Mixing the reactive with the personal: Opportunities for end user programming in Personal information management

Max Van Kleek[1], Paul André[2], David R. Karger[1], and m.c. schraefel[2]

[1]CSAIL, MIT
32 Vassar St.
Cambridge, MA, 02139, USA
{emax, karger}@csail.mit.edu

[2]Electronics and Computer Science
University of Southampton
SO17 1BJ, United Kingdom
{pa2, mc}@ecs.soton.ac.uk

Abstract. While human personal assistants routinely take actions on behalf of their supervisors, proactive assistance in personal information management (PIM) software today remains limited to basic reminding and e-mail filtering functions. In this chapter, we introduce a prototype information assistance engine that lets end-users delegate to it various simple, routine but context- and activity-reactive tasks. To track user context and activity, our system, Atomate, treats web feeds (e.g., RSS/ATOM) from "life-tracking" web sites as sensor streams. Information from these feeds is integrated with information from other online resources such as social networking sites, online calendaring tools, and messaging services to form a simple but robust internal RDF world model. This world model facilitates integration with heterogeneous services, making possible the construction of behaviors with the ability to carry out tasks ranging from context-aware retrieval, filtering, and message routing to social coordination. To make behaviors easy and natural for end-users to specify, Atomate features a constrained natural language interface (CNLI) that guides and accelerates input using predictive auto-complete and correction.

1. Introduction

While desktop and mobile computing through great strides have yielded the many information tools we rely upon on a daily basis, these tools exhibit a simple limitation -- they are designed to let users access, manipulate and aggregate information manually. In contrast, human personal assistants, such as secretaries and administrative assistants, routinely do things autonomously on behalf of their supervisors, such as taking calls, handling visitors, managing contacts, coordinating meetings and so on. In order for personal information management tools to become as helpful as human personal assistants, these tools will require greater autonomy, specifically the ability to work on the behalf of a user without explicit human attention.

Building information tools that serve as effectively as human assistants in their diversity of competences is at least in the short term, beyond our capabilities. However, if one considers the many tedious, attention-intensive tasks that involve purely the management and routing of information, executing such tasks are well-suited to current capabilities of Personal Information Management (PIM) software. Automating these tasks thus reduces to two challenges; first, establishing a means for users to articulate to what should be done and the circumstances under which to take action, and second, monitoring the information environment and detect when appropriate situations arise for taking action.

These two challenges have been prohibitive for several reasons. First, the information needs of individuals change dynamically from one activity (or situation) to the next, and providing appropriate assistance may depend on tracking a multitude of relevant activities. Second, specifying the exact conditions under which these actions should be taken (and how they should be taken) can be challenging particularly for non-programmers if these conditions need to be articulated precisely in terms of concrete, observable events and procedures. Finally, accessing all relevant information necessary may be difficult due to fragmentation; most people draw upon information stored in a variety of forms in many different locations, from Post-It notes, to e-mails, web sites, calendar tools to past conversations with friends in the process of planning and conducting everyday activities.

Fortunately, however these challenges are starting to be mitigated by the Web. First, "life-tracking" services such as Google Latitude[1] for location, Last.fm[2] for music listening, Wakoopa[3] for application use, etc., have started to make unobtrusive user activity tracking possible via users' ensembles of personal digital devices, and to make this data available in real-time via standard delivery formats and mechanisms (e.g, RSS/ATOM feeds and REST APIs). With respect to the problem of fragmentation, many new PIM applications are moving off of desktops and onto the Web. Tools that have taken an immense web presence already include e-mail, note-taking tools, calendars, to-do list keepers, and document creation suites. These tools are making available via standard web protocols and APIs information previously sequestered in difficult to reach or proprietary/inaccessible formats.

Optimistic towards determining whether these developments could be applied to help users with their personal information workloads, we created Atomate, an information assistance engine for end-users that combines information from Web sources to enable it to take care of simple information tasks. In this paper, we demonstrate that, indeed simple context-reactive automation is feasible using web sources today, and that such automation is sufficient for several classes of simple information tasks people currently do manually. Our second contribution surrounds an approach to integrating heterogeneous information on the web using a semi-automatic strategy involving "semantic repair and personalization" by the end-user. This personalization process allows users to attribute significance to information arriving from various streams during the integration process, making later access to this information natural and easy. The final contribution of this paper is a controlled natural language interface (CNLI) for

---

[1] Google Latitude http:///www.google.com/latitude

[2] Last.fm http://last.fm, http:///www.audioscrobbler.com

[3] Wakoopa: http://wakoopa.com

allowing users of the system to specify what they want the system to do for them, using an English-like syntax.

In the following sections we consider related work, followed by a brief walkthrough of Atomate's UI and some examples of its use. We then provide a detailed overview of the Atomate framework, discuss how the framework can be extended to new data sources and capabilities, and present ongoing work towards making Atomate behaviors sharable and rule conditions easier to express.

2. Related Work

As an end user, web-based reactive behavioral programming environment for personal information tasks that uses a natural language interface (NLI), Atomate sits at the intersection of several fields. We briefly summarize connections to related work across these fields here.

With respect to end-user context-sensitive reactivity, the ubiquitous computing research community has explored the space of end-user authored reactive behaviors for some time. For example, a macro recording system in the Intelligent Room let users program room configuration tasks physically by example [8]. Meanwhile, iCAP [22] let end-users sketch their desired rules for home automation tasks, such as controlling the home thermostat and automatically turning on and off devices when people were occupying a space. Perhaps most similar to Atomate was CAMP [20], a "magnetic-poetry" interface that used pseudo-natural language for the construction of behaviors pertaining to automatic capture and access tasks.

Towards end-user programming on the web, Atomate was preceded by Greasemonkey, Chickenfoot [3] and CoScripter [16] that introduced end-user customization of web pages and, for the latter two, the scripting of web navigation-related tasks. Atomate extends automation provided by Chickenfoot and CoScripter to context-reactive behaviors that work "off the page", e.g., with feeds and web services that accept queries and perform actions based on the user's activities. Unlike Chickenfoot or CoScripter, Atomate does not yet support the specification of behaviors by demonstration; behaviors have to be explicitly specified by the user. This is an area of future investigation.

Atomate's natural language interface for specifying behaviors is inspired by work on Constrained Natural Language Interfaces (CLNIs) from the Semantic Web community, particularly the ATTEMPTO Controlled English (ACE) project for the specification of ontologies in rules in natural language [15] and the GINSENG and GINO interfaces for guided input of queries and ontology construction [1]. Atomate's grammar is different in that it is tuned to the creation of several types of behavior rules (either time based or world-model based), which may include wildcard expressions that can bind to any value meeting specified criteria. Although Atomate currently uses a predictive CFG grammar with limited reordering, extending Atomate's UI to a "sloppy" parser as described in Chapter [??] should be feasible and is under consideration.

Architecturally, Atomate most resembles early AI work in multi-agent systems, particularly blackboard architectures, devised as

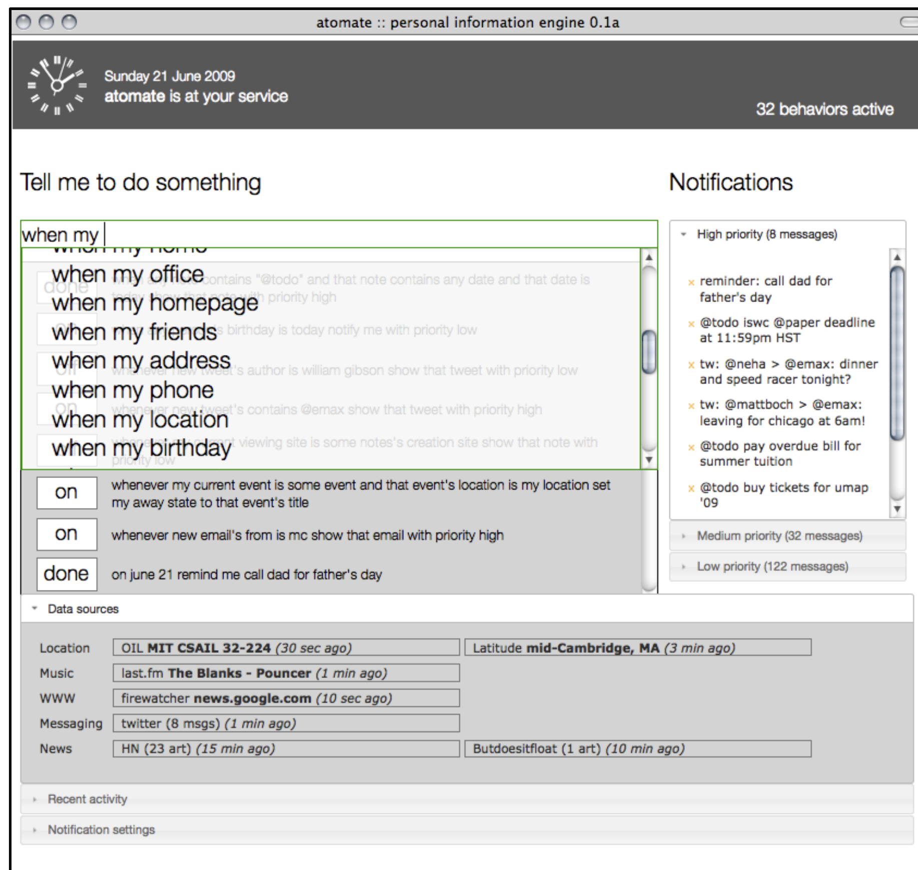

Figure 1 - Atomate personal mission control UI consisting of the rule creation interface on the upper left, filtered notifications on the sensor scoreboard showing latest information retrieved from web sources at the bottom.

problem-solving frameworks for heterogeneous information sources [21]. In a blackboard architecture, each knowledge source contributes evidence and expertise to help solve a particular set of problems or goals at hand. Programs attempting to solve problems or goals watched the blackboard for patterns as experts filled it, triggering (e.g., executing) when such patterns were found. These "programs" correspond to Atomate's end-user authored behaviors, while the central "blackboard" corresponds roughly to Atomate's internal RDF world model.

With respect to combining information from multiple web domains in end-user web environments, research surrounding web mash-ups have resulted in a number of end-user mashup authoring tools including Marmite [26], MashMaker [6], and Pipes [29]. The focus of these tools has been in facilitating the creation of combined feeds, views and simple visualizations that juxtapose or contextualize data from multiple sources. In particular, one paper [27] surveyed 22 mash-ups and their functions, and concluded that most mash-ups surrounded the construction of custom views of data, or bringing data to the desktop. There was an absence of any mash-ups which produced reactive behaviors (e.g., action) based on data from multiple sources.

3. Overview

Atomate helps users by letting them delegate simple actions to perform when certain conditions are met. These reactive conditional actions are called behaviors, and are expressed in the system as sets of rules (section **Error! Reference source not found.**). In the following section, we present a walk-through of how a user interacts with Atomate to set up a simple reactive behavior. Following this, we provide a number of behaviors to illustrate different types of tasks that Atomate can assume.

3.1 Walkthrough and Examples

Figure 1 illustrates Atomate's main user interface and the process of rule creation. In the figure, the user, Xaria, is setting up a simple rule for Atomate to remind her to call her father when she gets home. Xaria pulls up Atomate's UI using a bookmark on her Firefox toolbar. From this UI she can add new behaviors, edit, enable, or disable existing ones and monitor behavior execution.

To add a new behavior, Xaria focuses on the input box at the top of her list of behaviors and starts typing -- first, she types "when", indicating to Atomate that she is starting to author a behavior to be run only once when a certain condition is met. If she had instead started her behavior with the other behavior modifiers illustrated in Table 1, her rule would have been interpreted differently.

Following "when", Atomate's UI offers a list of all the entities representing people, places and things in its world model. It is asking her to start specifying the antecedent for her behavior, which describes the situations under which it should run. Antecedents are expressed as a a conjunctive sequence of subject-predicate-object expressions, where a subject can be a path query along properties of an entity in a posessive form, (e.g., "Joe Smith's current event's location"). Atomate uses `rdfs:label`s attached to entities as their "friendly names". Predicates, meanwhile are drawn from a pool of functions (overloaded by subject type), such as "is", "is near", or "occurs". Object values can either be path queries like subjects or primitive values, such as dates or arbitrary strings.

Since Xaria wants Atomate to remind her when she gets home, she simply types "my" (which Atomate recognizes as the possessive form of me), and it offers a list of all properties the system knows about with `rdfs:domain` Person: "age", "e-mail

at/on/no expression - *one shot alarm (time trigger)*
e.g.: "3pm tuesday", "on march 31"

every - *recurring alarm (time trigger)*
e.g.: "every tuesday", "every day at 3pm"

when - *one shot world-entity change trigger*
e.g.: "when my location is home",

whenever - *recurring entity-change trigger*
e.g.: "whenever any person's location is my office"

while - *recurring entity-change trigger w/ action reversal*
e.g.: "while my location is home"

Figure 2. Behavior trigger modifiers which modify how rules are executed

address", "location", "currently viewing website" (an activity she is engaged in), and so on. She starts typing "loc" and Atomate auto-fills the rest of "location", asking her next for a predicate ("is", "is near") restricted to only those that accept a place as its first argument. She selects "is", and Atomate offers a list of all the location entities it knows about. She selects "home", and then offers to let her either specify another clause ("and/or") to further refine the antecedent, as well as actions she might want to perform. She selects "notify me", which accepts an arbitrary string argument. She types "to call dad", and hits Enter to complete her rule. Atomate then creates a behavior (rule) to trigger the next time her entity is observed at being at location "home". She could have added multiple actions by typing "and" and naming another action and appropriate arguments. (Atomate's reminder/notification service can be configured via a rule, or manually through the UI at the bottom of Figure 1. Reminders can be sent through Growl, IM, email, synthesized text-to-speech, or mobile text messaging.)

If Xaria had instead wished to be notified whenever anyone's location changed, she could have used one of two special wildcard entity types: "new <type>" and "some <type>"/"any <type>". These entity expressions match incoming (new) entities or any entities of the particular specified type (e.g., person/friend, place, e-mail, document, tweet, et cetera). If she wished to refer back to the last bound wildcard entity in subsequent antecedent or action clauses, she can use the expression "that <type>" (e.g., "when *any person*'s location is my home and *that person* is not me notify me...").

Next, we present several examples intended to illustrate roles that Atomate can serve by integrating information and functionality from separate applications and services. These examples rely on retrieving and tagging notes from a note-taking tool such as List.it [24]. Other data sources involved include a calendaring service (e.g., Google Calendar), active indoor location information (e.g, OIL), and integration with messaging services, specifically e-mail and Twitter.

Reminding

Show notes tagged as a "todo" that contain a date occurring tomorrow or the day after:

```
while any note contains "@todo" and
    that note contains any date and
    that date occurs today
show that note with priority high
```

Activity-sensitive reminding:

Retrieve notes tagged to contain website passwords when the relevant page is visited:

```
while any note contains "@password" and
   my current viewing site is
   that note's creation site
show that note
```

Retrieves notes created at past sessions of a recurring event:

```
while my current event is any note's
  creation event
show that note
```

Tagging/organization

Automatically tag notes when created during a meeting:

```
while my current event is
  Haystack meeting and my location is
       CSAIL G531
  tag new notes "@haystack"
```

Message subscription management

Show messages from others "live tweeting" the event I am attending:

```
while my current event is not nothing and
  a new tweet's creator is any person and
  that person's current event is
  my current event
show that tweet
```

Social coordination

Inform the user when a specific individual arrives at a particular location:

```
when mc schraefel's location is
  mc schraefel's office
notify me
```

Set the user's away state to reflect attendance at an event on the user's calendar while they are actually attending it:

```
while my current event is any event and
  my current location is
  that event's location
set my away message to that event's title
```

Notify the user when an acquaintance looks at a research-related website that she has visited recently:

```
whenever any person's current
  viewing site is any site and
  my daily site viewing history
  contains that site and
  that site's topics contains
  "Semantic Web"
notify me
```

4. Atomate: The Information Assistance Engine

In this section, we describe salient features of Atomate's design, focusing on its core RDF world model, and the process by which new information is incorporated into it. A discussion of how new data sources can be added to Atomate follows, and how end-users can merge intentionally identical entities, assign personal aliases and relationships between entities using rules The rule engine and natural language parser are described last.

4.1 Internal world model

Atomate's world model is an RDF triple store [19] that contains two types of instances: entities and events. Entities represent the various types of "things" that can be referred to in behaviors, such as people, places, notes, web pages, e-mails, documents, and tweets. Events, meanwhile, are time-stamped observations of particular time-varying properties or activities of entities in the model.

To clarify the relationship between entities and events, we proceed with a simple example. When a user's location is identified using a localizer, a new event is created to represent this observation - which includes the time that the observation was made, the entity spotted (i.e., the user), the significance of this observation (that it represents the user's "location" property), the identity of the location, how the measurement/observation was made (e.g., via GPS or WiFi) and so on. The creation of this event then triggers a world model update rule (described in section 4.3) which updates the "current location" property of the corresponding person entity.

Maintaining a chronology of events that documents the evolution of entity state is useful for several parts of the system. As described in section 4.4, every rule firing also creates an event in the chronology. For "while" behaviors, these records are used to deduce the appropriate undo action (e.g., "rolling back" an entity property assignment) when a behavior's trigger ceases to match. Second, having time-stamped observations allows entity update rules to ensure that the most up-to-date observations are propagated to the entity store; observations themselves could arrive at the system at any time and out of order from when they were made. Finally, Atomate's UI uses the audit trail to let users inspect and understand why behaviors executed; this is important for letting users understand where things go wrong and to figure out the appropriate rule(s) or sources to change. This chronology will also eventually be used to derive the peRSSona, described in Section 5.3.

4.2 Integrating heterogeneous information

A fundamental design challenge surrounding Atomate's architecture was to allow users to add new data sources seamlessly to the system. This is a complex problem due to the number of levels at which information sources on the web are different -- the protocol level (e.g., the query interface or retrieval method), the syntactic level, consisting of the structure and container in which the information is delivered (feeds, web APIs), and finally the semantic level, consisting of what the data represents (e.g., is it a feed of news articles syndications? Observations of a user's activity?)

The problem of integration is further complicated by the fact that standardized containers for information (such as RSS 0.95/2.0/ATOM) often get appropriated and used to deliver information of fundamentally different structure. When this is done, fields are typically randomly appropriated and data is arbitrarily "shoehorned" into particular fields; for example, in Last.fm's audioscrobbler feed of a particular user's music listening activity, the song and artist names are concatenated into a single field, "Title". This kind of schema abuse is extremely common in "Web 2.0" sites today, and due to inconsistencies with which this is done, extracting data from these feeds automatically becomes a challenging extraction problem in its own right.

Atomate's philosophy to solving these integration problems is do as much as possible automatically, and make it easy for users to fix the rest. Specifically, where possible, Atomate attempts to take care of the first two of the three challenges above, and lets users handle semantic reconciliation and attribution. The reason for letting users handle semantics is, first, that is rarely possible to automatically discern the significance of arbitrary data streams, and second, it allows users, who possess external knowledge about how entities relate to them in to encode some of this knowledge into the world model. We describe specific examples of this in the Section 4.3.

In the case of that a data source represents and delivers its information in RDF, Atomate trivially solves the retrieval and syntactic handling problem by supporting the standard methods of publishing and parsing serializations of RDF (e.g., RSS 1.0, N3, RDF/XML, etc). Since RDF can be used to express arbitrary relations, this "shoehorning" problem does not arise. Furthermore, if the data source additionally either pre-aligns their graph to the Atomate schema or supplies an ontology definition (in RDFS/OWL) that is able to establish a mapping between the instances encoded in the document and Atomate classes (specifically observations and various entity types), these mappings are completed and entities are automatically aligned and added to the world model. An example of a feed from an RSS 1.0/RDF localizer service that expresses observations using the Atomate ontology is visible in Figure 3. If an RDF data source does not provide such a mapping and yet is able to identify instances (by examining their `rdf:type`s), it will import such instances verbatim as entities.

For data sources that do not use RDF, Atomate uses service wrappers to retrieve, parse, extract and transform the data into RDF instances. Due to the shoehorning problem above, this generally requires writing a new wrapper for every information different web site or information source. Wrappers are simple functions written in Javascript with the jQuery library, which facilitates AJAX calls and the parsing and selection of fields from XML feeds. When called, wrappers are responsible for retrieving new data, and updating the world model by either creating new events, entities, or updating existing ones. A javascript wrapper to the triple store facilitates the update process.

Another significant challenge with mixing heterogeneous non-RDF data sources is identifying co-referring entities, since such data sources do not use unambiguous URIs/IRIs. Such data sources do, however, typically provide an unambiguous key or identifier specific to that service for each particular entity or item. Atomate wrappers thus use this key to re-identify entities they have previously created in the triple store. However in many cases these entities may actually intentionally correspond to other entities already in the triple store, created from other information

```
johnsmith is Jonathan E. Smith
mc schraefel is monica (m.c.) schraefel

whenever some person's email is
     some other person's email
     the 1st person is the 2nd person

whenever some person's twitter id is
     some other person's twitter id
     the 1st person is the 2nd person
```

Figure 4. The first two rules demonstrate entity merging, where two entities corresponding to the same person but with slightly different labels were added by separate knowledge sources. The second pair of rules corresponds to inverse functional property creation, which fire whenever two emails or ids match. The final pair of examples creates aliases for common names of entities.

```
<?xml version="1.0"?>
<rdf:RDF  xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
  xmlns:atomate="http://plum.csail.mit.edu/atomate#"
  xmlns:sy="http://purl.org/rss/1.0/modules/syndication/"
  xmlns:mlt="http://mylifetracker.org/ont/svcdescription#"
  xmlns:gaz="http://mylifetracker.org/ont/gazetteer#"
  xmlns:geo="  http://www.w3.org/2003/01/geo/wgs84_pos# "
  xmlns="http://purl.org/rss/1.0/">

  <channel rdf:about="http://mylifetracker.org/">
    <title>My Life Tracking Service</title>
    <link>http://mylifetracker.org/lfeeds/emax</link>
    <description> Example Location Tracking Feed</description>
    <dc:date>2009-06-20T12:00+00:00</dc:date>
    <sy:updatePeriod>hourly</sy:updatePeriod>
    <sy:updateFrequency>10</sy:updateFrequency>
    <sy:updateBase>2009-01-01T12:00+00:00</sy:updateBase>
    <items> <!-- .. items omitted here for brevity --> </items>
  </channel>

  <item rdf:about="http://mylifetracker.org/lfeeds/emax/j200906loc1">
    <rdf:type rdf:resource="atomate:Observation"/>
    <title>Location observation</title>
    <atomate:dtstart >2009-06-21T11:37+00:00</atomate:dtstart>
    <atomate:dtlast >2009-06-21T11:48+00:00</atomate:dtlast>
    <atomate:subject rdf:resource="http://people.csail.mit.edu/~emax"/>
    <atomate:property rdf:resource="xsd:location"/>
    <atomate:value rdf:resource="gaz:MIT_CSAIL_32G531"/>
    <atomate:method rdf:resource="mlt:OrganicIndoorWiFi"/>
    <mlt:precision units="mlt:meters">3</mlt:precision>
</item>
```

Figure 3 - Partial listing of an RDF/RSS1.0 feed using Atomate's native schema for expressing observations. The duration of the observation has been highlighted in blue; the subject being observed (e.g., the user) in red, and location in orange.

```
whenever a new location observation's
  subject is some person and
  that location observation's
  value is some place
set that person's current location to
  that place

whenever a new web page view event's page is
  some web page and
  that web page view event's subject is
  some person
set that person's currently viewing site
  to that web page
```

Figure 5. Example rules that customize person and note entities to create properties "current location", and "current viewing site" out of appropriate events.

sources. We describe manual and semi-manual approaches to merging such co-referring entities next.

4.3 Semantic repair and personalization

As just described, importing data from various heterogeneous RDF and non-RDF data sources may result in inconsistencies in the world model, such as the duplicate entities referring to the same logical thing. Since automatically fixing semantic issues reliably and in general is difficult, we resort to making it easy for end-users to notice and fix these problems, using the same mechanism used to specify behaviors -- the creation of simple "world repair" rules. Duplicate entities are merged typically using one of two approaches: by specifying that two specific entities that are equivalent. The first corresponds to adding an `owl:sameAs` relation between them, which is expressed with a simple "is" statement visible in Figure 4. The second strategy is to match entities using property values. This is done by rules like the latter examples of Figure 4. The effect of such rules to the resulting view of the model has the same effect as would declaring these properties `owl:inverseFunctional`, but is easier for users to understand.

Such rules are useful for personalizing the world model beyond entity repair. First, a user may wish to relate a sequence of events to a time changing property of an entity, to support expressions such as "Joe Smith's location" instead of having to say "a new location observation about Joe Smith and that location observation's value is ..". The first simple rule in Figure 5 illustrates how such a property can be set from for any individual using location events. The second rule hooks up a user's web page viewing activity to a property of their entity. If the properties did not exist in the ontology, Atomate automatically instantiates a new `rdf:Property` with an `rdfs:label` corresponding to the provided property name.

Finally, to facilitate referring to commonly used entities, Atomate allows the user to add extra `rdfs:label`s as personal aliases to entities. The simple statement "call <entity> alias" can used to, for example, give the name "home" to the entity representing the user's home generated by the localizer.

4.4 Evaluating and executing behaviors

The duty of the Atomate rule engine is to run behaviors at the time the conditions specified on each rule are met. To do this, it applies a mixed strategy; for time-based rules, it computes the next relevant absolute time that the alarm should fire, and sets an OS/browser level callback to signal rule triggering. For when/whenever/while triggers, meanwhile, triggers are evaluated when changes are made to the world model. Since triggers for behaviors are represented as SPARQL queries and optional predicate applications, this amounts to running at least one query for every rule on every world update. To make this less expensive, the scheduler maintains a data structure that hashes rules by the entities they condition on; using this structure, rule triggers are only evaluated if the entities named in the trigger are involved in the change to the world model. Rules that have wildcard expressions (some/new <type>) are evaluated only if the changed entity is that type. We are currently investigating streaming SPARQL query engines that will allow rule triggers to be evaluated efficiently whenever triples in the store are added or modified.

Since rule conflicts could cause the system to enter infinite chaining loops, an important second responsibility of the rule engine is to detect rule conflicts and signal these to the user. To do this, the rule chainer keeps track of which rules were fired as the result of a single initial change to the world model. Any single rule that is fired twice as the result of a single initiating change is flagged as looping, is terminated, and the problem is signaled in the UI. This does not, however, catch external loops, which can result from state external to the system being set and reflected back into the system as a new observation. We are currently considering approaches to detect such behavior using patterns of rule re-firings.

4.5 Predicates and Actions

While the most basic predicate used in rule antecedents is "is" which merely compares an entity's property for equality to a particular value, other predicates may be added to Atomate to extend its ability to create sophisticated rules. Specifically, predicates may be functions that compute some derived value of the graph (for example, "number of friends"), or rely on external sources of information. An example of such a predicate is the "is within X miles of" predicate, which can rely on open GIS APIs to compute physical distances between landmarks. To prevent such external operators from having to be called incessantly, predicate applications are by default cached for a given set of arguments.

Currently, predicates can only be added to the system by users comfortable with Javascript programming. Predicates are simply Javascript objects that wrap boolean functions that take parameters of specific types, which are either entity or primitive (XSD) types. Currently, Atomate only supports binary predicates, but planned improvements to the parser should allow predicates with multiple modifier clauses. When multiple applicable predicates with the same name but different parameter types are declared, Atomate chooses the most specific predicate by determining at runtime the predicate with types that (cumulatively) have the smallest graph distance to the types of the arguments.

Action are implemented similarly to predicates in that they Javascript functions with strictly typed parameters; new ones may be added easily directly via a web interface by programmers. However, action functions are not assumed functional or idempotent, and therefore no execution caching is performed. Figure 5 lists a few of AM's actions. The simplest action, "set", assigns a property for a particular entity to a particular value, specified in its operands. Most of the other actions involve manipulating something external to the system; this is usually performed through a web API call (when available). However,

several actions have non-web destinations. For example, actions affecting the user's local machine, such as the "play media" "show document" action, are made through PLUM [25]. Still other actions, such as "filter notes" pertain to actions affecting Firefox extensions, such as List.It. and thus are dispatched to components directly within Firefox.

4.6 Interacting with the user

We anticipated that there would be two major dangers concerning how users might react to Atomate. The first was that users might perceive creating behaviors as "too much effort", both before and after learning how. Generations of devices from thermostats to VCRs have met this fate, resulting in their automation features to be ultimately left unused. The second peril was the feeling of unpredictability/untrustworthiness. If the system acted unexpectedly this could severely detract from perceived usefulness.

To address the former, we sought to make the system easy to learn by making the expression of behaviors resemble natural delegation. This inspired us to consider command-based natural language interfaces. We ultimately chose a simple controlled natural language interface (CNLIs) which allowed behaviors to be easily understood by any native speaker of the language, and yet could be parsed with little or no ambiguity. To create this interface, we hand-crafted a grammar (visible in Figure 6.) and added a fast-input framework to guide the user and auto-complete legal values as the user types. This not only accelerates input but also allows illegal input to be caught and corrected immediately.

At the core of the CNLI rule specification grammar is the simple antecedent clause ("antclause"), which, as can be seen follows a simple subject-predicate-object format. Subject expressions support indirection over properties using possessives (e.g., "Joe's location's latitude"). The terminals in bold represent the name of an entity or property in the RDF store. To support references to entities bound to the "any"/"some"/"new" wildcard entity expressions, the grammar supports demonstratives (e.g., "that") as well as indexical expressions ("first", "second") when multiple wildcard values are bound. The "other" adjective can be used to force wildcard specifiers to be bound to distinct entities; for example "if some person's location is home and some other person's location is home".

```
set <named-entity>'s <property> (to)
   <entity or value>
set <named-entity> is <named-entity>
notify <named-entity>
show me <entity-property-chain>
enable/disable rule <rule>
forward/send <entity-property-chain>
   to <entity-property-chain>
post tweet <text>
   | <entity-property-chain>
set priority of <entity-property-chain>
   to (low | medium | high)
say <entity/text>
play (song/media)
open <entity-property-chain>
set system power state/volume
search flickr/google images/wikipedia
map <entity-property-chain>
run function.... < code >
```

Figure 5. Basic Atomate actions

In addition to plain terminal and entity matching, Atomate's parser contains handwritten special-purpose code for parsing special data types such as date/time expressions. This parser in particular handles relative date/time expressions (e.g., "tomorrow", "next tuesday") as well as vague time expressions (e.g., "3pm"). It evaluates such expressions relative to the day the rule is created, and assumes that incomplete date expressions imply the soonest day, week or month that that expression applies.

With regard to making the system predictable, Atomate provides a real-time visualization of behaviors as they execute, along with a history of all the behaviors that have fired. Each history element explains what caused the behavior to fire, including the origin of the information that caused it to trigger. An ongoing effort surrounds implementing a behavior simulator so that the user can preview the effects of their behaviors as soon as they have composed them.

5. Ongoing and future work

The current status of Atomate demonstrates an approach that can support user-defined reactive behaviors based on extant Web 2.0 data feeds. In this section, we describe our current work towards extending Atomate to be easier to use, more predictable, and more useful, and finally, post public deployment, an evaluation of it uses in the real world.

5.1 Rule specification and debugging

As described earlier, the biggest challenge towards adoption surrounds making rules easy and quick enough to specify to be perceived as worthwhile. We have the following plans to improve Atomate's input interface in various ways.

5.1.1 Graphical input accelerator

We are currently working to extend the predictive input interface to facilitate input in two ways: first, by adding graphical feedback (e.g, icons and widgets) that let users more easily select specific entities and quantities (such as days with a date-picker widget) in rule expressions. For this, we will extend our prior work with Inky, the graphical command line for the Web . The second to add pure-pointer/graphical input support for constructing rules by touching to select relevant entities and desired actions.

5.1.2 Rule simulator

Since specifying correct (accurate and complete) antecedents for rules is often tricky for end-users but necessary for rules to work as intended, we are adding a rule simulator which will help the user immediately verify the behavior of their rules at time of specification. Our approach is to simulate rule execution using events from the user's recent past. Atomate will search backwards in its history from the moment the command was invoked, to find the most recent situation in which the particular rule would have triggered, and what the resulting action would have been. The output of this simulation will be displayed in a simple textual summary beneath the rule creation interface. The effects of actions will be described by combining the descriptions of the action operators invoked and the bindings that would have been in effect for the operands of these operators in each situation.

5.1.3 Picking behaviors by demonstration

Another enhancement to Atomate underway allowing behaviors to be specified using a Programming-By-Demonstration (PBD) approach [5]. To do this, we are creating the Situation Picker, which will allow rule antecedents to be chosen by example; this is done by presenting users' recent activity and context history in a

timeline, and letting users simply select an moment in their past they wished for Atomate to act. Atomate will then examine the state of the world model and user's entity at that given moment in time and automatically propose a rule antecedent. Using this feature in conjunction with the aforementioned rule simulator will then allow rules to be induced and debugged immediately as they are specified.

5.2 Sharing behaviors with a community

Many of the rules that update and repair semantics based on information retrieved from external data sources described in Section 4.3 map particular characteristics of the data sources to common changes of the world model. Thus it is likely that many of these rules will be useful to others using the same sources. To make it easier for users to get started with using Atomate, we plan to build sharing of behaviors such as these into the system. Around this sharing, we also will build a web site that supports discussion and showcasing of user rules. This way, we hope that users will trade ideas and tips on how to apply Atomate to automate aspects of their routines.

5.3 Publishing your life stream (peRSSona)

To make it easy for users to share updates about their activities, location and state with their friends, Atomate can be configured to publish state changes to particular entities in its world model as RSS 1.0 feeds. This feature, which we call *your peRSSona* will be configurable to provide differing degrees of disclosure for their activities. For example, a user might publish a public feed containing information about their contactibility but not their precise activity, while posting different perRSSonas containing more detailed activity information for their trusted friends -- information such as their whereabouts, music listening and web page browsing history. Users desiring more control over what gets posted to feeds can manually create feed entities, and set up rules that post to these feeds under arbitrary conditions. In the future, we plan to extend peRSSona support to provide differing levels of detail for a single property, for example, via summarization.

5.4 Planned study and deployment

While we have some insights already from research in end user programming on how to facilitate user-based programming, we need a significantly refined understanding of how users will engage with Atomate due to its breadth. Our approach will be two fold: a longitudinal field study with a dozen participants to be followed by a general web-based beta release. With these releases our key questions will be to investigate the kinds of behaviors users sought to have Atomate support and the degree to which Atomate satisfied those goals. We anticipate this study will help us understand how better to tune attributes such as interface, language expressivity , system predictability and reliability.

6. Discussion

In this chapter, we have described Atomate, a framework that enables the use of the web as a platform for context-sensitive personal reactive automation. In so doing, we demonstrated that with appropriate manipulation, many of the web data sources and APIs available today are suitable as information sources for driving a variety of simple but useful reactive personal information processes. These reactive processes can serve many roles in personal information workflow, including context-aware reminding, information filtering and social coordination.

Atomate benefited from several key architectural decisions. The first was the use of a single persistent internal representation containing simple key representations of people, places, events and resources. This intermediate representation ultimately served three important roles in the system. The first was in simplifying

statement → rule-expr | call-expr | is-expr
rule-expr → time-trigger-modifier *time-expr* actions |
    wm-trigger-modifier antecedents actions |
    actions "unless" antecedents
time-trigger-modifier → "at" | "on" | "every" | ∅
wm-trigger-modifier → "when" | "whenever" | "while"
antecedents → antclause | antclause "and" antecedents
actions → action-expr | action-expr "and" actions
antclause → subject-expr predicate-expr object-value-expr
subject-expr → entity-property-chain | wildcard-type | coref-expr
predicate-expr → **named-predicate** | *new-predicate-name*
object-value-expr → named-or-wildcard-entity | *primitive-value*
entity-property-chain → **named-entity** | **named-entity** "'s" properties
wildcard-type → "some" **named-type** | "some other" **named-type** |
    | "any" **named-type** | "any other" **named-type** |
    | "new" **named-type**
coref-expr → "that" **named-type** | "the" coref-place **named-type**
properties → **named-property** | **named-property** "'s" properties
coref-place → "first" | "second" | "third" | "1st" | "2nd" | "3rd" ...
action-expr → is-expr | set-expr | **named-action** action-arguments
action-arguments → argument-expr | argument-expr action-arguments
argument-expr → *argument-modifier* object-value-expr
set-expr → subject-expr "to" object-value-expr
call-expr → "call" **named-entity** *alias*
is-expr → **named-entity** "is" **named-entity**

Figure 6. CFG for Atomate's CNLI. Boldfaced terminals represent retrieved values from the triple store. Italicized terminals represent values which are handled by a custom parser for the particular value type.

representation reconciliation; having a single representation as a basis of aligning external sources of information avoids the pairwise-alignment problem that serves as a scalability limitation to many mash-ups today. The second surrounded its role as a single, unambiguous world model for the Atomate rule chainer. Third, this representation serves as an important abstraction barrier that decouples behavior rules from information sources; allowing information sources to be exchanged freely (or added for redundancy) without having to modify users behaviors.

Our second architectural insight was that web services and data feeds are increasingly useful sources for domain-specific knowledge about the world, and are thus suitable for use as predicates in evaluating relational information about specific data types such as locations, people and events.

An additional contribution is a simplified interface for supporting end-user programming across heterogeneous data types using a constrained simplified natural language interface. This approach reduces errors by eliminating the need for named entity reference resolution, making syntactic errors impossible, and providing just-in-time assistance that enumerates all possible values at each stage of rule specification. As a next step, we plan to add a rule simulator to further reduce the possibility for error by immediately demonstrating the behavior of a rule on the user's past historical data.

In summary, we have shown that web based personal information sources can be applied to enable a wide variety of simple but useful reactive processes. These personal reactive processes provide a glimpse of the potential for web data to do more for us, with less effort, than we may have previously imagined possible.

7. Availability

The entirety of Atomate, including source code is available for download under the MIT License at `http://code.google.com/p/atomate`.

8. Acknowledgements

This project was funded by MIT CSAIL and Nokia Research through the MIT Nokia alliance. It was also supported by WSRI and a Royal Academy of Engineering Senior Research Fellowship We thank our collaborators Mikko Pertunnen, Michael Bernstein, Katrina Panovich, Wolfe Styke, Greg Vargas, and Jamey Hicks for their contirbutions to the project, and Ora Lassila Mark Adler Brennan Moore, Wendy Mackay, and Michel Beaudoin-Lafon for their many ideas and suggestions.

9. REFERENCES

[1] Bernstein, A. and Kaufmann, E. GINO - A Guided Input Natural Language Ontology Editor. ISWC '06.

[2] Bernstein, M., Van Kleek, M., Karger, D. and schraefel, m.c. Information Scraps: How and Why Information Eludes our Personal Information Management Tools. ACM Trans. Inf. Syst. 26, 4 (Sep. 2008), 1-46.

[3] Bolin, M., Webber, M., Rha, P., Wilson, T., and Miller, R. C. Automation and customization of rendered web pages. UIST '05.

[4] Cron. http://en.wikipedia.org/wiki/Cron

[5] Cypher, A. Watch What I Do. MIT Press, 1993.

[6] Ennals, R. J. and Garofalakis, M. N. MashMaker: mashups for the masses. SIGMOD '07.

[7] Flot. Javascript plotting library. http://code.google.com/p/flot/

[8] Gajos, K., Fox, H., and Shrobe H. " Alfred: End User Empowerment in Human Centered Pervasive Computing", Pervasive 2002

[9] Google Maps API. http://code.google.com/apis/maps/

[10] Hogan, A., Harth, A., Decker, S. Performing object consolidation on the semantic web data graph. In Proceedings of 1st I3: Identity, Identifiers, Identification Workshop, 2007.

[11] Jena. Semantic Web Framework for Java. http://jena.sourceforge.net/

[12] jQuery JavaScript Library. http://jquery.com/

[13] JSON. JavaScript Object Notation. http://www.json.org/

[14] Kaiser, C. Ginseng–A Natural Language User Interface for Semantic Web Search. Thesis, Universität Zürich, 2004.

[15] Tobias Kuhn. How Controlled English can Improve Semantic Wikis. *Proceedings of the Fourth Workshop on Semantic Wikis, European Semantic Web Conference 2009*, CEUR Workshop Proceedings, 2009

[16] Leshed, G., Haber, E. M., Matthews, T., and Lau, T. CoScripter: automating & sharing how-to knowledge in the enterprise. CHI '08.

[17] Malone, T. W., Grant, K. R., and Turbak, F. A. The information lens: an intelligent system for information sharing in organizations. CHI '86.

[18] Miller, R., Victoria Chou, Michael Bernstein, Greg Little, Max Van Kleek, David Karger, mc schraefel. Inky: A Sloppy Command Line for the Web with Rich Visual Feedback, UIST'08, Monterrey, CA, October 2008.

[19] RDF. Resource Description Framework. http://www.w3.org/TR/rdf-concepts/

[20] SIMILE projects. http://simile.mit.edu/

[21] Winograd, T. 2001. Architectures for context. Hum.-Comput. Interact. 16, 2 (Dec. 2001), 401-419. DOI= http://dx.doi.org/10.1207/S15327051HCI16234_18

[22] Sohn, T. and Dey, A. iCAP: an informal tool for interactive prototyping of context-aware applications. CHI '03.

[23] Truong, K.N., Huang, E.M., Abowd, G.D. CAMP: A Magnetic Poetry Interface for End-User Programming of Capture Applications for the Home. UbiComp '04.

[24] Van Kleek, M., Bernstein, M., Panovich, K., Vargas, G., Karger, D., schraefel, mc. “Note-to-self: Examining personal information keeping in a lightweight note-taking tool.”, CHI 2009.

[25] Van Kleek, M. and Shrobe, H.E. A Practical Activity Capture Framework for Personal, Lifetime User Modeling. UM '07.

[26] Wong, J. and Hong, J. I. Making mashups with marmite: towards end-user programming for the web. CHI '07.

[27] Wong, J. and Hong, J. What do we "mashup" when we make mashups? WEUSE '08.

[28] XMLRPC. http://www.xmlrpc.com

[29] Yahoo! Pipes. http://pipes.yahoo.com/pipes/