behavior while maintaining the tractable reasoning described in Chapter 4. We also demonstrated that it is possible to verify the instantiation safety of a template in polynomial time, and showed how to generate explanations for the SEP's actions in polynomial time. Together, these techniques both simplify the task of the SEP author and improve the overall execution quality for the originator and participants of a SEP. In addition, our polynomial time results ensure that these features can scale to SEPs with large numbers of participants, choices, and goals. Consistent with our gradual adoption and ease of use principles, these features facilitate the development of a range of broadly-applicable, explainable SEPs that are guaranteed to be safely invocable by non-technical users via generic web browsers.

Our results for semantic email are also relevant to other agent systems. Many other agents (e.g., [15, 146, 132]) can be viewed as having an author, originator, and participants. Each participant may be a human or another agent, and may require some explanation for an intervention. For instance, RCal [146] presents users with a finite number of interactions that they may originate, and explanation would be a useful addition to the system (e.g., to justify meeting rescheduling). Likewise, McIlraith et al. [132] propose the use of a number of template-like generic procedures for travel planning, and it would be useful to be able to generate explanations, both for the originator (why can't I return home on Friday?) and for other participants, such a booking agents (what about the proposed itinerary is unsatisfactory?). We showed that generating explanations can be NP-hard in general, but that the combination of simple explanations and modest goal restrictions may enable explanation generation in polynomial time.

In addition, such agents frequently require a fair amount of flexibility in the specification of their goals, e.g., to support trips with variable numbers of destinations or meetings with variable attendance and RSVP requirements. We showed how a high-level, declarative template language could support a wide range of functionality, and explored how to ensure the *safety* of each possible use. There are several different types of safety to consider, including that of doing no permanent harm [180], minimizing unnecessary side-effects [180], and accurately reflecting the originator's preferences [23]. We motivated the need for instantiation

safety, a type that has been previously examined to some extent [65, 6], but is particularly challenging when the instantiators are non-technical users. Our results also highlight the need to carefully design template languages that balance behavior flexibility with the ability to efficiently verify such safety properties.

Thus, many agents could benefit from a high-level, declarative template language with automatic safety testing and explanation generation. Collectively, these features would simplify the creation of an agent, broaden its applicability, enhance its interaction with the originator and other participants, and increase the likelihood of satisfying the originator's goals.

Chapter 6

CONCLUSIONS

Our goal was to discover how to enable and motivate non-technical people to both utilize and contribute content to the Semantic Web. This chapter summarizes the contributions of this dissertation towards that goal and points to the many opportunities for future work.

6.1 Contributions

This dissertation proposed the use of three key design principles for Semantic Web systems, then described novel mechanisms and theory that support these principles in the construction of two new systems. Below we briefly examine these principles. Next, we discuss the contributions made in applying these principles to our two systems. Finally, we consider additional contributions related to the field of intelligent agents.

We proposed three design principles that should be followed by any Semantic Web system that seeks to have participation by non-technical people. First, the *instant gratification* principle requires that both application usage and content creation provide immediate, tangible benefit to the user. Applying this principle carefully not only forces designers to ensure that applications and authoring are well motivated, but also provides a metric for quickly ruling out a wide variety of previously-used techniques (e.g., aggregation solely via periodic web crawls, publication only after moderator approval) that impede this motivation. Next, the *gradual adoption* principle addresses the common chicken-and-egg problem of Semantic Web systems by ensuring that applications can be profitably invoked even when there are few existing system users and that content can be incrementally provided from previous representations. This principle thus serves to prime the network effect that helped to make the original web so successful. Finally, the *ease of use* principle insists that the basic system be as simple as possible to use, ideally requiring no special knowledge, training, or

software. This principle is essential given our desire to motivate the initial participation of non-technical people, but still permits more advanced interfaces for users that have become convinced of the system's utility. Together, these principles both simplify the development of a Semantic Web system (by focusing attention on its most important features) and greatly increase the chances of its adoption by non-technical persons.

6.1.1 Contributions of the MANGROVE *System*

The first of our implemented Semantic Web systems, MANGROVE, motivates the annotation of existing HTML content. MANGROVE's key contributions — its architecture, services, and MTS annotation syntax — directly support our three design principles. In particular, MANGROVE's architecture provides instant gratification with a loop that takes freshly published content to semantic services, and then back to the user through the service feedback mechanism. We described several such services that motivate the annotation of HTML content by consuming semantic information, and explained how MANGROVE's declarative approach can boost this motivation by leveraging the same content across multiple services. Second, MANGROVE enables gradual adoption by seeding its services with initial content and by enabling authors to incrementally annotate existing content with our MTS syntax. We showed how MANGROVE could utilize such incomplete content and thus provide tangible benefit to authors even when pages are only sparsely annotated. Finally, MANGROVE supports ease of use by providing a simple graphical annotation tool, deferring integrity constraints to the services, and reusing familiar interfaces and methods of determining the trustworthiness of data.

These contributions in MANGROVE have led to service execution and content creation by a variety of people with no knowledge of semantic representation. Thus, MANGROVE's design represents a concrete path for enticing ordinary people to contribute their existing content to the Semantic Web.

6.1.2 Contributions of the Semantic Email System

Our second implemented system introduced a paradigm for Semantic Email and described a broad class of semantic email processes (SEPs). In support of instant gratification, these automated processes offer tangible productivity gains on a wide variety of email-mediated tasks. We presented a formalization that teases out the issues involved, and used this formalization to explore several central inference questions. In particular, we defined and explored two useful models for specifying the goals of a process and formalizing when and how the manager of the process should intervene. For our logical model we showed how the problem of deciding whether a response was acceptable relative to a set of ultimately desired constraints could be solved in polynomial time for bounded constraints. In addition, with our decision-theoretic model we addressed several shortcomings of the logical model and demonstrated conditions in which the optimal policy for this model could be computed in polynomial time. In both cases we identified restrictions that greatly improved the tractability of the key reasoning problems while still enabling a large number of useful processes to be represented.

We also explored how to assist the adoption of Semantic Email by simplifying the task of specifying a new SEP. In particular, we designed a template-based approach that shifts most of the complexity of SEP specification from untrained originators onto a much smaller set of trained authors. We then addressed a number of challenges that arise in this approach. In particular, we discussed how high-level features of our template language enable the concise specification of complex SEP behavior. We also demonstrated conditions in which it is possible to verify the instantiation safety of such a template in polynomial time. Moreover, we described how to automatically generate explanations for the manager's interventions and identified cases where these explanations can be computed in polynomial time. These techniques both simplify the task of the SEP author and improve the overall execution quality for the originator and the participants of a SEP. In addition, our polynomial time results ensure that these features can scale to SEPs with large numbers of participants, choices, and goals.

Finally, we described our publicly available Semantic Email system and how it satisfies

Table 6.1: Summary of the roles of each person (or other agent) involved in the execution of an agent, and how they would benefit from a high-level, declarative template language with safety testing and explanation generation. This table shows that these types of features could benefit a broad range of agent systems, both email-based and otherwise.

Person and Role	Benefits described in this dissertation
Author: writes template using editor, often by modifying existing template.	• **Template approach:** allows authoring agent once for many uses • **High-level language primitives:** enables easy specification of complex goals and behavior • **Instantiation-safety testing:** eliminates need to exhaustively consider many possible template instantiations. • **Automatic explanation generation:** simplifies specification of high-quality agents
Originator: fills out form	• **Template approach:** permits agent instantiation via form, no need to understand RDF or programming • **Declarative templates:** allows any template to be instantiated on any agent server, no need to install procedural code • **Instantiation-safety testing:** ensures that any agent the originator instantiates will be executable
Participants: respond to requests	• **Automatic explanation generation:** explains reasons for interventions and how to successfully respond

the principles of gradual adoption and ease of use via its server-based implementation, simple web forms for SEP instantiation, and text messages that can be handled by any client without any software installation. The combination of these features with useful, flexible SEPs provides a platform enabling ordinary people to easily leverage lightweight semantics to accomplish common email-mediated tasks.

6.1.3 Implications for Intelligent Agents

Our results for semantic email are also relevant to other agent systems. Many other agents (e.g., [15, 146, 132]) can be viewed as having an author, originator, and participants. For such agents, Table 6.1 summarizes how a high-level, declarative template language with safety testing and explanation generation would benefit the author, originator, and participants. Collectively, these features would simplify the creation of an agent, broaden its applicability, enhance its interaction with the originator and participants, and increase the likelihood of satisfying the originator's goals. Our results both demonstrate the general importance of these different features as well as provide specific results (e.g., regarding the tractability of explanation generation for simple constraint systems) that may be directly applicable to other agent environments.

6.2 Future Directions

Despite the significant progress we have achieved, there remains room for much future work. This section details many such opportunities, beginning with extensions to the two example systems described in this dissertation and concluding with some broader themes.

6.2.1 Extensions to MANGROVE

Our goal in designing MANGROVE and deploying it locally was to test our design on today's HTML web against the requirements of ordinary users. Clearly, additional deployments in different universities, organizations, and countries are necessary to further refine and validate MANGROVE's design. In addition, new instant gratification services are necessary to drive further adoption and to explore what features are necessary in different domains. For instance, MANGROVE could benefit from a semantic browser that was able to profitably combine semantic data with ordinary HTML content. Likewise, there is the potential for interesting applications that aggregate community advice and recommendations.

Considering applications, MANGROVE's service construction template greatly simplifies the task of implementing robust services that can give immediate feedback to the author. Creating such services, however, still requires a fair amount of technical sophistication. We see two avenues for simplifying this process. First, we could expand our search service's query language to enable many more services to be written just as simple queries. Second, we could imitate our success with semantic email and create a higher-level, declarative language for constructing MANGROVE services. Creating such a language may be challenging because of the need to carefully express how to query for and cache data, what data transformations and cleaning to apply, and how to produce suitable outputs. Success in this endeavor, however, could greatly increase the number of MANGROVE applications and the number of people capable of producing such applications.

One obvious limitation of the current system is that all queries are processed by a single centralized database. As explained in Section 3.1.5, MANGROVE could be extended through the use of a peer-to-peer network for distributed content aggregation and querying. Future work should investigate this approach more fully and explore the importance of caching for

reducing system load and query latency. To better support scalability, we propose to make only a "best effort" to provide correct, complete, and fresh results. For instance, we expect that a more distributed system will have weaker guarantees on data freshness in general, but will continue to provide prompt access to newly published data for local services or for remote services that a user has specifically identified. This approach ensures that users can continue to get instant gratification when annotating their pages with a particular service in mind, yet weakens other, less critical, freshness guarantees to support scalability.

Finally, in this dissertation we strove to both enable and motivate non-technical users to contribute content to the Semantic Web. Overall, because of the larger amount of other research related to annotation, in MANGROVE we focused more on *motivating* rather than *enabling* such motivation. Hence, MANGROVE supports annotation via a text editor or a graphical annotation tool, but both methods have their shortcomings. In particular, if existing HTML authoring tools (e.g., FrontPage) are used on annotated pages, annotations may sometimes be discarded or corrupted. This problem could be remedied by incorporating annotation features directly into standard HTML editors, though at significant cost. Partial solutions could include having a separate MANGROVE service that automatically detects (and offers to correct) annotations that are lost from a previously published page, and improving the semantic parser to be more robust to malformed annotations.

6.2.2 Extensions to Semantic Email

There are a number of ways to improve how SEPs relate to authors, originators, and participants. We examine each in turn and then consider some additional opportunities for semantic email in general.

First, our declarative language greatly simplifies the task of authoring SEPs compared to the original procedural approach. However, it still requires writing a formal specification and hence a fair amount of technical skill. To simplify this task, we could instead treat this language as an intermediate representation that is the output of a graphical design tool. Authors could use this tool to construct a template by combining different building blocks for gathering information, enforcing goals, and sending notifications. Potentially, this tool

could even allow authors to automatically compose different SEPs together. One could even imagine advanced originators using the tool to directly create fully instantiated SEPs for a single-use, instead of creating parameterized templates.

Second, as discussed in Chapter 4, we think that even with our webform-based instantiation, invoking a new SEP still requires too much upfront work for originators. We could address this problem by providing basic versions of SEPs that provide defaults for almost all parameters, allowing rapid instantiation for common tasks. An additional improvement would be to allow originators to modify parameters while a process is executing. This both eliminates the need for upfront work and simplifies appropriate parameter selection, since the originator can delay this task until a few illustrative responses have arrived. Finally, we expect that integrating tools for launching a new SEP into common email clients, while still enabling participants to respond with any client, would ease SEP instantiation for many users.

Third, making SEPs even easier to use for participants is an important issue. For instance, SEPs currently require participants to respond using a fixed vocabulary. We could support more flexibility by incorporating recent work on schema and ontology mapping [48]. In addition, for responses we chose to use plain text forms for simplicity and interoperability. These forms, however, are sometimes misunderstood by participants. Sending participants a link to an appropriate web form to use for their responses might be a more attractive and reliable technique.

There are also a number of interesting ways to extend the basic model of SEPs. For instance, we identified specific cases where our reasoning is tractable, but how does increasing the expressive power of our constraint and utility language impact the tractability of inference and policy selection? Likewise, we described a model of SEPs with a very simple control flow structure, where the originator asks a single set of questions that are answered by a single set of participants. Clearly, richer models of collaboration are sometimes needed, e.g., to deal with multiple rounds of querying, intermediate decisions, and changing sets of participants (or where the number of participants is not known). Extending SEPs to support such flexibility, while maintaining their understandability and tractability, would be very useful.

Finally, the SEPs we focused on in this work are only one instantiation of semantic email, which far from exhausts its potential. For instance, semantic email could be used to update data sources or to pose/answer general questions, as briefly discussed in Section 4.1. Exploring these other avenues for semantic email seems like a promising direction for future work.

6.2.3 Interaction between MANGROVE *and Semantic Email*

Although MANGROVE and Semantic Email have been described separately, they are actually implemented within the same system. This provides us with a single RDF-based infrastructure for managing data and for potentially integrating email data with web-based data sources and services. Currently, only very basic interactions are performed. For instance, the MANGROVE web calendar accepts event information via email or from a web page. In the future, however, we would like to leverage MANGROVE's data to aid semantic email reasoning. For example, MANGROVE provides an RDF data source about courses, people, etc. that could be used to support the prediction of likely responses by the manager discussed in Section 4.2. Likewise, a semantic email client could utilize data from MANGROVE to answer common questions. When previously unknown questions are answered manually by the user, these responses could be stored for future use, thus enabling the automatic acquisition of semantic knowledge over time. Enabling such interactions is an important area of future work.

6.2.4 Collaborative Ontology Development and Evolution

Just as the web evolved in ways that its creators never anticipated, a Semantic Web system must permit uses never imagined when initially deployed. Thus, in MANGROVE annotators may immediately utilize new MTS tags simply by referencing a web-accessible schema document that they control. Future work, however, is necessary to permit users to declare the relationships between different tags, and to enable other users to discover the existence of new tags (cf., TRELLIS's ontology search techniques [20]). For instance, we intend to encourage schema convergence by maintaining statistics about tag usage, and *promoting*

popular tags to “primary status.” When tagging, users may then choose a schema view with all available tags or, for simplicity, just the primary ones. These techniques permit complete flexibility for users while encouraging schema re-use wherever possible. Related techniques are also needed to enable SEP originators to easily create or select appropriate semantic terms for instantiating a general SEP template, as discussed in Section 4.5.2.

6.2.5 Semantic Web Agents

The vision of the Semantic Web has always encompassed not only the declarative representation of data but also the development of intelligent agents that can consume this data and act upon their owner’s behalf. The combination of MANGROVE and Semantic Email represent a first step in this direction, as knowledge obtained with MANGROVE can be applied towards concrete information-management tasks mediated via email. This is only a first step, however, and there is much potential for agents to exploit other domains and communication mediums. For instance, practical agents are needed that can interact with web services for reservations, purchasing, and querying. Likewise, we could profitably implement semantic agents on other types of computing devices, such as cell phones, PDAs, or even common household appliances [15]. This is clearly a large field with substantial prior work, but there remains significant opportunity to make the application of semantics to these domains practical, perhaps based on extending the general SEP model.

6.2.6 Exploring other Information Sources

We’ve described a set of key design principles designed to help motivate users to structure their data, and deployed two systems that target existing web and email data. However, many other data sources exist. For instance, many users have significant amounts of data in relational databases, spreadsheets, contact lists, text files, bookmarks, etc. that they may be willing to structure and share under appropriate circumstances. In the future, we intend to extend our systems to such data sources and identify motivating applications for the structuring of data in these realms.

6.2.7 User Studies

Finally, we return to the impact of all this work on actual people. Our goal was to enable and motivate non-technical people to participate in the Semantic Web. Both MANGROVE and Semantic Email are fully deployed systems that can be used by such people in a variety of ways, and thus offer the potential to answer tangible questions about their modes of participation. To date, examination of these questions has been limited because the systems have reached a relatively small number of people. Yet, these systems could potentially attract many more users. For instance, with a little publicity and fine-tuning Semantic Email might easily reach thousands of users.

Thus, with a larger user base and some additional logging features installed, both MANGROVE and Semantic Email could enable a number of interesting user studies. For instance, when annotating content for MANGROVE, what quantity and types of annotations are used? Do users repeatedly annotate and publish just to fix errors or does seeing tangible results motivate the annotation of fundamentally new content? What fraction of people utilizing MANGROVE services such as the calendar also contribute content? Likewise, how do people make use of Semantic Email? Do originators exploit a wide range of SEPs, or do a few popular ones constitute the vast majority of use? How many participants are typically involved in a SEP, how quickly do they respond, and how do they react to the manager's interventions? In addition, how likely are previously unknown participants to later become originators themselves? Answering these questions would provide very useful information for the further development of both these systems and the Semantic Web in general.

BIBLIOGRAPHY

[1] http://www.bibserv.org.

[2] Skical. http://www.skical.org/.

[3] Serge Abiteboul, Victor Vianu, Bradley S. Fordham, and Yelena Yesha. Relational transducers for electronic commerce. In *PODS*, 1998.

[4] S. Adali, K. Candan, Y. Papakonstantinou, and V.S. Subrahmanian. Query caching and optimization in distributed mediator systems. In *Proc. of SIGMOD*, pages 137–148, Montreal, Canada, 1996.

[5] Sanjay Agrawal, Surajit Chaudhuri, and Vivek Narasayya. Automated selection of materialized views and indexes in Microsoft SQL Server. In *Proc. of VLDB*, pages 496–505, Cairo, Egypt, 2000.

[6] Eric E. Allen. *A First-Class Approach to Genericity*. PhD thesis, Rice University, Houston, TX, 2003.

[7] A. Ankolenkar, M. Burstein, J. Hobbs, O. Lassila, D. Martin, S. McIlraith, S. Narayanan, M. Paolucci, T. Payne, K. Sycara, and H. Zeng. DAML-S: A semantic markup language for web services. In *Proceedings of the Semantic Web Working Symposium*, pages 411–430, 2001.

[8] Franz Baader and Ulrike Sattler. Description logics with concrete domains and aggregation. In *Proeedings of the European Conference on Artificial Intelligence*, pages 336–340, 1998.

[9] Dave Banks, Steve Cayzer, Ian Dickinson, and Dave Reynolds. The ePerson Snippet Manager: a semantic web application. Number HPL-2002-328, 2002.

[10] Tom Barrett, David Jones, Jun Yuan, John Sawaya, Mike Uschold, Tom Adams, and Deborah Folger. RDF representation of metadata for semantic integration of corporate information resources. In *WWW2002 Workshop on Real World RDF and Semantic Web Applications*, 2002.

[11] Robert Baumgartner, Sergio Flesca, and Georg Gottlob. Visual web information extraction with Lixto. In *VLDB ́01*, 2001.

[12] Sean Bechhofer and Carole Goble. Towards annotation using DAML+OIL. In *K-CAP 2001 Workshop on Knowledge Markup and Semantic Annotation*, 2001.

[13] B. Benatallah, M. Hacid, C. Rey, and F. Toumani. Request rewriting-based web service discovery. In *Second International Semantic Web Conference*, October 2003.

[14] V. Richard Benjamins and Dieter Fensel. Community is knowledge! in (KA)2. In *Eleventh Workshop on Knowledge Acquisition, Modeling and Management*, 1998.

[15] T. Berners-Lee, J. Hendler, and O. Lassila. The Semantic Web. *Scientific American*, May 2001.

[16] Tim Berners-Lee. Notation 3. http://www.w3.org/DesignIssues/Notation3.

[17] Abraham Bernstein and Mark Klein. Towards high-precision service retrieval. In *First International Semantic Web Conference, Sardinia, Italy, June 2002*.

[18] D. Bertsekas. Dynamic programming and optimal control. Athena Scientific, 1995.

[19] Mikhail Bilenko, Raymond Mooney, William Cohen, Pradeep Ravikumar, and Stephen Fienberg. Adaptive name matching in information integration. *IEEE Intelligent Systems Special Issue on Information Integration on the Web*, September 2003.

[20] Jim Blythe and Yolanda Gil. Incremental formalization of document annotations through ontology-based paraphrasing. In *Proc. of the Thirteenth Int. WWW Conference*, 2004.

[21] Blai Bonet and Hector Geffner. Planning with incomplete information as heuristic search in belief space. In *Artificial Intelligence Planning Systems*, pages 52–61, 2000.

[22] Anthony J. Bonner. Workflow, transactions, and datalog. In *PODS*, pages 294–305, 1999.

[23] Craig Boutilier. A POMDP formulation of preference elicitation problems. In *AAAI-02*, pages 239–246, 2002.

[24] Craig Boutilier, Thomas Dean, and Steve Hanks. Decision-theoretic planning: Structural assumptions and computational leverage. *Journal of Artificial Intelligence Research*, 11:1–94, 1999.

[25] Craig Boutilier, Richard Dearden, and Moise's Goldszmidt. Exploiting structure in policy construction. In *Proc. of IJCAI-14*, 1995.

[26] Justin A. Boyan and Michael L. Littman. Exact solutions to time-dependent MDPs. In *Advances in Neural Information Processing Systems 13 (NIPS)*, pages 1026–1032, 2000.

[27] J. Broekstra, A. Kampman, and F. van Harmelen. Sesame: An architecture for storing and querying RDF data and schema information, 2001.

[28] Christoph Bussler, Dieter Fensel, and Alexander Maedche. A conceptual architecture for semantic web enabled web services. *ACM SIGMOD Record*, 31(4):24–29, 2002.

[29] Giuseppe Carenini and Johanna D. Moore. Generating explanations in context. In *Intelligent User Interfaces*, pages 175–182, 1993.

[30] Michael J. Carey and Donald Kossmann. Reducing the braking distance of an SQL query engine. In *Proceedings of the Twenty-fourth International Conference on Very Large Databases*, pages 158–169, New York City, USA, 1998. VLDB Endowment, Saratoga, Calif.

[31] Jeremy Carroll. Matching rdf graphs. In *First International Semantic Web Conference, Sardinia, Italy, June 2002*.

[32] Damianos Chatziantoniou and Kenneth A. Ross. Groupwise processing of relational queries. In *VLDB*, pages 476–485, 1997.

[33] Surajit Chaudhuri, Kris Ganjam, Venkatesh Ganti, and Rajeev Motwani. Robust and efficient fuzzy match for online data cleaning. In *Proc. of SIGMOD*, 2003.

[34] Sudarshan Chawathe, Hector Garcia-Molina, Joachim Hammer, Kelly Ireland, Yannis Papakonstantinou, Jeffrey Ullman, and Jennifer Widom. The TSIMMIS project: Integration of heterogeneous information sources. In proceedings of IPSJ, Tokyo, Japan, October 1994.

[35] L. Chen, N.R. Shadbolt, C. Goble, F. Tao, S.J. Cox, C. Puleston, and P. Smart. Towards a knowledge-based approach to semantic service composition. In *Second International Semantic Web Conference*, October 2003.

[36] Rada Chirkova, Alon Halevy, and Dan Suciu. A formal perspective on the view selection problem. In *Proc. of VLDB*, 2001.

[37] S. Cohen, W. Nutt, and A. Serebrenik. Rewriting aggregate queries using views. In *Proc. of PODS*, pages 155–166, 1999.

[38] William Cohen. Integration of heterogeneous databases without common domains using queries based on textual similarity. In *Proc. of SIGMOD*, Seattle, WA, 1998.

[39] Dan Connolly. A knowledge base about internet mail. http://www.w3.org/2000/04/maillog2rdf/email.html.

[40] DAML Services Coalition. DAML-S and OWL-S. http://www.daml.org/services.

[41] John Davies, Richard Weeks, and Uwe Krohn. QuizRDF: Search technology for the semantic web. In *Workshop on Real World RDF and Semantic Web Applications*, 2002.

[42] Maria Garcia de la Banda, Peter J. Stuckey, and Jeremy Wazny. Finding all minimal unsatisfiable subsets. In *Proceedings of the 5th ACM SIGPLAN international conference on Principles and practice of declarative programming*, pages 32–43. ACM Press, 2003.

[43] M. Dean, D. Connolly, F. van Harmelen, J. Hendler, I. Horrocks, D. McGuinness, P. Patel-Schneider, and L. Stein. OWL web ontology language 1.0 reference, 2002. Manuscript available from `http://www.w3.org/2001/sw/WebOnt/`.

[44] S. Decker, M. Erdmann, D. Fensel, and R. Studer. Ontobroker: Ontology based access to distributed and semi-structured information. In *Eighth Working Conference on Database Semantics (DS-8)*, pages 351–369, 1999.

[45] Nachum Dershowitz. Program abstraction and instantiation. *ACM Trans. Programming. Language Systems*, 7(3):446–477, 1985.

[46] Alin Deutsch, Liying Sui, and Victor Vianu. Specification and verification of data-driven web services. In *PODS*, 2004.

[47] Anhai Doan, Pedro Domingos, and Alon Halevy. Reconciling schemas of disparate data sources: a machine learning approach. In *Proc. of SIGMOD*, 2001.

[48] Anhai Doan, Jayant Madhavan, Pedro Domingos, and Alon Halevy. Learning to map between ontologies on the semantic web. In *Proc. of the Int. WWW Conf.*, 2002.

[49] S. A. Dobson and V. A. Burrill. Lightweight databases. *Computer Networks and ISDN Systems*, 27(6):1009–1015, 1995.

[50] Xin Dong, Alon Halevy, Ema Nemes, Stephan B. Sigurdsson, and Pedro Domingos. SEMEX: toward on-the-fly personal information integration. In *IWEB 2004*, 2004.

[51] Xin Dong, Alon Y. Halevy, Jayant Madhavan, Ema Nemes, and Jun Zhang. Similarity search for web services. In *Proc. of VLDB*, 2004.

[52] Robert B. Doorenbos, Oren Etzioni, and Daniel S. Weld. A scalable comparison-shopping agent for the world-wide web. In *Proceedings of the First International Conference on Autonomous Agents*, 1997.

[53] M. Druzdzel. Qualitative verbal explanations in bayesian belief networks. *Artificial Intelligence and Simulation of Behaviour Quarterly*, 94:43–54, 1996.

[54] Oliver M. Duschka and Michael R. Genesereth. Answering recursive queries using views. In *Proc. of PODS*, pages 109–116, Tucson, Arizona., 1997.

[55] Martin Dzbor, John Domingue, and Enrico Motta. Magpie — towards a semantic web browser. In *Second International Semantic Web Conference*, October 2003.

[56] Oren Etzioni, Alon Halevy, Hank Levy, and Luke McDowell. Semantic email: Adding lightweight data manipulation capabilities to the email habitat. In *Sixth Int. Workshop on the Web and Databases*, 2003.

[57] Ronald Fagin, Joseph Y. Halpern, Yoram Moses, and Moshe Y. Vardi. *Reasoning About Knowledge*. M.I.T Press, 1995.

[58] Alexander Felfernig, Gerhard Friedrich, Dietmar Jannach, and Markus Zanker. Semantic configuration web services in the cawicoms project. In *First International Semantic Web Conference, Sardinia, Italy, June 2002*.

[59] Ivan P. Fellegi and Alan B. Sunter. A theory for record linkage. *Journal of the Americal Statistical Association*, 64(328):1183–1210, December 1969.

[60] Dieter Fensel, Jrgen Angele, Stefan Decker, Michael Erdmann, Hans-Peter Schnurr, Rudi Studer, and Andreas Witt. On2broker: Lessons learned from applying AI to the web. Technical Report Institute AIFB Research report no. 383, 1998.

[61] Dieter Fensel, Jurgen Angele, Stefan Decker, Michael Erdmann, Hans-Peter Schnurr, Steffen Staab, Rudi Studer, and Andreas Witt. On2broker: Semantic-based access to information sources at the WWW. In *WebNet (1)*, pages 366–371, 1999.

[62] Mary Fernandez, Daniela Florescu, Alon Levy, and Dan Suciu. Declarative specification of web sites with strudel. *VLDB Journal*, 9(1):38–55, 2000.

[63] Francisco Ferreira, Daniel Schwabe, and Carlos Lucena. Using semantic web services now. In *IX Brazilian Symp. on Hypermedia and Multimedia*, 2003.

[64] C. Fillies, G. Wood-Albrecht, and F. Weichardt. A pragmatic application of the semantic web using SemTalk. In *WWW*, pages 686–692, 2002.

[65] P. Flener, K.-K. Lau, M. Ornaghi, and J. Richardson. An abstract formalisation of correct schemas for program synthesis. *Journal of Symbolic Computation*, 30(1):93–127, July 2000.

[66] Helena Galhardas, Daniela Florescu, Dennis Shasha, Eric Simon, and Cristian-Augustin Saita. Declarative data cleaning: Language, model, and algorithms. In *VLDB*, pages 371–380, 2001.

[67] H. Garcia-Molina, Y. Papakonstantinou, D. Quass, A. Rajaraman, Y. Sagiv, J. Ullman, and J. Widom. The TSIMMIS project: Integration of heterogeneous information sources. *Journal of Intelligent Information Systems*, 8(2):117–132, March 1997.

[68] Yolanda Gil and Varun Ratnakar. Trusting information sources one citizen at a time. In *First International Semantic Web Conference, Sardinia, Italy, June 2002*.

[69] Jim Gray, Adam Bosworth, Andrew Layman, and Hamid Pirahesh. Data cube: A relational aggregation operator generalizing group-by, cross-tab and sub-totals. In *Proc. of ICDE*, pages 152–159, 1996.

[70] G. AA. Grimnes, S. Chalmers, P. Edwards, and A. Preece. Granitenights - a multi-agent visit scheduler utilising semantic web technology. In *Seventh International Workshop on Cooperative Information Agents*, pages 137–151, 2003.

[71] Stephane Grumbach and Leonardo Tininini. On the content of materialzed aggregate views. In *Proc. of PODS*, 2000.

[72] N. Guarino, C. Masolo, and G. Vetere. Ontoseek: Content-based access to the web, 1999.

[73] R. Guha and Rob McCool. Tap: A semantic web platform. http://tap.stanford.edu/tap/papers.html.

[74] R. Guha, Rob McCool, and Eric Miller. Semantic search. In *World Wide Web*, 2003.

[75] Laura Haas, Donald Kossmann, Edward Wimmers, and Jun Yang. Optimizing queries across diverse data sources. In *Proc. of VLDB*, Athens, Greece, 1997.

[76] A. Halevy, O. Etzioni, A. Doan, Z. Ives, J. Madhavan, L. McDowell, and I. Tatarinov. Crossing the structure chasm. In *First Biennial Conferenece on Innovative Data Systems Research, Asilomar, CA, January 5-8*, 2003.

[77] Alon Y. Halevy, Zachary G. Ives, Peter Mork, and Igor Tatarinov. Piazza: Data management infrastructure for semantic web applications. In *World Wide Web*, 2003.

[78] Siegfried Handschuh and Steffen Staab. Authoring and annotation of web pages in CREAM. In *World Wide Web*, pages 462–473, 2002.

[79] Siegfried Handschuh, Steffen Staab, and Fabio Ciravegna. S-cream - semi-automatic creation of metadata. In *EKAW*, pages 358–372, 2002.

[80] Siegfried Handschuh, Steffen Staab, and Raphael Volz. On deep annotation. In *WWW*, 2003.

[81] S. Haustein and J. Pleumann. Is participation in the semantic web too difficult? In *First International Semantic Web Conference, Sardinia, Italy, June 2002*.

[82] J. Heflin and J. Hendler. A portrait of the semantic web in action. *IEEE Intelligent Systems*, 16(2), 2001.

[83] Jeff Heflin, James Hendler, and Sean Luke. SHOE: A knowledge representation language for internet applications. Technical Report CS-TR-4078, 1999.

[84] Jeff Heflin, James A. Hendler, and Sean Luke. Applying ontology to the web: A case study. In *IWANN (2)*, pages 715–724, 1999.

[85] Andreas Heß and Nicholas Kushmerick. Learning to attach semantic metadata to web services. In *Second International Semantic Web Conference*, October 2003.

[86] I. Horrocks, F. van Harmelen, and P. Patel-Schneider. DAML+OIL. http://www.daml.org/2001/03/daml+oil-index.html, March 2001.

[87] E. J. Horvitz, J. S. Breese, and M. Henrion. Decision theory in expert systems and artificial intelligence. *International Journal of Approximate Reasoning*, 2:247–302, 1988.

[88] Andreas Hotho, Alexander Maedche, Steffen Staab, and Rudi Studer. SEAL-II - the soft spot between richly structured and unstructured knowledge. *Journal of Universal Computer Science*, 7(7):566–590, 2001.

[89] Richard Hull, Michael Benedikt, Vassilis Christophides, and Jianwen Su. E-Services: A look behind the curtain. In *PODS*, 2003.

[90] David Huynh, David Karger, and Dennis Quan. Haystack: A platform for creating, organizing and visualizing information using RDF.

[91] Zachary Ives, Daniela Florescu, Marc Friedman, Alon Levy, and Dan Weld. An adaptive query execution engine for data integration. In *Proc. of SIGMOD*, pages 299–310, 1999.

[92] K. Jensen. Coloured Petri nets: A high-level language for system design and analysis. In G. Rozenberg, editor, *Advances in Petri Nets 1990*, pages 342–416. Springer-Verlag, 1990. (Lecture Notes in Computer Science 483).

[93] David S. Johnson. Approximation algorithms for combinatorial problems. *Journal of Computer and System Sciences*, 9:256–278, 1974.

[94] Ulrich Junker. QUICKXPLAIN: Conflict detection for arbitrary constraint propagation algorithms. In *IJCAI'01 Workshop on Modelling and Solving problems with constraints*, Seattle, WA, USA, August 2001.

[95] Narendra Jussien and Vincent Barichard. The PaLM system: explanation-based constraint programming. In *Proceedings of TRICS: Techniques foR Implementing Constraint programming Systems, a post-conference workshop of CP 2000*, pages 118–133, Singapore, September 2000.

[96] Narendra Jussien and Samir Ouis. User-friendly explanations for constraint programming. In *ICLP 11th Workshop on Logic Programming Environments*, Paphos, Cyprus, Dec. 2001.

[97] Jose Kahan and Marja-Ritta Koivunen. Annotea: an open RDF infrastructure for shared web annotations. In *World Wide Web*, pages 623–632, 2001.

[98] Aditya Kalyanpur, Bijan Parsia, James Hendler, and Jennifer Golbeck. SMORE - semantic markup, ontology, and RDF editor. http://www.mindswap.org/papers/.

[99] George Katsirelos and Fahiem Bacchus. Unrestricted nogood recording in csp search. In *Principles and Practice of Constraint Programming*, October 2003.

[100] Ralph Kimball and Kevin Strehlo. Why decision support fails and how to fix it. *SIGMOD Record*, 24(3):92–97, 1995.

[101] David A. Klein and Edward H. Shortliffe. A framework for explaining decision-theoretic advice. *Artificial Intelligence*, 67(2):201–243, 1994.

[102] Subhash Kumar, Anugeetha Kunjithapatham, Mithun Sheshagiri, Tim Finin, Anupam Joshi, Yun Peng, and R. Scott Cost. A personal agent application for the semantic web. In *AAAI Fall Symposium on Personalized Agents*, 2002.

[103] N. Kushmerick, R. Doorenbos, and D. Weld. Wrapper induction for information extraction. In *Proceedings of the 15th International Joint Conference on Artificial Intelligence*, 1997.

[104] Cody Kwok, Oren Etzioni, and Dan Weld. Scaling question answering to the web. In *Proc. of the Int. WWW Conf.*, pages 150–161, 2001.

[105] editor L. Fischer. *Workflow Handbook 2001, Workflow Management Coalition*. Lighthouse Point, Florida. Future Strategies, 2001.

[106] Eric Lambrecht, Subbarao Kambhampati, and Senthil Gnanaprakasam. Optimizing recursive information gathering plans. In *Proceedings of the 16th International Joint Conference on Artificial Intelligence*, pages 1204–1211, 1999.

[107] O. Lassila and R. Swick. Resource description framework (RDF) model and syntax specification. http://www.w3.org/TR/REC-rdf-syntax/, 1999. W3C Recommendation.

[108] Alon Y. Levy, Inderpal Singh Mumick, and Yehoshua Sagiv. Query optimization by predicate move-around. In *Proc. of VLDB*, pages 96–107, 1994.

[109] Alon Y. Levy, Anand Rajaraman, and Joann J. Ordille. Querying heterogeneous information sources using source descriptions. In *Proc. of VLDB*, pages 251–262, Bombay, India, 1996.

[110] Gangmin Li, Victoria Uren, Enrico Motta, Simon Buckingham Shum, and John Domingue. Claimaker: Weaving a semantic web of research papers. In *First International Semantic Web Conference, Sardinia, Italy, June 2002.*

[111] Thorsten Liebig. The travel agent of the next generation web. In *Poster presentation at the First International Semantic Web Conference*, 2002.

[112] Michael L. Littman. Probabilistic propositional planning: Representations and complexity. In *Proce. of (AAAI-97)*, 1997.

[113] Michael L. Littman, Thomas L. Dean, and Leslie Pack Kaelbling. On the complexity of solving Markov decision problems. In *Proc. of (UAI–95)*, 1995.

[114] Ling Liu, Calton Pu, and Wei Han. XWRAP: An XML-enabled wrapper construction system for web information sources. In *ICDE '00*, pages 611–621, 2000.

[115] Alan K. Mackworth. Constraint-based agents: The ABC's of CBA's. In *Constraint Programming*, pages 1–10, 2000.

[116] Alexander Maedche, Steffen Staab, Rudi Studer, York Sure, and Raphael Volz. SEAL - tying up information integration and web site management by ontologies. *IEEE Data Engineering Bulletin*, 25(1):10–17, 2002.

[117] Thomas Malone, Kenneth Grant, Franklyn Turbak, Stephen Brobst, and Michael Cohen. Intelligent information-sharing systems. *Comm. of the ACM*, 30(5):390–402, 1987.

[118] Ioana Manolescu, Daniela Florescu, and Donald Kossmann. Answering xml queries on heterogeneous data sources. In *Proc. of VLDB*, pages 241–250, 2001.

[119] P. Martin and P. Eklund. Manageable approaches to the semantic web. In *World Wide Web*, 2002.

[120] P. Martin and P. W. Eklund. Embedding knowledge in web documents. *WWW8 / Computer Networks*, 31(11-16):1403–1419, 1999.

[121] P. Martin and P. W. Eklund. Large-scale cooperatively-built KBs. In *ICCS*, pages 231–244, 2001.

[122] m.c. schraefel, Nigel R Shadbolt, Nicholas Gibbins, Hugh Glaser, and Stephen Harris. Cs aktive space: Representing computer science in the semantic web. In *Proc. of the Thirteenth Int. WWW Conference*, 2004.

[123] Brian McBride. Jena: Implementing the RDF model and syntax specification. In *Proceedings of the 2001 Semantic Web Workshop*, 2001.

[124] Andrew K. McCallum and Ben Wellner. Toward conditional models of identity uncertainty with application to proper noun coreference. In *IJCAI Workshop on Information Integration on the Web*, 2003.

[125] Luke McDowell, Oren Etzioni, Steven D. Gribble, Alon Halevy, Henry Levy, William Pentney, Deepak Verma, and Stani Vlasseva. Evolving the semantic web with Mangrove. Technical Report UW-CSE-03-02-01, February 2003.

[126] Luke McDowell, Oren Etzioni, Steven D. Gribble, Alon Halevy, Henry Levy, William Pentney, Deepak Verma, and Stani Vlasseva. Mangrove: Enticing ordinary people onto the semantic web via instant gratification. In *Second International Semantic Web Conference*, October 2003.

[127] Luke McDowell, Oren Etzioni, Alon Halevey, and Hank Levy. Semantic email. In *Proc. of the Thirteenth Int. WWW Conference*, 2004.

[128] Luke McDowell, Oren Etzioni, and Alon Halevy. The specification of agent behavior by ordinary people: A case study. In *Third International Semantic Web Conference*, November 2004.

[129] Luke McDowell, Oren Etzioni, and Alon Halevy. Specifying semantic email processes. In *WWW2004 Workshop on Application Design, Development and Implementation Issues in the Semantic Web*, 2004.

[130] Deborah L. McGuinness and Alexander Borgida. Explaining subsumption in description logics. In *IJCAI (1)*, pages 816–821, 1995.

[131] Deborah L. McGuinness and Paulo Pinheiro da Silva. Infrastructure for web explanations. In *Second International Semantic Web Conference*, October 2003.

[132] S. McIlraith, T. Son, and H. Zeng. Semantic web services. *IEEE Intelligent Systems. Special Issue on the Semantic Web*, 16(2):46–53, March/April 2001.

[133] S. A. McIlraith, T. C. Son, and H. Zeng. Mobilizing the semantic web with daml-enabled web services. In *Proceedings of the 2001 Semantic Web Workshop*, 2001.

[134] Robin Milner. *A Calculus of Communicating Systems*, volume 92 of *Lecture Notes in Computer Science*. Springer, 1980.

[135] C. Mohan. Workflow management in the internet age. www.almaden.ibm.com/u/mohan/workflow.pdf, 1999.

[136] B. Motik, A. Maedche, and R. Volz. A conceptual modeling approach for building semantics-driven enterprise applications. In *First International Conference on Ontologies, Databases and Application of Semantics (ODBASE)*, 2002.

[137] Saikat Mukherjee, Hasan Davulcu, Michael Kifer, Pinar Senkul, and Guizhen Yang. Logic based approaches to workflow modeling and verification. In *Logics for Emerging Applications of Databases*, 2003.

[138] A. Naeve. The concept browser - a new form of knowledge management tool. In *2nd European Web-based Learning Environments Conference (WBLE 2001), Lund, Sweden.*

[139] Alexander Nareyek. *Constraint-Based Agents*, volume 2062 of *Lecture Notes in Computer Science*. Springer, 2001.

[140] Robert Neches, William R. Swartout, and Johanna D. Moore. Explainable (and maintainable) expert systems. In *IJCAI*, pages 382–389, 1985.

[141] Wolfgang Nejdl, Boris Wolf, Changtao Qu, Stefan Decker, Michael Sintek, Ambjrn Naeve, Mikael Nilsson, Matthias Palmr, and Tore Risch. Edutella: a P2P networking infrastructure based on RDF. In *WWW*, pages 604–615, 2002.

[142] Ontoprise. Demo applications. http://www.ontoprise.de/com/co_produ_appl2.htm.

[143] Samir Ouis, Narendra Jussien, and Patrice Boizumault. k-relevant explanations for constraint programming. In *FLAIRS'03: Sixteenth International Florida Artificial Intelligence Research Society Conference*, St. Augustine, Florida, USA, May 2003. AAAI press.

[144] Massimo Paolucci, Takahiro Kawamura, Terry R. Payne, , and Katia Sycara. Semantic matching of web services capabilities. In *First International Semantic Web Conference, Sardinia, Italy, June 2002.*

[145] A. Patil, S. Oundhakar, A. Sheth, and K. Verma. METEOR-S web service annotation framework. In *Proc. of the Thirteenth Int. WWW Conference*, 2004.

[146] Terry Payne, Rahul Singh, and Katia Sycara. Calendar agents on the semantic web. *IEEE Intelligent Systems*, 17(3):84–86, 2002.

[147] Terry R. Payne, Rahul Singh, and Katia P. Sycara. RCal: a case study on semantic web agents. In *AAMAS*, pages 802–803, 2002.

[148] Filip Perich, Lalana Kagal, Harry Chen, Sovrin Tolia, Youyong Zou, Timothy W. Finin, Anupam Joshi, Yun Peng, R. Scott Cost, and Charles Nicholas. ITTALKS: An application of agents in the semantic web. In *ESAW*, pages 175–194, 2001.

[149] Amir Pnueli. The temporal logic of programs. In *Proc. of the 18th Annual IEEE Symposium on Foundations of Computer Science*, 1977.

[150] Martin Puterman. Markov decision processes. Wiley Inter-science, 1994.

[151] Dennis Quan, David Huynh, and David R. Karger. Haystack: A platform for authoring end user semantic web applications. In *Second International Semantic Web Conference*, October 2003.

[152] Dennis Quan and David R. Karger. How to make a semantic web browser. In *Proc. of the Thirteenth Int. WWW Conference*, 2004.

[153] Vijayshankar Raman and Joseph M. Hellerstein. Potter's wheel: An interactive data cleaning system. In *VLDB*, pages 381–390, 2001.

[154] Dave Reynolds. RDF-QBE: a Semantic Web building block. http://www.hpl.hp.com/semweb/publications.htm.

[155] Kenneth Ross, Divesh Srivastava, Peter Stuckey, and S. Sudarshan. Foundations of aggregation constraints. In *Principles and Practice of Constraint Programming.* LNCS, 874. Springer Verlag, 1994.

[156] Arnaud Sahuguet and Fabien Azavant. Building light-weight wrappers for legacy web data-sources using W4F. In *VLDB '99*, pages 738–741, 1999.

[157] Hiroyuki Sato, Yutaka Abe, and Atsushi Kanai. Hyperclip: A tool for gathering and sharing metadata on userś activities by using peer-to-peer technology. In *Workshop on Real World RDF and Semantic Web Applications*, 2002.

[158] Thomas Schiex and Girard Verfaillie. Nogood Recording fot Static and Dynamic Constraint Satisfaction Problems. *International Journal of Artificial Intelligence Tools*, 3(2):187–207, 1994.

[159] H. Schnurr, S. Staab, and R. Studer. Ontology-based process support. In *AAAI99 workshop on Exploring Synergies of Knowledge Management and Case-Based Reasoning*.

[160] Pinar Senkul, Michael Kifer, and Ismail H. Toroslu. A logical framework for scheduling workflows under resource allocation constraints. In *VLDB*, 2002.

[161] William M. Shaw Jr., Robert Burgin, and P. Howell. Performance standards and evaluations in IR test collections: Cluster-based retrieval models. *Information Processing and Management*, 33(1):1–14, 1997.

[162] A. Sheth, C. Bertram, D. Avant, B. Hammond, K. Kochut, and Y. Warke. Semantic content management for enterprises and the web. *IEEE Internet Computing*, July/August 2002.

[163] Bernd Simon, Zoltan Miklos, Wolfgang Nejdl, Michael Sintek, and Joaquin Salvachua. Elena: A mediation infrastructure for educational services. In *World Wide Web*, 2003.

[164] Evren Sirin, James Hendler, and Bijan Parsia. Semi-automatic composition of web services using semantic descriptions. In *ICEIS2003 workshop on Web Services: Modeling, Architecture and Infrastructure*, 2003.

[165] Stephen Soderland. Learning to extract text-based information from the World Wide Web. In *Knowledge Discovery and Data Mining*, pages 251–254, 1997.

[166] M. Solanki, A. Cau, and H. Zedan. Augmenting semantic web service description with compositional specification. In *Proc. of the Thirteenth Int. WWW Conference*, 2004.

[167] Steffen Staab, Jrgen Angele, Stefan Decker, Michael Erdmann, Andreas Hotho, Alexander Maedche, Hans-Peter Schnurr, Rudi Studer, and York Sure. Semantic community web portals. *WWW9 / Computer Networks*, 33(1-6):473–491, 2000.

[168] Larry J. Stockmeyer and Ashok K. Chandra. Provably difficult combinatorial games. *SIAM Journal on Computing*, 8(2):151–174, 1979.

[169] W. Swartout, C. Paris, and J. Moore. Design for explainable expert systems. *IEEE Expert*, 6(3):58–647, 1991.

[170] V. Tamma, M. Wooldridge, and I. Dickinson. An ontology-based approach to automated negotiation. In *AMEC-2002*, 2002.

[171] David Trastour, Claudio Bartolini, and Chris Preist. Semantic web support for the business-to-business e-commerce lifecycle. In *WWW*, pages 89–98, 2002.

[172] Olga De Troyer, Jo De Greef, and Peter Stuer. Web-for-web: A tool for evolving data-driven web applications.

[173] W.M.P. van der Aalst, A.H.M. ter Hofstede, B. Kiepuszewski, and A.P. Barros. Workflow patterns. *Distributed and Parallel Databases*, 14(3):5–51, July 2003.

[174] M. Vargas-Vera, E. Motta, J. Domingue, S. Buckingham Shum, and M. Lanzoni. Knowledge extraction by using an ontology-based annotation tool. In *K-CAP 2001 workshop on Knowledge Markup and Semantic Annotation, Victoria*, 2001.

[175] Raphael Volz, Daniel Oberle, Steffen Staab, and Boris Motik. Kaon server - a semantic web management system. In *Alternate Track of WWW2003*. ACM, May 2003.

[176] W3C. RDF calendar workspace. http://www.w3.org/2002/12/cal/.

[177] W3C. W3C issues recommendation for resource description framework (RDF). W3C press release 24 February 1999. http://www.w3.org/Press/1999/RDF-REC.

[178] W3C. Web-ontology (webont) working group. http://www.w3.org/2001/sw/WebOnt/.

[179] W3C. XML digital signatures activity statement. http://www.w3.org/Signature/Activity.html, 2002.

[180] Daniel Weld and Oren Etzioni. The first law of robotics (a call to arms). In *Proc. of AAAI*, 1994.

[181] Dell Zhang and Wee Sun Lee. Web taxonomy integration using support vector machines. In *Proceedings of the 13th conference on World Wide Web*, pages 472–481. ACM Press, 2004.

Appendix A

MANGROVE SCHEMA

Below we describe the schema used for annotating HTML documents in MANGROVE. This schema is formatted as an XML DTD for simplicity, and contains embedded comments that is used to automatically generate both HTML documentation for the schema and a definition file that is used to present the schema in the graphical annotation tool. The comments used for the HTML generation are XML comments that precede each XML `ELEMENT`. For instance, the `workAddress` element has the documentation "Work mailing address."

```
<!-- ******************************************** -->
<!--           FACULTY MEMBER ELEMENT           -->
<!-- ******************************************** -->

<!-- facultyMember Annotates semantic information about a faculty member. -->

<!ELEMENT facultyMember (#PCDATA | name | portrait | jobTitle | university | department |
             workAddress | office | workPhone | fax | workEmail | workHomepage |
             assistant |
             homeAddress | homePhone | cellphone | pager | personalEmail | personalHomepage |
             researchInterests | project |
             advisedStudent | publication | officeHours | value | ignore)*>

<!-- name Name of a person, project, course, etc.-->
<!ELEMENT name (#PCDATA | value | ignore)*>

<!-- portrait Person's photo. -->
<!ELEMENT portrait (#PCDATA | value | ignore)*>

<!-- jobTitle Job title (position) in the department (e.g., associate professor, research assistant professor, ect.) or in a company.  -->
<!ELEMENT jobTitle (#PCDATA | value | ignore)*>

<!-- university University name.-->
<!ELEMENT university (#PCDATA | value | ignore)*>

<!-- department Department name. -->
<!ELEMENT department (#PCDATA | value | ignore)*>

<!-- workAddress Work mailing address. -->
<!ELEMENT workAddress (#PCDATA | value | ignore)*>

<!-- homeAddress Home mailing address. -->
<!ELEMENT homeAddress (#PCDATA | value | ignore)*>

<!-- office Office number. -->
<!ELEMENT office (#PCDATA | value | ignore)*>
```

```
<!-- workPhone Work phone number. -->
<!ELEMENT workPhone (#PCDATA | value | ignore)*>

<!-- homePhone Home phone number. -->
<!ELEMENT homePhone (#PCDATA | value | ignore)*>

<!-- fax Fax number. -->
<!ELEMENT fax (#PCDATA | value | ignore)*>

<!-- workEmail Work (professional) e-mail address. -->
<!ELEMENT workEmail (#PCDATA | value | ignore)*>

<!-- personalEmail Personal e-mail address. -->
<!ELEMENT personalEmail (#PCDATA | value | ignore)*>

<!-- workHomepage Work (professional) homepage URL. -->
<!ELEMENT workHomepage (#PCDATA | value | ignore)*>

<!-- personalHomepage Personal homepage URL. -->
<!ELEMENT personalHomepage (#PCDATA | value | ignore)*>

<!-- assistant Staff assistant. -->
<!ELEMENT assistant (#PCDATA | name | portrait | workEmail | workPhone | workHomepage | office | value | ignore)*>

<!-- cellphone Cell phone number. -->
<!ELEMENT cellphone (#PCDATA | value | ignore)*>

<!-- pager Pager number. -->
<!ELEMENT pager (#PCDATA | value | ignore)*>

<!-- researchInterests Research interests. -->
<!ELEMENT researchInterests (#PCDATA | value | ignore)*>

<!-- project Research project. -->
<!ELEMENT project (#PCDATA | name | homepage | summary | participant | publication | value | ignore)*>

<!-- homepage Homepage URL; for person's homepage please see workHomepage and personalHomepage. -->
<!ELEMENT homepage (#PCDATA | value | ignore)*>

<!-- participant Project participant. -->
<!ELEMENT participant (#PCDATA | name | portrait | jobTitle | organization | university | department |
             workAddress | office | workPhone | fax | workEmail | workHomepage |
             homeAddress | homePhone | cellphone | pager | personalEmail | personalHomepage |
             researchInterests | project | publication |
             yearOfStudy | undergradUniversity |
             degreeCompleted | degreeGoal | programStatus |
             advisor |  thesisTopic |
             birthday | sportsPlayed | hobbies | otherInterests |
             assistant |
             advisedStudent | officeHours | value | ignore)*>

<!-- summary Summary. -->
<!ELEMENT summary (#PCDATA | value | ignore)*>

<!-- advisedStudent Student advised by that faculty member. -->
<!ELEMENT advisedStudent (#PCDATA | name | portrait | university | department |
              workAddress | office | workPhone | fax | workEmail | workHomepage |
              homeAddress | homePhone | cellphone | pager | personalEmail | personalHomepage |
              yearOfStudy | undergradUniversity |
              degreeCompleted | degreeGoal | programStatus |
              researchInterests | advisor | project | thesisTopic |
              publication | birthday | sportsPlayed | hobbies | otherInterests |
              jobTitle | organization | value | ignore)*>

<!-- publication Publication (e.g., TR, paper, etc.). -->
<!ELEMENT publication (#PCDATA | author | publicationTitle | forumName | forumType |
```

```
            forumLocation | year | publisher | file | value | ignore)*>

<!-- author Author of a publication. -->
<!ELEMENT author (#PCDATA | value | ignore)*>

<!-- publicationTitle Title of a publication. -->
<!ELEMENT publicationTitle (#PCDATA | value | ignore)*>

<!-- forumName Forum name. -->
<!ELEMENT forumName (#PCDATA | value | ignore)*>

<!-- forumType Forum type, e.g., conference, workshop, etc. -->
<!ELEMENT forumType (#PCDATA | value | ignore)*>

<!-- forumLocation Forum location. -->
<!ELEMENT forumLocation (#PCDATA | value | ignore)*>

<!-- file  Link to the file containing that publication/paper. -->
<!ELEMENT file (#PCDATA | value | ignore)*>

<!-- publisher Publisher name. -->
<!ELEMENT publisher (#PCDATA | value | ignore)*>

<!-- officeHours Office hours (location and/or time). -->
<!ELEMENT officeHours (#PCDATA | value | ignore)*>

<!-- value Represents the value of the tag. -->
<!ELEMENT value (#PCDATA | value | ignore)*>

<!-- ignore Explicitly annotates that that piece of data should be ignored by the applications. -->
<!ELEMENT ignore (#PCDATA | value | ignore)*>

<!-- ******************************************** -->
<!--           GRAD STUDENT ELEMENT             -->
<!-- ******************************************** -->

<!-- gradStudent Annotates semantic information about a graduate student. -->
<!ELEMENT gradStudent (#PCDATA | name | portrait | university | department |
            workAddress | office | workPhone | fax | workEmail | workHomepage |
            homeAddress | homePhone | cellphone | pager | personalEmail | personalHomepage |
            yearOfStudy | undergradUniversity |
            degreeCompleted | degreeGoal | programStatus |
            researchInterests | advisor | project | thesisTopic |
            publication | birthday | sportsPlayed | hobbies | otherInterests |
            jobTitle | organization | value | ignore)*>

<!-- yearOfStudy Year of study (e.g., first year grad student). -->
<!ELEMENT yearOfStudy (#PCDATA | value | ignore)*>

<!-- undergradUniversity The name of the university where the student earned his/her undergrad degree. -->
<!ELEMENT undergradUniversity (#PCDATA | value | ignore)*>

<!-- degreeCompleted Degree completed. -->
<!ELEMENT degreeCompleted (#PCDATA | value | ignore)*>

<!-- degreeGoal Degree pursued. -->
<!ELEMENT degreeGoal (#PCDATA | value | ignore)*>

<!-- programStatus Student status, i.e., Pre-Quals, Post-Generals, Ph.D. -->
<!ELEMENT programStatus (#PCDATA | value | ignore)*>

<!-- advisor Student's advisor. -->
<!ELEMENT advisor (#PCDATA | name | portrait | jobTitle | university | department |
           workAddress | office | workPhone | fax | workEmail | workHomepage |
           assistant |
           homeAddress | homePhone | cellphone | pager | personalEmail | personalHomepage |
```

```
          researchInterests |
          project | advisedStudent | publication | value | ignore)*>

<!-- thesisTopic Topic of the thesis. -->
<!ELEMENT thesisTopic (#PCDATA | value | ignore)*>

<!-- birthday Birthday (month and day). -->
<!ELEMENT birthday (#PCDATA | value | ignore)*>

<!-- sportsPlayed Sports played. -->
<!ELEMENT sportsPlayed (#PCDATA | value | ignore)*>

<!-- hobbies Hobbies. -->
<!ELEMENT hobbies (#PCDATA | value | ignore)*>

<!-- otherInterests Other interests. -->
<!ELEMENT otherInterests (#PCDATA | value | ignore)*>

<!-- organization Current affiliation, e.g., company name. -->
<!ELEMENT organization (#PCDATA | value | ignore)*>

<!-- ******************************************** -->
<!--            COURSE ELEMENT                   -->
<!-- ******************************************** -->

<!-- course Annotates semantic information about a course or seminar (in a course web page). -->
<!ELEMENT course (#PCDATA |
          courseCode | name | schoolQuarter | year |
          description | instructor | teachingAssistant | credits | textbook |
          startDate | endDate | time | location | event | value | ignore)*>

<!-- courseCode Course code, e.g., "cse590s". -->
<!ELEMENT courseCode (#PCDATA | value | ignore)*>

<!-- schoolQuarter Quarter when the seminar/course is offered. -->
<!ELEMENT schoolQuarter (#PCDATA | value | ignore)*>

<!-- year Year. -->
<!ELEMENT year (#PCDATA | value | ignore)*>

<!-- description Short description of the course. -->
<!ELEMENT description (#PCDATA | value | ignore)*>

<!-- instructor Instructor of a course/seminar. -->
<!ELEMENT instructor (#PCDATA | name | portrait | jobTitle | university | department |
              workAddress | office | workPhone | fax | workEmail | workHomepage |
              assistant |
              homeAddress | homePhone | cellphone | pager | personalEmail | personalHomepage |
              researchInterests | project |
              advisedStudent | publication | officeHours | value | ignore)*>

<!-- teachingAssistant Teaching Assistant. -->
<!ELEMENT teachingAssistant (#PCDATA | name | portrait | university | department |
                 section | officeHours |
                 workAddress | office | workPhone | fax | workEmail | workHomepage |
                 homeAddress | homePhone | cellphone | pager | personalEmail | personalHomepage |
                 yearOfStudy | undergradUniversity |
                 degreeCompleted | degreeGoal | programStatus |
                 researchInterests | advisor | project | thesisTopic |
                 publication | birthday | sportsPlayed | hobbies | otherInterests |
                 jobTitle | organization |
                     value | ignore)*>

<!-- section Which section the teaching assistant is teaching. -->
<!ELEMENT section (#PCDATA | value | ignore)*>
```

```
<!-- credits Course credits. -->
<!ELEMENT credits (#PCDATA | value | ignore)*>

<!-- textbook Course textbook. -->
<!ELEMENT textbook (#PCDATA | bookTitle | author | edition |
            publisher | year | value | ignore)*>

<!-- bookTitle Title of a book.-->
<!ELEMENT bookTitle (#PCDATA | value | ignore)*>

<!-- edition Edition of a book. -->
<!ELEMENT edition (#PCDATA | value | ignore)*>

<!-- time When the course meets (e.g., "W 4:30 - 5:20", "10:30 a.m."). -->
<!ELEMENT time (#PCDATA | value | ignore)*>

<!-- location Where the course/seminar meets (e.g., "EE1 031"). -->
<!ELEMENT location (#PCDATA | value | ignore)*>

<!-- event Represents a lecture, a seminar or other event. -->
<!ELEMENT event (#PCDATA | date | startDate | endDate | time | location |
          typeLecture | typeSection | presenter | topic | paper | slides | readings |
          remarks | value | ignore)*>

<!-- date Date (e.g., "Jul 21, 2002", "10/13/2002"). -->
<!ELEMENT date (#PCDATA | value | ignore)*>

<!-- startDate Start date for a reoccurring event. -->
<!ELEMENT startDate (#PCDATA | value | ignore)*>

<!-- endDate End date for a reoccurring event. -->
<!ELEMENT endDate (#PCDATA | value | ignore)*>

<!-- typeLecture Standalone tag classifying a given event as a 'lecture'. -->
<!ELEMENT typeLecture (#PCDATA | value | ignore)*>

<!-- typeSection Standalone tag classifying a given event as a 'section'. -->
<!ELEMENT typeSection (#PCDATA | value | ignore)*>

<!-- presenter Presenter's name. -->
<!ELEMENT presenter (#PCDATA | value | ignore)*>

<!-- topic Topic of a presentation, a lecture, a talk. -->
<!ELEMENT topic (#PCDATA | value | ignore)*>

<!-- paper Paper. -->
<!ELEMENT paper (#PCDATA | author | paperTitle | forumName | forumType |
        forumLocation | year | publisher | file | value | ignore)*>

<!-- paperTitle Title of a paper. -->
<!ELEMENT paperTitle (#PCDATA | value | ignore)*>

<!-- slides Lecture/seminar slides. -->
<!ELEMENT slides (#PCDATA | value | ignore)*>

<!-- readings Lecture/seminar readings. -->
<!ELEMENT readings (#PCDATA | value | ignore)*>

<!-- remarks Additional notes and remarks. -->
<!ELEMENT remarks (#PCDATA | value | ignore)*>

<!-- ******************************************* -->
<!--           STAFF MEMBER ELEMENT             -->
<!-- ******************************************* -->

<!-- staffMember Annotates semantic information about a staff member. -->
```

```
<!ELEMENT staffMember (#PCDATA | name | portrait | jobTitle | university | department |
             workAddress | office | workPhone | fax | workEmail | workHomepage |
             homeAddress | homePhone | cellphone | pager | personalEmail | personalHomepage |
             project | publication |
             value | ignore)*>

<!-- ******************************************** -->
<!--        UNDERGRAD STUDENT ELEMENT             -->
<!-- ******************************************** -->

<!-- undergradStudent Annotates semantic information about a undergraduate student. -->
<!ELEMENT undergradStudent (#PCDATA | name | portrait | university | department |
            workAddress | workPhone | workEmail | workHomepage |
            homeAddress | homePhone | cellphone | pager | personalEmail | personalHomepage |
            yearOfStudy | degreeCompleted | degreeGoal |
            researchInterests | advisor | project | publication |
            birthday | sportsPlayed | hobbies | otherInterests |
            value | ignore)*>
```

Appendix B

SEMANTIC EMAIL DECLARATIONS AND TEMPLATES

This appendix provides some technical details on the language used to represent SEP declarations and templates. First, Section B.1 gives an overview of how a completed SEP declaration is interpreted by the manager for execution. Next, Sections B.2 and B.3 define the ontology used for specifying a SEP template and SEP parameter declaration, respectively.

B.1 Interpretation of SEP *Declarations*

In this section we assume the manager has a been given a complete SEP declaration δ, and explain how the manager executes this declaration. The declaration is used for three primary purposes: 1.) sending the initial messages to the participants, 2.) processing responses received from the participants, and 3.) handling notifications. Below we first describe some general rules that apply to all of these situations, then explain each of these three cases in more detail. Note that this section describes the error-free execution of a SEP — see the proof of Theorem 5.4.2 for details on the template checking that is needed to ensure this occurs.

General rules: We make use of the following recursive definition:

Definition B.1.1 (**forward-reachable**) A statement s is *forward-reachable* from a resource r iff the subject of s is r, or the subject of s is *forward-reachable* from r. □

Intuitively, a statement s is forward-reachable from r if s can be found by starting from r and traversing statements in the subject to object direction.

A SEP declaration uses RDF resources to represent key elements such as questions, goals, and notifications, as well as lower-level primitives such as variable definitions. Every such resource r is processed using the following steps:

1. Evaluate any global defines. Using these defines, recursively substitute their values into any variable references (e.g., `$NumExpected$`) that are found in any statement that is forward-reachable from r.

2. Evaluate any `guard` statements whose subject is r. If the object of any such statement evaluates to false, stop (i.e., proceed as if this resource did not exist).

3. Evaluate the `forAll` statement whose subject is r. (If there is no such statement, proceed as if there were such a statement with a single possibility that defined no relevant variables). For each possibility defined by this quantification(s), repeat the following steps:

   (a) Evaluate the variables defined by the `define` statement whose subject is r, if any. Then use the resultant definitions, along with any variables defined by the current `forAll` instantiation, to again substitute variable references in any statement that is forward-reachable from r.

   (b) Evaluate any `suchThat` properties whose subject is r. If the object of any such statement evaluates to false, skip to Step (d).

   (c) Otherwise, perform the specific action indicated by r with this possibility. If r has any statements pointing to additional resources that need to be evaluated first, then execute this entire procedure beginning with Step 2 for each such resource r'.

   (d) After completely processing the quantification, revert any statements that were changed in Step (a) back to their unmodified state.

4. After completing processing this resource, revert any statements that were modified in Step 1 back to their original state.

In practice, instead of reverting statements, our implementation makes shadow copies of every statement before modifying them with variable substitutions, then discards the modified statements when they are no longer needed.

Sending initial messages to the participants: When a SEP is initially invoked, the manager parses the declaration and extracts the originator, participants, and initial prompt. The manager uses the prompt to compose a tentative message to the participants. To aid later interpretation of the responses, the manager also assigns a unique SEP identifier to the new process and includes this identifier in the message. The manager then parses the list of questions. For each question, the manager adds text to the message, asking the participant for the value desired by the question and providing a text form (with multiple choice options for enumerated questions) for the participant to use in their response. If a question is quantified (e.g., to ask for yes/no responses to a series of options that is provided by the originator), then the manager repeats this process for each option. Finally, the manager combines the RDQL queries from each question into a single RDQL query that is attached to the bottom of the message. The entire message is then sent to the participants.

Processing responses from the participants: When a message is received from a participant, the manager first uses the SEP identifier included in the message to identify the appropriate SEP and retrieve its declaration. The message's semantic response is extracted from the message using the associated RDQL query, and the resultant RDF is tentatively stored in the manager's RDF database.

The manager then evaluates each SEP goal in turn. For an L-SEP, if the constraint was previously satisfiable, but can no longer be satisfied by the new state of the database, then the response must be rejected. It is removed from the database and a message is sent to the participant notifying them of the rejection, possibly accompanied by an explanation associated with the goal. For a D-SEP, the message is always accepted. However, the manager will now evaluate the optimal policy choice given the current state of the database (this policy is calculated once for all possible states when the first response is received, then cached for later use). Based on this policy, the manager may choose to send a suggestion to the participant, asking them to change their response. In our current semantic email implementation, such suggestions are only sent immediately following a response by a participant.

Handling notifications: The manager processes notifications whenever a new response is handled (as described above) and periodically, to check for notifications triggered by a `OnDateTime` condition. If a response has just been accepted, then the data set used to compute values for the notification will include the new data. For each notification in the SEP's notification list, the manager checks to see if the corresponding condition has been satisfied. If so, the manager computes the message requested by the notification and sends it to the requested recipients. Alternatively, if the notification states that the `ProcessSummary` should be notified, then the manager uses the result to update a cached document that summarizes the responses received by a SEP along with any such SEP-specific notifications. The text of a notification may also make use of pre-defined variables (e.g., $Bringing.acceptable()$) that reasons about what responses are acceptable given the SEP's goals and current responses.

B.2 Ontology for Describing SEP *Templates*

At the highest level, a SEP template (of type `SemanticEmailProcess` in the ontology below) specifies a `title` of the process, a textual `summary` of what it does, a set of `parameters` to be used for instantiation (described in the next section), and a formal `definition`. The latter definition specifies the `originator` and `participants`, a `prompt` to send with the initial message, and RDF lists of `questions`, `goals`, and `notifications`. The complete ontology is given below, in N3 format.

Where appropriate, the ontology identifies cardinality restrictions. For instance, a `TradeoffGoal` must have exactly one `optimize` property (specified via a `owl:cardinality` property). In addition, because `TradeoffGoal` is a sub-class of `Goal`, it must have at most one `message` property (specified via a `owl:maxCardinality` property). Note, however, many such properties (e.g., `forAll, define`) point to a RDF list of resources, thus permitting an arbitrary number of quantifications, definitions, etc. per resource.

```
@prefix a:       <http://www.cs.washington.edu/research/semweb/semantic_email#> .
@prefix rdfs:    <http://www.w3.org/2000/01/rdf-schema#> .
@prefix rdf:     <http://www.w3.org/1999/02/22-rdf-syntax-ns#> .
@prefix :        <#> .
@prefix owl:     <http://www.w3.org/2002/07/owl#> .
```

```
a:OnConditionSatisfiedFirstTime
      a       owl:Class ;
      rdfs:comment """Fires when the given condition is satisifed, but only if
this is the first such occurrence in the life of the process.""" ;
      rdfs:subClassOf a:Notification ;
      owl:equivalentClass
              [ a       owl:Class ;
                owl:intersectionOf ([ a       owl:Restriction ;
                            owl:cardinality "1"^^<http://www.w3.org/2000/10/XMLSchema#nonNegativeInteger> ;
                            owl:onProperty a:condition
                          ])
              ] .

a:parameters
      a       owl:ObjectProperty ;
      rdfs:comment """Points to a list of parameter descriptions that the form
generator will use to create a form for instantiating
this process.""" ;
      rdfs:domain a:SemanticEmailProcess ;
      rdfs:range a:ParameterList .

a:value
      a       owl:DatatypeProperty ;
      rdfs:comment """Defines the object to be used a variable
assignment.""" ;
      rdfs:domain a:DefineNode .

a:notifications
      a       owl:ObjectProperty ;
      rdfs:comment "Points to a list of notifications for this process" ;
      rdfs:domain a:SemanticEmailProcessDefinition .

a:guard
      a       owl:DatatypeProperty ;
      rdfs:comment """This property is evaluated first when a new resource is encountered,
before any new variables are defined.  If it evaluates to non-zero,
then this resource is processed, otherwise it is ignored.""" ;
      rdfs:domain a:Question , a:DefineNode , a:Notification , a:EvaluateNode , a:Goal .

a:DefineNode
      a       owl:Class ;
      owl:equivalentClass
              [ a       owl:Class ;
                owl:intersectionOf ([ a       owl:Restriction ;
                            owl:cardinality "1"^^<http://www.w3.org/2000/10/XMLSchema#nonNegativeInteger> ;
                            owl:onProperty a:name
                          ] [ a       owl:Restriction ;
                            owl:maxCardinality "1"^^<http://www.w3.org/2000/10/XMLSchema#nonNegativeInteger> ;
                            owl:onProperty a:forAll
                          ])
              ] ;
      owl:equivalentClass
              [ a       owl:Class ;
                owl:unionOf ([ a       owl:Restriction ;
                            owl:cardinality "1"^^<http://www.w3.org/2000/10/XMLSchema#nonNegativeInteger> ;
                            owl:onProperty a:product
                          ] [ a       owl:Restriction ;
                            owl:cardinality "1"^^<http://www.w3.org/2000/10/XMLSchema#nonNegativeInteger> ;
                            owl:onProperty a:value
                          ] [ a       owl:Restriction ;
                            owl:cardinality "1"^^<http://www.w3.org/2000/10/XMLSchema#nonNegativeInteger> ;
                            owl:onProperty a:string
                          ] [ a       owl:Restriction ;
                            owl:cardinality "1"^^<http://www.w3.org/2000/10/XMLSchema#nonNegativeInteger> ;
                            owl:onProperty a:min
                          ] [ a       owl:Restriction ;
```

```
                        owl:cardinality "1"^^<http://www.w3.org/2000/10/XMLSchema#nonNegativeInteger> ;
                        owl:onProperty a:max
                      ] [ a       owl:Restriction ;
                        owl:cardinality "1"^^<http://www.w3.org/2000/10/XMLSchema#nonNegativeInteger> ;
                        owl:onProperty a:sum
                      ])
              ] .

a:Goal
      a       owl:Class ;
      rdfs:comment "A goal to be pursued by the process" ;
      owl:equivalentClass
              [ a       owl:Class ;
                owl:intersectionOf ([ a       owl:Restriction ;
                        owl:maxCardinality "1"^^<http://www.w3.org/2000/10/XMLSchema#nonNegativeInteger> ;
                        owl:onProperty a:define
                      ] [ a       owl:Restriction ;
                        owl:maxCardinality "1"^^<http://www.w3.org/2000/10/XMLSchema#nonNegativeInteger> ;
                        owl:onProperty a:forAll
                      ] [ a       owl:Restriction ;
                        owl:maxCardinality "1"^^<http://www.w3.org/2000/10/XMLSchema#nonNegativeInteger> ;
                        owl:onProperty a:message
                      ])
              ] .

a:forAll
      a       owl:ObjectProperty ;
      rdfs:comment """Points to a list of variable quantifications.  The system will
evaluate all possible combinations of each that are consistent
with the 'suchThat' properties, if any are present.""" ;
      rdfs:domain a:Question , a:DefineNode , a:Notification , a:EvaluateNode , a:Goal .

a:intervalSeconds
      a       owl:DatatypeProperty ;
      rdfs:comment "Specifies the number of seconds between repeats" ;
      rdfs:domain a:RepeatNode .

a:sum
      a       owl:ObjectProperty ;
      rdfs:comment """Operator in define statement.  Points to a resource,
will evaluate all possible quantifications for that resource and
return the sum of all of them""" ;
      rdfs:domain a:DefineNode ;
      rdfs:range a:EvaluateNode .

a:title
      a       owl:DatatypeProperty ;
      rdfs:comment "Title of the SEP" ;
      rdfs:domain a:SemanticEmailProcess .

a:enumeration
      a       owl:DatatypeProperty ;
      rdfs:comment """Restricts the legal answers to this question to be one of the string
identified in this expression, which follows the set syntax.""" ;
      rdfs:domain a:StringQuestion .

a:OnAllResponsesReceived
      a       owl:Class ;
      rdfs:comment "Fires once when the expected number of responses have been received" ;
      rdfs:subClassOf a:Notification .

a:MustConstraint
      a       owl:Class ;
      rdfs:comment "A constraint to be enforced so that every outcome of the process is guaranteed to satisify this constraint." ;
      rdfs:subClassOf a:Constraint .
```

```
a:untilDateTime
      a       owl:DatatypeProperty ;
      rdfs:comment """Specifies the time after which a reminder will not repeat.
In the same format as a 'dateTime' property""" ;
      rdfs:domain a:RepeatNode .

a:ProbabiliitiesSimple
      a       owl:Class ;
      rdfs:comment "Basic mechanism for expressing probabilities associated with a TradeoffGoal" ;
      rdfs:subClassOf a:ProbabilitiesNode .

a:costs
      a       owl:ObjectProperty ;
      rdfs:comment "Points to a list of cost information for a Tradeoff goal." ;
      rdfs:domain a:TradeoffGoal .

a:enforce
      a       owl:DatatypeProperty ;
      rdfs:domain a:Constraint .

a:SemanticEmailProcessDefinition
      a       owl:Class ;
      owl:equivalentClass
              [ a       owl:Class ;
                owl:intersectionOf ([ a       owl:Restriction ;
                            owl:maxCardinality "1"^^<http://www.w3.org/2000/10/XMLSchema#nonNegativeInteger> ;
                            owl:onProperty a:questions
                          ] [ a       owl:Restriction ;
                            owl:cardinality "1"^^<http://www.w3.org/2000/10/XMLSchema#nonNegativeInteger> ;
                            owl:onProperty a:subject
                          ] [ a       owl:Restriction ;
                            owl:cardinality "1"^^<http://www.w3.org/2000/10/XMLSchema#nonNegativeInteger> ;
                            owl:onProperty a:prompt
                          ] [ a       owl:Restriction ;
                            owl:cardinality "1"^^<http://www.w3.org/2000/10/XMLSchema#nonNegativeInteger> ;
                            owl:onProperty a:participants
                          ] [ a       owl:Restriction ;
                            owl:maxCardinality "1"^^<http://www.w3.org/2000/10/XMLSchema#nonNegativeInteger> ;
                            owl:onProperty a:notifications
                          ] [ a       owl:Restriction ;
                            owl:cardinality "1"^^<http://www.w3.org/2000/10/XMLSchema#nonNegativeInteger> ;
                            owl:onProperty a:originator
                          ] [ a       owl:Restriction ;
                            owl:maxCardinality "1"^^<http://www.w3.org/2000/10/XMLSchema#nonNegativeInteger> ;
                            owl:onProperty a:goals
                          ])
              ] .

a:ProbabilitiesNode
      a       owl:Class ;
      rdfs:comment """Abstract class for describing the probabilities governing
expected participant behavior in a TradeoffGoal.""" ;
      owl:equivalentClass
              [ a       owl:Class ;
                owl:intersectionOf ([ a       owl:Restriction ;
                            owl:maxCardinality "1"^^<http://www.w3.org/2000/10/XMLSchema#nonNegativeInteger> ;
                            owl:onProperty a:define
                          ])
              ] .

a:define
      a       owl:ObjectProperty ;
      rdfs:comment "Points to a list of definitions for later use." ;
      rdfs:domain a:ProbabilitiesNode , a:Notification , a:CostsNode , a:EvaluateNode , a:Goal .

a:DefineList
```

```
      a       owl:Class ;
      rdfs:comment "RDF list of define nodes" .

a:string
      a       owl:DatatypeProperty ;
      rdfs:comment """Defines a literal string to be used as the object
in a variable definition or an output statement.""" ;
      rdfs:domain a:DefineNode .

a:OnMessageReceived
      a       owl:Class ;
      rdfs:comment "Fires every time a message is received" ;
      rdfs:subClassOf a:Notification .

a:min
      a       owl:ObjectProperty ;
      rdfs:comment """Operator in define statement.  Points to a resource,
will evaluate all possible quantifications for that resource and
return the minimum one.""" ;
      rdfs:domain a:DefineNode ;
      rdfs:range a:EvaluateNode .

a:False
      a       owl:Thing .

a:product
      a       owl:ObjectProperty ;
      rdfs:comment """Operator in define statement.  Points to a resource,
will evaluate all possible quantifications for that resource and
return the product of all of them.""" ;
      rdfs:domain a:DefineNode ;
      rdfs:range a:EvaluateNode .

a:OnMessageAccepted
      a       owl:Class ;
      rdfs:comment "Fires every time a message is accepted" ;
      rdfs:subClassOf a:Notification .

a:ParameterList
      a       owl:Class ;
      rdfs:comment """RDF list of parameter nodes.
These nodes are given in a separate ontology:
http://www.cs.washington.edu/research/semweb/params#""" .

a:Boolean
      a       owl:Class ;
      owl:oneOf (a:True a:False) .

a:prompt
      a       owl:DatatypeProperty ;
      rdfs:domain a:SemanticEmailProcessDefinition .

a:TradeoffGoal
      a       owl:Class ;
      rdfs:comment """A goal to pursue some utility function, subject to given costs
of making suggestions.""" ;
      rdfs:subClassOf a:Goal ;
      owl:equivalentClass
              [ a       owl:Class ;
                owl:intersectionOf ([ a       owl:Restriction ;
                            owl:cardinality "1"^^<http://www.w3.org/2000/10/XMLSchema#nonNegativeInteger> ;
                            owl:onProperty a:probabiliities
                          ] [ a       owl:Restriction ;
                            owl:cardinality "1"^^<http://www.w3.org/2000/10/XMLSchema#nonNegativeInteger> ;
                            owl:onProperty a:costs
                          ] [ a       owl:Restriction ;
```

```
                            owl:cardinality "1"^^<http://www.w3.org/2000/10/XMLSchema#nonNegativeInteger> ;
                            owl:onProperty a:optimize
                          ])
              ] .

a:dateTime
      a       owl:DatatypeProperty ;
      rdfs:comment """String representing date and time in ISO 8601 format.
e.g., 2004-07-16T19:20:30+01:00""" ;
      rdfs:domain a:OnDateTime .

a:minInclusive
      a       owl:DatatypeProperty ;
      rdfs:comment "Minimum legal value for the response to a question" ;
      rdfs:domain a:DoubleQuestion , a:IntegerQuestion .

a:originator
      a       owl:DatatypeProperty ;
      rdfs:domain a:SemanticEmailProcessDefinition .

a:condition
      a       owl:DatatypeProperty ;
      rdfs:comment "Specifies a condition on which a notification depends" ;
      rdfs:domain a:OnConditionSatisfiedAnyTime , a:OnConditionSatisfiedFirstTime , a:OnConditionSatisfied .

a:participants
      a       owl:DatatypeProperty ;
      rdfs:domain a:SemanticEmailProcessDefinition .

a:BooleanQuestion
      a       owl:Class ;
      rdfs:comment "Question to ask the participants, of type boolean" ;
      rdfs:subClassOf a:Question .

a:repeat
      a       owl:ObjectProperty ;
      rdfs:comment """Points to a resource describing how often a OnDateTime notification
should repeat""" ;
      rdfs:domain a:OnDateTime ;
      rdfs:range a:RepeatNode .

a:Constraint
      a       owl:Class ;
      rdfs:comment "A constraint type of goal" ;
      rdfs:subClassOf a:Goal ;
      owl:equivalentClass
              [ a       owl:Class ;
                owl:intersectionOf ()
              ] .

a:OnMessageRejected
      a       owl:Class ;
      rdfs:comment "Fires every time a message is rejected" ;
      rdfs:subClassOf a:Notification .

a:query
      a       owl:DatatypeProperty ;
      rdfs:comment """A RDQL query specifying the semantic interpretation of the results
of this question.""" ;
      rdfs:domain a:Question .

a:IntegerQuestion
      a       owl:Class ;
      rdfs:comment "Question to ask the participants, of type integer" ;
      rdfs:subClassOf a:Question ;
      owl:equivalentClass
```

```
              [ a       owl:Class ;
                owl:intersectionOf ([ a       owl:Restriction ;
                            owl:maxCardinality "1"^^<http://www.w3.org/2000/10/XMLSchema#nonNegativeInteger> ;
                            owl:onProperty a:maxInclusive
                          ] [ a       owl:Restriction ;
                            owl:maxCardinality "1"^^<http://www.w3.org/2000/10/XMLSchema#nonNegativeInteger> ;
                            owl:onProperty a:minInclusive
                          ])
              ] .

a:name
      a       owl:DatatypeProperty ;
      rdfs:comment "Variable name for a question or a define node" ;
      rdfs:domain a:Question , a:DefineNode .

a:probFreeChoice
      a       owl:DatatypeProperty ;
      rdfs:comment """Points to expression representing probability of participant
initially responding with some choice that is not bound by the goal
(e.g., Not Coming)""" ;
      rdfs:domain a:ProbabiliitiesSimple .

a:SemanticEmailProcess
      a       owl:Class ;
      owl:equivalentClass
              [ a       owl:Class ;
                owl:intersectionOf ([ a       owl:Restriction ;
                            owl:cardinality "1"^^<http://www.w3.org/2000/10/XMLSchema#nonNegativeInteger> ;
                            owl:onProperty a:definition
                          ])
              ] .

a:suchThat
      a       owl:DatatypeProperty ;
      rdfs:comment """Like a guard statement, the subject resource is evalauted only if all 'suchThat'
properties evaluate to non-zero.  However, this kind of property is evaluated after
new variables are defined for the current resource.""" ;
      rdfs:domain a:Question , a:DefineNode , a:Notification , a:EvaluateNode , a:Goal .

a:probSwitchRefuse
      a       owl:DatatypeProperty ;
      rdfs:comment """Expression representing probability that participant will
refuse to switch their response when asked.""" ;
      rdfs:domain a:ProbabiliitiesSimple .

a:OnConditionSatisfied
      a       owl:Class ;
      rdfs:comment """Fires when the given condition (usually based on a query of the current state)
is satisfied, but was *not* true in the previous state.""" ;
      rdfs:subClassOf a:Notification ;
      owl:equivalentClass
              [ a       owl:Class ;
                owl:intersectionOf ([ a       owl:Restriction ;
                            owl:cardinality "1"^^<http://www.w3.org/2000/10/XMLSchema#nonNegativeInteger> ;
                            owl:onProperty a:condition
                          ])
              ] .

a:OnDateTime
      a       owl:Class ;
      rdfs:comment "Fires once when the current time equals the specifed time." ;
      rdfs:subClassOf a:Notification ;
      owl:equivalentClass
              [ a       owl:Class ;
                owl:intersectionOf ([ a       owl:Restriction ;
                            owl:cardinality "1"^^<http://www.w3.org/2000/10/XMLSchema#nonNegativeInteger> ;
```

```
                            owl:onProperty a:dateTime
                          ] [ a       owl:Restriction ;
                            owl:maxCardinality "1"^^<http://www.w3.org/2000/10/XMLSchema#nonNegativeInteger> ;
                            owl:onProperty a:repeat
                          ])
              ] .

a:Responders
      a       owl:Thing .

a:PossiblyConstraint
      a       owl:Class ;
      rdfs:comment """Constraint enforced as follows: if after accepting a message it is still possible for
the constraints to be satisfied after all messages have been received, then it is acceptable
with respect to this constraint.""" ;
      rdfs:subClassOf a:Constraint .

a:costSwitchRequest
      a       owl:DatatypeProperty ;
      rdfs:comment """Points to literal expression representing the cost of asking a
participant to switch their response to something else.""" ;
      rdfs:domain a:CostsSimple .

a:maxInclusive
      a       owl:DatatypeProperty ;
      rdfs:comment "Maximum legal value for the response to a question" ;
      rdfs:domain a:DoubleQuestion , a:IntegerQuestion .

a:probSwitchFree
      a       owl:DatatypeProperty ;
      rdfs:comment """Expression representing probability that, when asked
to switch, participant switches to a free choice
(e.g., Not Coming)""" ;
      rdfs:domain a:ProbabiliitiesSimple .

a:True
      a       owl:Thing .

a:EvaluateNode
      a       owl:Class ;
      rdfs:comment """A resource that should specifies some value with
an 'evaluate' property, possibly assisted by quantifications
and variable definitions.""" ;
      owl:equivalentClass
              [ a       owl:Class ;
                owl:intersectionOf ([ a       owl:Restriction ;
                            owl:maxCardinality "1"^^<http://www.w3.org/2000/10/XMLSchema#nonNegativeInteger> ;
                            owl:onProperty a:forAll
                          ] [ a       owl:Restriction ;
                            owl:cardinality "1"^^<http://www.w3.org/2000/10/XMLSchema#nonNegativeInteger> ;
                            owl:onProperty a:evaluate
                          ] [ a       owl:Restriction ;
                            owl:maxCardinality "1"^^<http://www.w3.org/2000/10/XMLSchema#nonNegativeInteger> ;
                            owl:onProperty a:define
                          ])
              ] .

a:definition
      a       owl:ObjectProperty ;
      rdfs:domain a:SemanticEmailProcess ;
      rdfs:range a:SemanticEmailProcessDefinition .

a:probBoundChoice
      a       owl:DatatypeProperty ;
      rdfs:comment """Points to expression representing probability of a participant
initially responding with a choice that is constrained by the goal.
```

```
If both are specified, this value must be (1 - probFreeChoice)""" ;
      rdfs:domain a:ProbabiliitiesSimple .

a:evaluate
      a       owl:DatatypeProperty ;
      rdfs:comment """Points to an expression to be evaluated and then
used as the value of this property's subject.""" ;
      rdfs:domain a:EvaluateNode .

<http://www.cs.washington.edu/research/semweb/semantic_email>
      a       owl:Ontology .

a:max
      a       owl:ObjectProperty ;
      rdfs:comment """Operator in define statement.  Points to a resource,
will evaluate all possible quantifications for that resource and
return the maximum one.""" ;
      rdfs:domain a:DefineNode ;
      rdfs:range a:EvaluateNode .

a:Question
      a       owl:Class ;
      rdfs:comment "An abstract question to ask the participants" ;
      owl:equivalentClass
              [ a       owl:Class ;
                owl:intersectionOf ([ a       owl:Restriction ;
                            owl:maxCardinality "1"^^<http://www.w3.org/2000/10/XMLSchema#nonNegativeInteger> ;
                            owl:onProperty a:forAll
                          ] [ a       owl:Restriction ;
                            owl:maxCardinality "1"^^<http://www.w3.org/2000/10/XMLSchema#nonNegativeInteger> ;
                            owl:onProperty a:query
                          ] [ a       owl:Restriction ;
                            owl:cardinality "1"^^<http://www.w3.org/2000/10/XMLSchema#nonNegativeInteger> ;
                            owl:onProperty a:name
                          ])
              ] .

a:AllParticipants
      a       owl:Thing .

a:StringQuestion
      a       owl:Class ;
      rdfs:comment "Question to ask the participants, of type string" ;
      rdfs:subClassOf a:Question ;
      owl:equivalentClass
              [ a       owl:Class ;
                owl:intersectionOf ([ a       owl:Restriction ;
                            owl:maxCardinality "1"^^<http://www.w3.org/2000/10/XMLSchema#nonNegativeInteger> ;
                            owl:onProperty a:freeChoices
                          ] [ a       owl:Restriction ;
                            owl:maxCardinality "1"^^<http://www.w3.org/2000/10/XMLSchema#nonNegativeInteger> ;
                            owl:onProperty a:freeChoices
                          ])
              ] .

a:CostsSimple
      a       owl:Class ;
      rdfs:comment "Basic mechanism for expressing costs for a TradeoffGoal" ;
      rdfs:subClassOf a:CostsNode ;
      owl:equivalentClass
              [ a       owl:Class ;
                owl:intersectionOf ([ a       owl:Restriction ;
                            owl:cardinality "1"^^<http://www.w3.org/2000/10/XMLSchema#nonNegativeInteger> ;
                            owl:onProperty a:costSwitchRequest
                          ])
              ] .
```

```
a:message
      a       owl:DatatypeProperty ;
      rdfs:comment """A string message to send to a participant in case this goal
causes their message to be rejected/suggested against.""" ;
      rdfs:domain a:Notification , a:Goal .

a:OnConditionSatisfiedAnyTime
      a       owl:Class ;
      rdfs:comment """Fires when the given condition is satisfied, after any change in the state
of the process.""" ;
      rdfs:subClassOf a:Notification ;
      owl:equivalentClass
              [ a       owl:Class ;
                owl:intersectionOf ([ a       owl:Restriction ;
                            owl:cardinality "1"^^<http://www.w3.org/2000/10/XMLSchema#nonNegativeInteger> ;
                            owl:onProperty a:condition
                          ])
              ] .

a:summary
      a       owl:DatatypeProperty ;
      rdfs:comment "Overall summary of the SEP" ;
      rdfs:domain a:SemanticEmailProcess .

a:notify
      a       owl:DatatypeProperty ;
      rdfs:comment "Specifies how to send a notification to when it is triggered." ;
      rdfs:domain a:Notification .

a:includeForm
      a       owl:ObjectProperty ;
      rdfs:comment """If true, specifies that the notification should include a form
for creating responses in the notification.  Useful for
reminder-like messages.""" ;
      rdfs:domain a:Notification ;
      rdfs:range a:Boolean .

a:questions
      a       owl:ObjectProperty ;
      rdfs:comment "List of questions to ask the participants" ;
      rdfs:domain a:SemanticEmailProcessDefinition .

a:goals
      a       owl:ObjectProperty ;
      rdfs:comment "Points to a list of goals to pursue" ;
      rdfs:domain a:SemanticEmailProcessDefinition .

a:optimize
      a       owl:DatatypeProperty ;
      rdfs:comment "Points to an expression representing the utility function for a TradeOff goal." ;
      rdfs:domain a:TradeoffGoal .

a:freeChoices
      a       owl:DatatypeProperty ;
      rdfs:comment """Specifies that the elements in this expression (a set) are choices
that must always be accepted (e.g., 'Not Coming').  This is
used in conjunction with 'enumeration'""" ;
      rdfs:domain a:StringQuestion .

a:Originator
      a       owl:Thing .

a:Notification
      a       owl:Class ;
      rdfs:comment "A notification to be triggered when some condition is satisfied." ;
```

```
      owl:equivalentClass
              [ a       owl:Class ;
                owl:intersectionOf ([ a       owl:Restriction ;
                            owl:cardinality "1"^^<http://www.w3.org/2000/10/XMLSchema#nonNegativeInteger> ;
                            owl:onProperty a:notify
                          ] [ a       owl:Restriction ;
                            owl:maxCardinality "1"^^<http://www.w3.org/2000/10/XMLSchema#nonNegativeInteger> ;
                            owl:onProperty a:forAll
                          ] [ a       owl:Restriction ;
                            owl:cardinality "1"^^<http://www.w3.org/2000/10/XMLSchema#nonNegativeInteger> ;
                            owl:onProperty a:message
                          ] [ a       owl:Restriction ;
                            owl:cardinality "1"^^<http://www.w3.org/2000/10/XMLSchema#nonNegativeInteger> ;
                            owl:onProperty a:define
                          ])
              ] .

a:RepeatNode
      a       owl:Class ;
      rdfs:comment "Gives info about how often and until when a reminder should repeat." ;
      owl:equivalentClass
              [ a       owl:Class ;
                owl:intersectionOf ([ a       owl:Restriction ;
                            owl:cardinality "1"^^<http://www.w3.org/2000/10/XMLSchema#nonNegativeInteger> ;
                            owl:onProperty a:untilDateTime
                          ] [ a       owl:Restriction ;
                            owl:cardinality "1"^^<http://www.w3.org/2000/10/XMLSchema#nonNegativeInteger> ;
                            owl:onProperty a:intervalSeconds
                          ])
              ] .

a:NotificationTarget
      a       owl:Class ;
      rdfs:comment "A possible target for a notification" , "The possible target of a notification" ;
      owl:oneOf (a:AllParticipants a:Originator a:NonResponders a:Responders) .

a:probabiliities
      a       owl:ObjectProperty ;
      rdfs:comment """Points to a list of probability information for a TradeoffGoal.
These encode expected participant behavior.""" ;
      rdfs:domain a:TradeoffGoal .

a:CostsNode
      a       owl:Class ;
      rdfs:comment """Abstract class for describing costs associated with
the actions of a TradeoffGoal.""" ;
      owl:equivalentClass
              [ a       owl:Class ;
                owl:intersectionOf ([ a       owl:Restriction ;
                            owl:maxCardinality "1"^^<http://www.w3.org/2000/10/XMLSchema#nonNegativeInteger> ;
                            owl:onProperty a:define
                          ])
              ] .

a:DoubleQuestion
      a       owl:Class ;
      rdfs:comment "Question to ask the participants, of type double" ;
      rdfs:subClassOf a:Question ;
      owl:equivalentClass
              [ a       owl:Class ;
                owl:intersectionOf ([ a       owl:Restriction ;
                            owl:maxCardinality "1"^^<http://www.w3.org/2000/10/XMLSchema#nonNegativeInteger> ;
                            owl:onProperty a:maxInclusive
                          ] [ a       owl:Restriction ;
                            owl:maxCardinality "1"^^<http://www.w3.org/2000/10/XMLSchema#nonNegativeInteger> ;
                            owl:onProperty a:minInclusive
```

```
                    ])
          ] .

a:subject
      a       owl:DatatypeProperty ;
      rdfs:domain a:SemanticEmailProcessDefinition .

a:NonResponders
      a       owl:Thing .
```

B.3 *Ontology for Describing* SEP *Parameter Descriptions*

A SEP parameter description (of type `ParameterList` in the ontology below) consists of a RDF list of `ParameterNode`s. `ParameterNode`s are either a `Description` node (providing some explanatory text), or are a subclass of `InteractiveParameterNode`. The latter specify a parameter whose value should be collected from the originator; each must specify a `name` for that parameter and optionally a `prompt` and a `default` value.

In our implementation, a SEP template identifies its associated parameter description via a `parameter` property that points to a `ParameterList`. Thus, the form generator can process a single URL that contains both the template and the parameter description.

The form generator creates a form by processing the `ParameterList` one at a time, generating HTML that places each `ParameterNode` in sequence. The author can place some control over this layout with `horizSubelementGroup` and `vertSubelementGroup` properties; these specify another `ParameterList` of elements that should be grouped together. If these properties are placed inside a `ChoiceNode` element (e.g., if a choice is offered between "strict" and "flexible" constraint enforcement), then any subelements are required to be filled in only if that particular choice is selected by the originator. The complete ontology is given below.

```
@prefix a:       <http://www.cs.washington.edu/research/semweb/params#> .
@prefix rdfs:    <http://www.w3.org/2000/01/rdf-schema#> .
@prefix b:       <http://www.cs.washington.edu/research/semweb/semantic_email#> .
@prefix rdf:     <http://www.w3.org/1999/02/22-rdf-syntax-ns#> .
@prefix :        <#> .
@prefix owl:     <http://www.w3.org/2002/07/owl#> .

a:showallChoices
      a       owl:ObjectProperty ;
      rdfs:comment """Applies where there is an enumeration property.  If true,
the form will show all choices at the same time, otherwise
a drop-down may be used.""" ;
      rdfs:domain a:InteractiveParameterNode ;
```

```
    rdfs:range b:Boolean .

a:ChoiceList
    a       owl:Class ;
    rdfs:comment "RDF list of enumerated choices for a parameter value." .

a:enumeration
    a       owl:ObjectProperty ;
    rdfs:comment "Points to a RDF list specifying choices for this parameter." ;
    rdfs:domain a:InteractiveParameterNode ;
    rdfs:range a:ChoiceList .

a:name
    a       owl:DatatypeProperty ;
    rdfs:comment "Variable name for this property" ;
    rdfs:domain a:InteractiveParameterNode .

a:TypeInteger
    a       owl:Class ;
    rdfs:subClassOf a:InteractiveParameterNode .

a:TypeEmail
    a       owl:Class ;
    rdfs:subClassOf a:InteractiveParameterNode .

a:content
    a       owl:DatatypeProperty ;
    rdfs:comment "The text to display" ;
    rdfs:domain a:Description .

b:ParameterList
    a       owl:Class .

a:TypeDouble
    a       owl:Class ;
    rdfs:subClassOf a:InteractiveParameterNode .

a:ParameterNode
    a       owl:Class .

a:minInclusive
    a       owl:DatatypeProperty ;
    rdfs:comment "Restriction on the legal values of an entered value." ;
    rdfs:domain a:TypeDouble , a:TypeInteger .

b:Boolean
    a       owl:Class .

a:ChoiceNode
    a       owl:Class ;
    rdfs:comment "A possible enumerated choice for a parameter." .

a:TypeString
    a       owl:Class ;
    rdfs:subClassOf a:InteractiveParameterNode .

a:prompt
    a       owl:DatatypeProperty ;
    rdfs:domain a:ChoiceNode , a:InteractiveParameterNode .

a:value
    a       owl:DatatypeProperty ;
    rdfs:comment "The value of an enumerated choice" ;
    rdfs:domain a:ChoiceNode .

a:InteractiveParameterNode
```

```
      a       owl:Class ;
      rdfs:comment "A node that the originator could enter a value for." ;
      rdfs:subClassOf a:ParameterNode .

a:TypeEmailSet
      a       owl:Class ;
      rdfs:comment "Gets a comma-separated set of email addresses." ;
      rdfs:subClassOf a:InteractiveParameterNode .

a:default
      a       owl:DatatypeProperty ;
      rdfs:comment "The default value for a parameter" ;
      rdfs:domain a:ChoiceNode , a:InteractiveParameterNode .

a:TypeStringSet
      a       owl:Class ;
      rdfs:subClassOf a:InteractiveParameterNode .

a:optional
      a       owl:ObjectProperty ;
      rdfs:comment """Explicit specification of whether a value is required for
this parameter.""" ;
      rdfs:domain a:InteractiveParameterNode ;
      rdfs:range b:Boolean .

a:TypeBoolean
      a       owl:Class ;
      rdfs:subClassOf a:InteractiveParameterNode .

a:vertSubelementGroup
      a       owl:ObjectProperty ;
      rdfs:comment """Points to a RDF list of parameter node that should be
grouped together vertically.""" ;
      rdfs:domain a:ParameterNode , a:ChoiceNode ;
      rdfs:range b:ParameterList .

a:maxInclusive
      a       owl:DatatypeProperty ;
      rdfs:comment "Restriction on the legal values of an entered value." ;
      rdfs:domain a:TypeDouble , a:TypeInteger .

a:horizSubelementGroup
      a       owl:ObjectProperty ;
      rdfs:comment """Points to a RDF list of parameter node that should be
grouped together horizontally.""" ;
      rdfs:domain a:ParameterNode , a:ChoiceNode , a:InteractiveParameterNode ;
      rdfs:range b:ParameterList .

a:Description
      a       owl:Class ;
      rdfs:comment "A static text field" ;
      rdfs:subClassOf a:ParameterNode .

a:subsetOf
      a       owl:DatatypeProperty ;
      rdfs:comment """Restriction on the legal values of an entered value.
Here, the value entered must be a set that is a subset of the
object of this statement.""" ;
      rdfs:domain a:TypeStringSet , a:TypeEmailSet .

<http://www.cs.washington.edu/research/semweb/params>
      a       owl:Ontology .
```

Appendix C

PROOFS

This appendix provides more details on the proofs for each of this dissertation's theorems, presented in the order they appear in the body. Throughout, we assume that a SEP has N participants. For the logical model, we assume that an L-SEP Λ has a current state D, constraints C_D, and that C_D refers to at most some constant H number of attributes. For the decision-theoretic model, we assume that each participant in a D-SEP δ will eventually send an original response, then only sends further messages if they receive a suggestion (which they will also eventually respond to). For convenience, we define the following notation: OptPolicy(δ) is the problem of determining the optimal policy $\pi^\star$ for a given D-SEP δ. OptUtility(δ,θ) is the problem of determining if the expected total utility of $\pi^\star$ for a given D-SEP δ exceeds some constant θ.

C.1 Proof of Theorem 4.3.1

We first show that ultimate satisfiability is NP-complete in the general case. Then the next section shows how this problem can be solved in polynomial time when the constraints are either domain-bounded or constant-bounded.

NP-complete for arbitrary constraints: First, observe that ultimate satisfiability is in NP – given an L-SEP Λ and a response r, we can guess a possible outcome of D that is consistent with r, then verify that the outcome satisfies the constraints.

Second, we show that ultimate satisfiability is NP-hard via a reduction from 3-SAT. Assume we are given a boolean formula ϕ of the form $\phi = L_1 \wedge L_2 \wedge ... \wedge L_m$ where $L_i = (w_{i1} \vee w_{i2} \vee w_{i3})$ for $1 \leq i \leq m$, and each w_{ij} equals some variable x_k or $\overline{x_k}$ for $1 \leq k \leq n$. The 3-SAT problem is to determine if ϕ is satisfiable for some assignment to the variables of w.

Given ϕ, we construct an L-SEP Λ where:

- Participants $P = \{p_0, p_1, p_2, ..., p_n\}$
- Data set D is a single table with one attribute value
- Responses $R = \{nil, r_1, r_2, ...r_n\}$
- Constraints $C_D = \phi$, with v_{ij} substituted for each w_{ij} where
  if $w_{ij} = x_k$, then we set $v_{ij} = [(\text{SELECT COUNT(*) WHERE value} = r_k) > 0]$
  otherwise $w_{ij} = \overline{x_k}$, and $v_{ij} = [(\text{SELECT COUNT(*) WHERE value} = r_k) = 0]$

This construction is polynomial in the size of ϕ. In the resulting Λ, there are $n+1$ participants that may each respond with one of $n+1$ values.

Given this constructed Λ, we now show that the 3-SAT formula ϕ is satisfiable iff an initially empty D for Λ is ultimately satisfiable w.r.t. C_D given response nil. First, given an assignment $x_1, ..., x_n$ that satisfies ϕ, a final state of D that satisfies C_D is as follows: p_0 responds nil, p_k responds r_k if x_k is true, otherwise p_k responds nil. This will set the corresponding x_k's in C_D to true, and since ϕ is satisfied, C_D will be satisfied in the resultant state, demonstrating that D is ultimately satisfiable given an initial response nil. Alternatively, if D is ultimately satisfiable given initial response nil, we can take a final state of D that satisfies C_D and construct an assignment $x_1, ..., x_k$ that satisfies ϕ as follows: if any participant has responded with value r_k, then x_k is true; otherwise, x_k is false. Thus, any 3-SAT problem with N variables can be solved by reduction to ultimate satisfiability with $N+1$ participants. Since 3-SAT is NP-complete in N, ultimate-satisfiability must be NP-hard in N.

C.2 Proof of Theorem 4.3.2

Polynomial-time when constraints are *domain-bounded*: In this case, the constraints refer to attributes whose domain size is at most some constant L. Since there are most H attributes, there are thus a total of LH possible responses.

We evaluate the constraints over D', a data set that distinguishes only representative

states that are different with respect to the constraints. In particular, all that matters for D' is the number of each type of response that has been received (i.e., aggregates of the responses). The number of possible states of D' is thus the number of ways of dividing N participants among $LH + 1$ possible responses (LH choices plus a "no response" option):

$$|D'| = \binom{N + LH}{LH} = O(N^{LH})$$

To determine ultimate satisfiability of D given r, we construct a data set D_r that is D augmented with the given response r. We then iterate over all possible values d of D'. For each value, if d is inconsistent with D_r (i.e., for some response type R_i, D_r shows more such responses than d does), we discard d. Otherwise, we evaluate C_D over d – this requires time linear in N and $|C_D|$ given a particular d. Given this procedure, D is ultimately satisfiable for r iff some d is consistent with D_r and satisfies C_D. Each step requires linear time, and there are a polynomial number of iterations ($O(N^{LH})$), so the total time is polynomial in N and $|C_D|$.

Polynomial-time when constraints are *constant-bounded*: This case uses a similar algorithm as when the constraints are *domain-bounded*. However, since each attribute may have a potentially infinite domain, we must keep track of the possible states differently. Here, we allow only COUNT aggregations, which may be of the form: COUNT(*) WHERE value $= v_i$ or an inequality like COUNT(*) WHERE value $> v_i$.

If C_D is constant-bounded, then there are most K constants $v_1, ..., v_k$ used in these aggregations. These constants divide the domain of each attribute into at most $K + 1$ regions. Thus, there are $K + 1$ possibilities for each of the H attributes of a response, yielding a total of $O(KH)$ possible responses. As with the analysis above, the number of possible states in the representative data set D' is thus $O(N^{KH})$, and the time to evaluate each state is linear. Since H and K are assumed to be constants, then the total time to check ultimate satisfiability is polynomial in both N and $|C_D|$.

C.3 *Proof of Theorem 4.4.1 – bounded suggestions*

For this first case, we assume that the manager can send at most some constant L messages to each participant. Below we prove that in this case OPTUTILITY(δ,θ) is PSPACE-complete, then use this result to prove that OPTPOLICY(δ) is PSPACE-hard.

OptUtility(δ,θ) is PSPACE-complete: First, we show that OPTUTILITY(δ,θ) is in PSPACE. Given δ, consider the tree representing all possible executions, where the root of the tree is the initial state and each leaf represents a possible halted state. From any state in the tree, the next state may result either from the manager making a suggestion or from receiving a response from some participant. Hence, the branching factor of the tree is $O(N)$. In addition, since the manager may make at most LN suggestions and each participant may send up to $L+1$ responses, the tree is acyclic and has total height $O(LN)$. Consequently, we can determine the expected utility of the optimal policy via a suitable depth-first search of the tree. Since the utility of a child node can be discarded once the expected utility of its parent is known, the total space needed is just $O(LN)$. Thus, OPTUTILITY(δ,θ) is in PSPACE.

Second, we show that OPTUTILITY(δ,θ) is PSPACE-hard by a reduction from QBF (quantified boolean formula). A QBF problem specifies a formula φ of the form:

$$\varphi = \exists x_1 \forall y_1 ... \exists x_k \forall y_k \ \phi$$

where ϕ is a 3-CNF boolean formula over the x_i's and y_i's. The computational problem is to determine if φ is true.

Given φ, we construct a corresponding D-SEP δ as follows:

- **Participants:** $P = \{A_1, ..., A_k, B_1, ..., B_k\}$, for a total of $N = 2k$ participants.

- **States:** A state $s = (a_1, ..., a_k, b_1, ..., b_k)$ where the a_i's and b_i's indicate each participant's current response ($True, False$, or $NoneYet$). The a_i's and b_i's correspond directly to the x_i's and y_i's in the formula ϕ. Thus, we say "ϕ is satisfied in s" if no a_i or b_i has the value $NoneYet$ and evaluating ϕ by substituting corresponding values for the x_i's and y_i's yields true.

- **Values:** $V = \{True, False\}$.

- **Actions:** $A = \{NoOp, Halt, SW_{p,true}, SW_{p,false}\}$, where $p \in P$.

- **Transitions:** We construct $T()$ so that the following steps will occur in order:

  1. **Choice:** In the initial state the manager may either perform $NoOp$ (to wait for responses) or $Halt$ (if it has no winning strategy).
  2. **A-Turn:** A_1 sends a $False$ response. The manager may choose either to execute $NoOp$ (thus accepting $a_1 = False$) or to suggest a change to A_1, in which case A_1 immediately agrees (so $a_1 = True$).
  3. **B-Turn:** The manager performs $NoOp$, and receives an random original response (either $True$ or $False$) from B_1. B_1 refuses any suggestions.
  4. **Repeat:** Repeat A-Turn and B-Turn for (A_2, B_2) ... (A_k, B_k), then $Halt$.

- **Utilities:** the only non-zero utilities are as follows:

$$\begin{aligned} U(s_0, Halt) &= 1 \quad (quitting\ from\ the\ initial\ state) \\ U(s, Halt) &= 1 + \epsilon \quad if\ s \neq s_0 \ \ and\ \phi(s) = True \end{aligned}$$

  where ϵ represents an infinitesimally small, positive value. Note that this use of ϵ does not introduce any serious computational difficulties. The expected utility of each state may be maintained in the form $(c + d\epsilon)$ – addition, multiplication, and comparison (over a total order) are easily defined for such values. In addition, since ϵ appears only in the utility function, higher-order values such as ϵ^2 do not arise.

The size of this D-SEP is polynomial in N and the whole reduction can be done in polynomial time. In particular, while an explicit representation of the transition and utility functions for every possible state would be exponential in N, the rules above allow all of the necessary functionality to be encoded concisely in terms of the current responses. For instance, the utility function representing one possibility for a B-turn (where b_i changes from $NoneYet$ to $True$) is:

$$
\begin{aligned}
T(\quad s \quad &, NoOp, s') = 0.5 \ where \\
s &= \{a_1, ..., a_i, None_{i+1}, ..., None_k, b_1, ..., b_{i-1}, \mathbf{None_i}, None_{i+1} ..., None_k\} \\
s' &= \{a_1, ..., a_i, None_{i+1}, ..., None_k, b_1, ..., b_{i-1}, \mathbf{True}, None_{i+1}, ..., None_k\}
\end{aligned}
$$

Note also that in several steps above we made statements like "The manager performs action $NoOp$," when really at each step the manager has a choice to make. However, since we can construct the transition function in any desired fashion, we can "force" the manager into any needed behavior by setting the transition probability for executing any other action to zero. The same control over the probabilities permits us to ensure that participants behave in certain ways and that messages arrive in a certain order.

We now demonstrate an additional result needed to complete the proof:

Definition C.3.1 (**guaranteed satisfying policy**) Given a D-SEP δ constructed from φ as above, a guaranteed satisfying policy is a policy that, if followed by the manager, guarantees that the SEP will terminate in a state that satisfies ϕ. □

Claim: A guaranteed satisfying policy for δ exists iff the expected utility of the optimal policy $\pi^\star$ for δ is greater than 1 (e.g., OptUtility(δ,$\theta = 1$) is true).

Proof: Clearly, the expected utility of a guaranteed satisfying policy for δ is $1 + \epsilon$, so any optimal policy must have utility at least this large, which is greater than 1. In the other direction, by examining the utility function we see that the only way for $\pi^\star$ to obtain a utility greater than 1 is for the SEP to halt with ϕ satisfied, yielding reward $1 + \epsilon$. If this outcome occurs with any probability $P_\phi < 1$ for $\pi^\star$, then the total expected utility will be less than 1. Thus, if the expected utility of $\pi^\star$ is greater than 1, some guaranteed satisfying policy must exist. □

Finally, we show that the QBF formula φ is true iff a guaranteed satisfying policy for δ exists. In the D-SEP, the manager can choose whether to set each a_i true or false by making a suggestion or not when A_i sends its response. This corresponds to the "exists" quantifications in φ – when trying to prove the formula true, we can choose any desired

value for x_i. On the other hand, the manager cannot influence the values of b_i – these are chosen at random. Thus, the manager will have a guaranteed satisfying policy iff it's policy can handle all possible choices of the b_i's. This corresponds exactly to the "for all" quantifications of the y_i's. Note that we don't depend on the precise values of the probabilities – all that matters is that both true and false can occur for each b_i with some positive probability. Thus, a guaranteed satisfying policy for δ exists iff the QBF formula is true. Since the latter problem is PSPACE-complete, then the problem of determining if δ has a guaranteed satisfying policy is PSPACE-hard, and hence (by the above claim) OptUtility(δ,θ) for a bounded number of suggestions must also be PSPACE-hard.

OptPolicy(δ) is PSPACE-hard: We show that OptPolicy(δ) is PSPACE-hard by reducing from OptUtility(δ,θ). Given a D-SEP δ, we construct δ' to be the same as δ except that it has a new initial state s'_0. From s'_0, the manager may choose $Halt$ in order to end the process and gain utility $\theta + \epsilon$, or may choose $NoOp$, in which case the process transitions to the original initial state s_0. This construction is easy to do and runs in polynomial time. The original D-SEP δ has an expected utility for $\pi^\star$ that exceeds θ iff the optimal policy for δ' specifies that the manager should perform the initial action $NoOp$. This follows since if the expected utility of δ is θ or less, the optimal decision is to $Halt$ immediately, taking the utility $\theta + \epsilon$. Thus, since OptUtility(δ,θ) is PSPACE-complete for a bounded number of suggestions, the corresponding problem of OptPolicy(δ) must be PSPACE-hard.

C.4 Proof of Theorem 4.4.1 – unlimited suggestions

For this second case, we assume that the manager may make an unlimited number of suggestions to any participant. Below we prove that in this case OptUtility(δ,θ) is EXPTIME-complete, then use this result to prove that OptPolicy(δ) is EXPTIME-hard.

OptUtility(δ,θ) is EXPTIME-complete : First, we show that OptUtility(δ,θ) is in EXPTIME. Given a D-SEP δ, we can convert δ into a Markov Decision Process (MDP) with $O(N)$ possible actions and one state for each state in δ. The MDP can be then solved with techniques such as linear programming that run in time polynomial in the number of

states [113]. For δ, the number of states is exponential in N, so the total time is exponential. Then the expected utility of $\pi^\star$ for δ exceeds θ iff the optimal value of the initial state of the MDP exceeds θ.

Second, we show that OPTUTILITY(δ,θ) is EXPTIME-hard by a reduction from the game G_4 [168]. This game operates as follows (description from [112]): The "board" is a 13-DNF (disjunctive normal form) formula φ with a set of assignments to its $2k$ boolean variables. One set of variables $x_1, ..., x_k$ belong to player 1 and the rest $y_1, ..., y_k$ to player 2. Players take turns flipping the assignment of one of their variables. The game is over when the 13-DNF formula evaluates to true with the winner being the player whose move caused this to happen. The computational problem is to determine whether there is a winning strategy for player 1 for a given formula from a given initial assignment of the variables. Without loss of generality, below we assume that the original formula has been transformed so that the corresponding initial assignment sets all variables to false.

Given an instance of the game G_4 over some 13-DNF formula φ, we construct a corresponding D-SEP δ as follows:

- **Participants:** $P = \{A_1, ..., A_k, B_1, ..., B_k\}$, for a total of $N = 2k$ participants.

- **States:** A state $s = (a_1, ..., a_k, b_1, ..., b_k, Pend, Last)$ where the a_i's and b_i's indicate each participant's current response ($True, False$, or $NoneYet$), $Pend$ is the set of participants that the manager has made a suggestion to that has not been responded to yet, and $Last$ indicates whether the last message that changed a value was from some A or some B. The a_i's and b_i's correspond directly to the x_i's and y_i's in the formula φ. Thus, we say "φ is satisfied in s" if no a_i or b_i has the value $NoneYet$ and evaluating φ by substituting corresponding values for the x_i's and y_i's yields true.

- **Values:** $V = \{True, False\}$.

- **Actions:** $A = \{NoOp, Halt, SW_{p,true}, SW_{p,false}\}$ where $p \in P$.

- **Transitions:** We construct $T()$ so that the following steps will occur in order:

1. **Choice:** In the initial state the manager may either perform $NoOp$ (to wait for responses) or $Halt$ (if it has no winning strategy).

2. **Startup:** Every participant sends in a response $False$. The manager then suggests a change to every B_i, who do not immediately respond.

3. **A-Turn:** The manager chooses some A_i to suggest a change to. A_i immediately agrees, flipping the current value of a_i. If φ is now satisfied, $Halt$.

4. **B-Turn:** The manager performs $NoOp$, and receives a response to a previous suggestion from some random B_i, flipping the value of b_i. The manager immediately sends another suggestion back to the same B_i, who does not yet respond. If φ is now satisfied, $Halt$. Otherwise, go back to A-Turn.

- **Utilities:** the only non-zero utilities are as follows:

$$U(s_0, Halt) = 1 \quad (quitting\ from\ the\ initial\ state)$$
$$U(s, Halt) = 1 + \epsilon \quad if\ s \neq s_0,\ s.Last = A,\ and\ \varphi(s) = True$$

The size of this D-SEP is polynomial in N and the whole reduction can be done in polynomial time. As with the bounded suggestions case, the explicit transition and utility functions are exponential in N, but the rules above allow all of the necessary cases to be represented concisely in terms of the current responses, $Pend$, and $Last$. Likewise, we can "force" the needed manager and participant behavior by appropriate setting of the transition function.

We now demonstrate an additional result needed to complete the proof:

Definition C.4.1 (**guaranteed A-Win policy**) Given a D-SEP δ constructed from φ as above, a guaranteed A-Win policy is a policy that, if followed by the manager, guarantees that the SEP will terminate in a state that satisfies φ and where the last step was an "A-Turn." □

Claim: A guaranteed A-Win policy for δ exists iff the expected utility of the optimal policy $\pi^\star$ for δ is greater than 1 (e.g., OptUtility(δ,$\theta = 1$) is true).

Proof: Analogous to the claim previously given for a guaranteed satisfying policy in the bounded suggestions case.□

Finally, we show that a winning strategy exists for player 1 in G_4 iff a guaranteed A-Win policy exist for δ. We consider the possible actions for the SEP manager, who represents player 1. In the initial "Choice" step, if the manager does not have a guaranteed A-Win policy, it is best to *Halt* immediately and settle for a utility of 1. If the manager decides to play, then it also has a choice in Step 3 of which A_i to suggest a change to – this corresponds to choosing which x_i for Player 1 to flip. Step 4 corresponds to Player 2's flip of some y_i, and the manager has no choice to make. Thus, given a winning strategy for Player 1 in G_4, it is easy to construct a guaranteed A-Win policy for δ (mapping x_i flips to A_i change suggestions), and vice versa. Since the problem of determining if Player 1 has such a winning strategy for G_4 is EXPTIME-hard, the problem of determining if δ has a guaranteed A-Win policy is EXPTIME-hard, and hence (by the above claim) the problem of OptUtility(δ,θ) must also be EXPTIME-hard.

OptPolicy(δ)is EXPTIME-hard: This proof follows exactly the same form as the proof of OptPolicy(δ) for the bounded suggestions case. Since OptUtility(δ,θ) is EXPTIME-complete for an unlimited number of suggestions, the corresponding problem of OptPolicy(δ) must be EXPTIME-hard.

C.5 Proof of Theorem 4.4.2

Here we show how to compute the optimal policy $\pi^\star$ in time polynomial in N, assuming a K-partitionable utility function and that the manager sends at most one suggestion to any participant. Although the formalisms are very different, the key observation underlying this proof is similar to that of Theorem 4.3.2. Here we also create a state space that only models the *number* of participants in each group, rather than their specific members.

We define a summary state function $S = \{\bar{C}, \bar{D}, \bar{E}\}$ where

- $\bar{C} = (C_1, ..., C_K)$ where C_i is the number of responses V_i that were received that do *not* have a suggestion pending.

- $\bar{D} = (D_1, ..., D_K)$ where D_i is the number of responses V_i that were received that *do* have a suggestion pending.

- $\bar{E} = (E_1, ..., E_K)$ where E_i is the number of responses V_i that were received as a response to a suggestion.

In what follows, the notation $\bar{C} - v$ indicates "subtract one from the variable in $\bar{C}$ specified by value v." Given S, we can define the following transitions (omitting details for states where everyone has already responded):

$$T(\{\bar{C}, \bar{D}, \bar{E}\}, SW_v, \{\bar{C}-v, \bar{D}+v, \bar{E}\}) = 1$$

$$T(\{\bar{C}, \bar{D}, \bar{E}\}, NoOp, \{\bar{C}+v, \bar{D}, \bar{E}\}) = \rho_o(\bar{C}, \bar{D}, \bar{E}) \cdot \rho_v$$

$$T(\{\bar{C}, \bar{D}, \bar{E}\}, NoOp, \{\bar{C}, \bar{D}-v, \bar{E}+w\}) = \rho_{sv}(\bar{C}, \bar{D}, \bar{E}) \cdot \rho_{vw}$$

The first equation represents the manager requesting that some respondent switch their response from the value v; the state is updated to note that a suggestion has been made (with probability 1). The next two equations handle the uncertainty when the manager decides to wait for the next message to arrive. Specifically, the second equation handles the case when the next message is an original response from a previously unheard from participant (probability $\rho_o(\bar{C}, \bar{D}, \bar{E})$), while the third equation handles the case where the next message is a response to a previously made suggestion to switch from value v (probability $\rho_{sv}(\bar{C}, \bar{D}, \bar{E})$).

At any time each participant's response is either counted once among the K variables of each of $\bar{C}$, $\bar{D}$, or $\bar{E}$, or has not yet been received. The number of possible states is thus the number of ways of dividing N participants among $3K + 1$ groups, which is:

$$|S| = \binom{N + 3K}{3K} = O(N^{3K})$$

Because of the restriction to send at most one suggestion to each participant, the graph formed by the transition function over these states is acyclic. Thus, the optimal policy may be computed via a depth-first search over the graph in total time $O(N^{3K})$.

C.6 Proof of Theorem 5.4.1

We are given a SEP template τ and a parameter description ϕ for τ, and wish to determine whether τ is instantiation safe with respect to ϕ. We will show that in general this problem is co-NP-complete.

First, observe that this problem is in co-NP: a non-deterministic algorithm can solve the complementary problem of determining instantiation (un)safety by guessing an assignment to all the parameters, then verifying that instantiation with those parameters results in an invalid declaration.

Next, we show that this problem is co-NP-hard by reducing from $\overline{SAT}$ (the problem of determining whether some boolean formula φ is *not* satisfiable). Given a formula φ over the K boolean variables $w_1, w_2, ..., w_K$, we construct a template τ with the following parts:

- **Participants:** fixed to a single arbitrary email address

- **Questions:** one boolean question named `Test` that is *guarded* so that it is only asked of the participants when the expression $\neg\varphi_q$ is true. φ_q is the formula φ where each variable w_i has been replaced by a boolean parameter q_i

- **Goals:** a single `MustConstraint` C_0 that is true whenever the `Test` variable is true.

- **Notifications:** none.

In addition, we construct a parameter description ϕ for τ that specifies K boolean parameters named $q_1, ..., q_K$. This construction is clearly polynomial time in the size of ϕ.

Then, φ is in $\overline{SAT}$ iff τ is instantiation-safe w.r.t. ϕ. More specifically, τ is not instantiation safe only if there is some way for the guard $\neg\varphi_q$ on the question `Test` to evaluate to `False`, in which case the constraint C_0 is invalid because it references the undefined variable `Test`. Thus, τ is instantiation safe w.r.t. ϕ iff $\neg\varphi_q$ is always true, which is the case iff φ_q is always false, which is the case iff φ is never satisfiable (e.g., if $\varphi \in \overline{SAT}$). Since the size of ϕ is proportional to the number of parameters, determining instantiation safety is thus co-NP-hard in the size of ϕ.

C.7 Proof of Theorem 5.4.2

We are given a SEP template τ and a parameter description ϕ for τ, and wish to determine whether τ is instantiation safe with respect to ϕ. Given the bounded conditions of the theorem, we can assume that each `forAll` and `enumeration` statement consist of at most some constant J set parameters combined with any set operator, and that each `guard` statement consists of conjunctions and disjunctions of at most J terms (where terms are boolean parameters, or compare a parameter with a constant/parameter). Initially, we assume that there are no quantifications on any question, then relax this assumption at the end. We begin by examining some general properties of `guard` statements that will be significant, then sketch how to solve this problem by examining each of three primary parts of the template.

Guard statements: Given the assumptions, a `guard` depends only on constants and parameters. Thus, for any node (e.g., a question, goal, or notification) in the template, the guard may be evaluated without considering the current state of the data set or any variables defined by that node. In addition, the remainder of a node is evaluated only if the `guard` evaluates to true (See Appendix B.1), in which case the remainder of the node must have no syntactic errors, undefined variables, etc.

In addition, a key issue is whether a `guard` property can ever evaluate to false. A guard may involve up to J terms that utilize up to $2J$ parameters. Suppose we are given some guard formula $\varphi = g(P_1, P_2, ..., P_{2J})$. Each parameter P_i may have some restrictions R_i associated with it that are defined in ϕ (e.g., restricting the minimal or maximal value). These restrictions involve only a single parameter, so have bounded size. Given these definitions, we construct:

$$\varphi' = g(P_1, P_2, ..., P_J) \wedge R_1 \wedge R_2 \wedge ... \wedge R_{2J}$$

φ' is a boolean formula with at most $O(J)$ terms and $O(J)$ parameters. By construction, the guard may evaluate to false in an instantiated template iff φ' evaluates to false for any choice of the parameters $P_1, ..., P_{2J}$. At worst, we can determine if this can ever occur by

considering all possible assignments of each *term* to true or false ($O(2^{O(J)})$ possibilities). Then, for each true possibility, we check to see if there is an assignment to the parameters that achieves those truth values for the terms and that is consistent with ϕ, via a linear program that can be solved in time polynomial in J. Thus, the total time is exponential in J but polynomial in the total size of τ and ϕ. We make use of this result below.

Questions: We process each question in turn, checking each of the three conditions given for a valid declaration (Definition 5.4.2). If a question has a `guard` property, we first evaluate whether that guard could ever evaluate to true in polynomial time, using the result above. If not, then we ignore the rest of the question. If so, we verify that the question has a valid type, contains all the necessary properties for that type, and has a valid question name that is not reproduced by another question. In addition, we must verify that each property is a valid expression, based on substituting candidate values for any parameter. This is easy to check because all that matters for whether the expression is valid is the type of the parameters, not their specific values. All of these steps are easily accomplished in time polynomial in the number of queries, and thus in the size of τ. In addition, we can handle each question separately, aside from verifying that each question has a distinct name.

Finally, we must verify that any `enumeration` property E is not the empty set. First, note that the (non-set) parameters used by a `guard` are disjoint from the (set) parameters used by an `enumeration`. Thus, we can ignore the guard after determining that it is possible for it to be satisfied. Next, we consider the possible values for the set parameters used in E. There are potentially an infinite number of such possible values. Note, however, that our only concern is whether any such choice will cause E to evaluate to the empty set, so we can consider a finite set of carefully chosen choices. In particular, we can consider each possibility where parameter P_i is empty or not and is related to every other parameter by a subset/superset/equals relation or none of those. We eliminate possibilities excluded by ϕ due to non-empty or subset parameter restrictions (see Definition 5.4.1) — the simple form of these restrictions ensures that this is easy to do, even if they refer to other set parameters not directly used by E. Since E has at most some constant J parameters, the total number of possibilities is exponential in J but polynomial in the total size of τ and ϕ.

Goals: As with questions, we process each goal in turn, discarding those for which the guard will never evaluate to false. Likewise, we then check that the goal has an appropriate type, appropriate properties for that type, and that each expression used by the property is valid after substituting candidate parameters.

There are two significant differences vs. the verification of questions. First, goals may contain quantifications. In particular, we must verify that any `forAll` property has a valid set expression, and that any `suchThat` property is a valid boolean expression. Both of these are easy to do. Note that we do *not* have to determine if there exists some possible choice of the quantified variables so that all `suchThat` properties are satisfied — if not, the node will not be executed, but it must still be valid in terms of legal expressions, referencing only defined variables, etc.

Second, a goal may refer to the value of certain questions (e.g., to test how many responses of a certain type have been received). We must ensure that these references are not invalid because of an unsatisfied guard on those questions. Assume momentarily that a goal refers to exactly one such question. Let φ_g be the guard on the goal and φ_q be the guard on the question. As before, we can construct a new formula φ' that is the conjunction of φ_g, φ_q, and any parameter restrictions R_i from ϕ on the parameters in this formula. This formula has at most $4J$ parameters and can still be solved in time polynomial in the size of τ and ϕ. If this formula can ever evaluate to false, then this goal will be invalid for some parameter assignment and thus τ is not instantiation-safe.

We now consider guards that refer to more than one question. A key observation is that the goal is invalid if and only if there exists a parameter assignment such that the guard on the goal is true and the guard on some question q is false for any q that this goal references. Thus, we can apply the test with φ' independently to every question referenced in the guard, and the template is not instantiation safe if any φ' can evaluate to false.

Thus, overall we can verify one goal in time polynomial in the size of τ and ϕ. Furthermore, each goal can be considered independently, since goals do not define symbols used elsewhere.

Notifications: The basics of dealing with guards, quantifications, and checking questions is

the same as for goals. There are just a few differences in properties that have to be checked. For instance, we must check that there is exactly one `notify` and `message` property. As with goals, the entire testing can be done in polynomial time.

Conclusion: Thus, questions can be verified in polynomial time, and each goal and notification can be verified in polynomial time while considering at most one question at a time. Since τ is proportional to the number of questions, goals, and notifications, under the given assumptions we can determine the instantiation safety of τ w.r.t. ϕ in total time polynomial in the size of τ and ϕ.

We now briefly consider the issue of quantified questions, In this case, verifying instantiation safety remains polynomial time, but there are a number of additional issues. First, questions must have a unique name, distinct for each quantification possibility. Second, goals/notifications may reference these quantified questions, and we must ensure that each reference is to a defined variable. The template language addresses both of these issues by having the template provide only a base name for each question, then automatically computing composite names by adding a quantifier ID to the base name for each possibility. Goals/notifications may access these names via a quantification over variables such as `$Opt.range()$`, where `Opt` is the question base name. Finally, in a question an `enumeration` property may make use of quantified variables, and we must test that the enumeration cannot result in an empty set. We can solve this problem using the same general technique that was applied to checking enumerations before (iterating over all representative possibilities). However, the addition of quantifications means that we also must consider representative values for each quantified variable defined by a `forAll` property, restricted by any `suchThat` properties. Since each `forAll` and `enumeration` property references at most J parameters, the total number of possibilities considered is exponential in J but still polynomial in τ and ϕ. Thus, given the conditions of Theorem 5.4.2, templates with quantified questions may still be verified in polynomial time.

C.8 *Proof of Theorems 5.5.1 and 5.5.2*

For these theorems we are given an L-SEP Λ, current state D, and some `PossiblyConstraints` C_D, and wish to compute the acceptable set A of Λ. We consider the two cases where C_D is and is not bounded:

NP-hard for arbitrary constraints: For this case we show that computing the acceptable set is NP-hard by a reduction from ultimate satisfiability: given an L-SEP Λ with N participants, data set D, constraints C_D, and a possible response r, Λ is ultimately satisfiable for r iff r is in the acceptable set A for Λ. This relationship follows directly from the definition of the acceptable set, and the reduction is clearly polynomial time. Since ultimate satisfiability is NP-complete in N for arbitrary constraints, computing the acceptable set must be NP-hard in N.

Polynomial time for bounded constraints: We can determine whether any particular response r is in A via testing ultimate satisfiability: r is in A iff D is ultimately satisfiable w.r.t. C_D for r. Since C_D is bounded, Theorem 4.3.2 states that this satisfiability testing can be done in time polynomial in N and the $|C_D|$. In addition, since C_D is bounded, either there are only a small number of possible responses (if C_D is domain-bounded), or there are only a bounded number of responses that are distinguishable w.r.t. the constraints (if C_D is constant-bounded, as discussed in the proof of Theorem 4.3.2). In either case, there are only a constant number of different responses r that must be tested. Thus, by testing each representative response, we can determine the entire acceptable set (representing it as ranges of acceptable values) in time polynomial in N and $|C_D|$. If we actually construct the entire set A (as described in the theorem), then there is an additional polynomial time dependence on $|A|$.

C.9 *Proof of Theorem 5.5.3*

This theorem follows from the proof from Theorem 4.4.2, since computing the optimal policy in this situation involves computing and comparing the expected utility of all possible states.

C.10 Proof of Theorem 5.5.4

Here we are given an L-SEP Λ, current state D, constraints C_D, and a response r, and wish to compute the minimum sufficient explanation E for rejecting r. This theorem has different results depending on whether C_D consists of `MustConstraints` or `PossiblyConstraints`:

Polynomial time for `MustConstraints`: For a `MustConstraint`, the size of the minimum sufficient explanation is always one. We can compute this explanation by adding r to D and then testing each constraint to see if it is unsatisfied in this new state; any such constraint is a minimum explanation. Testing each constraint on a given state is polynomial in N, and there are at most $O(|C_D|)$ constraints, for total time polynomial in N and $|C_D|$.

NP-hard for `PossiblyConstraints`: In this case computing a minimum explanation is NP-hard in two different ways. First, a reduction from ultimate satisfiability: given an L-SEP Λ, D, C_D, and r, D is ultimately satisfiable for r iff the minimum explanation for rejecting r on D does *not* exist. This relationship follows from the definition of an explanation, since if an explanation exists it rules out any way of satisfying the constraints, and the reduction is clearly polynomial. Thus, since determining ultimately satisfiability is NP-complete in N (Theorem 4.3.1), then computing the minimum explanation is NP-hard in N.

Second, a reduction from SET-COVER, which is defined as follows: We are given a set $X = \{1, 2, ..., N\}$ and family of subsets of $F = \{S_1, S_2, ..., S_M\}$ such that every $S_i \subset X$ and every element of X is contained in some S_i. A cover for this problem is a set $F' \subset F$ such that the union of all $S_i \in F'$ contains every element of X. The problem is to determine whether there exists a cover of size J or smaller for X.

We construct the following L-SEP Λ with:

- **Participants:** $P = \{p_0, p_1, p_2, ..., p_N\}$.

- **Data set:** D is a single table with one boolean attribute R

- **Constraints:** a set of `PossiblyConstraints` $C_D = C_0 \wedge C_1 \wedge C_2 \wedge ... \wedge C_M$ where

$$\begin{aligned} C_0 &= (R_{yes} = 0) \\ C_i &= \bigwedge_{j \in S_i} (R_{true} \neq j) \; for \; 1 \leq i \leq M \\ R_{yes} &= (COUNT \; (*) \; WHERE \; \mathsf{value} = True) \end{aligned}$$

Constructing this L-SEP is clearly polynomial time in the size of the SET-COVER problem.

Given this construction for Λ, we now show that a set cover for X of size J exists iff the minimum explanation E for rejecting a response r of `False` for Λ with an initially empty state D contains $J+1$ constraints. First, given an explanation E with $J+1$ constraints, a minimum cover F' is the set of all S_i such that C_i is present in E, for $i \neq 0$. (Every sufficient explanation E contains C_0; it is a special case included just to handle the situation where all participants respond `No`. Hence, F' will be of size J.) To see why this works, consider an example set $S_7 = \{3, 5\}$. This set is mapped to the constraint $C_7 = (R_{true} \neq 3) \wedge (R_{true} \neq 5)$. A sufficient explanation for rejecting r must cover every possible outcome of the L-SEP, and two such outcomes are for either 3 or 5 participants to respond `True`. Thus, if response r is to be rejected, the explanation E must cover these two cases, either by choosing C_7, or by choosing some other constraint(s) that also covers the cases of 3 or 5 `True` responses. This follows exactly the same rules as a solution to SET-COVER. Likewise, given a cover F' for X of size J, a minimum explanation for rejecting an initial `False` response is the conjunction of C_0 together with all constraints C_i where S_i is in F', for a total size of $J+1$. Thus, any input to the SET-COVER problem can be reduced to solving the minimum explanation problem. Since the former problem is NP-complete in the number of sets (M), the latter problem must also be NP-hard in number of constraints ($|C_D|$). Combining this with the previous result, we see that computing the minimum sufficient explanation for `PossiblyConstraints` is NP-hard in N and NP-hard in $|C_D|$.

C.11 Proof of Theorem 5.5.5

We are given an L-SEP Λ with N participants, current state D, constraints C_D, and a response r and wish to find the minimum sufficient explanation E for rejecting r, assuming that C_D is bounded and that the size of a minimum E is no more than some constant J. If C_D consists of `MustConstraints`, then we already know that this problem is polynomial time in N and $|C_D|$ from Theorem 5.5.4.

If C_D is made up of `PossiblyConstraints`, then we can test if any particular explanation E is a sufficient explanation via ultimate satisfiability: E is a sufficient explanation iff $E \subseteq C_D$ and D is *not* ultimately satisfiable w.r.t. E for r. Since the constraints are bounded, Theorem 4.3.2 states that this testing can be performed in time polynomial in N and $|C_D|$. In addition, since any minimum explanation E contains only terms from C_D, restricting E to at most size J means that the total number of explanations that must be considered is $O(2^J)$. which is polynomial (constant) in $|C_D|$. Thus, we can compute the minimal explanation by testing the sufficiency of every possible explanation of size J or less and picking the smallest sufficient explanation. This algorithm runs in total time polynomial in N and $|C_D|$.

C.12 Proof of Theorem 5.5.6

This proof is explained when the theorem is introduced in Section 5.5.3.

VITA

Luke McDowell was born and grew up in Wilmington, Delaware. He went on to study Electrical Engineering at Princeton University, graduating with a B.S.E. degree in 1997. After working for several years in the field of computer vision at Sarnoff Corporation in Princeton, New Jersey, he entered graduate school in the Computer Science Department at the University of Washington. Luke has been a recipient of both a National Science Foundation Graduate Research Fellowship and a Microsoft Endowed Fellowship. In his graduate work, he pursued research in the fields of computer architecture, databases, and intelligent Internet systems, focusing on the Semantic Web. He received a M.S. in Computer Science in 2001 and a Ph.D. in Computer Science in 2004, both from the University of Washington.