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

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

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

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

Most people rely on their ability to effectively draw upon, process, and use a wide variety of information in order to plan and execute their work and leisure activities each day. The field of personal information management (PIM) has sought to understand and build tools to support people's information needs, in particular to help people effectively remember, manage and recall large quantities of information. Today, several classes of such tools dominate the PIM landscape; in particular, electronic calendars, e-mail clients, address books, to-do item managers and various note-taking tools.

While these tools have become increasingly capable and available to us pervasively through the web and our mobile devices, they all exhibit a simple limitation -- they require explicit user interaction. By contrast, human personal assistants (PAs), such as secretaries and administrative assistants, routinely do things on behalf of their supervisors, such as taking calls, handling visitors, coordinating meetings and daily schedules, to name a few. Our perspective is that in order for personal information management tools to start to approach the helpfulness of human personal assistants, they will require some degree of autonomy, and ability to work on the behalf of a user without explicit human attention.

Designing PIM tools to approach the helpfulness of human PAs is, of course, difficult for a number of reasons. Human PAs possess rich extensive general knowledge of the world, deep domain knowledge pertaining to the tasks they need to perform (as professionals), and a thorough understanding of the person(s) whom they serve, including the person's preferences and needs. Building a digital personal assistant endowed with such levels of expertise and competence, while conceivable, is outside our present capabilities.

One alternative approach is to let users specify their desired autonomous or reactive autonomous behaviors directly, essentially programming PIM software to perform tasks for them. In the same way that computer system administrators routinely automate their workflows using tools such as shell scripting languages and cron [4], users should be able to delegate simple, ordinarily attention-intensive but well-defined personal information-related tasks to their personal information tools. A simple example of PIM automation features that have become already commonplace are e-mail filters and calendar alarm/reminder functions, which reduce the attentional demands of our e-mail clients and calendaring tools, respectively. In this chapter, we describe an experiment in expanding the role of end-user automation towards other kinds of more general personal PIM tasks, through a rule-based reactive programming language designed for end users, driven by rich Web-based personal information sources.

1. Personal reactive automation through aggregated information from the Web

The growth of web-based PIM applications (such as Google Calendar[1], RememberTheMilk[2] and del.icio.us[3]) and social networking and activity sharing portals (such as Facebook[4], last.fm[5], Twitter[6], and Plazes[7]) have made available an unprecedented quantity of rich personal information about people and their activities on the Web. Several of these sites (including Plazes and last.fm) publish near-real-time updates of people's locations and activities through automatic tracking software installed on users' personal devices.

In this chapter, we describe a system called AtomsMasher[8] (AM) that applies the rich, timely information published about people on the Web to trigger and drive end-user constructed adaptive, reactive behaviors that can be made to perform simple and useful PIM-related tasks. For example, AM can be used to set up an adaptive "Away responder" that can automatically determine, based on a user's calendar entries or location, when and how to automatically reply to messages of particular types. (See Section 3.1 for example AM behaviors.)

To support reactive behaviors using Web 2.0 information sources, we need to bridge two lines of research. Mash-ups [5][26] typically blend two or three data sources to re-present a view of the sources' data, but usually provide no means of scripting actions based on this data. Active scripting in end-user programming [3][14] affords customizable actions to be assigned to particular conditions, but so far have been built for closed domains where entities and actions are known and specified in advance.

By contrast, AM dynamically builds a data model based on information aggregated from heterogeneous sources on the web and maintains this model in a simple RDF representation. A simple rule chainer at the heart of the system triggers based on changes to this model, and maintains an open registry of simple

[1] Google Calendar: http://calendar.google.com

[2] Remember The Milk: http://www.rememberthemilk.com

[3] del.icio.us : http://del.icio.us

[4] Facebook : http://www.facebook.com

[5] last.fm : http://www.twitter.com

[6] Twitter : http://www.twitter.com

[7] Plazes : http://www.plazes.com

[8] AtomsMasher is open source under the MIT License and can be downloaded at http://plum.csail.mit.edu/atomsmasher

predicates and actions. Together, these components create a complete reactive rule-based system based on data derived from web information sources. To make specifying behaviors in AM intuitive and easy for end-user non-programmers, AM provides a constrained simplified natural language (CSNL) UI, with which users state simply *what* they want to happen and *when* they want the behavior to be activated. This CNL UI is intended to let programming behaviors resemble instructing a human PA to do something later, except using a simplified constrained vocabulary to simplify interpretation.

In designing AM, we sought to answer the following questions: a) whether current data feeds have sufficient, timely information to drive useful reactivity b) finding a flexible and scalable method for consolidating and integrating heterogeneous information from diverse web sources c) whether a CLUI would make the specification of desired automation easy and error-free. In the following sections we consider related work, followed by a brief walkthrough of AM's UI and some examples of its use. We then provide a detailed overview of the AM framework, discuss how the framework can be extended to new data sources and capabilities, and describe ongoing work towards making AM behaviors sharable and rule conditions easier to express.

2. Related Work

AtomsMasher draws on work from several fields: it is an end user, reactive behavioral programming environment, for personal information tasks, driven by heterogeneous web data sources. We report on the most similar and seminal works in the following related areas: symbolic rule based expert and control systems (AI), end-user programming, context-aware computing, and most recently web mash-up and semantic web research.

An early system that combined similar goals with a similar approach (rule based system) was the Information Lens [15], a system that introduced end-user constructed rules for more effective filing and management of (the closed world of) e-mail. With respect to featuring context-sensitive reactivity, context-aware and ubiquitous computing systems have sought to achieve physical, environmental and user task context-aware applications based on perceptions primarily from sensors. A few of these systems focused on end-user construction of reactive behaviors. These included: a macro recording system in the Intelligent Room let users program macros physically by example [7], iCAP [20] which let end-users sketch their desired situations and actions, and CAMP [20] which used a magnetic-poetry metaphor for the construction of similar behaviors.

With respect to end-user programming on the web, Chickenfoot [3] and Co-Scripter [14] introduced programming environments for letting users automate common repetitive web tasks and customizing pages with additional functionality. These systems dealt primarily with the domain of the web page, with objects and actions pertaining to pages, their structure, and navigation. AtomsMasher extends these systems by providing a rule engine for automatic execution of scripts, a repository of external information that scripts can use in their actions to be more adaptive, and actions that support scripting "off the page", e.g. web services.

With respect to combining information from multiple web domains in end-user web environments, web mash-ups have recently popularized the act of taking live data provided by one site and using it in the context of another, such as for producing visualizations. One paper [24] surveyed 22 such mash-ups and their function, and concluded that most mash-ups surrounded the construction of custom views of data, or bringing data to the desktop. There was an absence of any mash-ups which produced reactive behaviors (e.g., action) based on data from multiple sources.

The end-user construction of such mash-ups has been the focus of a number of web-based tools including Marmite [23], MashMaker [5], and Pipes [26]. The focus of these tools has been in facilitating the creation of combined feeds, views and simple visualizations of combined information from arbitrary feeds or services on the web. AtomsMasher's approach in retrieving and aligning sources differs in several ways. We use a relational representation in RDF (unlike most mashups that align at the

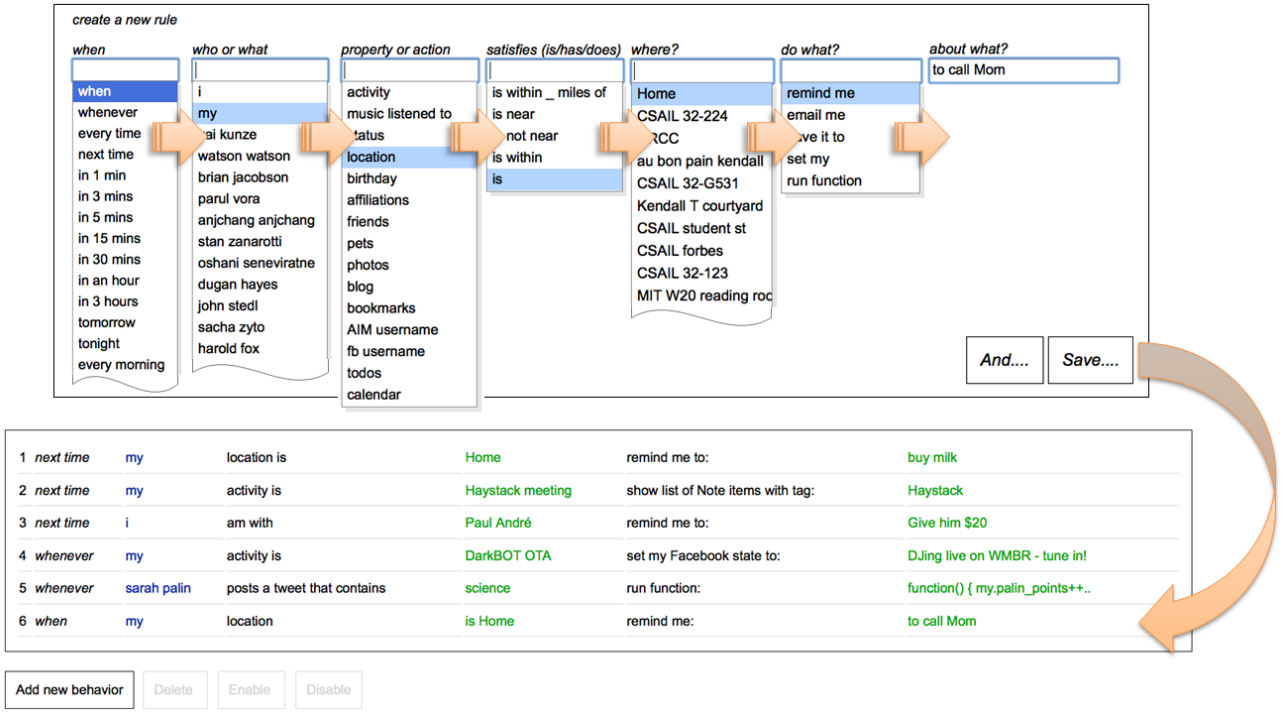

Figure 1: AtomsMasher's simplified Constrained Natural Language (CNL) interface for rule specification, described in S3.1

syntactic or structural level), which supports rich linking of related data items, simplifies integration and scaling to new data sources and types, and reduces dependence on the source representation. AM also supports the integration of private data sources such as e-mail and sources on the user's desktop.

From the semantic web community, AM relies on various RDF technologies including OWL [16] for expression of its internal representation, and toolkit Jena [9] for storage, reasoning and query. As described in section 4.4, AM's UI was inspired by work on constrained input and simplified natural language interfaces (NLI), particularly GINO [1] and GINSENG [13] method of constrained input interaction.

3. Overview

AtomsMasher helps users by letting them delegate simple actions to perform when certain conditions are met. These reactive conditional actions are called behaviors, and are expressed in the system as sets of rules (section 4.2). Like in standard rule-based systems, each rule contains an expression of the condition(s) under which it should execute, and the action to be performed when its antecedent is satisfied. In AM, rule antecedents are comprised of conjunctions of predicates which represent relational constraints between entities in AM's internal model, known as AM's knowledgebase (KB). These predicates and relations, described in section 4.3.1, typically are surrogates for tangible real-world things in the user's life, such as people, places, and events, and real-world properties/relations among those things, The AM KB is constructed and maintained by AM data source modules, which, as described in section 4.3.2, use information from both web and desktop based sources to build the KB. The updates that these data source modules perform to the KB cause conditions of rules to be satisfied, which, in turn, drives AM to perform actions, described in section [4.2.2].

3.1 Walkthrough and Examples

Prior to describing the system components, we run through an example of how a user interacts with AM to set up a simple reactive behavior. Following this, we provide a number of other examples of typical uses for AM.

Figure 1 illustrates AM's main interface and the process of rule creation. In the figure, the user, Xaria, is setting up a simple rule for AM to remind her to call her mother when she gets home. When Xaria selects the AM bookmark on her browser, AM's rule user interface appears (bottom of figure). From this UI she can add new rules, monitor their execution (not shown) or selectively enable and disable rules in her collection.

In this scenario, she clicks "Add new behavior". Here she begins to specify the antecedent for her rule, which describes the situations under which the rule should run. AM prompts her with an auto-complete input box labeled "when". AM is asking her whether she wants her rule to run at a specific time (which she can type directly) or based upon some event or situation. She types "when", which indicates to the system that she wants it to run the rule only once. (She could have typed "whenever" if she wished for the rule to run whenever she got home). AM then prompts her with a second input box: "What?". This box is pre-populated with a list of entities in her life: people she knows, places she has been (and heard of through event web sites), and files on her computer. It also includes the special terms "some <type>" and "new <type>", where types correspond to the major types of things AM knows about: people, places, web sites, and events. She selects "my", indicating to AM that she wants the rule to run when one of her properties assumes a particular value. AM then asks her to specify which property (she selects "location"), a comparison verb phrase (she "is"), and finally, the name of a place ("Home"). In each step, AM only presents possible options that are valid based upon what she has specified thus far. Finally, AM prompts for the (re-)action to take and arguments, in this case “remind me”, “to call Mom”. This new behavior is then saved to the rule interface. The “remind me” action can be configured to deliver its reminder notifications through a variety of channels, including desktop-based (Growl, IM, email, synthesized TTS) or mobile (SMS). A further example of context-based reminding can be seen in a social-contact event, for example:

next time i am with Max Van Kleek
remind me to “tell him about short film festival”

With respect to facilitating focus on particular tasks, AM can be used to automatically retrieve information depending on a user’s location or activity, set via their calendar for example.

next time my activity is Haystack meeting
show bookmarks tagged “haystack”
open document http://groups.csail.mit.edu/haystack/blog

Social Co-ordination and Awareness. AtomsMasher can be used to publish custom feeds tracking particular updates in a user's life activities. This is to allow users to better control their privacy while allowing their coworkers and friends alike to keep track of their whereabouts.. To do this, an AM user can simply set up a behavior to post to a particular feed, identified by a name. If a feed of that name does not exist already, AM creates one. In the following simple example, Xaria has specified a rule for having posted on a feed known to lab colleagues only when she's at lab:

whenever my location is some place
and that place is located in or part of MIT CSAIL
then post to feed labmates-feed
“Xaria is now at ”, my location

In addition to generating ATOM/RSS feeds, AM can be used to post updates to social networks (such as updating an IM state, Facebook status or Twitter).

Communications Mediation. Social coordination overhead often occurs due to lack of awareness in both synchronous and asynchronous communication. For example, it may be difficult to tell an appropriate time to interrupt someone over IM, or how long to one might have to wait to expect a reply via e-mail. AM can be used to assist in such situations in several ways. In addition to automatically setting one's IM/away status based on aspects of a user's recent activity, AM can also be used to route and handle personal communication. This is an example of a deluxe e-mail "auto responder" which engages automatically when the user has not been at their computer for two days, and supports the option of forwarding messages automatically to one's mobile phone based upon who sent the email.

whenever I receive new Email
and that email’s sender is some Person
and that Person in Friends
and my idle time is greater than 1 day
and yesterday not in holidays
and yesterday is a weekday
send reply to that Email with contents “Hi, Sorry (...) If your message is urgent, please reply with subject starting “urgent...”

whenever I receive a new Email
and that email’s subject starts with “urgent”
and that email’s sender in my friends
send a text message to me with that email

Mobile Information Retrieval. AM can be used to retrieve private information from your personal calendar, workstation or social network using SMS text messages, e-mail or your communications medium of choice.

whenever I receive a new IM
and that IM is from me
and that IM contents is "@am today"
then find all events with start date is today
send reply to that IM with those events

This last example used a special construct (find/those) which syntactically resembles an antecedent and returns the entire set of bindings that satisfy the clauses. In the next section, we examine the individual components of AtomsMasher needed to realize these examples.

4. Architecture and Data Model

In this section we describe how AtomsMasher works, including how rules are evaluated, predicates and actions are implemented, and AM's KB is built from external information. We highlight the key issues in each.

4.1 System Architecture

AtomsMasher differs from traditional web applications in that it runs entirely on the client (the user's workstation). It requires a browser (currently Firefox 3.0+ is supported) and a background (Java-based) process running on the user's machine. The browser hosts AM's user interface and web-based data components, including data sources, web-based predicates and action functions. The Java process maintains persistent state and the core rule monitoring/execution loop, as well as integration with PLUM [21], a desktop activity monitoring framework .

This browser-Java split was motivated by three major reasons. First, it enabled the use of a large number of convenient APIs available on both sides. For the UI, AM relies on various web APIs including (YUI [27], Google Maps [8], Simile Exhibit, Timeline and Timegrid [19], flot [6]) and animation tools (jQuery

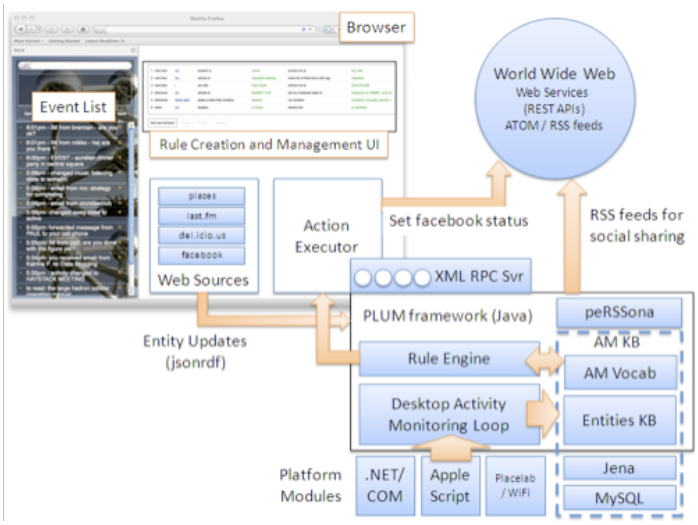

Figure 2. AtomsMasher architecture and components

[11]). For connectivity to web-based data sources, AM relies on various web service client libraries (Google Calendar Client, Facebook Connect API), and facilities for retrieving and parsing XML from feeds easily and efficiently (jQuery). For persistence, AM uses a Java RDF graph API that provides OWL reasoning (jena [9]) and reliable persistent storage (JDBC to mysql/postgresql).

The second motivation for splitting the architecture was that several components that were considerably compute-intensive – rule based trigger evaluation and RDFS reasoning [18] over entities. By pushing these to the Java process, we are liberated from single-process/thread JavaScript limitations in browsers, and can readily take advantage of multi-core architectures when permitted by the Java VM. The final reason to split the architecture was motivated by the desire to support code-sharing and end-user extension to new data sources, operators, and actions We therefore designed these components to be modular and implemented in JavaScript. The two halves of AM communicate using an RDF data model [17] expressed in JSON syntax [12] over XML-RPC [25].

4.2 Rules in AtomsMasher

Rules constitute the basic unit of AtomsMasher's reactive behaviors. Like rules of other rule-based systems, each of AM's rules consist of a set of conditions that characterise the situation under which a rule should be executed (known as the *antecedent* or *if-part*), and the set of actions to be carried out when these conditions are met (known as the *consequent*, or *then-part*). The conditions in the antecedent are expressed as simple conjunctions of predicates, each of which may take entities or values as arguments.

While AM internally makes no distinction among rule types, in practice rules in AM consist of one of two types: those that update AM's internal world model based on new events arriving externally (also known as *updatelets*), and *behavior rules* that take external action based on updates to the internal model. As described later, keeping rules separated into these types decouples knowledge sources from action, making it easier to scale AM to new sources and actions.

4.2.1 Rule triggering conditions (Antecedents)

Antecedents for rules in AtomsMasher are either time or relation-based. To make it convenient for users to create behaviors that only execute once, all antecedents may be declared to be one-shot, triggering only the next time an event that satisfies the conditions arrives, or recurring, triggering for each new event that occurs indefinitely.

In their most basic form, time-based constraints let users specify an exact date and time of execution e.g., "11:00am Monday November 3, 2008", like standard alarms available in many applications today. However, AM also permits time constraints to be partially left unspecified, such as the day, hour, or minute. This capability was motivated by our prior work in information scraps [2], in which we found that people rarely specified times to-the-minute; instead, they often used relative specifiers and more vague descriptors of time such as "tonight", "tomorrow morning" and "next week". Since the goal of AM was to be a personal automation assistant, we felt it important to support these natural forms of time expression. Finally, since events in the world are often repeated, time-based antecedents can also be recurring, such as "every weekday morning", "every third Sunday".

While time-based antecedents only specify the time to execute a rule, relation-based constraints specify conditions on relations among entities in AM's knowledgebase. Such constraints are arranged as a conjunction of predicates applied to one or more operands. These operands can stand for a specific entity (e.g., a

person, place or thing), a property of a specific entity (e.g., the name/latitude/start date of a person, place or thing), or any entity of a specific type (e.g., the arrival of a new e-mail, calendar event or friend profile). The most common predicate, "is", when applied to a property of an entity, merely checks for the presence of a particular relation between entities in the KB. Other predicates can be used to test more elaborate relations between entities, including relationships not explicitly stated in KB model.

4.2.2 Rule Consequents

Rule consequents ("then parts") consist of a sequence of calls to action functions. These action functions, described further in section 4.3.1, are applied similarly to predicates; they can take any number of operands, each of which may be an entity or a primitive value. These operands may consist of references to entities bound in the antecedent, using pronouns, as described in section 4.4.

4.2.3 Triggering and execution semantics

The rule scheduler ensures that rules are run precisely when their antecedent conditions are met. To make this process computationally feasible (e.g., to not have to test every rule antecedent every second), AM's scheduler only considers rule antecedents when they might trigger. This process is described as follows.

AM's scheduler first determines whether rule antecedents are time-dependent or event-dependent or both, and the specific types of events upon which they depend This event could be the creation of a new entity, or the modification of a property of a particular entity. When such an event occurs, the scheduler only considers the rules that could be affected by that event. For example, a rule that conditions only on the user's location would only be considered whenever the user's location changes, while a rule conditioning on the user's location and new arriving email would be considered whenever either event occurred.

Rules with time dependencies are also scheduled lazily when possible. For rules with simple time constraints (such as "3pm tomorrow"), AM's scheduler registers a system timer callback to signal when to next consider the rule. For recurring time constraints, the scheduler sets an alarm for the soonest definite moment that the event will be satisfied, which is re-scheduled when triggered. The remaining discussion surrounds time constraints that result from antecedent clauses comparing properties of entities to "now", such as "whenever an event's start time is now". For such antecedent clauses, the scheduler's policy depends on the predicate being applied; for "is", "intersects" or "is after", AM's scheduler reads the property's value and sets an alarm callback to check back at that moment. For "is sooner than" AM merely includes the rule as a candidate until the tested time expires, and removes it afterwards. If AM's scheduler cannot figure out how to lazily handle an expression involving "now" (such as with new operators in the future), it schedules it for consideration once a minute.

Once a rule antecedent has been evaluated and if it is satisfied, AM will immediately execute the actions named in the antecedent. If the rule antecedent is evaluated and not satisfied, it does not have to take any particular action because the rule will be re-considered when the operands of the clause(s) preventing satisfaction receive updates. This combination of event and time-based lazy triggering evaluation saves considerable computational burden by reducing the need to evaluate rule antecedents

Web sources:		
google calendar	Event information from calendars	Event
twitter	Retrieves all posts by people user is following	Tweet
facebook	Retrieves personal info and updated status of all friends	Person
plazes	Retrieves Plazer sightings of friends and self	Sighting
last.fm	Retrieves scrobbled music listening activity for friends and self	MusicListeningAction
upcoming.org	Retrieves musical & performing arts-related events in local area	
rememberthemilk	Retrieves task items created on the RTM service	Todo
iwantsandy	Retrieves to-do list items created on the Sandy service	Todo
PLUM sources:		
IMAP	Retrieves and indexes all emails	Email
PlumPlaces	Identifies current location using WiFi (and Intel Placelab)	Location
Outlook/AppleMail	Logs message view, compose actions	EmailAction
MS Office	Identifies which documents are being viewed/edited	DocumentAction
iTunes	Identifies which songs are being played	MusicListeningAction
Firefox & IE	Identifies web pages visited/tabs switched	WebpageViewAction
Adium/iChat	Identifies chats	IMAction
HIDIdle	Identifies periods of inactivity at the keyboard/mouse	UserIdleAction
Finder/Explorer	Identifies use of Finder/Explorer	FIlemanagerAction
fsevent/md	Identifies filesystem modficiation	FilesystemEvent
MarcoPolo (*in progress*)	Identifies various low-level system state changes in OS X	SystemStateChange

Figure 3. AtomsMasher Web and PLUM-based data sources

repeatedly, and is entirely transparent to the user.

4.2.4 Rule inconsistency detection

Inconsistent rules can be problematic in a reactive behavioral system because they can create infinite execution loops. AM does limited testing for such problems by analyzing its firing log for a sequence of re-firings of a rule, where a re-firing is defined as firings immediately occurring after a previous finding without additional information from any external data sources. Note that this only catches loops internal to AM; detecting loops caused by inconsistent rules firing through actions and perceived through external data sources is more difficult. Investigation into techniques for efficiently doing so is currently underway.

4.3 Data Model

AtomsMasher's data model grounds the meaning of the AM's predicates used in rule antecedents, and actions in rule consequents. As mentioned previously, these predicates and actions operate over representations called entities that are surrogates for real-world tangible things such as people, places, and resources, as well as abstract quantities such as events and observations of actions or changes. These representations are created and maintained by data sources, which bring information from external sources into the system. These representations are expressed as an RDF graph, and stored in a persistent triplestore kept on the user's machine. This section describes each of the aforementioned components.

4.3.1 Predicates and Actions

As described earlier, AtomsMasher's predicates are boolean functions that make up rule antecedents and are used to express constraints for forming the conditions under which the rule should fire. While most predicates consist of graph queries over the underlying AM KB, AM does not limit predicates to KB queries. Specifically, predicates may also be functions that compute some derived value of the graph (for example, "number of friends"), or rely on external sources of information. An important example of such a predicate is the "is within distance of" predicate, which requires multiple calls to the Google Maps API. To prevent such external operators from having to be called incessantly, predicate applications are by default cached for a given set of arguments.

Predicates within AtomsMasher are declared to take parameters of specific types, which are either entity or primitive (XSD) types. When multiple applicable predicates with the same name but different parameter types are declared, AtomsMasher chooses the most specific predicate by determining at runtime the predicate with types that (cumulatively) have the smallest graph distance to the types of the arguments.

```
set <entity> <property> <entity or value>
enable/disable rule <rule>
remind/notify me (via growl, SMS, IM, speech) with <entity> <text>
send email/text/IM to <person> <text>
post tweet <text>
set IM/Facebook status <text>
post to feed <feed> <text>
show <document / web page>
set note / todo / RSS reader filter
say <entity/text>
play (song/media)
set system power state/volume
search flickr/google images/wikipedia
run function.... < code >
```

Figure 4. Sample list of AtomsMasher actions

Action, used in rule consequents, are implemented similarly to predicates in that they are modular (Javascript) functions with strictly typed parameters. However, action functions are not assumed functional or idempotent, and therefore no execution caching is performed. Figure 4 lists a few of AM's actions. The simplest action, "set", assigns a property for a particular entity to a particular value, specified in its operands. Set is used for rules that we previously referred to as *updatelets* responsible for updating AM's state given new incoming items (see Figure 6). Most of the other actions involve manipulating something external to the system; this is usually performed through a web API call (when available). However, several actions have non-web destinations. For example, actions affecting the user's local machine, such as the "play media" "show document" action, are made through PLUM. Still other actions, such as "filter notes/todos/RSS" pertain to actions affecting Firefox extensions (an RSS reader and an in-browser note taking client) and thus are dispatched to components directly within Firefox.

4.3.2 Data Sources

Data sources in AtomsMasher play the role of keeping AM's internal knowledgebase up-to-date based on information from the outside world. Each data source is typically responsible for creating or updating a single type of entity: people, places, resources (documents, web sites, media files), messages, events, or activity observations.

AM's data sources come in two flavors. Web-based data sources consist of Javascript code modules that fetch items from web services (via REST APIs) and feeds (such as RSS or ATOM). PLUM data sources, on the other hand, are designed to capture in real time activities that the user performs through his or her computer. These activities include viewing an editing documents or web pages, reading and sending e-mails, watching movies, listening to music, and changing physical locations (by carrying one's laptop). Through integration with various PIM and desktop applications, PLUM also harvests resources such as files, e-mails, and personal contact information, making these entities available to AM through the KB. PLUM is described in greater detail in [21].

The role of data sources in AM distinguishes it from other web based mash-ups. While data mash-ups focus on alignment and syntax reconciliation from multiple structured data feeds, AM's data sources extract information from feeds for updating its internal model, the AM KB. Unlike schema alignment, this extraction process often involves full parsing of particular fields of feeds (such as dates and times), and resolution of references to named entities to corresponding entities in AM's model.

4.3.3 Semantic reconciliation

An important constraint in AM is for entities in the real world to have at most a single corresponding entity in AM's model. Without such an assumption, multiple representations for a single logical thing may appear in the user interface, which could confuse users. Even worse, this could result in model graph fragmentation, resulting in inconsistent behavior. This suggests the need to address the entity resolution (or "semantic reconciliation") problem [2], which can be described as follows: given two extensionally different descriptions for an entity, how can a system determine whether these descriptions intentionally refer to the same entity (e.g., person place or thing), or to two different entities? In the context of AM, this question is faced by data sources, which are beset with the responsibility of updating AM's internal KB: given some external description of an entity, (arriving from some web feed or PLUM observation, for

example,) should it update an existing entity in the KB or create a new one?

Due to the potentially thorny nature of this problem, AM manages semantic reconciliation in three ways: through support for entity-resolution strategies for data sources, reasoning-supported entity equivalence heuristics in the KB, and as a last resort, an interface by which users can manually correct the system. We briefly describe each of these facilities below.

4.3.3.1 Data source strategies

AM prescribes two basic strategies to data sources for mapping updates onto entities in the KB. The first is "update only what I've written", in which data sources are responsible only for the data they have contributed to the pool. To ensure that entities can later be unambiguously identified as derived from a particular source record, data sources rely on URIs (when available) or hints to function as URIs generated by source-specific identifiers. The benefit of this strategy is that it is simple to understand and implement (for new data sources); the disadvantage is that it relies on the other mechanisms (equivalence reasoning or manual reconciliation below) to merge representations from disparate sources.

The second, more aggressive strategy is "update the best one". Using this strategy, data sources use an a entity resolution function to identify the closest entity in the KB matching a particular new piece of datum, and have to make a decision whether this is in fact the intended entity To make this process easier, AM provides several convenient entity resolution methods in its core API for data source writers. These methods perform either name or structure comparison, additionally several methods encode custom name-mangling logic for several common types, such as persons, places and events.

These same entity resolution methods are also used to link records to entity mentions (in both strategies). For example, if a data source encoding information about an event has a list of attendees, and a named location, it must attempt to resolve these references so that entities may be linked properly in the underlying model.

4.3.3.2 Equivalence reasoning in the KB

The idea with the second approach is to take advantage of ontology-based reasoning to make entities that are actually disparate in the KB appear to be the same to all consumers of that KB. One particular instantiation of that idea is the use of inverse functional properties. This strategy suggested by others [9] relies on the insight that entities often have properties that uniquely identify a particular for a particular value of that property, i.e., that no two entities share the same value for a property. AM comes with a pre-set list of functional properties for its built-in types, and lets advanced users configure which properties are inverse functional (which we call "uniquely identifying properties"); such declarations are then asserted into the underlying KB. The OWL reasoning engine takes care of merging the view of inferred intentionally equal entries in its view of the graph.

4.3.3.3 Manual entity reconciliation

AM provides a final resort for semantic reconciliation: manual override. To let users specify that two entities are really the same, the AM UI includes a "glossary" view which let users construct "simple rules" that states that two entities are the same (using an interaction method similar to but simpler than, rule antecedent specification, i.e., entity1 is same as entity2). While these are meant to look like rules to preserve a uniform UI across AM's interfaces, internally these become simple OWL sameAs [16] assertions which get added to the AM KB. AM's internal OWL reasoner takes care of the rest.

4.4 Controlled NLI for rule creation

Section 3.1 walked through the AM rule creation process for a user. This UI design was chosen because it supported the natural expression of rules in an English-like syntax, but mapped unambiguously to logical expressions that could be used by the system. This design was inspired by previous work in "controlled english" natural language interfaces to semantic KBs, in particular GINO [1] which were used to allow the encoding of formal knowledge in a natural-seeming way, and GINSENG [13], which allowed for the controlled expression of queries.

We extend their approach in two ways to make their approach suitable and convenient for rule expression. First, we introduce wildcard types "new <type>" and "some <type>" to match new entities of a particular (RDF) class, or any entities of a particular class. This greatly expanded the expressiveness of rules beyond named entities as arguments to predicates. The challenge, however, with the use of these wildcard specifiers is the following: if such a specifier is used several times in an antecedent, should it always refer to the same entity or different entities?

We have opted to side with English and to support both, through the simple use of demonstrative pronouns. To refer back to a particular entity previously identified using a wildcard specifier, one can use the simple demonstrative phrases "this <type>" and "that <type>". Between these two, the distal ("that") refers to the further of the two closest previous bindings for a particular type, whereas the proximal ("this") refers to the closer. For convenience, the AM UI also adds the pronoun aliases "him" and "her" for "this person" and "it" for all other types.

For example, in the rule:

> whenever some person's status contains "sick",
> send an email to him with text "Get well soon!"

the wildcard entity specifier "some person" gets bound to a particular person when their status contains the string "sick", and co-refers to the same person as the pronoun "him", an alias for the phrase "this person" in the action.

Besides these extensions, the applications of ideas from the aforementioned previous systems

```
whenever a new plazes story arrives and
  that story's location is some location and
  that story's author is some person then
  set that person's location to that location

whenever some event's start time is now and
    that event's source is my personal calendar then
    set my activity to that event

whenever some person's location is my location then
    set my with people to that person
```

Figure 6: *Updatelets* specify rules that connect new incoming events to property updates of existing entities in AM's KB

```
atomsmasher.ds.register( {
   type : "http://plum.csail.mit.edu/08/10/plum.n3#AtomsMasherDataSource",
   uri  : "http://geo.bar.org#MyFirstGeoLocator",
   settings: [ { username: "", service_url: "http://geo.bar.org/api/locations", spotted_entity: "me" } ],
   preferred_update_freq: atomsmasher.constants.HOURLY,
   async: true,
   implementation: function(continuation) {
       var this_ = this;
       jQ.ajax({ type: "GET", url:this.settings.service_url,
          arguments: { username: this.settings.username },
          dataType: "xml",
          success: function(xml){
              //This function be called with a list of stories representing
              //sightings in atom0.95+geo, some of which we have seen before
              var results = [];
              jQ("feed > entry", xml).each(function(){
                   var timestamp_ = atomsmasher.util.parseDateISO8601(jQ("timestamp",this).text());
                   var id_ = jQ("id").text().trim();
                   var lat_ = parseFloat(jQ("geo\\:lat",this).text());
                   var long_ = parseFloat(jQ("geo\\:long",this).text());
                   // the first thing we do is find out whether we've already posted
                   var match = atomsmasher.entities.matcher({
                       type: atomsmasher.ontology.Sighting,
                       ds: this_.uri,
                       siteid: id_,
                   });
                   if (match.length == 0) {
                       // we dont have this entry yet, so create a new Sighting entity to be posted
                       results.push(new atomsmasher.ontology.Sighting({ds: this_, siteid:id_,
                           observation_time : timestamp_, lat : lat_, long : long,
                           spotted_entity: atomsmasher.settings.User )
                       }));
                   } else { /* no need to do anything, since this service never modifies old entities */ }
              });
              continuation.success(results);
          },
          error: continuation.fail
       });
   }
} );
```

Figure 5: A self-contained example of an AM data source for a geolocation service. Such data sources construct AM entities out of information contained in various feeds from web sources.

4.5 Simulating rules using history

Since specifying correct (accurate and complete) antecedents for rules is often tricky for end-users but necessary for such rules to behave as desired, AM attempts to help the user immediately verify the behavior of their rules by simulating the rule using events from the user's recent past. AM searches backwards in its history from the moment the command was invoked, to find the (N=5) most recent situations in which the particular rule would have triggered, and what the resulting action would have been. The output of this simulation is displayed in a simple textual summary beneath the rule creation interface. The effects of actions are described by combining the descriptions of the action operators invoked and the bindings that would have been in effect for the operands of these operators in each situation.

4.6 Sharing slices of your life: your peRSSona

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

PeRSSona feeds can be served directly from AM, or published to any WebDAV server, set via AM's preferences. Since most users are not likely to be running AM on a machine that has a static, world-visible IP address, we provide a publicly usable WebDAV server at our laboratory that is selected by default.

5. Extensibility and robustness to change

AtomsMasher was designed to accommodate changing data sources and web resources in two ways. The first, which is the primary topic of this section, is to design the system to be extensible to new data sources, predicates, and actions (services). We describe how this is done next.

The second pertains to robustness against failure of data sources and services. AM supports graceful degradation from failure of web data sources in part directly through its world model. By acting as a strict abstraction barrier between data sources and use, the model prevents rules from relying on a specific source. Thus if a user merely adds redundant sources of data for particular types, behavioral rules will continue to operate unaided.

5.1.1 Connecting AM to new data sources

One of the primary advantages of using an RDF data model should be in sharing data with external data sources, such as web sites. When such RDF data sources become available, it should be possible to make adding a new data source to AM as easy as pointing AM to the URL of a service description document or feed. In order for this to work, a generic AM data source would have to read the ontology of the service, and automatically determine an effective mapping between structures in the services ontology and those of AM's. However, due to the lack of availability of services offering such RDF feeds and the complexity of this process, we have not been able to demonstrate the general feasibility of such an approach.

In the meantime, our strategy has been to make it easy for advanced users to build wrapper code for transforming arbitrary source formats into AM-RDF. These source-specific data source modules can be implemented in Javascript, to be able to take advantage of the various client-side web service libraries becoming available. Figure 5 lists the complete, self-contained code for a simple data source wrapper of a hypothetical geo-location service that produces an ATOM feed of the user's movements. This specification declares a simple JS object which declares its URI, type, preferences for scheduling, user-settings and implementation. The implementation function performs actual data retrieval; it first retrieves new information from the feed and walks over the results using jQuery. Then, for each such entry, it creates a new AM Sighting entity for each new record it creates. Since this (hypothetical) service (like many others) never modifies old entries after they are published, entities corresponding to previously created entities are skipped.

5.1.2 Adding new predicates and actions

Adding new predicates or actions is a very similar process. Two additionally required fields for action and predicate specifiers are the list of arguments (and their types), as well as a readable English phrase to use in the NLI.

5.1.3 Extending the vocabulary

If a user wishes to create new data types or extend any of AM's existing types with new properties, AM makes it possible to extends its ontology using two methods: by directly modifying the definition (declared in N3); and programmatically through AM's JS entities API. This API lets users effectively add arbitrary definitions to the ontology directly from code such as a data source, predicate or action.

6. Future work

The current status of AM demonstrates an approach that can support user-defined reactive behaviors based on extant Web 2.0 data feeds. There are several challenges related to user interaction, however, that we are addressing to enable us to optimize the usability of this currently rather novel way of engaging with web based information sources. We touch on these briefly here as we hypothesize they will be significant for greater tool uptake by everyday web users.

6.1 Social sharing: Atom Stasher

In section 5 we state that a motivation for decoupling rules that updated AM's KB given external events ("updatelets") from those that performed useful external action was that this decoupling allowed updatelets (which were fairly generic across users) to be made shareable.

To make such sharing easy, we are undertaking the construction of the AtomStasher, a site for supporting a community of AM users by letting users post behaviors and extensions (data sources, predicates and actions) they create, show off their creations, and exchange ideas and behaviors with others. The needs for AtomStasher are several. First, not all users of AM will be comfortable writing their own behaviors.. The AtomStasher provides a mechanism let anyone put behaviors to use without having to role their own. Second, sharing behaviors may both reduce duplication effort. Third, opportunities to share and extend existing code will accelerate the development of code that will integrate AM with other services.

6.2 Antecedents by example: Situation Picker

We plan to make it easier for everyday users to determine rule conditions and gain a better visualization of rule effects. For the former, a Situation Picker will present users' recent activity and context history in a timeline, and let users simply select an example that best resembles a situation where they wished for AM to act. AM will then automatically create an antecedent based on that example a starting point for the user's rule's antecedent. For the latter, to simulate rule actions, AM's UI will output simulated actions from the user's past, using the same visualization, to make it easier to see if the rule is executing at the desired moments.

6.3 Planned study and deployment

While we have some insights already from research in end user programming on how to facilitate user-based programming, we need a significantly refined understanding of how users will engage with such a global system as AM. We are currently working on a study to improve aspects of AM's user interface in preparation for a full-scale deployment of AM. Our approach will be two fold: a longitudinal field study with a dozen participants to be followed by a general web-based beta release. With these releases our key questions will be to investigate the kinds of behaviors users sought to have AM support and the degree to which AM satisfied those goals. We anticipate this study will help us understand how better to tune attributes such as interface, language expressivity and system responsiveness.

7. Summary

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

AtomsMasher benefited from several key architectural decisions. The first was the use of a single persistent internal representation containing simple key representations of people, places, events and resources. This intermediate representation ultimately served three important roles in the system. The first was in simplifying representation reconciliation; having a single representation as a basis of aligning external sources of information avoids the pairwise-alignment problem that serves as a scalability limitation to many mash-ups today. The second surrounded its role as a single, unambiguous world model for the AM rule chainer. Third, this representation serves as an important abstraction barrier that decouples behavior rules from information sources; allowing information sources to be exchanged freely (or added for redundancy) without having to modify users behaviors.

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

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

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

8. Acknowledgements

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

9. REFERENCES

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

[16] OWL. Web Ontology Language. http://www.w3.org/TR/owl-features/

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

[18] RDFS. Resource Description Framework Schema. http://www.w3.org/TR/rdf-schema/

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

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

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

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

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

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

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

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

[27] YUI. Yahoo! User Interface Library. http://developer.yahoo.com/yui/