Atomate It! End-user Context-Sensitive Automation using Heterogeneous Information Sources on the Web

Max Van Kleek, Brennan Moore, and David Karger
MIT CSAIL 32 Vassar St.
Cambridge, MA, 02139, USA
emax, zamiang, karger@csail.mit.edu

Paul André, and m.c. schraefel
Electronics and Computer Science
University of Southampton
SO17 1BJ, United Kingdom
pa2, mc@ecs.soton.ac.uk

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 paper, we describe a next-generation 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.

Categories and Subject Descriptors

H.4 [**Information Systems**]; H.5 [**Information Interfaces and Presentation**]

General Terms

Design, Human Factors.

Keywords

Context-aware computing, end-user programming, reactive automation, mash-ups

1. INTRODUCTION

Most of the digital information tools we rely upon on a daily basis are still designed to facilitate manual, user-initiated information access and manipulation. In contrast, human personal assistants, such as secretaries and administrative assistants, work on behalf of their supervisors, autonomously 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, they will need to demand less explicit attention from users and gain a greater ability to initiate and carry out tasks without supervision.

The transition of personal information management (PIM) tools to the web presents an unprecedented opportunity to give these 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 a wealth of near real-time information about people, such as their whereabouts, current activities, and opinions, as well as more slowly changing information, such as people's relationships with and preferences toward other people and things.

Today, the data produced by these social networking sites and services are geared toward human consumers, e.g., letting people view each others' updates easily in RSS news aggregators. In this paper, we demonstrate that by treating feeds from these services as sensor streams, the same data feeds can also be used to drive simple adaptive, context-reactive automation, which can be used routinely to take care of a wide range of simple tasks without user supervision. To this end, we introduce Atomate, a web-based reactive personal information assistance engine. 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.

Atomate relies on two key components: the first is a uniform internal RDF 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 to modify the data model. For the latter, we introduce an guided-input, controlled natural language interface (CLNI) [1] designed to let non-programmers accurately and easily specify actions to be performed, and to inspect and edit the data model. While the use of CNLs has previously been explored for ontology authoring and semantic annotation, Atomate is the first system that uses a CNLI for letting users specify rules for reactive automation.

[1]Controlled Natural Language Interfaces apply simple grammars derived from human natural language for communicating with users unambiguously. For more information, see: http://www.ics.mq.edu.au/~rolfs/controlled-natural-languages

WWW2010, April 26-30, 2010, Raleigh, North Carolina.

In the remainder of this paper, 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.

2. ATOMATE WALK-THROUGH

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

2.1 Scenario 1: Simple contextual reminding

There are certain household chores Xaria needs to do on particular days of the week, but she often forgets to do them. Since she sometimes works late, a regular calendar alarm would interrupt her at the wrong time – either on her drive home, or while she's still at the office. With Atomate, she can set up a reminder action to trigger precisely when she gets home, delivered via a variety of mechanisms — SMS, e-mail, or via desktop notification.

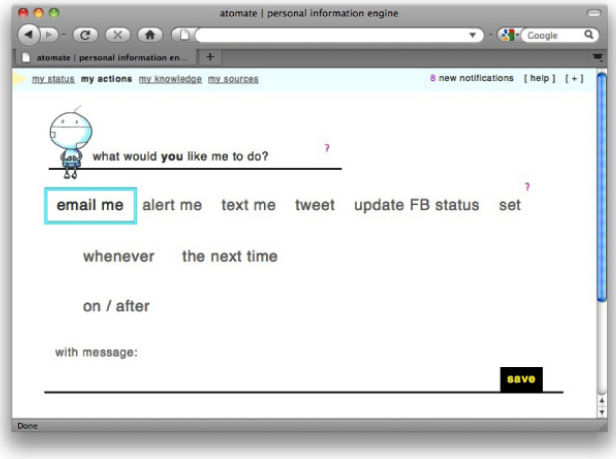

Figure 1: Atomate behavior creation interface. Actions are visible along the top row, followed by situation clause, time constraints, and optional messages.

To set up an Atomate reminder to take the trash out when she gets home on Tuesday evenings, Xaria clicks on the Atomate tab in her Firefox browser, and clicks "My tasks". This displays the interface visible in Figure 1. She first specifies what she wants done: she selects "alert me" (Figure 2.1), which defaults to a a combination of desktop notification and an SMS 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" (2.2).

Clicking on "whenever" makes visible the situation specifier, visible as the "who does what" line (2.3). This interface is used to easily specify the contextual and situational conditions under which an action should occur. Such constraints are evaluated against Atomate's RDF world model, which is formed from the integration of various near-real-time web data streams about various people, places, things and their relationships. Since Xaria wants to be reminded *when she gets home*, she clicks the "who" slot, which produces a list of all the entities with properties in the world

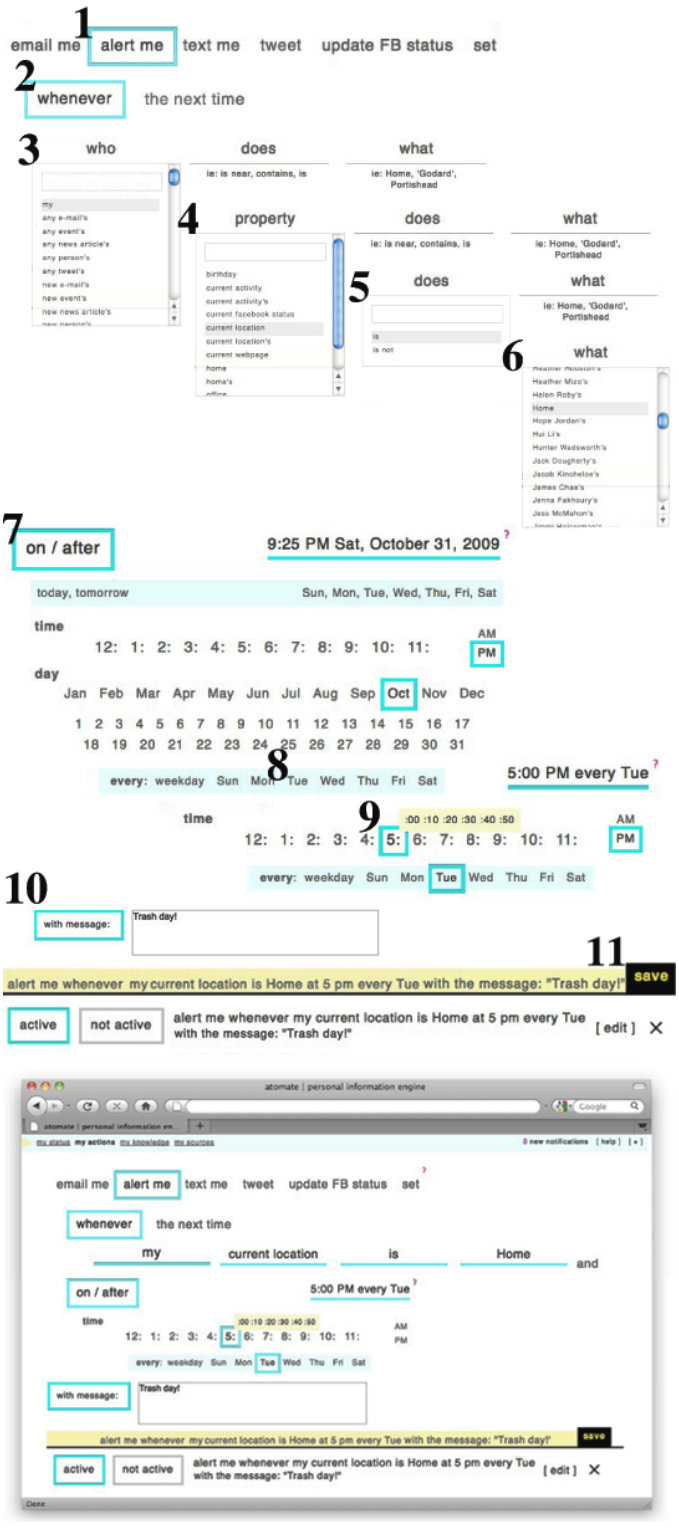

Figure 2: Top: A step-by-step montage illustrating the construction of the reminder behavior described in Scenario 1, using Atomate's constrained simplified natural language UI (CLNI). Bottom: The completed behavior.

model, and selects the top choice, "my" which represents her entity. Atomate displays all her entity's properties, including her "current activity", "birthday", "current location" and so on. She selects "current location" (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" (2.5), and this moves the selector to the object ("what") slot, where she selects "Home" (2.6). (She could have equivalently selected "[[my home]]" , which would refer to the same entity via her entity's "home" property).

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 (2.7). This produces a calendar widget, where she selects "every Tuesday" (2.8). Finally, she specifies the reminder message by clicking "message" (2.10) and typing "Trash day!".

As Xaria constructed her new behavior, Atomate displayed its partial interpretation of the rule as-completed at the bottom of the screen. Now that the behavior is finished, Xaria reads this interpretation to herself to verify its correctness: "Alert me when my location is home on/after Tuesdays at 5pm with the message: Trash day!". She confirms the rule by hitting Save (2.11) and the behavior appears in her Atomate's list of tasks.

Although it is ambiguous as specified, it is more likely that Xaria intends on having the system remind her *once* on Tuesday upon reaching her home than over and over repeatedly *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 specifically to execute once, only when the world model's state changes to satisfy conditions for execution. We describe this in detail in Section 4.3.

2.2 Scenario 2: 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 he's forgotten about them, in other cases he misses events deliberately when he's busy with other things. In either case, he wants to be notified which of his friends (if any) attended events he's scheduled, so that he can intervene (either to to tell them he's coming), ask them about it afterwards, or just even to decide whether or not to go.

To set up an appropriate event, Ben opens Atomate and clicks "New task". He selects the "notify me" action, and "whenever" to specify a repeating action. Next, he constructs the first condition clause by selecting, in sequence "[my] [current calendar event's] [location] [is not] [my] [location]". Next, he adds a second clause: "[any friend's] [location] [is] [my] [current calendar event's] [location]". This clause uses the "any" wildcard entity specifier, which is evaluted against all entities of the specified type. Ben verifies that the rule looks correct by inspecting the English rendering and hits save.

For this rule to work, Ben needs to have provided Atomate with two types of data sources: a location source for him and his friends, and his calendar[2]. His location data

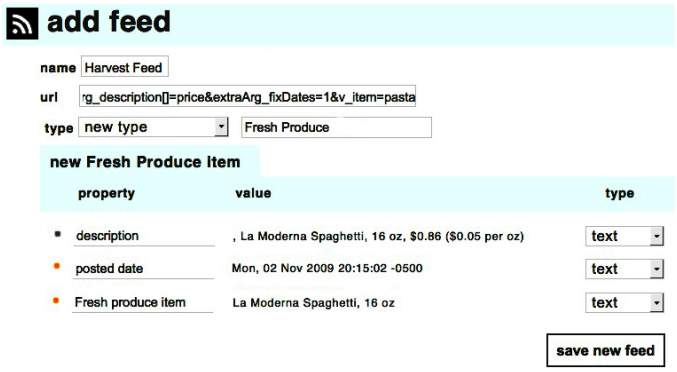

Figure 3: Scenario 3: Sherry adding a new data source to Atomate

source is configured to set the "current location" property on entities representing him and his friends in Atomate's world model, while his calendar is configured to update the "current calendar event" property on the entity representing him. Section 5.4 describes how new data sources can be configured to update properties automatically.

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."

2.3 Scenario 3: Extending the system

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

Sherry prefers buying groceries from Cropz, a local organic grocer, but the availability of produce items there changes few every days. Cropz recently 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.

Her grocer does not publish an RSS feed, so she uses DAPPER(.net, the data mapper) to construct a mapper to scrape the grocer's site every couple days. Once she constructs the mapper, she copies the address of the resulting RSS feed to her clipboard, and clicks on the My Feeds section of Atomate (Figure 3. She selects "Create a data source" and names it "Cropz Feed". She then pastes the DAPPER feed URL into the "source" field. Atomate retrieves the feed and displays the current feed entries below. Next, she changes the type of the entries from the default ("News Story") to "Fresh Produce" by clicking on the "type:" property value, selecting "new type" and replacing the entry value. This causes all entries in the feed to assume the same type — a new *rdfs:Class*, since Atomate had no previous record of a

[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.

class named "Fresh Cropz". 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 Cropz and any Fresh Cropz's posted date is within the past 24 hours".

3. RELATED WORK

Architecturally, Atomate resembles work on blackboard systems [11], early sensor-fusion architectures for pattern-directed problem solving using heterogeneous information sources. Atomate's information sources (e.g., web sites), correspond roughly to a blackboard's "experts", while its end-user authored behaviors correspond to the pattern-directed programs in these systems.

Context-reactive automation has been explored primarily in the ubiquitous computing domain, such as with architectures such as the Context Toolkit [8], and ReBA [7]. Research exploring interfaces for end-user construction of such automation include iCAP [9], which proposed a sketch-based interface for programming home automation, and CAMP [10], which explored a magnetic-poetry metaphor for the construction of behaviors by end-users.

The design of Atomate's rule creation UI (Section 4.4.2) was inspired by research from the semantic web community pertaining to simplified and constrained-input natural-language interfaces for knowledge capture and access. In particular, the CLOnE [5] and ACE (Attempto Controlled English) [4] work introducing controlled language languages (CNL), and related GINO [2] and GINSENG interfaces for guided input interfaces for CNLs were the basis of Atomate UI's design. Since Atomate uses a rule based system at its core, emerging Semantic Web work pertaining to rule languages (such as SWRL and RuleML), and efficient chainers for these languages are currently of great relevance and interest to us well.

Web mash-ups have explored the potential for combining information from multiple sources on the web. Systems for enabling the end-user construction of such mash-ups include MashMaker [3], while tools like Pipes (http://pipes.yahoo.com) have focused on end-user processing and manipulation of web feeds. A survey of 22 mash-ups [12] concluded that most mash-ups enabled the construction of custom visualizations, while none were designed to deliver reactive behaviors (e.g., actions) based upon mashed-up data. In this work, we show that reactive actions based on mashed-up data are both easily created and useful for saving people time and effort.

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.

4.1 Data flow

Atomate works in a reactive loop retrieving information from the web, updating its world model, and triggering user-specified rules. As portrayed in Figure 4, Atomate's feeder component periodically retrieves from Atomate's RDF model a list of all the data sources the user has created, and polls each. For most data sources (corresponding to RSS/ATOM feeds), this poll 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.

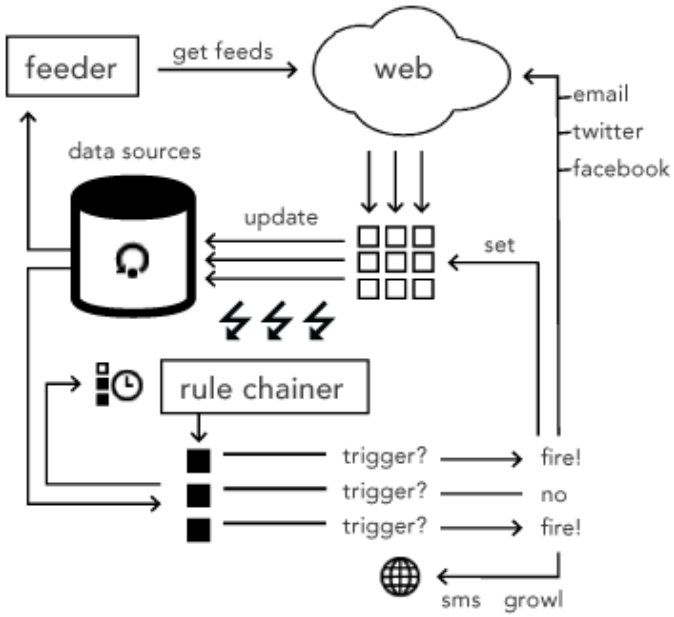

Figure 4: Atomate data flow. Atomate pulls data from the web and updates the world model. Model updates are fired and picked up by the rule chainer, which evaluates triggers for user-specified behaviors (rules). Triggered rules that depend on changed entities are fired. The effects of such behaviors can be to update the world model (Set) or to call external services.

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 the user initially sets up the data source. The process of creating such mappings is described in Section 5. For example, a Twitter feed would cause new *Tweet* entities to be created in the RDF world model, while entries from Facebook would cause updates to and creation of new *Person* entities.

Updates to entities in the world model are noticed by the rule chainer. When such updates occur, the rule chainer retrieves all *Rule* entities from the world model. It evaluates each rule antecedent, and fires all rules whose triggered antecedents depend on the entities that changed. As described further in Section 4.3.1, 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 world model (if they have a "set" clause, described in Section 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. As described in Section 5 new actions can be easily added to call arbitrary web services. If the world model is updated as a result of this firing, the chainer is informed and the rule trigger evaluation starts again.

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 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 5. It also facilitates integration with new external data sources, also as described in Section 5.

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. The antecedent of a rule in Atomate consists of conjunctions ("ands") of clauses. Each clause can consist of a either a predicate applied to the current time (to cover temporal constraints), or a binary relational predicate which can be applied to values in Atomate's RDF entity database. Each argument of a binary relational predicate can be either a simple primitive type (number, date, or string) or a *path query* over Atomate's world model. Each path query originates at either a single specific entity (node) or all entities of a particular type. This latter situation occurs when the user specifies a condition with the "any" expression, such as "any Person", "any Event" or "any e-mail", described in 4.4, which lets users express situations pertaining to any instances of a particular type. The tail of a path query consists of properties which are traversed from the origin node(s). These paths of properties can be of arbitrary length and are expressed as chains of possessives in the CNLI. A rule is considered triggered when an interpretation can be found that maps entities to query expressions and satisfies all of the clauses in the antecedent.

An additional feature supported by the rules is the "that" expression, 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.

A rule's consequent consists of a single action; this action either directly changes the world model through the *Set* function (which updates the property of an arbitrary entity in the system), or calls upon an external service. The default set of actions provided in Atomate are set, e-mail, alert, text, post a tweet, update Facebook status and '*Set*'. Section 5 describes how actions can be added to the system by advanced users. When an action is called, it is provided as an argument to 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. The convenience method *toSimpleEnglish()* can be used to render path queries into simplified english. The *Set* action allows users to create rules that update an entity's properties, as described in 5.4.

4.3.1 Triggering and scheduling

Computationally, the rule chainer is a naive forward rule chainer that uses a brute-force approach to evaluate rule antecedent clauses. For each behavior, the rule chainer determines 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 dereferenced by the appropriate path queries, yield values that satisfy the predicates in the antecedent. For evaluating existential "any <type>" expressions, the chainer searches all nodes of the appropriate type to determine satisfaction.

As introduced in Scenario 1, most behaviors exhibit the assumption that they will fire once when the rule's antecedents transition from not-satisfied to satisfied. Atomate's rule chainer implements this assumption as follows: when an entity is updated in the world model, the chainer is informed of the change, in particular the entity, property and value(s) affected. Then, upon evaluating the antecedents for all behaviors, the chainer only fires rules if the values that caused them to trigger corresponded to any of update operations.

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.

4.4.1 "My stuff": The Entity Explorer

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 autocompleting, 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. The user can merge any redundant items by selecting them and clicking "merge items." Entities 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. As entity properties are changed in the world model,

the interface reflect such changes using fading animations to make the changes clear. Users can inspect the recent change-history of a particular property's value by right-clicking on the value.

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, or easier, to use than standard reminder-alarm dialog boxes often visible in PIM calendaring tools such as Outlook, and yet provide vastly more expressive reminder capabilities, using a constrained-input guided simplified natural language interface as explored by the semantic web community for ontology creation and annotation.

Scenario 1 walked through one example of using the rule creation interface visible in Figure 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. Actions consist of notifications such as "alert me" and "email me," social actions such as "tweet" and "update my facebook status" and the "set" action for updating the property value of an arbitrary entity in the world model. 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 2:3-6. Atomate continually modifies the displayed options such that an invalid rule cannot be created. The user may add additional clauses by clicking "and." In addition to the condition clauses, the user may specify a time constraint for the action to occur; this is done using the calendar widget shown in Figure 2:8-9 which appears by 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.

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 MozStorage APIs, which stores all data in an SQLite file in the user's Firefox profile. Atomate's RDF-JS 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 rule scheduler, database engine and feeder are packaged in a Firefox XPCOM component, and remains a singleton per Firefox process (which is limited by FF 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.

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, and GData JS can be readily used to obtain data.[3] New data sources with custom client libraries can be added by web developers without any need for special integration work.

[3]Atomate complies with Facebook's TOS by tagging data obtained from FB with a 24-hour expiration date.

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.

5.1 Adding predicates and actions

New predicates can be added to Atomate to add more powerful entity comparison operators or to overload operators for new entity 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.[4] Such an implementation may call an external web service to perform the relevant action or comparison evaluation.[5]

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(.net/open), Babel (http://simile.mit.edu/babel), or Yahoo Pipes 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 [6], which lets the user construct a visualization from multiple sources with dissimilar

[4]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 and is easily accessible from within the entity creation panel.

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

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 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 which is used to structurally convert incoming items on its feed prior to being added to the world model.

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.

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. To do this, the user performs 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 an superclass 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.

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 world model. 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 4.3 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 world model.

6. EVALUATION

Since a users' understanding and use of the rule creation interface will largely determine their success with the system, our first goal for the evaluation of Atomate 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 explore 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 Table 1 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.

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 (or due to technical error we are missing it). 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,

Rule 1	You have a meeting with a colleague tomorrow at 3pm. Set a reminder.
Rule 2	You have to provide a work status report every Thursday at 2pm. Set a reminder.
Rule 3	Set up an alert that notifies you whenever anyone you know is near your house.
Rule 4	Set an alert that notifies you when your boss, John von Neumann, arrives at his office.
Rule 5	You often forget to bring your shopping list with you to the store. Have atomate text you your new shopping list (1. eggs. 2. bread. 3. milk) to you when you arrive at your local grocery store (Cropz).
Rule 6	You have been buying too many books from Amazon.com. Remind yourself every time you visit amazon.com to check your local public library for the book.
Rule 7	You are working on an urgent project with Vannevar Bush and want to make sure to not miss new e-mails about it. Have Atomate alert you when you receive a new email from him containing the word "MEMEX" in the subject line.
Rule 8	Have Atomate automatically update your facebook status when you are at a concert.
Rule 9	Have Atomate send you a text message when you have an activity scheduled in 5 minutes that is not close to where you are.

Table 1: The nine rules participants were asked to create in the evaluation.

and thoughts as to how and when participants would like to use it now or in the future.

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

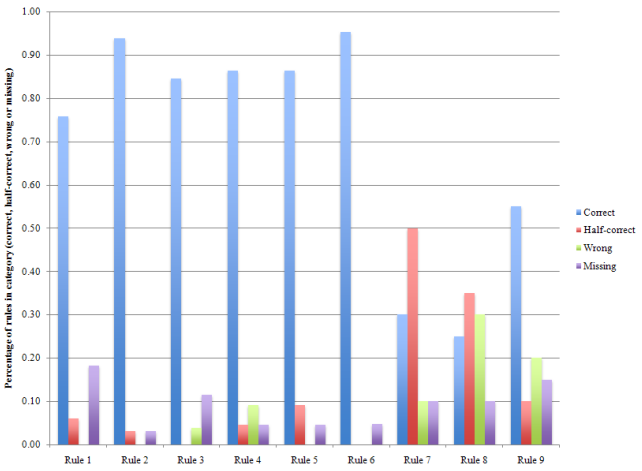

Figure 5: Percentage of created rules that were correct, half-correct, incorrect, or missing.

and in what ways participants thought such a system would be useful to them.

Accuracy and Ease of Use

As described in the previous section, we classified each answer into one of four categories: correct, half-correct, incorrect or missing. Figure 5 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', 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 6(a) 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 8.

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

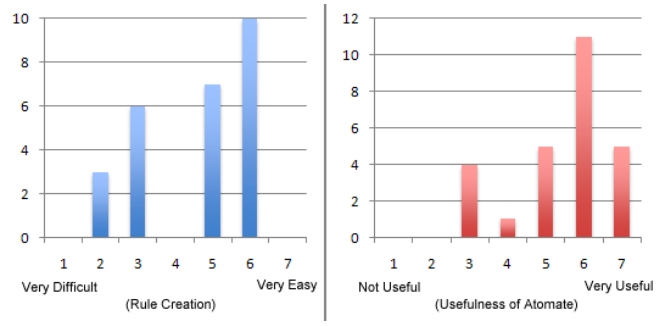

Figure 6: Results from the rule creation survey. (a) How easy was it to create each rule? (b) Do you think Atomate would be useful to you?

alterations to the language displayed in the interface, such as changing "at" to "on / after" for adding dates and and "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 8.2, 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.

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 6(b)). 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.

8. ONGOING WORK

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

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 "engageable" actions through "while" rules. Unlike the "next time"/"whenever" rules described earlier, "while" rules will engage an action when a rule triggers, and disengage the action 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.

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.

8.3 Sharing behaviors and activity feeds

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 a online community repository where users can upload entities of their choice, and publish them with descriptors so that they can be easily found, reviewed, revised, and re-used.

To make it easy for users to connect their Atomates, we plan to add functionality to let Atomate publish state changes as an RSS 1.0 feed known as *your peRSSona*. Each Atomate will support publishing multiple *peRSSona*s providing different degrees of disclosure/privacy for different consumers – controlled simply via behaviors that invoke "post to feed" actions based on particular state changes.

8.4 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. Because 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[6] with which feed clients register for change notifications at "hubs" for a num-

[6]http://code.google.com/p/pubsubhubbub/

ber of feeds. New information is pushed to clients only when changed.

9. DISCUSSION AND CONCLUSION

We have presented Atomate, a system to allow end-users to utilize 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 initial 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 [1]. 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 DL 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.

The web of today that Atomate works with 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 sources to the system. As described in Section 5, the lack of semantics forces users of Atomate to have to manually assign, for each new data source, mappings between fields and the schemata used in the rest of the system. Further, due to the "embedding" behavior exhibited in many RSS feeds, feeds often have to be first "scraped" to separate data and eliminate presentation markup from overloaded fields using a tool such as Yahoo Pipes, prior to being added to the system. Finally, users have to grapple with abstract concepts such as unique IDs (inverse functional properties) in order to make it possible for the system to identify corresponding entities to update provided the arrival of new information along any particular feed. These tasks raise the barrier of letting end-users extend and appropriate the system to their needs.

However, given the positive responses surrounding the perceived usefulness of the system from study participants, 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 described 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.

10. REFERENCES

[1] T. Berners-Lee, J. Hendler, and O. Lassila. The semantic web: Scientific american. *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 ont. 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 Semant.*, 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.