Information Agent Design Notes

Keith S. Decker

January 30, 1996

1 Overview

An “information agent” is a software agent that is closely tied to a source or sources of data, as opposed to being tied closely to a human user’s goals (so called “interface agents”), or the processes involved in carrying out an arbitrary task (so called “task agents”). In general such distinctions are necessarily part of a spectrum, but in this document we use the term “information agent” to denote a specific class of implemented agents with certain input/output/process behavior.

aside: [possibly the main diff is very shallow task structure, allowing certain efficiencies in meta-control, i.e. the planner becomes a query optimizer, and we can use a fairly specialized RT decision-theoretic scheduler]

This document will detail the design of a basic information agent class that provides two general behaviors:

- “Query-answering” behavior where an information agent responds to non-time-contextual queries, such as “what is the recent price of IBM” or “what where the earnings of IBM for the 3rd quarter of 1990”. Note from the examples that temporal information may be part of the query, but not the query context (other than the fact that all queries have an implicit “recent” or “current” context—what is your current information about X, reply ASAP). Contrast this with monitoring behaviors, below. Query behaviors are in response to KQML messages such as ask-one, ask-all, and stream-all.

- “Monitoring” behavior where an information agent handles queries with a temporal context, such as “tell me the quarterly earnings consensus for IBM every morning” or “tell me within 10 minutes if the price of IBM drops more than 25% of its current value”. Monitoring behaviors are in response to KQML messages such as monitor, cancel-monitor, and alter-monitor. An information agent, by accepting a monitoring query, takes on a clear organizational role (long-term commitment to a class of actions).

1.1 Levels of Service—Agent Competencies

In the design of such information agents, we need to keep in mind not only input/output behavior, but process behavior as well. A particular agent architecture instantiation may advertise externally various levels of service, which indicate certain internal agent competencies. Some of these services (in no particular order) are:

- basic—the default competency to answer simple queries; none of the following things can be assumed.
- monitoring conditions—basic query agents might not handle monitoring requests. Monitoring agents (information agents with the monitoring behavior) may provide information from a database at a specified frequency, or only when information in the database has been added or changed (within some specified time period). Meta-requests, such as monitoring changes in the underlying database schema, are also possible.
- relational queries—basic agents only handle flat file DB queries and simple triggers. More complex agents can deal with relational queries crossing multiple tables and multi-condition monitoring triggers.
- reliability/multiple sources—the agent has access to multiple sources or multiple methods to increase reliability and responsiveness.
- uncertainty—the agent can deal with an uncertainty representation both for limiting queries (i.e., need data of a certain level of certainty) and marking the replies. For example, reasons to believe or disbelieve, sources of uncertainty, and numerical summations are all possible representations.
- coordinated retrieval of multiple requests—the agent can handle the service of multiple requests from multiple agents. Implies soft real-time and result commitments (even for simple queries)
- data integrity—the agent uses some model(s) to check data before it has been returned, possibly adding it's own certainty stamp (i.e. does this data make sense?)
- hard real time guarantees—the agent guarantees monitoring services (on a request-by-request basis) in the hard real-time sense of the term.

Obviously, agents can offer multiple competencies. I doubt if we will be able to mix-and-match them at compile time, however. In the long run, customer/client-agents might also pay for different levels of service.

1.2 General Design

We will view the overall task of a basic information agent as the maintenance of, and query processing for, a single flat file database. We will eventually extend this to allow multi-table, relational queries. Actually, since this limitation is being imposed only on the interface that the information agent advertises to the external world, and not to the underlying data sources, this is not quite as serious a limitation as it may first appear. For example, a stock ticker agent may advertise a single table of company name, ticker symbol, price, and time but actually retrieve the ticker symbol using the company name as a key in external database 1 and then retrieve the price using the ticker symbol as a key in external database 2.

In any case, limiting the agent's task environment to database processing allows us to make a host of assumptions in fixing the details of the agent's internal architecture. Using the general DECAF architecture as a guide:

Planning: basic information agents do not plan in the classical AI sense. However, agents with higher competencies (multiple sources, multiple requests) might do a bit of query optimization (grouping similar or identical queries, for example). Katia has proposed grouping all periodic queries (on a single symbol) into a single query that would be run at the shortest requested period.

Coordination: basic info agents do not need a coordination component. Agents that provide a coordinated retrieval service should be able commit to when a query response will be available in the standard GPGP way. They should also be able to handle queries that are related by coordination relationships. *aside: [Such basic agents are really operating within implicit commitments—even a fancy* TÆMS *agent scheduler could interact with such an agent by explicitly marking the implicit commitment.]*

Scheduling: basic query agents do not need complex scheduling, but by the nature of the task even an basic monitoring agent will need a rudimentary periodic task scheduler. The problem can be made much harder by requiring hard guarantees. *aside: [We should eventually think about allowing some threading/multiprocessing here to alleviate net lag, but not right away!]*

Decision-making: not needed by these fairly simple agents. For readers unfamiliar with DECAF, an agent's decision making component chooses what utility criteria the agent will attempt to maximize at the current point in time. For example, often an agent's local scheduler will return several possible schedules that maximize different utility criteria, and the decision-maker chooses between them.

Execution monitoring: basic info agents will need to timeout on network connections, deal with moved or altered information sources and so on. Also data collection for modeling the sources themselves (i.e., reliability, response time, etc.)

Belief database data structures: There are three major components to an information agent's beliefs: First, the Local DB copy of whatever External DB(s) the agent is advertising services for. This can be thought of as a table with extra attributes, such as the previous value of each field and a timestamp. The second important structure is the agent's Task Structure Database: What requests has the agent received, and how is the agent planning to handle them? Part of this structure can be viewed as the set of actions that the agent needs to schedule and execute. The third important structure is the agent's stored Plan (or behavior) Library. This includes information on how different requests can be serviced, and what domain-specific code is available (along with details like how long it takes to run, etc. (i.e. duration and quality data)).

Planning, scheduling, execution, and the major data structures are described in much more detail in Section 3.

1.3 Architecting for Re-Use

For the basic competency level, the idea is to provide an agent template, a simple query language, a database definition language, and a simple plan template language. To make a specific information agent, one would then provide the agent template with a DB definition, a set of site-dependent methods for filling the rows of the table, and a small set of plan templates. Everything else should happen automatically. One would imagine that the site dependent executable methods can be re-used to a great extent (i.e., getting the "abc" field from URL1 is about the same as getting the "QED" field from URL2); see also Choon's table parser. The plan templates will be almost completely reusable, except that they'll call different site-dependent executable methods and have different durations, etc.

1.4 The Format of the Rest of This Paper

The next section covers the external information agent interface (i.e., the database schema definition language and the query language used inside the KQML :CONTENT slot). Section 3 then covers the basic internal architecture of the information agents. Section 4 discusses how to program information agents. Section 5 goes on to discuss how that architecture is expressed in our actual Perl5 implementation. Finally, Section 6 gives some examples of information agents, their database definitions, and possible queries, drawn from the financial portfolio management domain.

2 External Information Agent Interface

This section will describe the external interface that an information agent presents to the world of other agents. The key components of this interface, beyond KQML messaging, are the agent's advertised database definition and the concomitant simple query language (what goes into the KQML :CONTENT slot).

2.1 Database Definition Language

We view a simple flat-file database as a table with concept-labeled columns (fields), where each row is a record. Each database has an ontology associated with it that links concept-names to their data types (and must match on any incoming KQML message). The database also has a set of attribute-names associated with every entry in the database (usually a timestamp and the previous value, but it could be more than that). The attribute types will also be drawn from the database's ontology. Each database definition also includes what query languages can be accepted (currently only the simple-query language, see Section 2.2.1). Each column of the table (field) then has these descriptors:

concept-name: the column or field name, the term referring to what the value in the database represents. Concept-name plus the ontology implies a data-type (string, integer, float, data/time, enumerated, etc.)

advertised flag: A field is flagged "advertised" if it possible to use that field to select records in a query. If the external database is a full-fledged modern relational DB, than usually all fields will be flagged "advertised". However, many of the databases available over the WWW do not allow record selection by every field, and so the agent needs to include this in its specification. An agent can also intentionally not advertise a field if it would be extremely time-consuming to do so (for example, an agent working with a real database might choose to set the "advertised" flag only for indexed fields). In short, every query (see the query definition below) on a database must include at least one legal clause involving an advertised field. For example, we do not want the stock ticker agent to advertise that can handle a query for "all stocks that cost more than $100" (see Section 6.2).

unique flag: A field is flagged "unique" if every record in the database must differ on that field. Used to automatically determine if a record is new or an update. Every database definition must have at least one unique field. This field does not have to be advertised. In certain applications, it could in fact be internally generated; but that discussion is beyond the scope of this interface specification.

predicates: an optional field. If present, it limits the predicates that can be used in queries on this field. If absent, all of the natural predicates associated with the data-type for the concept-name can be used (could store this info in the ontology). Typically, a key field might limit the operators to EQUAL only.

range: An optional field gives information on any range limits associated with the field in question. If not present, the natural limits for the field's data type are assumed.

Thus, something like this will work to specify a database and schema:

```
(DATABASE <name>
  :ONTOLOGY <ontology>
  :ATTRIBUTES <attributes>
  :QUERY-LANGUAGE simple-query
  [:MONITORING-COST (< 15 minutes, infinite)]
  :SCHEMA
  (<concept-name> [:ADVERTISED]
                  [:UNIQUE]
                  [:PREDICATES <predicate>+]
                  [:RANGE <range-spec>])
  ...)
```

Note that there could be new fields as we build more complex agents that deal with the new external levels of service. In the example above, we've proposed that a database could advertise different costs for different monitoring intervals (the smaller the interval, the more expensive).

Note that I've linked a DB and a single schema; if we really think that schemas will be shared, then we could unlink them, but I'm not sure that would be too common...

2.1.1 Database Attributes

Attributes are meta-information stored about a field in a database record. By default, our information agents will keep track of two attributes: timestamp and previous value. The timestamp indicates when the field value in the record was last updated.[1] Note that when a database is formed from multiple underlying databases or information sources, the timestamps on every field n a record may be different. The previous value is the value the field had before the last database update. Very important: *No Absolute Temporal Relationship Between The Current Value and Previous Value Can Be Assumed*. Only the relative relationship ("before") can be assumed. This places a limit on the types of triggering queries that are possible. You cannot (directly) say "tell me if IBM's price rises more than $10 over 1 hour". You can say "tell me if IBM's price rises", or "Tell me IBM's price every hour". From the second query another

[1] The timestamp refers to the record update time. The confusion occurs when records represent time-stamped information, like a stock quote. Thus a stock quote record might be updated at 13:25 (the local DB timestamp), and the record may contain the quote "$23 1/2" *and* the time of the quote "13:05"—free quotes are often delayed by 20 minutes.

agent can construct the desired composite trigger. Another example would be using a second agent to create a composite trigger for “Tell me if IBM’s price crosses it’s 200-day moving average”.

The reason for this limitation is simplicity. In the future, we may extend the default implementation to provide a new level of service, the “history” database attribute. The history attribute allows one to construct queries that rely on arbitrary historical value changes in a database field (essentially, all record field value changes would be recorded and timestamped).

2.2 Queries

aside: [Anandeep's triggering agents have three kinds of triggers; I'd like to try folding record addition and field updates together, and leave schema-alteration as something separate]

In general, as mentioned earlier, there are two types of interactions that one can have with an information agent: simple queries and monitoring requests. Since any simple query can also be used as a monitoring request, we will discuss simple queries first.

2.2.1 The Simple-Query Query Language

A simple query on a database, given its schema, is a set of predicate clauses joined by an implicit AND (OR clauses can be handled by using separate queries)used to select a record or records from the database. At least one clause must be on an advertised field (as defined in the database schema), and the predicates must obey any restrictions imposed by the schema (such as equality testing only). So we get something like (could define a BNF for this):

```
(ask-all
 :sender <x>
 :receiver <y>
 :language simple-query
 :ontology <ontology>
 :reply-with <reply-tag>
 :content
  (query <database>
   :clauses
   (<predicate> $<concept-name> <arguments>)
   (<predicate> (<attribute> $<concept-name>) <arguments>)
   ...
   :output :ALL
   ...)
```

Simple query rules:

1. The first clause must refer only to a single, advertised field using a legal predicate.

2. (we may relax this one fairly easily) Each clause can only refer to a single field and its attributes.

Queries may contain an output specification, which tells what information to report back once some set of records have been selected. The default output specification, mentioned above, is :ALL (for every matching record, report all fields and values).

If the user desires attributes to be reported for all fields, then use the output specification `:OUTPUT :ALL <attr`
Then every field will be reported with its value, and the values for each attribute (such as TIMESTAMP and/or PREVIOUS VALUE).

Finally, the user may specify the output fields individually:

```
:output
 (<field> [<attr1> <attr2> ...])
 ...
```

which means only output the desired fields, values, and listed attributes (which could be different for every field).

If both the `:ALL [<attr>*]` tag and individual tags are present, then all fields are output with the global attribute list, EXCEPT for fields that are overridden by a specific output spec. Thus

```
:output :ALL
 (price previous-value timestamp)
```

in a query for the security-apl stock database (Section 6.2) would return all fields and values, but additionally would return the previous value and timestamp for the "price" field.

The reply (for each record) is a simple list-of-lists:

```
((<field-name> <value> [<attr-name> <attr-value>]*)*)
```

We still have the option to eventually allowing some full query language like SQL at some point in the future, but it seems like overkill for a flat file DB.

Queries that would return multiple records behave as specified by the wrapping KQML statement (ask-one returns only the first record, ask-all returns a list of all matching records, stream-all returns each record in a separate message).

2.2.2 Monitoring Queries

The canonical monitoring queries would be (the first runs the query every hour, the second one is a trigger on the change of a field value (could as easily be a threshold, relative change, etc.)):

```
;; run a query every hour and send the result to me, whatever it is
(monitor
   :sender <x>
   :reliever <y>
   :language simple-query
   :ontology <ontology>
   :reply-frequency (1 hour)
   :content (query <database>
               :CLAUSES (eq $<key-concept> <value>)
               :OUTPUT :ALL))

;; notify me within 15 minutes if the result of the query changes.
(monitor
   :sender <x>
   :receiver <y>
   :language simple-query
   :ontology <ontology>
   :notification-deadline (15 minutes)
   :content (query <database>
               :CLAUSES
               (eq $<key-concept> <value>)
               (!= $<concept-name> (previous-value $<concept-name>)
               :OUTPUT :ALL))
```

The keywords :REPLY-FREQUENCY and :NOTIFICATION-DEADLINE can be used together. The :REPLY-FREQUENCY keyword is used to force the execution of a query at specified time intervals. The :NOTIFICATION-DEADLINE keyword indicates that the agent only needs to know the result of the query within the specified time *if the result has changed*.

Note that a query does not have to reference an attribute such as "previous-value" in order to make sense for a notification-deadline monitoring request. In the example above, we use the previous-value attribute to detect a *change in an existing record*. A client might also be interested in the addition of a completely new record, (for example, a new news story). In this case a query that simply matches on the story (with a `(=~ $subject /IBM/)` clause orsomething similar) will work—you're not looking for a change in an existing news story (like you were with the stock price), but rather you're looking for any new news records that get added to the DB and which match your criteria.

Implementation note: There are two types of underlying external databases, at least. The first type is like Anandeep's example—database changes can be intercepted directly (close coupling). The second type are like the Security APL server—they operate at arms length and changes in them can only be detected by polling (loose coupling).

In closely coupled databases, reply-frequency triggers are run every appropriate time period, while notification-deadline triggers only need to be run when the database changes (in some databases, i.e. a blackboard system blackboard, these change events can be limited to precisely the clauses in the query).

In loosely coupled databases, the best we can do is run both frequency and notification triggers at their periods, plus rerunning all notification triggers whenever the *local* database changes.

In both cases, queries can be grouped and organized so as to optimize performance by either making fewer tests or (especially in the case of loosely coupled databases) fewer database accesses.

More details can be found in Section 3.

3 Implementation Details

In this section we will discuss the major and minor data structures used, and the basic tasks the agent will execute. We conclude with an operational example.

There are three major complex data structures: the local internal domain database, the task structure database, and the plan template database.

There are nine major operational tasks, consisting of three groups: the initialization and shutdown tasks; the metacontroller tasks [planning, scheduling, execution monitoring]; and the domain tasks [information-gathering, run-query, register-trigger, send-results].

We discuss each in turn.

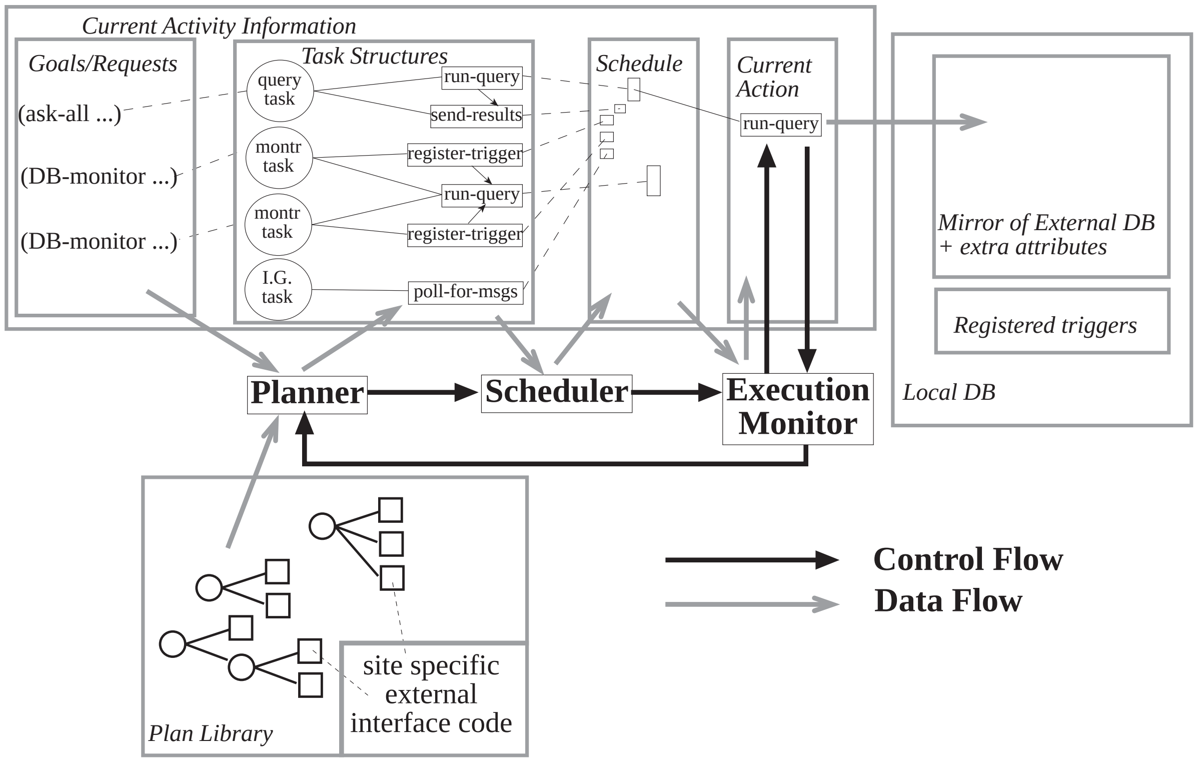

Figure 1: Overall view of data and control flow in an information agent.

3.1 The Local Domain Database [Local DB]

The Local DB comprises:

- Mirrored domain records from the External DB.
- Attributes on each field such as *timestamp*, *previous-value*.
- Attributes on each record, including a list of the task-group roots to which the record has already been sent. When an agent sends a message requesting a service from a second agent, the second agent creates a new *task group* that represents the request in the Task Structure Database (see the next section). Thus every domain action is part of one or more task groups.
- A data structure to store triggers. Agents that request notification monitoring provide a trigger query. These triggering queries are stored (either in a simple list or some more efficient RETE-net-like structure) and checked against any new Local DB updates (see the description of Run-Query below).

3.2 The Task Structure Database [Task DB]

The Task DB comprises:

- Tasks that the agent is trying to accomplish. When a request is sent to an agent, the Information Gathering action creates a task group to represent that request and store all the necessary information about it by creating a new root task (see the description of Info Gathering below). The planner (see below) will use this root task as an index into the plan template DB to retrieve and instantiate a plan for accomplishing the task. In more complex agents, this will be an abstract plan to be continually elaborated, but in the simplest information agents this can be simply a set of base action instances (executable methods).

- Executable methods, or Instantiated Actions, which the agent can schedule and execute (transfer control to). Every abstract plan bottoms out in actual action instances (code and contextual data; subroutine with calling parameters, etc.)[2]
- Relationships between these objects. Yes, the traditional hard relationships (A enables B) but eventually soft relationships as well. Subtasks or executable methods may be part of multiple task groups (i.e., Katia has suggested lumping all monitoring queries into a single Run-Query action instantiation).

Please realize that tasks and executable methods are fairly large objects—the root task stores all of the information from the incoming KQML message (who sent the message? what was the query? how do I reply? what's the deadline/repeat frequency/notification deadline?). Executable methods contain the parameters to be passed to the executing code, plus duration estimates, scheduling periods, etc. (retrieved from the plan template DB and used by the scheduler).

3.3 The Plan Template Database [Plan DB]

The Plan DB first of all includes *declarative* knowledge about the structure of the agent's environment, in the form of *plan templates*. These templates:

- are indexed by the task(s) they can accomplish
- include either what simpler tasks to do, or what individual actions to instantiate
- include info about how to instantiate actions (what resources will they need? how long will they take?).

The Plan DB may also contain procedural knowledge (code) for domain-specific plan optimization (at least until we develop some declarative way of representing such knowledge!).

3.3.1 The Basic Information Agent Plan Template DB

Currently, every info agent starts out with a basic Plan Template DB for the tasks of information gathering, running simple queries, and so on. Here is a list of those top-level task templates, and the low-level task actions they comprise:

[2]In more complex agents, the planner may fail to reach this point, and a coordination mechanism can instantiate a request for help from other agents

Task::Information-Gathering	Action::PollForMsgs
Task::ErrorReporting	Action::ReportError
Task::SimpleQuery	Action::RunQuery Action::SendResults *RunQuery enables SendResults*
Task::TrigMonitor	Action::RunQuery Action::CheckTrigger Action::SendResults *CheckTrigger is added to LocalDB* *CheckTrigger enables SendResults*
Task::RebuildLocalDB	Action::RebuildLDB

3.4 Processes, Actions, and Executable methods

Now that we have briefly discussed the Local DB, Task DB, and the Plan Template DB as the core data structures of an information agent, we can turn our attention to the processes an agent executes and the flow of control. Briefly (see Figure 1), a simple information agent have a fixed meta-controller that calls the agent planner, then the scheduler, and then the execution monitor, who in turn releases control to the next executable method (action instance) on the agent's agenda (as chosen by the scheduler). The thread of control then returns to the execution monitor and finally the loop is repeated.

Small changes in this meta-control architecture can result in increased performance and higher levels of service, but at the cost of making some of the components more complex. We can experiment with these tradeoffs in future work. For example, one extension is to allow multiple concurrent method executions in order to increase throughput. Another extension is to remove planning and scheduling from the meta-control loop, so that they must also be scheduled (the complete hard real-time OS approach)

3.4.1 Initialization and Shutdown

Fairly obvious. Initialization starts the KAPI and registers the agent with an ANS (Agent Name Server), and sets up error handlers to trap fatal interrupts. Shutdown attempts to correctly take the agent offline, including decommiting the agent from all of its tasks, unregistering from the ANS, and stopping the KAPI. Specific info agents built on the basic info agent class may add other specialized actions, such as linking to and writing out persistent data stores (the stock-display agent does this with it's historical lists of stock prices and news headers).

3.4.2 Planning

What behaviors are necessary for what tasks? The planner examines any new tasks that have appeared since it last ran. New tasks might come internally from other agent modules, or externally from a KQML message (processed by the Information Gathering Task). If a new task has no plan, one is retrieved for it. Currently, the only tasks that the planner (really, just a plan retriever) can plan for are incoming KQML messages. The planner uses a table that is indexed by the KQML performative (ask-one, stream-all, monitor, etc). Each performative has a small code fragment that pulls apart the KQML message and instantiates a task to handle the message.

Reply messages are an interesting subclass. A reply is really a "non-local" enablement of some local action. Whenever a task makes a request to another agent (e.g. to monitor a stock price), it also creates an action to receive the reply messages. A unique ID is chosen for the KQML :REPLY-WITH field, and this ID is registered in a second planner table. Whenever a REPLY comes in, then, the planner looks up the :IN-REPLY-TO ID and enables the appropriate existing task action.

3.4.3 Scheduling

Where multiple behaviors come together: ordering, delaying action choice, temporal location. The current scheduler is a simple earliest-deadline-first scheduler. Every task action is placed on the schedule, which is ordered by deadline. The action with the earliest deadline that is also enabled, is chosen for execution. Currently the scheduler does not even idle if the next task is far from its deadline. For practical purposes this doesn't matter much because Poll-For-Messages has a short (1 minute) period, and thus is always executed until one minute before any other task action's deadline. Obviously we can improve on this quite easily, as well as with more complex methods such as rate monotonic scheduling. When a periodic task is chosen for execution, it is automatically rescheduled. Non-periodic tasks are, of course, not rescheduled.

3.4.4 Execution Monitoring

The initial simple execution monitor simply executes the next intended action (as chosen by the scheduler). In practical terms, some piece of code is executed with the supplied arguments, and bookkeeping functions mark down the start and finish time, enable any local tasks that need to be enabled, and so on. Eventually, more complex execution monitoring facilities will allow for concurrent monitoring of task action execution to recognize early failures or possible temporal delays that might effect the agent's overall schedule.

3.5 Basic Information Agent Actions

Note: in earlier drafts and my thesis work, *actions* are called *executable methods*.

3.5.1 Poll-for-messages)

Read incoming KQML messages , create new root tasks.

3.5.2 Run-Query

domain code: some specific to all information agents, some tied deeply to the particular external database. includes updating the local DB, and checking the triggers. If a trigger goes off, it may enable some "send-result" guys just sittin' around waiting...

3.5.3 Check-Trigger

when monitor is requested, this action is attached to the the LocalDB.

3.5.4 Send-Results

Happy day, have an answer/answers, ship them back home. Remember, need to follow the requester's :OUTPUT spec. Need to respond the right way to: ASK-ONE, ASK-ALL, STREAM-ALL...

3.5.5 Others

Other actions include Termination (cause the agent to peacefully Shutdown), SendKQML (send an arbitrary KQML message object), Idle (sleep for n seconds), ReportError (send a KQML error message to an offending party), and RebuildLDB (jettison the current Local DB and rebuild it from the initial schema info; one might do this to periodically clear out space to conserve process memory).

3.6 Operational Example

Let's give an example of the operation of this agent, using the Security-APL stock database (Section 6.2).

When the agent is initialized, it either is compiled with or reads data from a file that gives it an identity: a name, address, ANS name, the DB definition, and the set of table-specific methods. The agent then registers itself with the agent nameserver, and advertises its database (under the default organization—let's not get too complicated to begin with!).

The agent then enters its main loop, which in the most simple agent is: retrieve a plan for any new tasks; schedule the planned actions; execute the next thing on the schedule/agenda. For the simplest information agents, each *action* on the agenda has a deadline and (optionally) a period. Our initial scheduler can just execute actions earliest-deadline-first. If an action is periodic, then once it is executed it is re-inserted into the agenda with a new deadline equal to the old deadline + the current time. Smarter schedulers can use more information, i.e., average execution times, coordination relationships, etc.

In our example, initially the only action on the agenda is *poll-for-messages* (information gathering). This first action does a KAPI:KGet (non-blocking read). The action is periodic with a fairly short period. Let's assume a query comes in:

```
(ask-one
 :SENDER fred
 :RECEIVER security-apl-agent
 :LANGUAGE simple-query
 :ONTOLOGY stocks
 :REPLY-WITH ibm-query-1
 :CONTENT (query security-apl :CLAUSES (eq $symbol "IBM") :OUTPUT :ALL)
```

The message is picked up by the *poll-for-messages* action and a handler parses it and places a representation of this particular query task on the agent's internal list of task groups. The plan-retriever then looks up a plan: run the query against the database and send the results to :SENDER fred. Each action is non-periodic, and given a fairly short default deadline (alternately, the agent fred might have specified a deadline in the KQML statement). The second action is enabled by the first one (so their order is fixed).

So now the agent executes the action *run-query*, with the appropriate context. A table-specific method is called that accesses `http://www.secapl.com/cgi-bin/qs` and retrieves the data for IBM. The data is then stored in the local database table—this includes checking to see if any events are triggered as in Anandeep's system, but right now in our imaginary example there are none. The agent then executes the *send-results* action, in which the record is packaged up and sent back via a KQML REPLY(ies), depending on the original performative. Each record keeps an internal field that records to whom the record has been sent, and when, to avoid duplicate messages (this will be more important when we start monitoring tasks, next).

Handling a MONITOR message with only a FREQUENCY component is exactly the same as a one-shot query, except that the run-query and send-results actions become periodic with a period equal to the frequency.

Handling a MONITOR with a NOTIFICATION-DEADLINE (edge trigger) is much more interesting. Let's assume that we recieve a monitoring request:

```
(monitor
 :SENDER barney
 :RECEIVER security-apl-agent
 :LANGUAGE simple-query
 :ONTOLOGY stocks
 :REPLY-WITH ibm-query-2
 :NOTIFICATION-DEADLINE (15 minutes)
 :CONTENT (query security-apl
                 :CLAUSES
                 (eq $symbol "IBM")
                 (< $52-week-low (previous-value $52-week-low))
                 :OUTPUT :ALL)) % $
```

Again, the communication is picked up by the *poll-for-messages* information-gathering action, and instantiates a monitoring task. This time the plan-retriever retrieves three actions: *run-query*, *check-trigger*, and *send-results*. The check-trigger action is stored in an agent Local DB data structure so that whenever the local database is updated, these clauses can be checked, and if they match, a *send-result* action is enabled. As mentioned before, we remember that a record has been sent to an agent for a query so as not to send duplicates.[3]

More complex information agents that offer more services (greater competency) can have better schedulers, more complex plans, coordination mechanisms for helping to order actions and make commitments, complex reasoning associated with *run-query* and having multiple sources, some parallelism, etc.

4 Programming an Information Agent

4.1 A Simple Information Agent Definition

Programming a simple information agent is very straightforward. It requires you to define a database specification as shown in section 2.1, and write an external query function. Here is the code for the Security APL Stock Ticker Agent. It defines a new agent object "SecAPL-Agent" that ISA Agent::InfoAgent. It defines the local DB spec as a string. It then makes an instance of the agent by calling "new" and passing it the name of the dbspec and the external query function. We'll discuss what goes into the external query function next.

```
package SecAPL_agent;
use Agent::InfoAgent;
use Agent::Task::Action::EQSecAPL;

@SecAPL_agent::ISA = qw( Agent::InfoAgent );
```

[3] Note that we have to assume that an agent/query pair is a single thing, in case an agent asks multiple queries that happen to contain the same record in the reply.

```
$dbspec = "(DATABASE security-apl
 :ONTOLOGY stocks
 :ATTRIBUTES previous-value timestamp
 :QUERY-LANGUAGE simple-query
 :SCHEMA
  (symbol :ADVERTISED :UNIQUE)
  (price )
  (reference-date )
  (exchange )
)";

$agent = SecAPL_agent->new('db-spec'           => $dbspec,
                           'external-query-fn' => \&external_query_secapl);
$agent->main();
```

4.2 Defining an External Query Function

An external query function takes three arguments: an agent object, an action object, and a query object. The function returns a reference to a list of record objects to be added to the Local DB. We use the Perl Exporter module to formally define the external query function as the only exported interface in the file. The external query modules are considered part of an agent Action, so they are stored under Agent/Task/Action/.

Note: This is an example, not the real News Agent code. I've removed the details of doing NNTP.

```
package Agent::Task::Action::EQNews;
use Agent::LocalDB::Record;
use Agent::LocalDB::Query;
use strict;
require Exporter;

@Agent::Task::Action::EQNews::ISA = qw(Exporter);
@Agent::Task::Action::EQNews::EXPORT = qw(external_query_news);

sub external_query_news {
  my $agent = shift;
  my $action= shift;
  my $query = shift;

  # each clause in the query is (operator field index-pattern)
  my $index = $query->find_clause_index_for_field('fieldname');

  # THIS NEXT LINE IS PSUEDOCODE!!!
  my @msgs = do_stuff_to_get_data_using_index($index);

  # At this point, the array @msgs contains a list of good stuff.
  my ($newrec, @newrecs);
  foreach $msg (@msgs) {
    # create an empty localdb record
    $newrec = Agent::LocalDB::Record->new($agent->{'localDB'}->schema());

    # fill in the field values
    $newrec->set_value('newsgroups', $msg[0]);
    $newrec->set_value('date', $msg[1]);
    $newrec->set_value('subject', $msg[2]);
    # etcetera etcetera....
    push @newrecs, $newrec;
  }

  return [@newrecs];
}
```

4.3 Limited Interface Abilities

Although Information Agents are not Interface Agents, since we don't have any set Interface Agent class you can do simple things with the information agent graphics window. In particular, the function `$self->make_entry("Label");` will add a labeled box under the "quit" button. This function should be called from Initialize. The matching function `$self->get_entry("Label")` retrieves what the user has typed into the box at that moment. See the airfare-agent for an example.

4.4 Current List of Agents

Of course the easiest way to write a new agent is to modify an old one that is similar. Here is the current set of agents.

news-agent: A pure information agent that can access Usenet, Clarinet, and Dow-Jones news via NNTP from localhost.

secapl-agent: A pure information agent that accesses stock quotes from the Security APL / PAWWS server using HTTP "PUT"s.

matchmaker-agent: A fairly pure information agent that serves other informaiton agent's DB specs. It has no external query function. Instead, it understands "ADVERTISE" KQML messages, which store other agent's DB specs in it's own local DB.

airfare-agent: A pure information agent mixed with an interface agent. The airfare-agent serves the lowest one-way fare (based on roundtrip prices) between two cities as retrieved over HTTP from the Internet Travel Network. It *also* has an interface that constructs the appropriate query from the user and sends it *to itself*. When it receives an answer (from itself), it sends email to Keith. Obviously we could generalize it a bit more :-)

stock-display-agent: This is a fairly impure information agent. It serves a database of recent stock prices and times of the most recent news article. It does this by using the news and secapl agents. Not only does it serve the aforementioned data, but it produces a GIF of stock prices and News articles, and an HTML page that has this GIF at the top and the list of news articles in temporal order at the bottom.

portfolio: The portfolio agent registers a set of stocks with the stock-display-agent and produces a summary table in HTML of the current portfolio value and the value and last news date for each stock it follows. The summary table contains hyperlinks to the stock-display-agent's news-and-price HTML pages.

5 Detailed Implementation Notes

This part is omitted in the short version of this report.

6 Examples

Here are some examples, just so that we can stay grounded. The first three are probably very representative of the three major types of financial data: company fundamentals, historical data, and news.

6.1 The STOCKS Ontology

At the moment I'm using this as a flat data-typing mechanism. Obviously this can eventually become richer.

symbol	string
company	string
exchange	string
price	real
volume	integer
reference-date	date/time
day-high	real
day-low	real
day-close	real
52-week-high	real
52-week-low	real
beta	real
shares	integer
dividend-yield	real
EPS	real
market-capitalization	real

6.2 Security APL

The only key here is the ticker symbol. All data here is listed exactly as it really appears, except I have removed all the formatting, and items in italics which are comments added by me.

Symbol	GH
Exchange	New York Stock Exchange (NYSE)
Description	GENERAL HOST CORP
Last Traded at	5.6250
Date/Time	Oct 11 11:52:45
$ Change	-0.1250
% Change	-2.17
Volume	13100
# of Trades	6
Day Low	5.6250
Day High	5.7500
52 Week Low	3.7500
52 Week High	7.6250

A link from here to the EDGAR project (still, a search keyed by the ticker symbol) gets us more information. Note that we can merge this in with the previous data.

research firm	Ford
date	10/03/95
MktCap	146
Beta	0.84

research firm	ZACKS
date	10/05/95
ShareOut	22
DivYld	6.47
EPS	-0.37

EDGAR also contains links to the company's SEC filings (annual and quarterly reports, who owns more than 5% of the companies stock, etc. Basically, these forms are the raw data from which many of these fundamental numbers originate (for the initiated, that's the 10-K's, the 10-Q's and the DEF-14's).

Our Security APL DDL statement might then be:

```
(DATABASE security-apl
 :ONTOLOGY stocks
 :ATTRIBUTES previous-value timestamp
 :QUERY-LANGUAGE simple-query
 :SCHEMA
  (symbol :ADVERTISED :UNIQUE :PREDICATES =)
  (company :ADVERTISED)
  (exchange :RANGE (NYSE AMEX NASDAQ))
  (reference-date :RANGE *today*)
  (price )
  (volume )
  (day-high )
  (day-low )
  (52-week-high )
  (52-week-low )
  (beta )
  (shares )
  (dividend-yield )
  (EPS )
  (market-capitalization )
)

; recent IBM record
(query security-apl
   :CLAUSES (eq $symbol "IBM") :OUTPUT :ALL)

; recent records of all companies with "gold" in thier names and a
; beta of less than 0.5
(query security-apl
   :CLAUSES
   (=˜ $company /gold/i)
   (< $beta 0.5)
   :OUTPUT :ALL)

; tell me whenever Oracle reaches a new 52-week high price
; This statement is only useful as a monitoring query.
(query security-apl
   :CLAUSES
   (eq $symbol "ORCL")
   (> $52-week-high (previous-value $52-week-high))
   :OUTPUT :ALL)
```

6.3 MIT Experimental Stock Market Data

Here is the start of the historical data at MIT for MCD (McDonald's Corp.).

Date	High	Low	Close	Vol	Price*Volume
950929	39.250	38.125	38.250	2729.1	104.3881
951002	38.375	37.750	38.000	3871.9	147.1322
951004	39.250	38.375	39.000	1842.7	71.8653
951010	39.125	38.375	38.750	1464.6	56.7533
951011	39.125	38.750	39.000	1417.3	55.2747

And it's DDL and some queries (note that we don't have to advertise the price*volume field, which is just calculated...but we could do it):

```
(DATABASE mit-stock-data
 :ONTOLOGY stocks
 :ATTRIBUTES nil
 :QUERY-LANGUAGE simple-query
 :SCHEMA
  (symbol :ADVERTISED :UNIQUE)
  (company :ADVERTISED)
  (reference-date :RANGE (> 930101))
  (day-high)
  (day-low)
  (day-close)
  (volume)
)

; historical data on McDonald's between Jan and Dec of this year
(query mit-stock-data
   :CLAUSES
   (eq $symbol "MCD")
   (> $reference-date 950101)
   (< $reference-date 951231)
   :OUTPUT :ALL)
```

6.4 News Stories

Here we mean unparsed news. Once a news story is parsed into a record or set of records, then it fits neatly into the table model. Here are examples of both Dow Jones (WSJ) and Associated Press news available here at CMU (Note that the AP news from ClariNet is more heavily keyworded and indexed).

```
Newsgroups: dow-jones.indust.earnings-projections
Subject: *Musicland Puts 3Q Oper Loss At 19c A Share >MLG
Date: Thu, 12 Oct 1995 07:01:27 -0400

  (More) Dow Jones news 10-12-95
    7:01 am
--
Newsgroups: dow-jones.indust.earnings-projections
Subject: *Musicland 3Q Oper Loss View To Include 10c/Shr Charge >MLG
Date: Thu, 12 Oct 1995 07:02:34 -0400

  (More) Dow Jones news 10-12-95
    7:02 am
--
Newsgroups: dow-jones.indust.earnings-projections
Subject: *Musicland Taking 3Q Goodwill Writedown Of $4.09 A Share >MLG
Date: Thu, 12 Oct 1995 07:05:16 -0400

  (More) Dow Jones news 10-12-95
    7:05 am
--
Newsgroups: dow-jones.indust.earnings-projections
Subject: Musicland Estimate -2-: 3Q Goodwill Writedown $138M >MLG
Date: Thu, 12 Oct 1995 08:15:23 -0400

  MINNEAPOLIS -DJ- Musicland Stores Corp. (MLG) expects to report a third-quar-
ter operating loss of about 19 cents a share.

  In a press release, the music retailer said its estimated loss includes
a pretax charge of $5.4 million, or 10 cents a share, for a reserve related
to 35 planned mall music store closings to start in early 1996.

  The company also plans a non-cash writedown of goodwill in the third
quarter totaling $138 million, or $4.09 a share. The writedown will reduce
goodwill amortization by about $4.2 million a year, which will help net
income by 3 cents a share in the fourth quarter and 13 cents a share
in fiscal 1996, the company said.

  Third-quarter expenses will be partly offset by pretax income of $8.8
million, or 16 cents a share, from the termination of certain service
and business development agreements.
```

```
  In the year-ago third quarter, the company had a loss of $2.6 million,
or 7 cents a share, on sales of $302.5 million.

  Year-ago fourth-quarter results were not immediately available.

  Musicland expects to report third-quarter results Oct. 25.

  (END) DOW JONES NEWS 10-12-95
    8:15 AM
--
Supersedes: <Aearns-chryslerURerl_5OB@clari.net>
Newsgroups: clari.local.michigan,clari.biz.earnings,clari.biz.industry.automotive
Subject: Chrysler Earnings Drop 46 Pct.
Keywords: General financial/business news, urgent, Chrysler Corp, tick=C,
        General Motors Corp, tick=GM, Ford Motor, tick=F, Business,
        Automotive, Transportation
Organization: Copyright 1995 by The Associated Press
Date: Wed, 11 Oct 1995 15:00:31 PDT
ACategory: financial
Slugword: Earns-Chrysler
Threadword: earns
Priority: important

        DETROIT (AP) -- Costs and lost production from the launch of its
new generation of minivans combined with a steep sales plunge in
Mexico to help depress Chrysler Corp.’s third-quarter profits by 46
percent.
        The results released by the No. 3 domestic automaker Wednesday
weren’t unexpected, since the third quarter historically is the
weakest period of the year for the Big Three.
[...]
        The automaker reported net earnings for the July-September
period were $354 million, or 91 cents a share, compared with $651
million, or $1.76 a share, in last year’s third quarter.
[...]
        Chrysler’s revenue for the quarter rose to $12 billion from
$11.7 billion in the same 1994 period.
        For the first nine months of this year, Chrysler profits have
fallen to $1.08 billion, or $2.82 a share, from $2.5 billion, or
$6.92 a share, in the same 1994 period. Revenue was relatively flat
at $38.1 billion vs. $38 billion a year ago.
[...]
        No. 1 General Motors Corp. reports earnings Tuesday and Ford
Motor Co.’s report is due a day later. Those figures also are
expected to show year-over-year drops in profit.
        Combined, the Big Three’s third-quarter profits are likely to
total about $1.2 billion, down from about $2.3 billion a year ago.
        Chrysler’s stock price closed up 12 1/2 cents a share at $52.75 on
the New York Stock Exchange.
```

News stories will require a new ontology (actually, we could define this hierarchically, starting with “news” in general, then “financial news” and “dow-jones news”, “clarinet-news”, etc.

newsgroups	string
subject	string
date	date/time
body	file

And a query for every article on IBM earnings:

```
(DATABASE dow-jones-news
 :ONTOLOGY news
 :ATTRIBUTES nil
 :QUERY-LANGUAGE simple-query
 :SCHEMA
  (newsgroups :ADVERTISED :RANGE "dow-jones.*")
  (message-id :UNIQUE)
  (subject)
  (date :RANGE (> (- *today* 5days)))
  (body))

(monitor
 :SENDER barney
 :RECEIVER news-agent
 :LANGUAGE simple-query
 :ONTOLOGY news
 :REPLY-WITH ibm-query-2
 :NOTIFICATION-DEADLINE (30 minutes)
 :CONTENT (query news
                 :CLAUSES
                 (=˜ $newsgroups "dow-jones.indust.earnings-projections")
                 (=˜ subject "IBM")
                 :OUTPUT :ALL))
```

6.5 Standard & Poors Company Reports

Not usually available for free, except for a few issues.

- General Information
  1. Company name
  2. Ticker symbol
  3. CUSIP identifying number assigned to each company
  4. S&P 500 indicator
  5. Standard & Poor's earnings and dividend ranking
  6. Symbol under which options are traded
  7. Amount of shares held by reporting institutions
  8. S&P ranking of issues from 1-5 stars, 5 is highest
  9. Beta – measure of price volatility relative to S&P 500
- Balance Sheet
  1. Current assets (millions)
  2. Current liabilities (millions)
  3. Balance sheet date

    4. Long-term debt (millions)
    5. Common shares outstanding
    6. Preferred shares outstanding
    7. Footnotes to common shares

- Price and Dividend Data
    1. P/E ratio, computed against last 12 months EPS
    2. Average daily trading volume for the last 30 days
    3. 52-week high/low
    4. Calendar year high/low
    5. % Yield of indicated dividend rate and current stock value
    6. Amount of latest dividend paid
    7. Indicated annual dividend rate
    8. Date at which latest dividend is to be paid
    9. Ex-dividend date
- Earnings
    1. Earnings per share (EPS), last 12 months
    2. 5-year growth, compounded annual rate of per-share earnings growth for the past 5 years
    3. Interim earnings – for the longest accounting interval since the last fiscal year-end and compared with those for the year-earlier period
    4. Interim earnings period – reporting period for the current and prior interim period earnings per share
    5. Reference year for actual and estimated EPS
    6. Current year EPS
    7. Next year earnings per share estimate
    8. Earnings footnotes
    9. Month which ends the fiscal year
- Historical Data
    1. High/low price ranges for last 4 years
    2. Dividends for last 4 years
    3. Earnings per share for last 4 years
    4. Equity per share for last 4 years
    5. Revenue per share for last 4 years
    6. Net income per share for last 4 years
    7. Most recent split date and divisor

6.6 Zack's Investment Research

This is the company that specializes in tracking earnings reports estimates and recommendations from other analysts.

IBM	4/16/95					
		Mean	High	Low	# Est	Mean ∆ last month
FISC YR END	12/96	8.55	10.20	6.10	17	0.26
FISC YR END	12/95	7.27	8.25	5.30	24	0.16
ACTUAL	12/94	4.92				
QUARTER END	3/95	1.33	1.45	1.10	18	0.07
ACTUAL	3/94	0.54				
QUARTER END	6/95	1.78	2.04	1.51	12	0.06
ACTUAL	6/94	1.14				
NEXT 5 YR GRTH (%)		11.33	30.00	3.00	12	0.33