Reflective Collaborative Agents for Complex Service Integration

P. Maurine Hatch and Eleni Stroulia
Computing Science Department
University of Alberta
Edmonton, AB, Canada T6G 2E8
{maurine,stroulia}@cs.ualberta.ca

Abstract

With the advent of more and more services available on the Web, a user can have a difficult job of assembling the various pieces of a complex task to arrive at a final solution. Not only would the user need to access each web-based resource through its individual client-side interface, but she would also need to interpret its response to her request, and to manually combine the multiple responses from the different resources to accomplish the complex task.

In this paper, we discuss a multiagent, XML-based framework that supports the development of aggregate applications that rely on semantic-based reflective monitoring and collaboration among several agents to complete the user's task. Our framework makes use of declarative models of the domain information, the task-specific information, and the semantic constraints of this information. Each agent uses these models to interact with the user, to coordinate the information exchange with the various web resources, to monitor and control the execution of the applications, and to take action when a failure is detected. When an agent detects a failure, it collaborates with other agents by distributing the tasks to those agents which are capable of completing the task, thus ensuring successful completion of the user's request.

We illustrate our approach and the architecture of the aggregate applications that it produces using a book-buying assistant as an example.

1. Motivation and background

As the World Wide Web continues to grow exponentially, querying it for a particular service can be a frustrating experience for a user. There are many different web-based applications and they usually do not interoperate. Not only does a user with a complex task need to decide in which order to access multiple applications, but she also needs to decide how to combine the information in their responses in order to formulate requests to yet other applications, and how to construct the solution to the task at hand based on all the responses received. The user's task would be much easier if she could simply request the complex service and have the final solution returned to her. This requires the development of an application that combines the functionalities of multiple, independently developed, web-based applications. The application aggregation would need to control the information flow of the overall task, translate the information among the existing applications, and combine the many responses received into a final solution

This interoperation of web-based applications is a challenge because there exists neither an agreed upon representation and semantics for the information required and produced by these applications, nor a uniform access mechanism for their services. Our work towards addressing this challenge has resulted in a multi-agent framework for task-specific aggregation of web-based applications. Each agent within this framework uses an eXtensible Markup Language (XML) [2] -specified domain model for the information representation and semantics, an XML-specified task structure model to represent the workflow and a task agent component that controls and coordinates the process by using these models.

Our framework is based on the TaMeX framework [13]. Similarly to TaMeX, *wrappers* are used to encapsulate the web-based applications that provide the information and services in a given domain. The role of a wrapper is that of an adapter between the application's original user interface and a new interface based on a common XML vocabulary for the domain. The wrappers interact with a *task agent* that acts as an intelligent, task-specific intermediary between the user and all the wrapped applications that are needed to accomplish the user's task. The task agent uses the *task structure model*, which is an explicit representation of a set of steps to accomplish the user's complex tasks, to control the process and information flow. The task agent also uses a *domain model*, which is a specification of the concepts of the subject area of interest, to establish a common language to be used by the user, the wrappers, and the task agent itself.

We have extended TaMeX to include an explicit representation of semantic constraints on the task and domain models of the task agent and the wrappers, and also a model of the distribution of capabilities between several agents. These more complex representations also required that the TaMeX models are specified using XML schema [6,7,8] technology. We have further extended the TaMeX framework by adding reflective monitoring with

domain-specific and task-specific semantics. This means that the execution is monitored and the various constraints on the information, both domain-specific and task-specific, are checked. This constraint checking is to ensure that there is valid input to and valid output from each step involved in the successful completion of the user's request. We also converted the framework to a multi-agent framework with the ability for the user-contacted agent to identify and collaborate with other agents. This collaboration is necessary when the reflective monitoring of the user-contacted agent detects a failure condition on one of its subtasks. Other agents are then called into service to ensure that the user's request is satisfactorily deployed.

The rest of this paper is organized as follows. Section 2 discusses the overall architecture of our multi-agent framework for aggregation of web-based applications. Section 3 describes and illustrates the run-time behavior of an aggregate application by using a prototype book-buying assistant as an example. Finally, section 4 summarizes the approach and concludes by identifying the contributions of this work.

2. Overall architecture

The overall architecture of our multi-agent XML-based framework consists of two loosely coupled environments: a design-time environment and a run-time environment. A multi-agent system consists of a group of agents that interact and cooperate to accomplish some set of tasks in a distributed way. Each agent offers a specific service or services to other agents. The solution to a large task is accomplished by combining and coordinating the different services offered by the individual agents. In our case, each agent is responsible for collaborating with the other agent(s) to accomplish the specific tasks of their own users. Once the task is accomplished, each agent is responsible for returning the combined results to their own users.

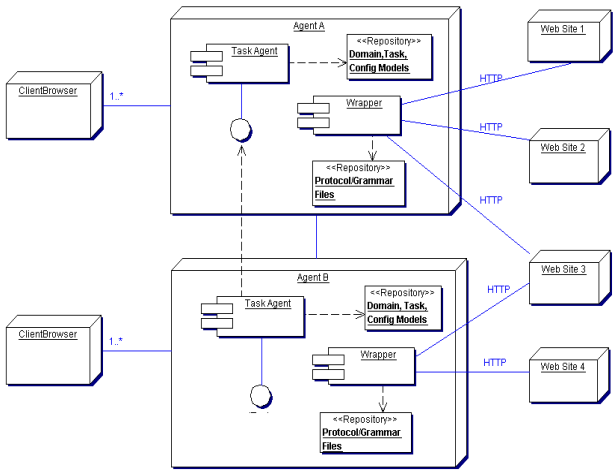

Figure 1: Run-time architecture of our multi-agent XML-based framework.

The run-time architecture is diagrammatically depicted in Figure 1, consisting of two agents. The main components of this architecture are the web-based applications being accessed and the agents that control the processing between the users, these web-based applications, and other agents. Each agent has the same structure and is composed of a task agent component, a wrapper component, and the repositories for the supporting files for these two components. The important supporting files for the task agent component are the domain model, the task structure model, and the constraints specified within these models. These models are computer-usable, declarative representations of the domain and the task structure respectively. The domain is the subject area involved in the specific application being run. The task structure is the specification of the workflow of the overall task of the application. The important supporting files for the wrapper component are the learned request protocol and grammar rules.

The design-time environment supports the development of these models/files needed for the execution of the task agent and the construction of the wrappers for the existing web-based applications for each new user task. These XML models and wrapper-construction files are used as input to the run-time environment. Using these models, each agent within the run-time environment monitors the execution, checks the constraints, and aggregates the outputs of the underlying wrapped web-based applications for its user(s) tasks.

2.1 The task agent

The role of a task agent is to interact with the user of the multi-agent system, to evaluate the applicable conceptual and functional constraints, and to coordinate the information exchange among the wrappers of the existing underlying applications and other agents needed for accomplishing the user's task.

To accomplish this coordination between the wrappers and other agents, the task agent uses a declarative, task-structure [10,11,12] approach to the representation of its processing mechanism. In this approach, a task is characterized by the type(s) of information it consumes as input and produces as output, and the nature of the transformation it performs between the two. A complex task may be decomposed into a partially ordered set of simpler subtasks. A simple, non-decomposable task, i.e., a leaf task, corresponds to an elementary procedure. The control of processing moves from higher-level complex tasks to their constituent subtasks. At the same time, information flows through the task structure as it is produced and consumed by the tasks. The actual process is non-deterministic because there may be alternative decompositions for a given task, applicable under different conditions, and the subtasks resulting from the decomposition of a complex task are only partially ordered. Complex, high-level tasks get accomplished when all their subtasks are accomplished.

The XML-based task structure model used by the task agent is described in section 2.2.

Each task agent implements a set of these XML-specified tasks, with non-deterministic decompositions. The user's interaction with the task agent decides which of the alternative decompositions is employed, and which task is active at each point in time. Also, through some user-interaction leaf tasks, the user specifies the problem and receives the results as they get produced. The wrappers of the individual underlying applications implement elementary leaf tasks using the services of their underlying sources (information-collection tasks). When a particular task is sufficiently decomposed into elementary tasks, the task agent requests the relevant wrappers to accomplish them and to return their results

As well as using a task structure model, the task agent also uses a system configuration model and a domain model for its processing. The system configuration model specifies information needed for execution of the system, such as the name of the server that the task agent is running on, the list of the registered task models, and the list of all the resources that the system knows about. The domain model specifies the entities of the application domain and their constraints. The task agent uses the constraints specified in the models to validate the user's input and the successful execution of the subtasks involved.

2.2 The task-structure model

The role of the task structure model is to enable the task agent to:

1. decompose high-level tasks into elementary (leaf) ones
2. present a hierarchical menu of tasks to the user
3. compose individual wrappers' results in a coherent solution to the overall user's task

The task structure is specified using XML. A generic task structure is specified with an XML schema and is part of the design-time environment. See Appendix B for the XML structure diagram and XML schema for the generic task structure used in our framework. An application-specific XML task structure instance is specified, in conformance with the task structure schema, for each high-level complex task to be executed by the user. This task structure instance is used by the task agent to present the task structure to the user as a hierarchical menu where the high-level tasks correspond to the top-level menus and their alternative decompositions are represented in sub-menus. The tasks presented are color-coded so that the user knows which tasks are executable and which have been completed. This representation style gives any specific task agent a simple intuitive interface through which the user can traverse the task structure and invoke the tasks that it can accomplish.

There are five types of tasks in the generic task structure. There are the two user-interaction tasks: the input task that gathers information from the user and the output task that presents results to the user. For each input and output task there is an associated XSL stylesheet [3] for user presentation. There is the wrapper task that is an information-collection task and accesses an external resource. For each wrapper task, a wrapper must be constructed to provide the functionality specified by the task. There is the internal task that performs non-interactive processing. For each internal task, a new component must be implemented to provide the functionality specified by the task. Finally, there is the grouping task that is used to structure the other tasks in sub-groupings to correspond to the intermediate sub-goals of the user's overall task.

2.3 The domain Model

The role of the domain model is to establish a common language for specifying the application information that can be understood by the user, the task agent and the wrappers of all the integrated applications. The application information that is specified using this domain model is the information provided by the user, the information produced by the task agent as it elaborates the user's problem specification, and the information provided to and by the wrappers and agents.

The domain model, used by the task agent, specifies:

1. the entities of the application domain
2. the attributes of these entities
3. the composition relationships between these entities
4. the logic constraints of these entities

The domain model is specified using XML documents. An application-specific domain model and some of its constraints are specified with an XML schema. Constraints that cannot be specified in the XML domain schema are specified in an XSLT stylesheet.

2.4 Constraints and Reflective monitoring

The role of the constraints is to provide the semantics used by the reflective monitoring process of the task agent. At run time, the task agent checks the constraints to ensure that the information consumed and produced at each stage of the processing is valid. Upon discovery of invalid information, the task agent takes action, such as invoking the services of another agent, to resolve the problem.

There are two main categories of constraints: conceptual constraints, which are domain-specific constraints defining the assumed ontology of the domain, and functional constraints, which are task-specific constraints specifying the nature of the information transformation expected of each task. Table 1 shows a breakdown of the types of constraints within each category, along with their description and where they are specified within our framework.

	Constraint Type	Description	Where Specified
Conceptual	Value Constraint Single Field	-constraints on the value of a single field of information in the domain model	XML domain schema
	Value Constraint Multiple Fields	-semantics on the relationship between values of multiple fields in the domain model	XSLT domain stylesheet
Functional	Structural Constraint	-semantics on the relationship between the tasks in the task structure model	XML task structure schema and instance
	Task Parameter	-semantics on the relationship between the tasks in the task structure model	extension to XML domain schema

Table 1: Constraints used by reflective monitoring in our framework

2.5 The Wrappers

The wrapper components in the architecture shown in Figure 1 act as adapters between the web-based resource's original API and a new API based on a common XML vocabulary for the domain. The wrapper receives an information-collection request from the task agent that is the problem specified in terms of this XML domain model. The wrapper responds to this request by returning an answer in the form of an XML document containing one or more instances of some concept (or concepts) in the domain model. This concept is called the target concept and is the entity (or entities) of the XML domain model that answer the problem specification from the task agent.

Our wrapper-construction process is based on the approach developed by Stroulia, Thomson, and Situ [13]. This approach exploits the hierarchical structure of both XML and HTML documents. XML documents are structured hierarchically where each XML element can be composed of simpler XML elements. We rely on this characteristic of XML for creating the semantic map of documents used in wrapping the WWW applications. The XML documents used are those for specifying the domain model, the information-collection request to the wrapper, and the answer returned from the wrapper that contains the instances of the target concept.

Our approach also utilizes the hierarchical structure of HTML. Currently, most web-based applications provide information encoded using HTML. These applications automatically generate HTML documents in response to a particular request type. The visual layout of these generated HTML documents for individual responses is similar, even though the actual content is different. In order to extract an instance of the target concept from one of these HTML responses, the hierarchical structure of this response is traversed to discover and select the needed elements of the target concept. This is done by parsing the HTML document into a tree representation rooted at the < html > tag, so that document subtrees that may contain the information of interest can be efficiently located, using a DOM-like API [1]. The wrapper uses a set of rules, or grammar, to traverse this document tree to locate the instances of the target concept's constituent elements. This extraction grammar is learned at design-time and is formulated using XPATH [5] expressions and is specified in an XML document.

In order to wrap a web-based application, the wrapper needs to know the protocol by which the responses of interest can be requested from the application server, and the grammar for extracting the instances of the target concept from the server's responses. The XML files that contain this information, the learned request protocol file and the grammar rules file, are stored in a repository for use by the wrapper.

The learned request protocol is the interaction protocol between the user's browser and the application. This protocol is specified in XML. Its role is to provide the wrapper component with a means to translate an XML problem specification into an appropriate request to the application server.

The grammar rules are the XPATH expressions for extracting each component of the target concept from the application's response. They are specified in XML. The role of the grammar rules is to provide the wrapper component with a means to extract the instances of the target concept from the application server's responses.

3. Run-time Behavior

To illustrate the run-time behavior of the aggregate applications developed with our multi-agent framework, we will use as an example a book-buying assistant prototype. This prototype was built to support explorative comparative book shopping, and more specifically book price checking. Today there are many websites offering books for sale. Knowledgeable consumers with specific constraints and preferences must access several different sites to identify available options. Then they have to compare the results from these different sites prior to making a decision. The aggregation of existing book-selling applications to support tasks such as comparative shopping is a compelling instance of web-based application aggregation. Our book-buying prototype application integrates two different book-selling web-based applications.

As the first step in the development of this application, the entities, their attributes, their compositions, and their constraints needed for the book-buying domain are specified in an XML schema and XSLT stylesheet. This book-buying domain XML schema file and XSLT stylesheet are shown in Appendix

A. A pictorial representation of the book-buying domain model is shown in Figure 2.

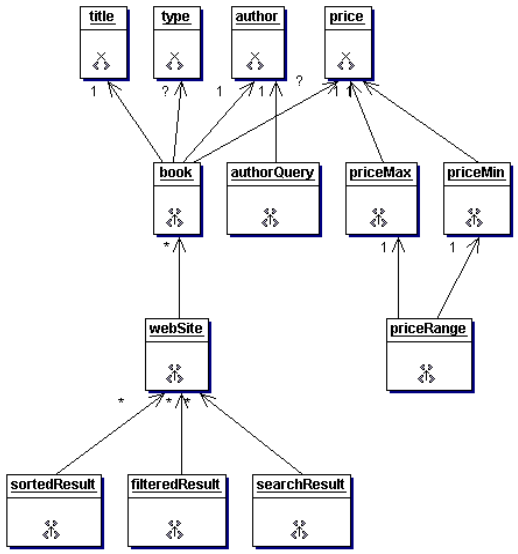

Figure 2: Domain Model XML Structure Diagram for the Book Buying Assistant.

This representation is an XML structure diagram generated by Together 5.5 from the XML schema file of the domain model. The multiplicities are shown on the diagram by '?', '1', or '*', where '?' signifies zero or one instance, '1' signifies exactly one instance, and '*' signifies zero up to an unlimited number of instances. The composition relationships can be seen clearly from this diagram. For example, it can be seen that the searchResult, the sortedResult, and the filteredResult, are all composed of books, with each book being composed of a mandatory title and author and an optional price and type of book. Examples of types of books are hardcover or softcover.

Our example of the book-buying assistant prototype is run with three agents. Each agent contains the same book buying assistant's task structure as diagrammatically depicted in Figure 3. The book buying assistant's XML task structure instance is shown in Appendix C.

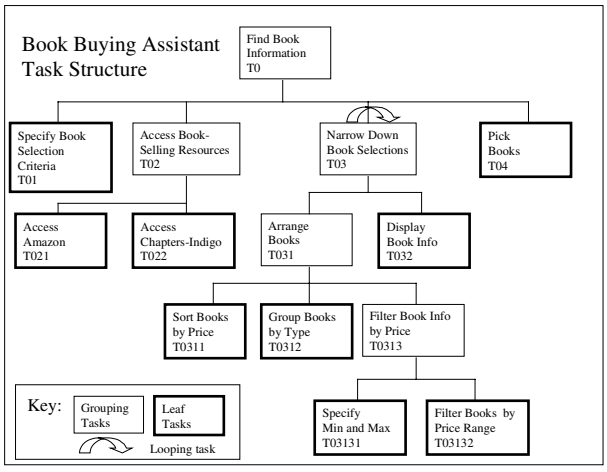

Figure 3: Task Structure for the Book Buying Assistant .

Though each agent uses the same task structure, each agent has different capabilities within this task structure. Agent A can perform all the tasks except for any of the "Access Book-Selling Resources". Agent B can perform all the tasks. Agent C can perform all the tasks except "Access Chapters-Indigo" and "Group Books by Type". In our example, Agent C is the agent that interacts with the user. Agent A and Agent B are the agents that have task capabilities within the book-buying assistant application that Agent C does not have. Agent C will accomplish the user's overall task by traversing the book-buying assistant task structure and performing the tasks within that structure. Agent C will either perform these tasks directly by itself, if capable, or will identify and then collaborate with other agents that are capable of performing these tasks.

In this task structure, the grouping tasks that are used to arrange the other tasks into sub-groupings are shown as regular solid boxes. The leaf tasks, which do the actual processing, are shown as heavy-lined solid boxes. The overall task of the book buying assistant, i.e., to *find book information*, gets decomposed into the tasks *of specifying book selection criteria*, *accessing book-selling resources* via the wrapped web-based applications, *narrowing down the retrieved book selections*, and finally by *picking the books* to purchase from the narrowed down list of books. The task agent of an agent capable of executing the book buying assistant uses the XML-specified model of this task structure to produce a task menu on the user's browser interface. This menu, shown in Figure 4, is the user's means of interacting with the task agent. The first step in the process is to specify the problem inputs. This is the user-interaction (input) task of "Specify Book Selection Criteria (T01)". The task agent sends to the browser the *query* entry form, which is generated by the XSL stylesheet corresponding to this input task. Using this form, the user specifies the name of the author whose books he wants to find. For the problem at hand, the user specifies the author to be "Sheri Reynolds" by typing "Sheri Reynolds" in the "author's name" entry field.

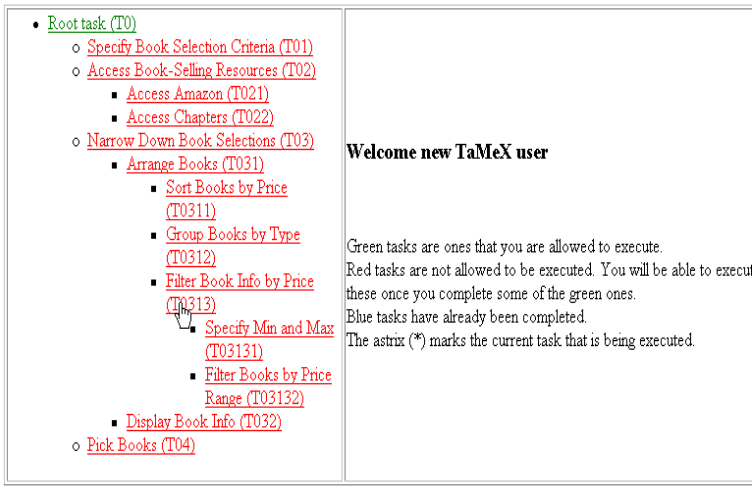

Figure 4: Task Menu for the Book-Buying Assistant

After the desired book selection criteria have been specified, the task agent proceeds to accomplish the next task, i.e., to access the book-selling resources (T02) via the wrapped web-based applications. This grouping task is decomposed into a set of information-collection (wrapper) tasks, one for each wrapped web-based application. In this example there are two wrapper tasks, "Access Amazon (T021)" and "Access Chapters-Indigo (T022)", to access the wrapped applications of the Amazon (www.amazon.com) and Chapters-Indigo (www.chapters.ca) book selling websites respectively. Each wrapper task is accomplished by invoking a request to the task-specified wrapped web-based application. This process is started when a user selects a wrapper task from the menu. The task agent responds by sending a request to the particular wrapper of the selected application that will produce the desired output given the already-entered set of inputs.

Since, in this example, the user wants to see all of the books that are available she selects each wrapper task in turn. When the user selects the first wrapper task of "Access Amazon (T021)" Agent C's task agent responds directly (since Agent C is capable of performing this task) by sending a request to the particular wrapper of the Amazon application that produces a list of books given the author's name. The author's name that is used for this access was previously entered by the user in the "Specify Book Selection Criteria (T01)" task and is "Sheri Reynolds". So, in this case, the T021 task will return a list of all the books authored by Sheri Reynolds that can be found at the Amazon website.

The user then selects the next wrapper task of "Access Chapters-Indigo (T022)." However, this time Agent C fails at this task because it does not have the capability to access the Chapters-Indigo website. Agent C takes corrective action and identifies that Agent B is capable of performing task T022. After having identified Agent B, Agent C begins its collaboration with this agent by sending Agent B the entire DOM with an indication to perform the T022 task. Rather than starting at the beginning of the book buying assistant task structure, Agent B jumpstarts its processing by starting with the T022 task. Agent B's task agent executes T022 by sending a request to the particular wrapper of the Chapters-Indigo application that produces a list of books given the author's name. After getting the results, Agent B updates the DOM with the retrieved information and then returns this updated DOM to the originating Agent C. Agent C has now completed the "Access Book-Selling Resources (T02) task.

Agent C then continues traversing the book buying assistant's task structure by performing the next task of "Narrow Down Book Selections (T03)". This grouping task is an iterative task that the user keeps performing until she has the selection of displayed books to her liking. Each iteration of this task consists of an "Arrange Books (T031)" task and a "Display Book Info (T032)" task. The "Arrange Books (T031)" grouping task allows the user to choose one of three choices: *sort the books by price*, *group the books by type of book*, or *filter the book information by a price range* entered by the user. The first two subtasks within T031, "Sort Books by Price (T0311)" and "Group Books by Type (T0312)" are internal tasks and are executed as soon as they are chosen by the user. The third subtask within T031, "Filter Book Information by Price (T0313)", is a grouping task that is made up of *specifying the minimum and maximum price* to use for filtering the books and then the actual *filtering of the books by using this price range.*

In our example, the user wants to group the books by whether they are hardcover or paperback so she selects the "Group Books by Type (T0312)" task from her browser's menu. Agent C fails at performing this task directly because it does not have the capability to group books by type. Agent C identifies that Agent A is capable of performing task T0312 and sends Agent A the entire DOM with an indication to perform this task. Agent A performs its internal task of T0312 by using the DOM passed to it as the input to the task. After execution, Agent A updates its own DOM with the results of the task execution which are a new grouping of books by type of the book information. Then, Agent A returns this updated DOM to the originating Agent C. Agent C then executes another user-interaction task, the output task of "Display Book Information (T032)" and displays this grouping of books to the user. The first iteration of T03 has now been completed.

The user decides that she is satisfied with the selection of books displayed on her browser and proceeds to select the task "Pick Books (T04)". This user-interaction task allows the user to pick the books she would like to buy by entering an indication by the appropriate displayed books.

The extended version of the prototype , supporting the example just described, is currently under development.

4. Conclusions

In this paper, we discussed an XML-based framework for the aggregation of web-based applications involving agents that collaborate to ensure that the requested service is delivered. The two main contributions of this work are

(a) the semantic-based reflective monitoring of the task-agents' behavior, and
(b) the agents' collaboration and tasks distribution that occurs when an agent fails.

In our framework, agents use declarative models of the application domain, the task structure and the semantic constraints specifying the information in this domain and the nature of the transformations it suffers in the task structure. Each agent uses these models to interact with the user, to coordinate the information exchange among the wrappers of the underlying web applications, to monitor and control the execution of the applications, and

to collaborate with the other agents when it cannot alone complete the desired task.

References

1. Document Object Model (DOM) Level 2 Specification, http://www.w3.org/TR/1999/CR-DOM-Level-2-19991210/
2. Extensible Markup Language (XML), http://www.w3.org/XML/
3. Extensible Stylesheet Language (XSL) Version 1.0 W3C Recommendation 15 October 2001, http://www.w3.org/TR/xsl/
4. Muffin, World Wide Web filtering system http://muffin.doit.org/
5. XML Path Language (Xpath) Version 1.0 W3C Recommendation 16 November 1999, http://www.w3.org/TR/xpath/
6. XML Schema Part 0:Primer W3C Recommendation, 2 May 2001, http://www.w3.org/TR/xmlschema-0/
7. XML Schema Part 1:Structures W3C Recommendation, 2 May 2001, http://www.w3.org/TR/xmlschema-1/
8. XML Schema Part 2:Datatypes W3C Recommendation, 2 May 2001, http://www.w3.org/TR/xmlschema-2/
9. XSL Transformations (XSLT) Version 1.0 W3C Recommendation 16 November 1999, http://www.w3.org/TR/xslt/
10. B. Chandrasekaran, "Task Structures, Knowledge Acquisition and Machine Learning", *Machine Learning*, 4:341-347, 1989.
11. E. Stroulia and A.K. Goel, "A Model-Based Approach to Blame Assignment: Revising the Reasoning Steps of Problem Solvers", *Proceedings of the 13th Annual Conference on Artificial Intelligence*, pp. 959-965, AAAI Press, 1996.
12. E. Stroulia and A.K. Goel, "Redesigning a Problem Solver's Operators to Improve Solution Quality. Proceedings of the 15th International Joint Conference on Artificial Intelligence, pp. 562-567, 1997.
13. E. Stroulia, J. Thomson, and Q. Situ, "Constructing XML-speaking wrappers for WEB Applications: Towards an Interoperating WEB", *Proceedings of the 7th Working Conference on Reverse Engineering,* 23-25 November 2000, Brisbane, Queensland, Australia, pp. 59-68, IEEE Computer Society Press.

The appendices referred to in this paper are included in the Computing Science Department, University of Alberta, Technical Report **TR02-06.**