[ChNum] Chapter 7

[ChTitle] Mixing the reactive with the personal: Opportunities for end-user programming in personal information management (PIM)

[Authors] Max Van Kleek, Paul André, Brennan Moore, David Karger and mc schraefel

[Abstract]

The transition of personal information management (PIM) tools off the desktop to the Web presents an opportunity to augment these tools with capabilities provided by the wealth of real-time information readily available. In this chapter, we describe a personal information assistance engine that lets end-users delegate to it various simple context- and activity-reactive tasks and reminders. Our system, Atomate, treats RSS/ATOM feeds from social networking and life-tracking sites as sensor streams, integrating information from such feeds into a simple unified RDF world model representing people, places and things and their time-varying states and activities. Combined with other information sources on the web, including the user's online calendar, web-based e-mail client, news feeds and messaging services, Atomate can be made to automatically carry out a variety of simple tasks for the user, ranging from context-aware filtering and messaging, to sharing and social coordination actions. Atomate's open architecture and world model easily accommodate new information sources and actions via the addition of feeds and web services. To make routine use of the system easy for non-programmers, Atomate provides a constrained-input natural language interface (CNLI) for behavior specification, and a direct-manipulation interface for inspecting and updating its world model.

[H1] 7.1. Introduction

The end-user programming systems presented in this book feature the common goal of empowering users who are non-programmers to construct custom computational processes thato help them with perform various types of tasks. In this chapter, we examine the application of end-user programming techniques to the management of personal information – a task which, for most of us, ordinarily consumes significant time and effort. Since different individuals' needs and strategies surrounding their personal information vary greatly, as do their approaches at managing it, the focus on empowering users to construct their own custom automation is of great promise to this problem.

The focus of our approach in this chapter is the construction of end-user scripts designed to execute automatically when certain conditions arise, which we refer to as *reactive automation*. Our motivation for examining reactive automation in personal information management (PIM) derives from the fact that the majority of While most of the PIMpersonal information tools we rely upon on a daily basis are still designed to facilitate user-initiated, manual access to and manipulation of information. By contrast,, human personal assistants, such as secretaries and administrative assistants, work autonomously on behalf of their supervisors, taking calls, handling visitors, managing contacts, coordinating meetings and so on. In order for personal information management tools to

approach the helpfulness of human personal assistants, these tools will need to demand less explicit attention from users and gain a greater ability to initiate and carry out tasks without supervision.

In this chapter, we introduce Atomate, a web-based reactive personal information assistance engine that uses information streams about people, places, events to drive end-user constructed custom automation. Atomate is implemented as an add-on for the Firefox browser. Unlike the personal agents portrayed in Semantic Web scenarios, Atomate requires no complex inference, learning, or reasoning -- data available from the feeds can be used directly to drive useful behaviors.

The transition of personal information management (PIM) tools to the web presents an unprecedented opportunity to give tools greater capabilities. The web is a data-rich environment with an increasing amount of machine-readable (semi-structured) information that systems can leverage directly to take action. In particular, the rise of social networking and life-tracking web sites have brought to the web a wealth of near-real time information about what people are doing, where they are, and the relationships between individuals, places, and things. Today, these social networking sites and services are geared toward human consumers, i.e., letting people view each other's activities. Atomate, however, treats feeds from these sites as sensor streams, and by doing so, demonstrates that these web data feeds can also be used to drive adaptive, context-reactive behaviors that perform various simple actions without user supervision.

Atomate relies on two key components: the first is a uniform internal RDF [Berners-Lee 2001] data model that simplifies the integration of heterogeneous information. The second is an interface that makes it easy for end-users to create behaviors and annotate the data model. To make this possible, we introduce a constrained-input, controlled natural language interface (CLNI)[1] designed to let non-programmers accurately and easily specify actions to be performed. While the use of CNLIs has previously been explored for ontology authoring and semantic annotation, Atomate is the first system to use CNLIs for scripting personal reactive automation.

In the rest of this chapter, we first present examples of Atomate behaviors and how they are expressed. These examples are followed by a brief discussion of related work, and a detailed description of Atomate's architecture. We then describe a user study to evaluate the creation of rules with Atomate, and discuss results and ongoing work.

[H1] 7.2. Atomate Walk-through

1 Controlled Natural Languages are subsets of natural language whose grammars and dictionaries have been restricted in order to reduce or eliminate both ambiguity and complexity. See: http://www.ics.mq.edu.au/~rolfs/controlled-natural-languages

We present three scenarios illustrating how a user interacts with Atomate to script useful simple reactive automation.

[H2] 7.2.1 Scenario: Simple contextual reminding

There are certain household chores Xaria needs to do on particular days of the week, but like many busy individuals, she has trouble remembering what day of the week it is. Since she sometimes works late, setting a regular calendar alarm would cause her to be reminded either on her way home or while she's still at the office. With Atomate, she can set up a reminder action to trigger precisely at the right time (right after she gets home), and to be delivered her via a variety of mechanisms -- an SMS text message to her mobile phone, e-mail, or a desktop notification.

*** Insert Figure 7.1 ***

[Caption] Atomate's interface for creating new behaviors. A new behavior is created by specifying the action first (top row), situations under which the behavior should act (second row), temporal constraints, and optional action parameters, such as the message to be delivered.

To set up a reminder to take out the trash when she gets home on Tuesday evenings, Xaria clicks on the Atomate tab in her Firefox browser, and clicks “My tasks”. This pulls up the interface visible in Figure 7.1. She first specifies what she wants done: she selects “alert me” (Figure 7.2.1), which defaults to a a combination of desktop notification and an SMS text message. The next two options are “whenever” or “next time”, which let her specify recurring or one-shot actions. Since she has to take the trash out every week, she chooses “whenever” (Figure 7.2.2).

*** Insert Figure 7.2 ***

[Caption] Xaria filing out the situational conditions for the statement “Alert me when my location is home on/after Tuesdays at 5pm with the message: Trash day!” The top portion of this figure shows a pieced together view of each click in the rule creation process. The bottom portion shows an in browser view of the final statement.

Clicking on “whenever” makes visible the situation specifier, visible as the “who does what” line (Figure 7.2.3). This interface is used to specify situational constraints that characterize the conditions under which the action should be taken. Such constraints could pertain to the user's state, (such as his or her location) as well as the state of any other entity Atomate has in its world model (described in Section 7.4.2). Such entities typically include the user's friends, acquaintances, calendar events, all of the locations the user commonly visits, web sites, music

tracks, news articles, and e-mails. This world model is built from information arriving from all of Atomate's various web-based information streams, as described in Section 7.4. Returning to the scenario, since Xaria wants to be reminded when she gets home, she specifies a constraint pertaining to her location: she clicks the "who" slot, which produces a list of possessive forms of people's names, and clicks the top choice, "my (Xaria's)". This produces a list of properties pertaining to her entitiy, including "current activity", "birthday", "current location" and so on. She selects "current location" (Figure 7.2.4). This moves the entity selector to the predicate position (labeled "does") and displays a list of all predicates that take a Location as the first argument, such as "is", "is near", "is not near". She selects "is" (Figure 7.2.5), and this moves the selector to the object (labeled "what") slot, where she selects "Home" (Figure 7.2.6). (Xaria could have equivalently selected "my", and subsequently "home", which also refers to her home.)

At its current state, the behavior would execute whenever she got home. To constrain the behavior to execute only on Tuesday evenings, she selects the "on/after" button (Figure 7.2.7). This produces a calendar widget, where she selects "every Tuesday" (Figure 7.2.8). Finally, she adds a message by clicking "message" (Figure 2.10) and typing "Trash day!".

She verifies that the behavior is what she intended by reading the simplified English rendering of the rule visible at the bottom of the screen: "Alert me when my location is home on/after Tuesdays at 5pm with the message: Trash day!". She hits Save (Figure 7.2.11) and the behavior appears in her Atomate's list of tasks.

Note that, in this example, it is likely that Xaria intends on having the system remind her*once* on Tuesday upon reaching her home, *not* constantly while at home on Tuesday evenings. This assumption is common to most kinds of behaviors that users create in the system, and thus Atomate's rule chainer is designed to work on change of state. We describe this in detail in Section 7.4.3.3.

[H2] 7.2.2 Scenario: Social coordination

In this second scenario, we illustrate how Atomate can be used in a simple social-coordination task.

Ben is often overloaded with things to do and misses events on his calendar -- sometimes because he's forgotten about them, while other times he was not sure he wanted to attend them to begin with. In either case, he wants to be notified which of his friends (if any) are attending events he's scheduled but not attending, to both serve as a reminder and potentially help him decide whether or not to go.

To set up an appropriate event, Ben opens Atomate and creates a “New task”. He begins by clicking “notify me” and “whenever” to specify that he wishes to be alerted whenever the condition is met. Then he constructs the first condition clause specifying that he wishes the behavior to execute when he’s missing an event on his calendar, by selecting, in sequence, “[my] [current calendar event's] [location] [is not] [my] [location]”. Next, he adds a second clause to restrict notifications to when his friends arrive at that event: “[any friend's] [location] [is] [my] [current calendar event's] [location]”. Ben verifies that the rule looks correct by inspecting the English rendering and hits save.

Since Ben did not specify a message, Atomate will tell him about the situation that caused the rule to trigger. Atomate uses the same mechanism for previewing rules in English to render notifications in simplified English as well. For example, if Ben was missing Liz's birthday party that his friend John is attending, the notification would appear “Ben's current location is not Liz's Birthday Party location. John Smith is at Liz's Birthday Party.”

Note that for this rule to work, Ben needs to have provided two types of data sources: a location data source for him and his friends, and his online calendar[2]. His location data source is configured to set the “current location” property on entities representing him and his friends in Atomate's knowledgebase, while his calendar is configured to update the “current calendar event” property on the entity representing him. Section 7.5.4 describes how new data sources can be configured to update properties automatically.

[H2] 7.2.3 Scenario: Extending the system

*** Insert Figure 7.3 ***

[Caption] Scenario 3: Sherry adding a new data source to Atomate

The final scenario illustrates how Atomate can be easily extended to new data sources.

Sherry prefers buying groceries from a local organic grocer, but the availability of produce items there changes every few days. Recently, her grocer started posting the arrival of fresh items on a simple static web page. Since she never remembers to manually check the page before going shopping, she sets up an Atomate script to inform her to not buy available organic items whenever she arrives at the supermarket.

2[Examples of services that could be used for location include Plazes (http://plazes.com) and Google Latitude (http://latitude.google.com). As most of these services do not provide indoor localization, we have also created our own WiFi-based room-granularity location-sensing service called OIL (Organic Indoor Location). For calendaring, any of the popular online calendaring services like Google Calendar would suffice

Her grocer does not publish an RSS feed, so Sherry uses DAPPER[3] to construct a data mapper to scrape data off of the grocer's site every couple days. Once she constructs it, she copies the address of the resulting RSS feed to her clipboard, and clicks on the "My Feeds" section of Atomate (Figure 7.3). She selects "Create a data source" and names it "Harvest Feed". She then pastes the dapper feed URL into the appropriate field.

When she does this, Atomate retrieves the feed and displays retrieved entries in the interface below. Seeing that Atomate has mis-identified the type of entry as "News Story", Sherry clicks on the "type:" property to change it. Since Atomate does not yet know about produce, she creates a new type for these items; to do this, she clicks "new type" and names it "Fresh Produce".

Behind the scenes, these actions cause Atomate to create an *rdfs:Class*[4] for the new type, with label "Fresh Produce", and to attach a property to the data source ("Harvest Feed") so that the arrival of new entries from this source will result in the creation of entities of the proper type in the world model in the future. Thus, in this way, new RDF classes can be spontaneously created from data arriving from new sources; and multiple sources can be configured to create the same data types. Sherry can further customize the names of the other properties of each of the Fresh produce items, or filter out unnecessary properties by un-selecting the row(s) corresponding to the properties she wishes to filter.

Next, she constructs a rule to use the new information. She goes to "My tasks" and creates a rule to trigger based on her location and available fresh produce items: "Notify me when my location is Super Market Shop and any Fresh produce item's posted date is within the past 24 hours".

[H1] 7.3. Related work

Architecturally, Atomate can be viewed as a type of blackboard system, an architecture for sensor-fusion and pattern-directed problem solving using heterogeneous information sources [Winograd 2001]. Atomate's information sources (e.g., web sites), correspond to a blackboard's "experts", while its end-user authored behaviors correspond to the pattern-directed programs in these systems.

3 Dapper - the Data Mapper http://www.dapper.net/open

4 In RDF, the entity used to represent classes of other entities defined in http://www.w3.org/TR/rdf-schema/

Towards end-user specified reactive-behaviors, a number of research systems have emerged from the context-aware and ubiquitous computing communities, including the Context Toolkit [Salber 1999], and ReBA [Kulkarni 2002]. A number of these projects have looked at different interfaces for making it easy to specify such behaviors; the iCAP, for example, proposed and studied sketching-based interfaces for programming home automation [Sohn 2003]. CAMP, meanwhile, explored a whimsical, magnetic-poetry metaphor [Truong 2004].

Atomate's approach to rule creation, meanwhile, was inspired by research systems examining the use of simplified natural languages for knowledge capture and access on the Semantic Web [Berners-Lee 2001]. In particular, the CLOnE [Funk 2008] and ACE (Attempto Controlled English) [Fuchs 2008] work introducing controlled language languages (CNL), and related GINO [Bernstein 2006] and GINSENG interfaces for guided input interfaces for CNLs were the basis of Atomate UI's design.

Other relevant research from the semantic web community includes work on rule representations (such as SWRL and RuleML), and efficient chainers and reasoners for rule systems. In the future, we aim to transition Atomate's internal rule representation to one of these standard representations and chainers to enable greater interoperability and more efficient rule evaluation.

Atomate also borrows inspiration from the “web mash-ups”, systems that have explored the potential for combining data from multiple web sites. A survey of 22 mash-ups [Wong 2008] concluded that most mash-ups provided new visualizations based on “mashed-up” data from heterogeneous feeds, but found no mash-ups designed to deliver reactive behaviors (e.g., actions) based on this same data. Atomate demonstrates thus that reactive mash-ups are both easily created and potentially useful for saving people time and effort. Since Atomate actually allows the user to extend the system and add new data sources, Atomate requires capabilities similar to end-user mash-up construction environments such as Intel's MashMaker [Ennals 2007].

[H1] 7.4. Atomate

In the following sections, we describe details of Atomate's design, explaining at a high level how data flows through the system, followed by a description of the data model, the rule chainer, and the user interface.

*** Insert Figure 7.4 ***

[Caption] Atomate data flow. Atomate pulls data from the web and updates the world model. Model updates are fired and picked up by the rule chainer, which evaluates triggers for user-specified behaviors (rules). Triggered rules that depend on changed entities are fired. The effects of such behaviors can be to update the world model (Set) or to call external services.

[H2] 7.4.1 Data flow

Atomate works in a reactive loop, as illustrated in Figure 7.4, consisting of the steps of data retrieval, world model update, and behavior execution. These behaviors, in turn, can cause changes to the knowledgebase that cause more behaviors to run.

Step 1: Retrieving new data - Atomate's feeder component periodically retrieves from Atomate's RDF model a list of all the data sources the user has created, and pulls data from each. For most data sources (corresponding to RSS/ATOM feeds), pulling data corresponds to a simple GET operation of the particular feed in question. For data sources that provide push (asynchronous call-backs), the feeder registers a callback on startup and waits for asynchronous notifications.

Step 2: Updating the world model - For each item retrieved from a source, the feeder updates the corresponding entity in the world model, or creates a new entity if no such corresponding item is found. This updating is fully deterministic as established through per-source mappings created by the user when he or she sets up the data source. For example, a Twitter feed would cause new *Tweet* entities to be created in the RDF knowledgebase, while entries from Facebook would cause updates to and creation of new *Person* entities.

Step 3: Triggering - Updates to entities in the knowledgebase are noticed by the rule chainer. When such updates occur, the rule chainer retrieves all *Rule* entities from the knowledgebase. It evaluates each rule antecedent, and fires all rules whose triggered antecedents depend on the entities that changed. As described further in Section 7.4.3, rule antecedents can be either entirely time-based or based on the state of entities in the world model.

The effects of rule firings depend on the type of rule: rules can either cause updates to properties of entities in the knowledgebase (if they have a “set” clause, described in Section 7.4.3), trigger a message or notification to be sent to the user, or cause a tweet or other message to be posted via a web service. For other actions, Atomate can be extended to call arbitrary web services as described in Section 7.5.2. If the knowledgebase is updated as a result of an action firing, the chainer is informed and the rule trigger evaluation starts again.

[H2] 7.4.2 Atomate's world model

Atomate internally represents all data it manipulates as RDF entities, including information about things in the physical world (people, places, events), digital items such as messages, articles, tweets, and e-mails, as well as Atomate-specific rules, predicates, and data sources. The uniform representation reduces overall system complexity and makes it possible for users to examine, access and update parts of the world model using the

entity inspector user interface (see Section 7.4.4.1), which acts as a “global address book” for the system. For example, users can redefine or add their own predicates and actions by creating new entities of the appropriate types, as described in Section 7.4.4.1. It also facilitates integration with new external data sources, as described in Section 7.5.2.

[H2] 7.4.3 Rules

Behaviors in Atomate are represented internally as rules, consisting of an antecedent (which characterize the conditions under which the rule should execute) and the consequent, which specifies the action to be taken.

[H3] 7.4.3.1 Specifying conditions for behavior execution

As described earlier, behavior antecedents can consist of two types of constraints: time constraints and situational constraints. Each behavior may have one time constraint, one or more situational constraints, or both a time and one or more situational constraints. The time constraint is simply an expression of when the behavior should execute. Like most calendar alarms, Atomate supports recurring time constraints based on day-of-week and time-of-day (e.g., "every Tuesday", "every day at 3pm").

Situational constraints, on the other hand, specify that particular conditions in the world model must be met for the rule to execute. Since Atomate's world model is designed to mimic (in a greatly simplified form) the dynamics of the real world, constraints on the world model correspond, in Atomate's terms, to situational constraints on the real world. As described in 2.1, these constraints could pertain to the user, people the user knows or any other entities (people, places, events, music tracks, resources) being tracked by the system.

Situational constraints are expressed as conjunctions (“ands”) of clauses, each with one binary relational predicate applied to two arguments. The simplest such binary relational predicate is “is” which merely checks to see whether the two arguments are the same or equal. Each argument of a binary relational predicate can be either a simple primitive type (number, date, or string) or a *path query* into Atomate's world model. Each path query originates at either a single specific entity, or any entity of a particular type through “any” expressions (e.g., “any Person”, “any Event” or “any e-mail”). Atomate also supports the restricted wildcard expression “new”, which is evaluated like “any” but only for newly created entities, to make it convenient to write rules that trigger based on the arrival of new information (e.g., “new Email”, “new instant message”, etc.). These wildcard expressions expand the expressiveness of Atomate's antecedents beyond propositional logic and allow for the construction of general rules that work over any (including yet unseen) entities of a particular type. The tail of a path query consists of properties which are traversed from the origin node(s) to values. For example, "[any e-mail's] [author's] [current location]" would be a path query starting from all e-mail entities to the locations of the authors

of those entities. Atomate supports path queries of arbitrary lengths. A rule is considered triggered when an interpretation (i.e. a set of mappings) can be found from entities to clause predicate arguments such that resolved values from those entities (through path queries) satisfy each corresponding clause predicate.

The use of “any” versus “new” can yield very different behaviors; for example, a rule with antecedent “[new email's] [author's] [location] [is] [my] [current location]” would trigger at most once: upon arrival of a new e-mail if the email's author was co-located with the user at the time the e-mail arrived. The use of “any” instead of “new”, meanwhile, (i.e., “[any email's] [author's] [location] [is] [my] [current location]”) would trigger whenever the author of any e-mail (including those previously received) became co-located with the user.

Atomate supports a third type of expression, “that”, which is used to refer to entities that were implicitly referenced in preceding clauses. Implicit references arise in two situations: at the end of a path queries (e.g., “my current location”) or via the wildcard expression “any” (e.g. “any Location”) described earlier. The subsequent use of “that Location” in either case would allow the specific entity (either the value of my current location) or the value being bound to “any Location” to be applied to another predicate, “that” binds to only the last such reference.

[H3] 7.4.3.2 Expressing actions to be taken

The default set of actions provided in Atomate are set, e-mail, alert, text, post a tweet, update Facebook status and “Set”. The *Set* function updates the property of an entity in the system to a particular specified value, and is typically used to automatically update properties of entities based on the arrival of information to the system, as described in Section 7.5.4.

New actions can be added to Atomate as described in Section 7.5.1. When an action is called, it is provided as an argument the antecedent that caused the rule to fire. This can be used to make the action adaptive, or to include a helpful message to the user about why the action was taken.

[H3] 7.4.3.3 Triggering and scheduling

The goal of the rule chainer is, for every given behavior, to determine whether predicates can be satisfied by some entity (or entities) under the current state of the world model. This corresponds to determining whether nodes named in each clause, when de-referenced by the appropriate path queries, yield values that satisfy the predicates in the antecedent. If the clause contains wildcard expressions (such as “any”, “new” described previously), the chainer must search over all nodes of an appropriate type to determine satisfaction. Computationally, Atomate's

approach is naive; it is a forward rule chainer that uses a brute-force approach to evaluate such rule triggers. While it is likely that an efficient solution to this problem exists, our implementation searches the entire space of instances of a particular type in the worst-case scenario to evaluate triggers.

One assumption embodied in the trigger surrounds when rules are considered for execution. As introduced in Scenario 1, most behaviors exhibit the assumption that they will fire once when the rule's antecedents become satisfied. Atomate's rule chainer implements this assumption as follows: when an entity is updated in the knowledgebase, the chainer is informed of the change, in particular the entity, property and value(s) affected. Then, upon evaluating the triggers for all behaviors, the chainer only fires rules if the values that caused them to trigger corresponded to any of update operations.

Our ongoing work seeks to apply previous work in reasoning engines to make evaluation of triggers more efficient, and to relax the assumption surrounding rule execution to support ongoing behaviors.

[H2] 7.4.4 User interface

The Atomate interface consists of two integrated interfaces: the delegation interface, where users construct, activate, and monitor the execution of reactive behaviors in the system, and the entity explorer interface, where entities can be viewed, edited, and linked.

[H3] 7.4.4.1 “My stuff”: The Entity Explorer

*** Insert Figure 7.5 ***

[Caption] The entity inspection UI lets users view and annotate entities in the world model

The entity explorer known as “My stuff” displays all entities in the system, filtered by type. It is designed to act as a general “global address book” for the user, a single unified interface to reference information obtained from web sources about people, places, events, articles, and so on. Clicking on any of the type names along the top of the displays a list of all the entities of that type. Clicking further on any item expands the item and displays its properties. Property values can be edited by clicking on them, which opens an auto-completing, incremental-search entity selector box. This box can be used to assign a primitive typed-value (an integer, string, or date), a single, or multiple entities as values. Entities can also be set as values of properties by dragging the entity from any list to the property value slot.

The “me” object displays the user's own entity object and state, which, due to its importance is promoted in the interface to the top (type) selector. Users can use this interface to quickly update properties of their own entity's state, which may cause relevant behaviors they have scripted to trigger.

[H3] 7.4.4.2 Rule creation and management interface

Scenarios 1-3 stepped through the rule creation user interface in detail. This interface was designed to be as easy to use as reminder-alarm dialog boxes often visible in PIM calendaring tools such as Outlook, and yet provide more expressive, context-aware reminder capabilities.

Scenario 1 walked through one example of using the rule creation interface visible in Figure 7.2. This interface has two sections, a section for choosing an action and a section for stating the condition(s) required to make that action occur. The user may choose for an action to occur “whenever” the conditions are satisfied or only “the next time” and then is guided through crafting the “who does what” clause, shown in Figure 7.2.3-7.2.6. Atomate continually modifies the displayed options such that an invalid rule cannot be created. The user may add additional clauses by clicking “and.” To specify a time constraint, the calendar widget shown in Figure 7.2.8-7.2.9 can be used, which becomes visible upon clicking the “at/on/after” button. At any time, the user may add a message by clicking “with message” and entering the message in the text field provided. While a user is creating a rule, it is incrementally displayed next to the save button in plain English to reduce the likelihood of error. After reviewing their rule, the user may alter their rule or click “save” to have it recorded and displayed below the rule creation interface in the rule management interface where it may be toggled active or inactive, and edited or deleted.

[H2] 7.4.5 Implementation

Atomate is implemented as a Firefox browser add-on, and is written entirely in Javascript. Atomate's RDF data storage layer is kept in-browser using the client-side persistence API mozStorage[5], which stores all data in an sqlite[6] file in the user's Firefox profile. Atomate's RDF-JS object relation mapper (ORM) allows RDF nodes to be treated like other Javascript objects in the code, enabling such objects to be serialized and inherit methods through *rdfs:subClass* chains. (Code for entity instance methods are serialized in the triple-store as string properties.) The

5 mozStorage APIs for Mozilla Firefox - https://developer.mozilla.org/en/Storage

6 Sqlite Database Engine – http://sqlite.org

rule scheduler, database engine and feeder are packaged in a Firefox XPCOM component, and remains a singleton per Firefox process (which is limited by Firefox to one per user). The Atomate UI, meanwhile resides in a web page which calls upon the components and renders/lays out the UI using jQuery[7].

The advantage to having Atomate in the browser is that the many Javascript client libraries and APIs available for letting sites access data, such as Facebook Client-Connect[8], and GData JS[9] can be readily used to obtain data[10]. New data sources with custom client libraries can be added by web developers without any need for special integration work.

[H1] 7.5. Extending Atomate

Since there are an enormous number of web sites that publish potentially useful information for use in Atomate scripts, it was impossible to anticipate and pre-integrate all possible data sources that users might need. Instead, we opted to make it easy for users to extend Atomate to use arbitrary new sources and actions as they become available.

The architecture of Atomate was designed to support extension in three ways. The first is the introduction of new capabilities (actions) and relation comparison functions (predicates) for user behaviors. The second is the addition of data sources for supplying new information about types of things Atomate already knows about. The third is to extend Atomate to entirely new types of information and representations, as illustrated in Scenario 3. In this section, we describe in greater detail how new sources are added to introduce new information into Atomate's world model.

[H2] 7.5.1 Adding predicates and actions

7 jQuery Framework for Javascript - http://jquery.org

8 Client Connect for Facebook - http://wiki.developers.facebook.com/index.php/JavaScript_Client_Library

9 Google Data APIs – http://code.google.com/apis/gdata/

10 In order to comply with Facebook's Terms of Service, Atomate tags and refreshes or expires data it has gathered from facebook within a 24-hour period.

New predicates can be added to Atomate to, for example, add more sophisticated support for comparing entities of certain types, while new action functions can be used to extend Atomate's capabilities. Currently geared towards end-users comfortable with writing Javascript, adding a predicate or action involves merely creating a new entity of the appropriate type (i.e. *Predicate* or *Action*), specifying required argument types, and attaching an implementation property with a string value consisting of the appropriate Javascript implementation[11]. Such an implementation may call an external web service to perform the relevant action or comparison evaluation[12].

[H2] 7.5.2 Adding new data sources

The most common form of extension is adding new data sources to provide instances of existing types. For the purposes of this discussion, we assume that Atomate had no prior record of the schema or representation used by this new data source.

Adding a new source requires two steps: adding the source to Atomate's world model, and telling Atomate how to interpret retrieved data. The first step is simple: if the data source exports an RSS/ATOM or other public XML feed, a new Data Source entity is created with the source URL pointing to that feed. If the data source provides its data in a proprietary format, embedded in HTML, or using a custom library, the user can either write a wrapper themselves in Javascript, or use a third-party service such as Dapper, Babel[13], or Yahoo Pipes[14] to convert data into an RSS feed first.

The second step is to establish a mapping between the new type and an existing Atomate type. We model our approach to that of Piggybank [Hyunh 2007], which let the user construct a visualization from multiple sources with dissimilar schemas using drag and drop gestures. Atomate first retrieves and displays a selection of raw (unaligned) items from the new source. Next, the user selects the Atomate type that best matches the type of the entities retrieved from the new source. For a micro-blogging service such as Twitter, for example, the user could select the closest entity, such as *Tweet*. Atomate then inserts in the display of each record of the new source items

11[We omit details on the requirements of such a function for sake of brevity, but a tutorial geared at advanced users of the system is included in Atomate's online documentation.

12 As predicates are re-evaluated frequently, they should be made efficient and properly cache computed values.

13[Babel translator service - http://simile.mit.edu/babel

14 Yahoo Pipes – http://pipes.yahoo.com

the properties that are found in instances of the chosen destination type. Atomate automatically maps properties in the destination type that match source type properties exactly. For the remaining properties, the user can manually match properties by dragging property names on top of other property names. Behind the scenes, Atomate creates a mapping descriptor to the data source that is used to structurally convert incoming items on its feed prior to being added to the knowledgebase.

Atomate does not require all properties in either source or target schemas to be aligned; new properties can be introduced into the type by leaving source properties unpaired. Similarly, properties of the target type can be left unpaired; this will cause entities from the particular source to have undefined values for these properties.

[H2] 7.5.3 Extending Atomate's schemas

If Atomate does not already have an appropriate class to represent a particular type, a new class can be created directly at the time the new source is added. This process, illustrated in Scenario 3, is done by performing the same steps as described above for adding a new data source. Then, when entities are retrieved from the source and displayed as described above, the user types the name of a new class instead of selecting an existing class as a target destination type. The user can customize the names of property fields and set property field destination types as in the alignment case described above. (Atomate initially guesses the value types by trying to coerce values in the new feed.) The user may optionally specify a super-class by selecting one during this process.

To complete the process of adding a schema, the user needs to specify at least one property to uniquely identify the entity to update when a new piece of information arrives from each source. Defining inverse functional properties serves as Atomate's simple solution the *entity resolution* problem, and avoids the need to deal with ambiguity in updates to the world model. For example, for schemas representing a person, their e-mail address and phone number could be a useful choice for such a property, as e-mail addresses or phone numbers usually uniquely identify individuals.

[H2] 7.5.4 Automatically-updated properties

The introduction of new sources and types just described result in the creation of new entities and classes to the Atomate knowledgebase. In many situations it is useful to assign created values as property value of some other entity in the system. For example, if a user adds new entities of type “GPS observation” from his car's navigation system, hooking up these values to the user's “current location” property would enable rules that condition on the user's current location property to work unmodified with this new data source.

This is where the *Set* action described earlier comes in. To create properties that automatically update in response to the arrival of new information items, a user creates a new rule using the*Set* action. *Set* takes as argument an entity, property, and value. The “that” expression described in Section 7.4.3.1 is particularly useful in*Set* rules, as it allows for the succinct expression of general property-update rules. For example, for the previous scenario, the rule “whenever [any GPS Observation's] [user id] is [any Person's] [gps service username], set [that Person's] [current location] to [that GPS Observation's] [location].” would connect the locations associated with any new GPS Observations to appropriate Person entities in the knowledgebase.

[H1] 7.6. Evaluation

Since a users' ability to easily and correctly create rules using Atomate's UI is potentially the greatest obstacle towards successful adoption of the system, our first goal was to investigate whether users could understand and create rules. Secondly, we were interested in users' thoughts about the potential value for such a system -- if and how they may use it now, and future desires for functionality. To answer these questions, we performed two studies as follows:

Study 1: Design Review - An informal design review was held with 15 user interface researchers (some of whom have experience in designing end-user programming interfaces) to discuss the rule creation process and interface. Asked to think about both personal and lay-user preferences, this discussion was used as early feedback and an opportunity to alter the surfacing, presentation, or explanation of the components before the rule creation study.

Study 2: Rule Creation - This study was designed to test our hypothesis that a constrained natural language input interface allows end-users to easily and quickly create rules of varying complexity, and to enquire about the system's current or future value to users. Using an example data set (consisting of locations, events, contacts and emails) we asked users to create nine rules (see Figure 7.6 for a full list). These rules ranged from simple to complex, and tested a range of possible behaviors (one-off/repeat reminding, different predicates and actions, multiple clauses, etc.).

We estimated the study would take 10 minutes, at 1 minute for each of the nine rules and another for the survey. We first piloted the study with three colleagues in our lab, observing as they thought-aloud through the study, before releasing it to a wider audience online. A two minute video was available to explain the system prior to use.

*** Insert Figure 7.6 ***

[Caption] The nine rules participants were asked to create in the evaluation.

We used one of four categories to classify each created rule. 'Correct' -- the rule does exactly what the instructions specified; 'half-correct' -- the rule would work but may fire too often as the result of it being inadequately specific, or it was obvious what the participant was trying to achieve; 'incorrect' -- the rule would not achieve the role as set out in instructions; or 'missing' -- the participant did not complete a rule. While accuracy was of course desirable, one of the goals of this study was the process participants went through to create rules, and the (potentially creative or misunderstood) solutions they may come up with, in order for us to refine the system for extended use. Thus, we were particularly interested in the 'half-correct' and 'incorrect' rules as these would point to areas or concepts that participants found difficult to comprehend or input, allowing future improvement.

An exit survey measured quantitative response to how easy the system was to use and how useful it was deemed to be, qualitative feedback on the ease of use of the system, and thoughts as to how and when participants would like to use it now or in the future.

[H1] 7.7. Results

Study 1: Design Review - Feedback from the design review resulted in a number of changes to the interface to make the rule creation process clearer and more intuitive. These included: labeling the three autocomplete boxes with examples, and simplification of the time and date entry fields.

Study 2: Rule Creation - Three colleagues performed the rule creation study in our lab, talking aloud so we could get feedback on what was confusing or unclear about the system. Positively, the feedback mostly concerned minor improvements as opposed to higher level concerns about the grammar or interface. Based on those concerns, we improved feedback on what an alert or e-mail would display, on whether a rule was one-time or to be repeated, and clarified our instructions before advertising the study online.

In total, 33 participants began the study, but because of time limitations or technical issues with smaller screens, 26 participants fully completed all rules and the final survey. The group's age ranged from 25-45, 14 of whom had some previous programming experience. In the sections below, we first examine how participants used the system, including measures of accuracy and ease of use, and discuss how these results suggest design changes and secondly, look at whether and in what ways participants thought such a system would be useful to them.

[H3] 7.7.1 Accuracy and Ease of Use

*** Insert Figure 7.7 ***

[Caption] Percentage of created rules that were correct, half-correct, incorrect, or missing.

As in the previous section, we classified each answer into one of four categories: correct, half-correct, incorrect or missing. Figure 7.7 details these scores as a percentage of all answers. The first six rules were correct over 75% (and mostly over 85%) of the time. The final three were more problematic and raised some interesting issues. Rule 7 (Memex mail): Many participants achieved 'half-correct' answers on this rule, leaving out one of the two clauses needed (either 'from V Bush' or 'subject contains 'Memex"). Rule 8 (concert): This was an intentionally tricky rule, and open to interpretation and how each person would set up their Atomate system. The 'correct' way was to specify a current activity type of concert, but many participants used 'current activity's description contains 'concert". This could feasibly work, but only if when setting up the concert feed, they make sure to precede each description with the word 'concert', otherwise it would most likely just look like 'nine inch nails @ middle east, 9pm'. The incorrect rules here were varied, and in feedback participants said when they could not quite grasp the correct way to do it they just moved on with whatever they had. Rule 9 (event in 5 minutes): The half and incorrect rules for rule 9 mainly missed out the second desired clause (and that event's location is near me), meaning the rule would fire for all events starting soon, regardless of location.

Figure 7.8.1 details responses to the question "After reading and understanding the instructions for each rule, how easy was it to create the rule?" Feedback suggests (as we intended) that the rules were of varying complexity, but overall it was encouraging that with only 2 minutes training in the form of a video, 65% of participants found it easy to create rules. However, 35% said they found it difficult, and we discuss improvements in Section 7.8.

*** Insert Figure 7.8 ***

[Caption] Results from the rule creation survey. (1) How easy was it to create each rule? (2) Do you think Atomate would be useful to you?

The half-correct and incorrect answers, along with feedback from the survey, suggest a number of both simple design changes to improve the system, as well as interesting directions for future work.

Simple improvements were implemented to address some of the issues we observed. These included an easier and more prominent way to "and" clauses together to form a rule and alterations to the language displayed in the interface, such as changing "at" to "on / after" for adding dates, and changing "if" to an option of "whenever" or "the next time" for adding clauses.

We are working on several improvements to the rule creation process described in Section 7.6, including rule sharing and approaches at specifying rules by demonstration. As a specific example, to mitigate the problem of

clause omission, which causes rules to act too often (e.g., with insufficient specificity), we are working on a rule simulation environment which will let the user immediately see the effects of a rule on the user's recent past.

[H3] 7.7.2 Usefulness

The second goal of our study was to explore how helpful or useful participants would find the current implementation, and what other uses they would like Atomate to perform.

After completing the example rule creation, users were asked 'Do you think Atomate would be useful to you?' (Figure 7.8.2). On a scale of 1 (Not Useful) to 7 (Very Useful), the mean response was 5.5. Participants valued a number of uses of Atomate, such as forwarding certain e-mails to a phone, e-mail me the weather forecast when I am in a new location, or reminders at certain dates (pay credit card when due) or for certain events and locations (bring earplugs to a concert, remind me to thank people for gifts when in their vicinity). A number of participants also reported they thought it may encourage more social uses of the web, easing the barriers for acts of sharing. One participant said:

"When done manually, letting your friends know every time you're at a bar could be tedious ... but with automation it would be more of a friendly standing invitation and may actually drive me to use social networking sites more."

A number of creative future uses for Atomate were also posited. The ability to add sensors to other objects was widely desired, such as a roomba vacuum cleaner, a microwave, a fridge, or even toilet paper! Integration with cellphones was also seen as valuable, declining calls if in a certain location, or replying to missed calls when in transit.

In summary, the majority of participants found it easy to create rules, and thought the system would provide value, positing potential supported uses. A number of rule creation mistakes and feedback suggested both short and long term design improvements. We discuss some of the longer term improvements in the following section.

[H1] 7.8. Ongoing Work

In this section, we describe our current work towards extending Atomate to make it easier to use and more versatile.

[H2] 7.8.1 Extending the rule language

Ternary and higher-arity predicates are often useful for expressing value-modified predicates such as “within N minutes of” or “within N miles of”; thus we are adding support for such predicates in the chainer and UI. Second, to realize an initial goal of better supporting message filtering/routing applications, we will support “engage-able” actions through “while” rules. Unlike the “next time”/”whenever” rules described earlier, “while” rules will enable something when a rule triggers, and disable it automatically when the rule ceases triggering. For example, “while my location is home, send notifications through my computer” would allow Atomate's notification policy to change based on the user's location.

[H2] 7.8.2 Simulation and by-demonstration UI

To reduce the likelihood of errors at rule creation time, we are developing an interface that will immediately simulate, when a rule is specified, the conditions under which it will fire. This simulation will display and replay a recent history of the user's world model, to demonstrate what caused the behavior to fire. Such a simulation we believe would help users identify, for example, when they've left out a clause in a rule antecedent that would cause it to fire more frequently than desired, or if their antecedent is too restrictive. In addition to firing simulation, we wish to provide a “programming by demonstration” approach to start specifying rules. The idea is to use the visualization idea just mentioned to let the user select situations in which they wish they want the new behavior to execute.

[H2] 7.8.3 Community sharing (the Atom Garage)

While advanced users and enthusiasts may extend the system to new data sources and property-updating rules in the ways described, casual end-users are unlikely to bother. To allow the effort of the few “power-users” of Atomate to benefit the casual users, we are building an online repository where users can upload entities of their choice, and publish them with descriptors so that they can be easily found.

[H2] 7.8.4 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 how they preferred to be contacted 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.

[H2] 7.8.5 From pull to push: PubSubHubbub

Atomate's current method of polling of ATOM/RSS feeds is inefficient because feeds are pulled repeatedly, which causes entries previously seen entries to be repeatedly parsed. The current architecture requires that Atomate subscribe to a potentially large number of feeds, hundreds, for example, if the user is to track the location and activity state changes of all her friends. To reduce load on clients, we are adding support for pubsubhubbub[15] with which feed clients register for change notifications at "hubs" for a number of feeds. New information is pushed to clients only when changed, saving clients from polling repeatedly and repeatedly re-obtaining data they already have.

[H1] 7.9. Discussion and Conclusion

We have presented Atomate, a system to allow end-users to utilise web-based data sources to create reactive automation. The system comprises a uniform internal data model, and a constrained-natural-language interface to create rules. Through an evaluation, we have demonstrated that the majority of participants found it easy to create rules, and thought Atomate would provide value in a number of personal information related settings.

Some of the reactive behaviors demonstrated in this paper bear resemblance to scenarios proposed by Berners-Lee et al. in the original vision for the Semantic Web [Berners-Lee 2001]. However, there are a number of differences between Atomate and these Semantic Web agents. First, unlike the agents in the Semantic Web scenarios, which can "roam from page to page to carry out sophisticated tasks for users", Atomate acts directly based on the state of its model and information on incoming feeds, and lacks a sophisticated description logic inference engine, or capacities to learn, search, or act on the user's behalf beyond the rules set up by the user. However, the comparative simplicity of Atomate's approach makes it easier to understand, and potentially more predictable than approaches that use more sophisticated reasoning mechanisms.

Furthermore, the web of today is very different from the web portrayed in Semantic Web scenarios. Very few of today's "Web 2.0" RSS/ATOM feeds employ RDF; instead, they "shoehorn" data into simpler schemas intended for news article syndication, often embedding HTML into fields and overloading field semantics. This has not been a problem until now, as feeds have been consumed exclusively by humans through feed aggregators. But these behaviors cause a wealth of challenges for Atomate, directly complicating the process of adding new data

15 The PubSubHubBub Project - http://code.google.com/p/pubsubhubbub/

sources to the system. Due to the lack of semantics and this scheme-overloading behavior observed in RSS feeds, users of Atomate have to manually assign, for each new data source, mappings between fields and the schemata used in the rest of the system. In schema overloading situations, feeds have to first be "scraped" to extract data from overloaded fields and to eliminate presentation markup (using a tool such as Yahoo Pipes) prior to being added to the system. Finally, users have to grapple with obscure concepts such as unique IDs (inverse functional properties) in order to make it possible for the system to identify entities to update provided the arrival of new information along any particular feed.

Given the positive responses from participants of our study surrounding the perceived usefulness of the system, we anticipate that reactive automation driven by data from web feeds and APIs will soon emerge in various forms. To make use of feeds for such applications easier, content publishers and designers of content delivery and publishing platforms for the web should consider ways to improve the quality of feeds to make them more suitable for such applications.

First, data feeds should avoid the schema overloading, appropriation, and presentation markup embedding practices just described, as this creates the need for an additional "scraping/extraction" step to syntactically clean up and unpack feeds prior to use. Second, meta-data could be added to facilitate the process of schema integration. Life-logging services in particular could add meta-data describing what it is that the feed represents - the type of activity or state (locations, music listening, web page viewing, hours slept), and, even more importantly an identifier of the person or agent that is the subject of observation. Of additional benefit for interfaces supporting constrained natural language expression of entities would be natural-language labels (common names) to use to describe observed entities or observations in the interface. Finally, the use of URIs, or an indication of the inverse functional properties to use in the schema of a feed would eliminate the need for users to specify this manually in the interface.

We note that such recommendations would be easy to implement using RSS 1.0/RDF which seemingly has fallen out of favor among content publishing platforms and feed readers. The richer RDF representation would allow data items to be presented in their original schemas, since RDF permits a natural incorporation of definitions and properties from external schemata within a RSS-schema structure documents. With RSS 1.0, information about different subjects could be combined in the same feed, reducing the fragmentation that occurs with the more restrictive feed types.

[H1] 7.10. Availability

Atomate, including its source is available as free, open source software under the MIT License at http://atomate.me . The authors view Atomate as currently primarily a proof of concept, and hope that other developers will download, examine, use, appropriate, extend and improve upon the ideas presented therein. A

forum, bug tracker and message board for researchers and developers is available there for discussing ideas and collaboration opportunities.

[H1] 7.11. Acknowledgements

Atomate was developed with support from MIT CSAIL, the National Science Foundation, the Web Science Research Initiative, and a Royal Academy of Engineering Senior Research Fellowship. We would like to thank Jamey Hicks, Ora Lassila, Mark Adler, Wendy Mackay and Michel Beaudoin-Lafon for their helpful ideas, input and suggestions

[H1] References

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

[2] A. Bernstein and E. Kaufmann. GINO - a guided input natural language ontology editor. In ISWC 2006, pages 144–157, 2006.

[3] R. Ennals, E. Brewer, M. Garofalakis, M. Shadle, and P. Gandhi. Intel mash maker: join the web. SIGMOD Rec., 36(4):27–33, 2007.

[4] N. E. Fuchs, K. Kaljurand, and T. Kuhn. Attempto controlled english for knowledge representation. pages 104–124, 2008.

[5] A. Funk, V. Tablan, K. Bontcheva, H. Cunningham, B. Davis, and S. Handschuh. CLOnE: Controlled language for ontology editing. In The Semantic Web, volume 4825 of LNCS, pages 142–155. Springer, 2008.

[6] D. Huynh, S. Mazzocchi, and D. Karger. Piggy bank: Experience the semantic web inside your web browser. Web Semantics, 5(1):16–27, 2007.

[7] A. Kulkarni. Design Principles of a Reactive Behavioral System for the Intelligent Room. 2002. To appear.

[8] D. Salber, A. K. Dey, and G. D. Abowd. The context toolkit: aiding the development of context-enabled applications. In CHI ’99, pages 434–441, 1999.

[9] T. Sohn and A. Dey. icap: an informal tool for interactive prototyping of context-aware applications. In CHI ’03, pages 974–975, 2003.

[10] K. N. Truong, E. M. Huang, and G. D. Abowd. CAMP: A magnetic poetry interface for end-user programming of capture applications for the home. In UbiComp 2004, pages 143–160, 2004.

[11] T. Winograd. Architectures for context. Hum.-Comput. Interact., 16(2):401–419, 2001.

[12] J. Wong and J. Hong. What do we “mashup” when we make mashups? In WEUSE ’08, pages 35–39, 2008.