Adding AI to Web Services

Charles Petrie[1], Michael Genesereth[1], Hans Bjornsson[1], Rada Chirkova[1], Martin Ekstrom[1], Hidehito Gomi[2], Timothy Hinrichs[1], Rob Hoskins[1], Michael Kassoff[1], Daishi Kato[2], Kyohei Kawazoe[3], Jung Ung Min[1], and Waqar Mohsin[1]

[1] Stanford Computer Science Department
Stanford University,
Stanford, CA 94305, USA
{petrie, all}@stanford.edu
http://logic.stanford.edu
[2] NEC Corporation
7-1, Shiba 5-chome
Minato-ku, Tokyo 108-8001, Japan
daishi@cb.jp.nec.com, gomi@az.jp.nec.com
[3] Intec Web and Genome Corp.
1010 El Camino Real #370
Menlo Park, CA 94025, USA

Abstract. The FX-Agents project consisted of members of the Stanford Logic Group and industrial visitors from NEC and Intec Web & Genome working together to develop new technologies based upon the combination of Web services and techniques from artificial intelligence, using our experience in AI-based software agents. This two-year project ran from April 2001 until March 2002 and explored the then emerging functionality of Web services. This paper is a result of our findings. In particular, this paper discusses the shortcomings of current Web service standards like WSDL and how logical AI techniques like declarative commands, agents, and planning can be used to address some of these shortcomings. The primary problems that we address are automated Web service discovery and composition of Web services.

1 Introduction

A key topic in knowledge management is the discovery of useful structured information when it is required. Web services generalize this issue to discovery of useful actions as well as information. By "Web services," we do not mean just any standard way to access business logic. We mean a way of publishing an explicit, machine-readable, common standard description of how to use a service and access it via another program using some standard message transport. If one is to use agents, one must have a machine readable format, preferably one that industry will use.

Fortunately, the use of *SOAP*[4] and *WSDL*[5] has been strongly embraced by industry and has become a major standard for such Web services. WSDL has

[4] http://www.w3.org/TR/SOAP/
[5] http://www.w3.org/TR/wsdl

had fast industrial acceptance, not in spite of its simplicity in comparison with more advanced systemic standards such as *ebXML*[6], but because of it.

The importance of SOAP and WSDL is that they offer the possibility of a simple industrial standard for reading what input and output messages a service accepts and sends, and then sending those messages over a standard transport. This *loose coupling* means it is irrelevant what kind of client or server software is at either end. Since WSDL is machine-readable and since SOAP allows loose-coupling, in principle it is possible to use arbitrarily complex software programs to read WSDL and use SOAP to invoke services by exchanging messages. On the Web service-enabled Internet, no one knows if you are a software agent.

However, there are problems with WSDL. The object-oriented community with experience in distributed transactions has already pointed out some of the problems with these standards [4]. Moreover, there is has been an expectation that we will be able to achieve the goal of discovering services when we need them, and, what is more, assembling them into new composite services, dynamically on-the-fly, as needed.

We show that WSDL-related technologies, such as *WSCI*[7], *WSCL*[8], *XLANG*[9], and *WSFL*[10] (the latter two having morphed into *BPEL4WS*[11]), which collectively we refer to as "WSxL" standards, are inadequate for dynamic discovery and integration, and thus inadequate for virtual enterprises (VEs). We also discuss the issues in providing the missing functionality.

2 Ultimate Requirements for Deployment of Web Service Agents

Ultimately, there are four primary functions we need to automate in building a network of service agents:

- Discovery, consisting of information sufficient for:
  - Search
  - Use
- Service Integration
- Process Integration
- Process Control

[6] http://www.ebxml.org/
[7] http://wwws.sun.com/software/xml/developers/wsci/
[8] http://www.w3.org/TR/wscl10/
[9] http://www.gotdotnet.com/team/xml_wsspecs/xlang-c/default.htm
[10] http://www-4.ibm.com/software/solutions/webservices/pdf/WSFL.pdf
[11] http://www-106.ibm.com/developerworks/library/ws-bpel/

We require that the information about the Web services be sufficiently machine-readable so that the above functions can be automated. Further, use of these functions should follow two **Fundamental Principles**:
Any publicly registered Web service can be consumed:

– **by anyone**, without requiring any change to the service (the **Democratic Principle**); and
– **at any time**, without prior arrangement (the **Just-in-Time Principle)**.

The FX-Agents approach is to identify the deficiencies of WSDL and associated technologies, and then to address them with the universality and flexibility of **Declarative Logic**.

3 Why UDDI and WSDL Aren't Enough

The most widely deployed emerging technology for Web service search is *UDDI*[12]. But a slight bit of experimentation will reveal that this technology, as it stands, is not adequate to allow service agents, or even humans, to discover new services. First consider the discovery component of *Service Search*.

Try an experiment. Go to `https://uddi.ibm.com/ubr/registry.html` and ask the computer equivalent of "Are there any Web services that book flights?"

UDDI *Find* allows you to search using the directory's taxonomy, but still you will not be able to search on services that "book flights." This is because **UDDI does not contain the actual service descriptions** which are distributed. But **UDDI does not perform distributed search**.

Suppose it did and found the **WSDL** description of the service and its operations. How would a software program know if a service booked flights or not?

To continue our experiment with the UDDI registry, if one searches the taxonomy "WAND" for "Travel," one finds "Travel Adventures Unlimited." One can then manually find the WSDL. Note that a program can find the WSDL too, but UDDI provides no search capability for the WSDL.

From examining the WSDL, one can see that:

– WSDL operations have names like "P3Type3"
– The *Input Message Part Name* for this operation is "airlineID"
– The *Output Message Part Name* is "body"
– No sequence of operations is specified.
– There is no specification of what the operations do

This can be seen in another (vendor-neutral) service registry and with a completely different example. Search `http://www.xmethods.com/` for a lookup service for Swedish telephone numbers and *somehow* find `http://www.marotz.se/scripts/searchperson.exe/wsdl/ISearchSwedishPerson`, which has five operations:

[12] `http://www.uddi.org/specification.html`

– “HTMLSearchAddress”
– “HTMLSearchPhone”
– “XMLSearchAddress”
– “XMLSearchPhone”
– “IsAlive”

The *input message* name is “HTMLSearchAddressRequest.” That message has part names of:

– “fName”
– “lName”
– “Address”
– “ZipCode”
– “City”

It will be very difficult for automatic software in real time to understand what these terms mean. And the *output message* has a single part: “return.”

The first and fundamental technical requirement we can see from this is that we require **semantics** to understand the terms in this XML structure. The second fundamental requirement is that we require additional information to be specified in the WSDL. I.e., we need “WSDL++.” As these technologies stand today, **UDDI/WSDL are Inadequate for Search**.

If we consider the last example in thinking about solutions,

– *Airline ID, City* can possibly be unified using distributed ontologies or industry taxonomies.
– The W3C answer is *OWL*[13] based upon *DAML+OIL*[14]

DAML-S may be too “heavyweight” for industry, but some set of distributed taxonomies about the service components, rather than just the businesses and kinds of services is necessary. Industry should develop its own simple taxonomy, starting with UBL[15] as a kernel. In any case, suppose we understand the terms of the operations messages.

4 Understanding What Web Services Do

The FX-Agents project created a set of Web services that performed actions in the world. One of these that we studied extensively was a Catering Service that was created from the Waiter.com Web site. We learned that the required application functions could be expressed as a sequence of WSDL operations[16].

Furthermore, there is at least one WSDL-based technology that can handle the expression of such a sequence. W3C/HP *WSCL* can describe the sequence of operations

[13] `http://www.w3.org/TR/owl-guide/`
[14] `http://www.w3.org/TR/daml+oil-reference` based on the Resource Description Framework.
[15] `http://www.oasis-open.org/committees/ubl/`
[16] `http://fxagents.stanford.edu/~gomi/CateringService.wsdl`

1. Get types of restaurants
2. Select type, get names of restaurants
3. Select restaurant, get menu
4. Select menu item, get set of choices (like pizza toppings)

So UDDI/WSDL needs something like WSCL. Then, what else would be required? The FX-Agents project team noted the following issues were not addressed by WSDL-based technologies:

- **Unplanned options** are not handled by WSDL:
  Preplanned sequence specifications, such as WSCL, are inadequate. For example, *id = pizzas* is a special case that may require a choice of toppings, that change often, and these values may affect the proper process sequence.
- No standard representation of **authorization** mechanisms.
  The membership registration is required for this case, but WSDL provides no standard way of discovering this.
- No standard representation of **cancellation terms**.
  By what hour/day could one cancel with what penalties?
- No standard representation of **effects**.
  What are the specific actions of this service? In this case, food ships 90 minutes after order. And there is a phone call and email sent.
- No standard representation of **pre-conditions**.
  - The minimum order is $80.
  - Delivery hours are different than takeout hours
  - No deliveries on Mondays.
- No standard representation of **payment terms**
  - There is a delivery charge of $8.95.
  - A 15% driver support charge is added to each order.

From this simple case, we can see that as a minimum, we need to be able to represent in machine-readable format:

- Pre-conditions and actions of each operation
- Tags that refer to protocols/terms of
  - Payment
  - Cancellation
  - Authorization

  and
- Representation for dynamic options

The W3C/DARPA *DAML-S*[17] is one solution to representing the preconditions and side-effects of Web services. DAML-S

- Provides an ontology of services and processes
- Is based on DAML+OIL/RDF
- Is being integrated with UDDI
- Could be too "heavyweight" but does address these issues

[17] `http://www.daml.org/services/`

– Does not provide distributed search

And, importantly, because it uses the current UDDI T-Model mechanism, it only describes pre-conditions and effects of the whole Web service, but **not of the individual operations.**

To really implement our democratic and just-in-time service principles, ordinary Web services are not enough. If we augment them with pre-conditions, effects, payment methods, authentication, etc., then we have what we may call **Business Services**. The FX-Agents project aims to show how this can be done, especially for financial services using logic-based technologies.

5 FX-Agents Approach

The FX-Agents project took a different but compatible approach. We built a **Business Service Directory (BSD)** based on Infomaster [2] that provides distributed search using logic as "glue." In addition, we have constructed a prototype *Internet Naming Service (INS)* to provide universal non-ambiguous resolution of all names. And finally, the prototype *Kiosk* provides the semantics for terms used in the descriptions of the services and their operations in the BSD.

In building the BSD, we wanted for humans to be able to use it as a *Web service browser*. In addition to the other defficiencies of WSDL, it does not contain enough information to dynamically generate a Graphical User Interface (GUI) for invoking Web services. So the group developed the Web Service GUI (WS-GUI) Engine [1], which does allow dynamic GUI Generation.

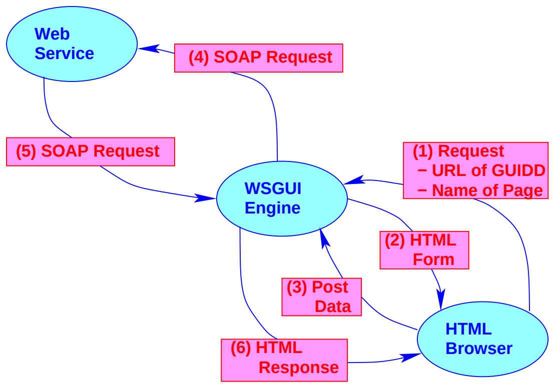


Fig. 1. WSGUI Architecture

In particular, WSGUI

– allows encoding of an *abstract interface*

– combines with stylesheet for concrete interface
– provides pull-down menus and similar standards
– enables *dynamic enumeration specification*
  E.g., new pizza toppings
– and allows *virtual operations* to be defined that allow Web services to be composed and invoked with default parameters.

6 Incremental Automation

With the ability to browse Web services, we investigated the issue of how people could easily build *macroservices.* These are an integrated set of services connected by a process. Unlike industrial approaches, our process will not be completely defined prior to runtime. A *macroservice* is a process of integrated Web services that can be encapsulated as a type of business service, but would be difficult and perhaps impossible to encapsulate as a simple Web service, because of the complexity of the messages that may be required to accomplish the task. I.e., we want to avoid the necessity of claiming a simple Web service encapsulation is possible for the complex processes we are investigating.

The major point here is to determine what information software agents would need to integrate services in a dynamic process, but also be able to easily be instructed by people about this integration. Before we program software agents to use Web services, we should understand how a person could do it.

Suppose we wanted to compose a set of Web services, and do so incrementally over a period of time as we discovered more useful services from the BSD. What are the issues involved? To investigate this, we constructed an application and populated it with Web services.

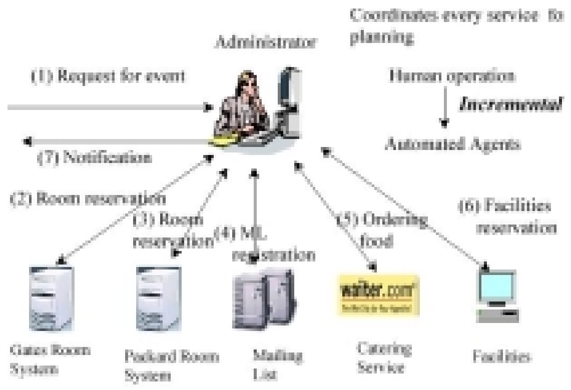


Fig. 2. Event Planning System

We decided upon *event planning* as a macroservice application as it was one that we understood. We further constructed Web services from existing Web-based services. The result is a set of SOAP-accessible services, with WSDL, and also a GUIDD, new format for specifying presentation using WSGUI, that allowed dynamic Web generation, and additional information in the BSD about pre- and post-conditions, as well as logic axioms, that allowed these services to be “glued” together easily.

Only the Gates room reservation system, already build on Infomaster, had an existing SOAP interface. The rest were constructed by initially screen-scraping, and then building a SOAP wrapper that sends HTTP messages to the original Web-based applications as if they were being sent by a Web browser.

The reader may ask the difference between this and a registry of existing Web-based services, since the latter already have Web interfaces defined. The difference is that the information about the Web services, and their interfaces, is now declared in a machine-readable format. Further, the presentation has now been separated from the application semantics. We thus built a *sandbox* for agents and Web services.

7 Steps in Automating Web Service Composition and Integration

Our strategy for producing the technology for agent-based Web service integration may summarized as follows:

Web Service Browsing by humans
(using BSD and WSGUI)
$\Longrightarrow$
DECLARATIVE WEB:
Incremental Automation
(using logic hidden by GUI and virtual operations)
$\Longrightarrow$
Agent-Mediated IA:
Software Agents consuming and integrating Web services
(using wrappers and logic-based interactions)
$\Longrightarrow$
Condition-based Integration
(service description allows dynamic planning)

Fig. 3. Sevice Integration Strategy

As we moved beyond the Web service browser capability, we entered a broad area of capability and approach we call the *Declarative Web* in which we take advantage of the fact that we had not only made machine-readable descriptions of the Web services, but have used logic to declare what the Web services do. We increasingly take advantage of this declarative logic in the sophistication of the integration of Web services.

8 Incremental Automation is an Alternative to Industry Approaches

In our next steps, we would like to provide a more **dynamic alternative** to the industrial standards for Web service integration and composition. As described in [3], these approaches are inadequate as they:

– require a static process model;
– are not easily changed, maintained, or controlled; and
– change requires coding of new WSDL operations by the service provider.

Such alternatives strongly violate our democratic and just-in-time principles.

As a first step beyond a Wizard and towards sophisticated dynamic Web service integration, we can note that if integrated applications require *some* coding in advance, then we can at least we can take advantage of Web services, software agents, and computational logic and avoid coding of new Web services by the service provider as is required with, say, BPEL4WS.

As indicated above, the first step of Incremental Automation is to provide an overall GUI that, together with Virtual Operations describing some logic functionality, allows the user to manage macroservices. Initially, we constructed a *"Wizard"* to guide humans through the use of the event planning services to create a new event. We used this application to organize at least one Steering Committee Meeting, including ordering lunch.

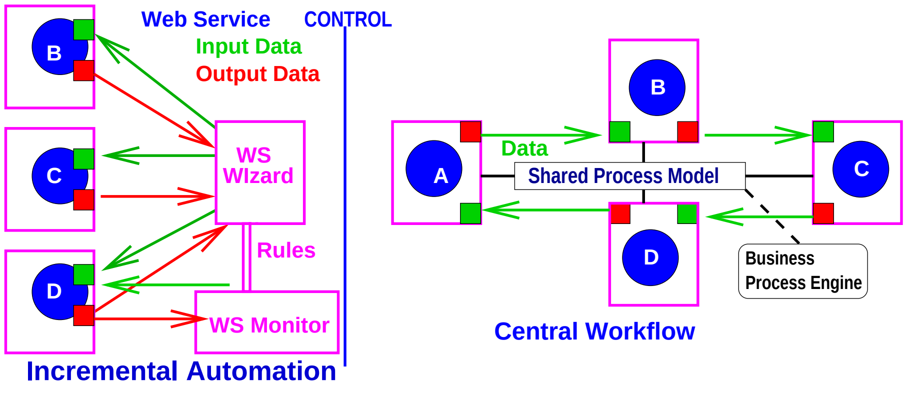


Fig. 4. Incremental Automation

The Immediate advantages of the IA/Wizard approach are that no **service-side** code changes are needed to add new services, and the user has complete control of the integration, instead of the process being controlled by a workflow engine of some kind. Further, it is now easy to provide a monitoring capability so that Web services are invoked whenever some condition changes, detected by polling the Web services every so often. Nothing needs to be changed on the service provider side and our democratic and just-in-time principles are preserved.

We intended further to build a human interface for building rules to integrate the Web services as a next step in Incremental Automation. Since each Web service operation has a set of well-defined inputs and outputs, as well as pre- and post-conditions, it is possible to display each operation as a table of parameters, with inputs and outputs. Then one can use a pull-down menu of operators and relations to create conditions among these parameters, including critical values (numbers or symbols pulled from the XML schemas) with actions being to transfer output values from the invocation of one Web service to another.

The only hard part about this idea was design of the user interface. While working on this, we decided to skip it and go directly to the next stage of service integration automation. *It remains as an easy idea for someone to implement and could result in a commercial product.*

9 Agent-Mediated Incremental Automation

The Wizard can guide the user in direct manipulation and integration of Web services. What we would like to do next is to construct more of a *"cockpit"* for the user.

Fig. 5. Cockpit Analogy

We want the user to be able to refer to a system of controls that perform actions in the world, which are not directly related to the services themselves, but directly related to actions of the enterprise the user is running. These actions are then related to a composite of Web service actions. I.e., the user sees complex actions directly related to what is desired, just as a pilot desires to "fly North at 10,000 feet" and this is enabled by a sequence of flaps and motor changes that correspond to the primitive Web services require to "make it so." Our approach is to use software agents to mediate between the user cockpit controls and the actual Web services.

Because we ultimately wanted to work on a financial system of some kind, we chose to implement a **Multi-Bank Cash Management System**. In this application, the "cockpit" is a Corporate Treasury Workstation. The Treasurer is the user and desires to manipulate the various bank accounts of the enterprise so as to optimize the cash flow, including managing Accounts Payables and Receivables.

This process is informed by a standard mathematical model called the *Miller-Orr model*, so we needed a specialized agent that, given the required inputs, would return a target distribution of cash among liquid and interest-bearing accounts. We also required a *Plan Revision Agent* that would generate and revise plans for transferring cash at different dates during the target period of time.

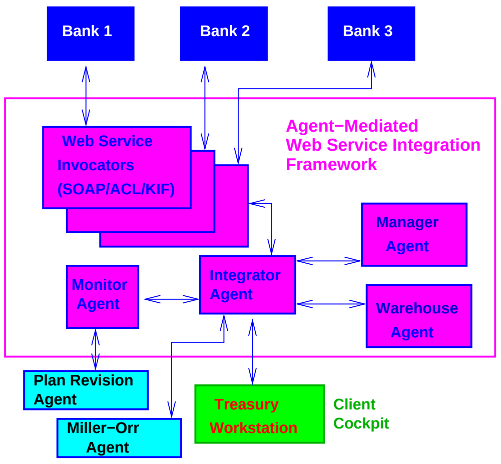


Fig. 6. Cash Management System

As can be seen from Figure 6, these were two special agents. Also the Treasury cockpit had to be designed especially for the application because the GUI has to be special in particular. The banks are assumed to have existing Web services for account information and transfer. With respect to this application, such Web services are *legacy systems* even though at the time of this work, they are just being constructed by real banks.

The mauve-colored agents in between the special agents and the legacy Web services comprise a reusable *Agent-mediated WS Integration (AWSI)* system and comprises a new type of Declarative Web.

– AWSI Agents "hold" KIF knowledge and are reusable.
– There is no coding by service providers.
– The agents speak an Agent Communication Language (ACL) among themselves and are capable of arbitrarily complex behavior, including negotiation.
– And users, such as a Corporate Treasurer, can "program" using IA rules.

Further contributing to the reusuability and easy development of the agents is that they are built on top of the reusable JKIP platform.

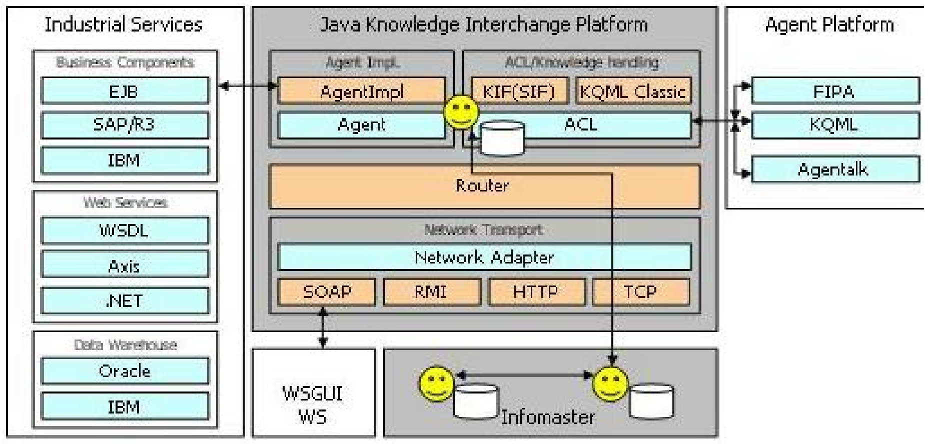


Fig. 7. JKIP Architecture

The Java Knowledge Interchange Platform (JKIP) is another technology from this project that allows agents to be easily built from Java templates and run inside one computer with a single IP address, but separately addressable from the outside.

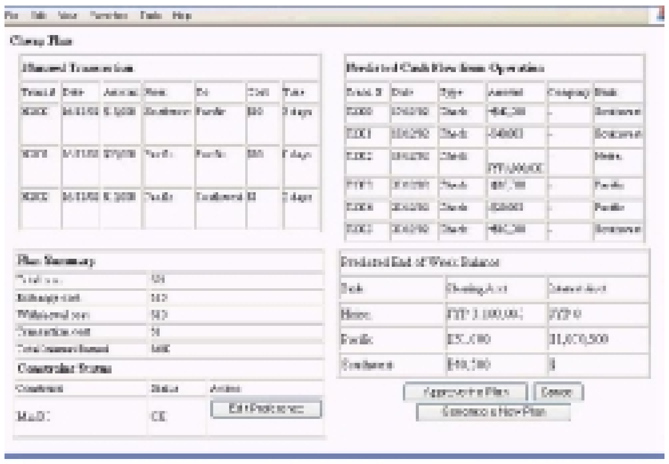
That the AWSI agents use **Logic-Based Integration** also makes them reusable. They each are capable of holding whatever axioms and statements are useful to any application. The **Integrator** agent theorem-proves over results from other agents and the **Manager** agent knows what predicates handled by what agents. The Manager agent uses the special predicate *Specialty* to relate task predicates to the application agents, such as:

- `Specialty convert FX`
- `Specialty plan Monitor`
- `Specialty bank.units Warehouse`
- `Specialty balance Warehouse`
- `Specialty minimum.balance Miller-Orr`

An example of the kind of information the Integrator agent has for this application is:

```
(≤ (request-transfer ?amount ?bank1 ?bank2 ?account)
(> (- (balance ?bank1) ?account) (min-balance ?bank1 ?account))
(= (bank.units ?bank1)(bank.units ?bank2))
(do (transfer ?amount ?bank1 ?new-amount ?bank2 ?account)))
```

This logic[18] states that a requested transfer of an amount from any bank1 to any bank2 should be done, if this will not cause the balance of the bank1 to drop below its minimum, and if both banks use the same units of currency. If the last condition does not hold, then another rule will come into play that ensures the best conversion of currency.



[illegible] Plan

Planned Transaction						
Trans #	Date	Amount	From	To	Cost	Time
[illegible]	[illegible]	[illegible]	[illegible]	Pacific	$[illegible]	2 days
[illegible]	[illegible]	[illegible]	Pacific	Pacific	$[illegible]	[illegible] days
[illegible]	[illegible]	[illegible]	Pacific	[illegible]	$[illegible]	2 days

Predicted Cash Flow from Operations					
Trans. #	Date	Type	Amount	Company	Bank
[illegible]	[illegible]	Check	[illegible]	-	[illegible]
[illegible]	[illegible]	Check	[illegible]	-	[illegible]
[illegible]	[illegible]	Check	[illegible]	-	[illegible]
[illegible]	[illegible]	Check	[illegible]	-	Pacific
[illegible]	[illegible]	Check	[illegible]	-	Pacific
[illegible]	[illegible]	Check	[illegible]	-	[illegible]

Plan Summary	
Total cost	$[illegible]
Exchange cost	$[illegible]
Withdrawal cost	$[illegible]
Transaction cost	$[illegible]
Total Interest Earned	$[illegible]

Constraint Status		
Constraint	Status	Action
[illegible]	OK	Edit Preferences

Predicted End of Week Balance		
Bank	Clearing Acct	Interest Acct
[illegible]	JPY [illegible]	JPY 0
Pacific	$[illegible]	$1,070,500
[illegible]	$[illegible]	$[illegible]

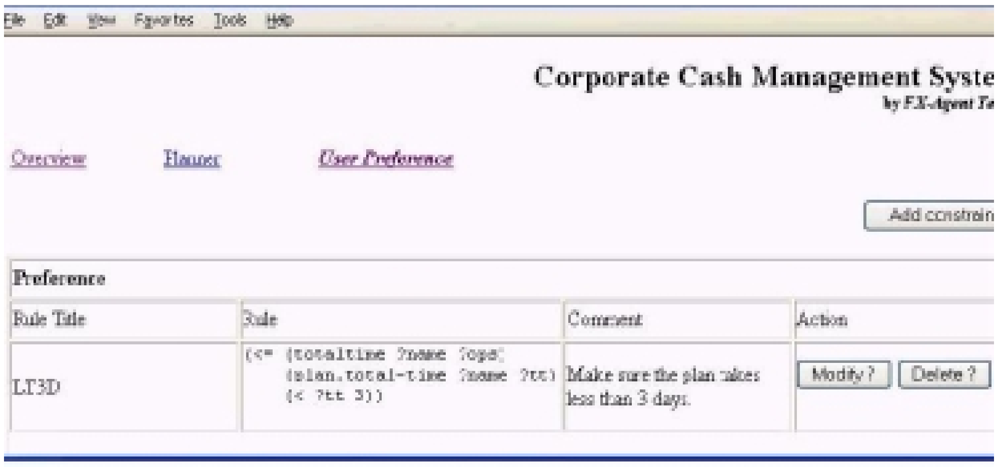
Fig. 8. Cash Plan

This is an example of a **Cash Plan** that may be displayed at the Treasury Workstation. Here, we do not address how such a plan is prepared. It may be done by a standard Integer Programming system. The first point is to encapsulate such systems as either Web services or agents. Next, we would like to make it easy for the Treasurer to create incrementally constraints on the rules while asking for a change to the plan.

10 Cash Plan Change Becomes a Rule

Logic allows us to create arbitrarily complex requests for changes to the cash plan (and to format such requests in the language that the cash plan mechanism understands), and also to record such requests as rules. The hardest part is to develop the appropriate GUI for this capability. GUI development was not a focus of our research. The above example is a very simple way to represent constraints over single variables as a result of plan changes. In this example, the Treasurer would like the duration of the total plan not to exceed three days and this is simply expressed in a KIF rule. It could also have been expressed more simply in an advanced GUI by allowing the user to select from a pull-down

[18] In this notation, "?" denotes a logical variable that will be unified.



Corporate Cash Management Syste
by FX-Agent Te

Preference

Rule Title	Rule	Comment	Action
LT3D	`(<= (totaltime ?name ?ops) (plan.total-time ?name ?tt) (< ?tt 3))`	Make sure the plan takes less than 3 days.	Modify ? Delete ?

Fig. 9. Plan Change

menu of all variables (selecting “total-time” in this case), selecting a relationship (“<”), input a number (3), and select units (“days”). The conversion to KIF on the backend is trivial. But more complicated rules would require a more complex GUI. We simply left this as an SEP (Someone Else’s Problem)

The important point is that we can write rules that would be interesting to a Treasurer. For instance:

– If the cash balance at a bank will fall below the minimum for that bank if payment is not received by an account known to be unreliable, then make a contingency plan.
– If the cash balance at a certain bank will fall below the minimum, find a high-interest account to transfer from.
– If money needs to be converted from Yen to USD, find the best rate, including transfer fees.

Such rules can be easily expressed in logic, with logical unification providing the “glue” among constraints and Web Service preconditions, postconditions, and axioms. Furthermore, they can refer to conditions in the world, such as the current US treasury index, and make actions happen. This can be done today with procedural encoding. The point here is that logic is a much more flexible way to encode such business logic, and is more easily maintained.

11 Declarative Web: Condition-based Integration

Our final step takes full advantage of the fact that we have business services, with pre-conditions and effects, rather than just Web services.

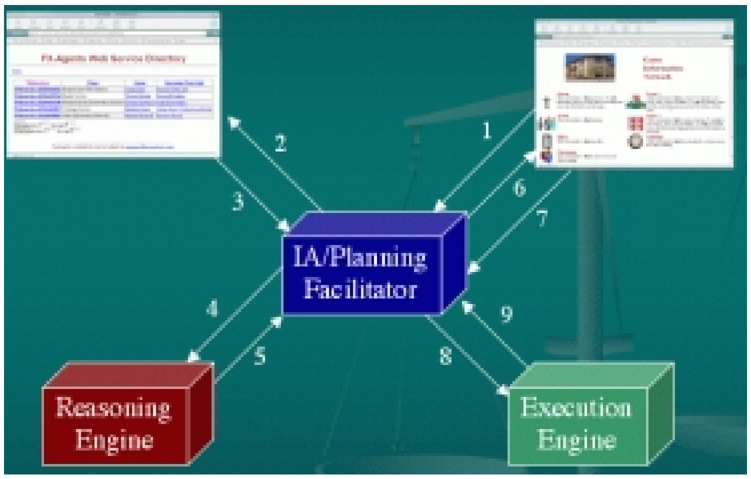


Fig. 10. Architecture of Logic-Based Integration

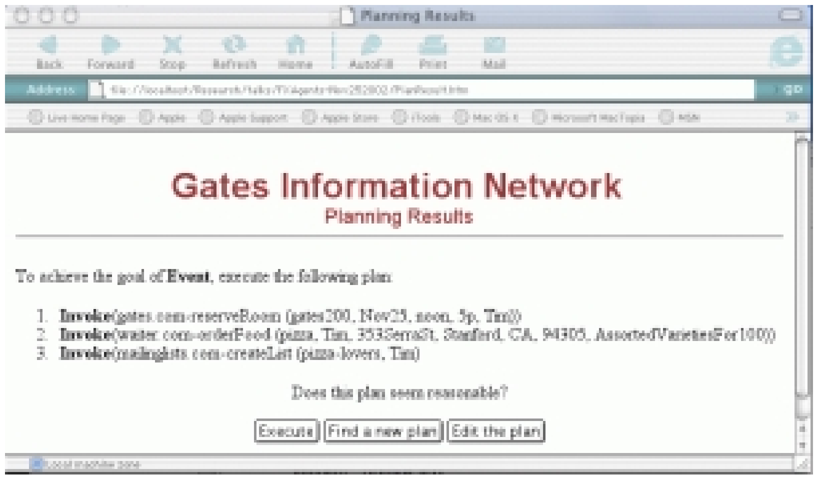
We have constructed this prototype logic-based integration of services and tested it with the event planning system (we have not found an appropriate financial application though the home mortgage process is one that we have examined and remains a candidate.) With such a system, we have **any** possible integration of **any** registered services
for **any** application, **anytime**.

This works as follows. A user requests that some large action be taken in the world, such as preparing for an event. The user simply says something like "make it so." This request is formatted and passed on to an Incremental Planning Engine and Facilitator in step 1. In steps 2 and 3, a set of services are extracted from a BSD, using the pre-conditions and effects of the services. This set of actions and associated axioms are then passed to a general theorem prover in step 4 and a plan is returned in step 5.

In steps 6 and 7, the user reviews and approves the plan, or asks for a change. The central planner and facilitator then requests execution of the plan and monitors the execution.

The "magic" is that we are able to take advantage of a suite of well-understood AI planning technologies for sequencing actions in a consistent way to achieve complex goals. With business services, we can move from specialized systems (military logistics for instance) to a general way to run business systems in the real world.

However, there is an important piece of this that we are skipping over. Such a general solution requires semantics and naming from INS/Kiosk, which are still under development. The good news is that not only the Stanford Logic Group but also a large number of both academic and industrial groups are attempting to solve the problem of semantic integration.


Gates Information Network

Planning Results

To achieve the goal of **Event**, execute the following plan:

1. **Invoke**(gates.com-reserveRoom (gates200, Nov25, noon, 5p, Tim))
2. **Invoke**(waiter.com-orderFood (pizza, Tim, 353SerraSt, Stanford, CA, 94305, AssortedVarietiesFor100))
3. **Invoke**(mailinglists.com-createList (pizza-lovers, Tim))

Does this plan seem reasonable?

Fig. 11. Plan Result

12 Good Research Issues Opened

I.e., lots of hard work to be done.

– How to make a real usable Business Service Registry based on the BSD
– How to integrate AWSI systems, more dynamically?
– How to account for *real business processes?*
– How to make sophisticated GUIs for this technology?
– How to generate logical constraints from simple GUIs?
– How to generally *monitor and* **control***?*
I.e., consider a recursive value network and the unsolved problem of controlling it while protecting information access.

This last research issue involves not just advanced development, but must address difficult computer science issues of distributed computing with robustness and security as well as control of information transparency.

13 Acknowledgments

The FX-Agents project concluded in March of 2003, but this research and development continues, most notably in cooperation with and supported by CommerceNet. We thank not only CommerceNet but also Christoph Bussler of Oracle for help in our Web service technology analysis.

References

1. M. Kassoff, D. Kato, and W. Mohsin, “Creating GUIs for Web Services,” *Internet Computing*, September/October 2003, **7** 5.
   Available at `http://logic.stanford.edu/~mkassoff/papers/wsgui.pdf`.

2. M. Genesereth, Keller, and Duschka, “Infomaster: An Information Integration System,” *Proc. 1997 ACM SIGMOD Conference*, May 1997.

3. C. Petrie and C. Bussler, “Service Agents and Virtual Enterprises: A Survey,” *Internet Computing*, July/August 2003, **7** 4.
   Available at `http://snrc.stanford.edu/~petrie/fx-agents/xserv/icpaper/`.

4. S. Vinoski “Web Services Interaction Models Part 1: Current Practice,” *Internet Computing*, May/June 2002 **6** 3, pp. 89-81.