AtomsMasher: Personalised Context-Sensitive Automation for the Web

Max Van Kleek[1], Paul André[2], Mikko Perttunen[3], Michael Bernstein[1], David Karger[1], Rob Miller[1] and m.c. schraefel[2]

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

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

[3]Dept. of Electrical and Information Engineering
University of Oulu, Finland
mikko.perttunen @ee.oulu.fi

ABSTRACT
This paper introduces AtomsMasher, an environment for creating reactive scripts that can draw upon widely heterogeneous information to automate common information-intensive tasks. AtomsMasher is enabled by the wealth of user-contributed personal, social and contextual information that has arisen from Web2.0 social networking content sharing and micro-blogging sites. Starting with existing web mashup tools and end-user automation, we describe new challenges in achieving reactive behaviours: deriving a consistent representation that can be used to predictably drive discrete action from a multitude of noisy, incomplete and inconsistent data sources. Our solution employs a mix of automatic and user-assisted approaches to build a common internal representation in RDF, which is used to provide a simplified programming model that lets Web2.0 programmers succinctly specify behaviours in terms of high level relationships between entities and their current contextual state. We highlight the advantages and limitations of this architecture, and conclude with ongoing work towards making the system more predictable and understandable, and accessible to non-programmers.

ACM Classification: D2.6 Programming Environments, D3.3 Language Constructs and Features, H5.2 Information interfaces and presentation: User Interfaces.

General terms: Design, Experimentation, Human Factors, Languages, Standardization

Keywords: toolkit, programming language, end user automation, rdf, context aware, mashup, reactive behaviours

1. INTRODUCTION

We find ourselves in many scenarios that potentially require retrieving, consulting and consolidating multiple sources of information. Automating these actions could save us time and effort in scenarios such as:

- context-based reminders -- remind me to call my mother when I get home;
- status update multicast -- forward my Twitter updates (that I send from my mobile phone) to Facebook and Jabber too;
- activity-based information filtering -- during meeting-packed days, make my RSS reader meeting-sensitive, to serve as a way to get e-mails and messages pertaining to my current meeting;
- event consolidation -- I subscribe to many different cinema feeds; consolidate these sources, and remove redundant entries so I can view them in my calendar;
- evening planning -- Find out if there are any bands I like playing tonight, and which of my friends that like similar music are free to come.

While programmers could write custom applications to realise each of these desired behaviours, doing so would require repeatedly solving the same problems a number of times from scratch. The existence of such common problems, from parsing and aligning data schemas, entity resolution between items from different sources, continually monitoring for creating reactive behaviours, as well as sources for context, indicates the need for common support to solve these problems. We have been developing an approach to make it easier for Web2.0 programmers to write behaviours (mini-applications) that incorporate these attributes of mixing public/social/personal data, context-awareness, and reactivity.

What makes these kinds of context-rich applications possible now (as opposed to with a previous lack of sources) is the rise of Web2.0 services promoting users to publish information like activity, location and schedule, giving unprecedented access to a rich sea of public, social and personal information, much of it available in semi-structured or structured form. Our approach has been to investigate *passively* filtering and blending this information, as in web/data mashups, while *actively* scripting automation based on this information, as in web end-user automation, to realise broader and richer functionality in *reactive* behaviours (doing something active in response to blended data).

To this end, we present AtomsMasher (AM), a context-aware reactive-behaviour authoring environment that allows an author to write simple rules to realise such scenarios. This personal automation tool is aimed at a similar audience to that of most mashups and EUA, the "*growing* *group of web designers and developers familiar with scripting languages*"[4].

AtomsMasher provides a common representation and consistent data model that unifies heterogeneous sources; the use of JavaScript to express rule conditions and consequents for querying, filtering and specifying behaviours; a rule engine that determines when scripts should run based on incoming information; and finally, a script authoring environment which makes it possible to understand, predict and debug script behaviour.

The rest of the paper is organised as follows. We first examine related work before elaborating on these challenges and describing the architecture of AM. We show how addressing these challenges enables our original scenarios, before discussing future work in extensibility and sharing, and engaging with end-users.

2. RELATED WORK

End-user automation (EUA) tools are used today to perform tasks more quickly and efficiently with reduced effort. For example, UNIX power users routinely optimize their workflows by writing shell scripts that often process data from multiple sources and applications, using files, process pipes, and character and byte stream transformation operators. AM similarly aims to facilitate cross-application data sharing and manipulation by facilitating the transformation of data into a flexible common representation consisting of structured entities representing the kinds of objects we commonly use to describe common personal information tasks – people, places, documents, and events. These abstractions are intended to facilitate the application of AM to personal-information related tasks.

On the Web, Chickenfoot [2] has sought to do for web programmers what shell scripting languages did for UNIX programmers - permit automation and customisation of the environment to accelerate common tasks and better fit users' needs. Chickenfoot's programming model avoids use of invisible complex selectors such as XPaths to describe items on pages, instead supporting relational descriptions of visible items, to make it easier for programmers to script complex actions involving complex pages. This design inspired our goals for AM 's query language, in which we hide the complexities of query and data heterogeneity by using a familiar javascript object model. Furthermore, we are working to fully support the use of Chickenfoot actions in AM 's action vocabulary. In turn, our system extends Chickenfoot by providing a rule engine (to support automatically executing scripts), a repository of external information which Chickenfoot scripts can use in their actions to be more adaptive, and actions that support scripting "off the page" -- e.g. web services.

2.1 Web Mashups

AM's approach towards retrieving and aligning information from multiple, heterogeneous sources on the web differs from typical mashup systems [3,17] in several ways. First, it uses obtained information to construct a relational representation in RDF, unlike most data mashups which align two or more structured data streams at the syntactic or structural level. The RDF model, which supports rich linking of related data items, is what is seen by the rule engine and users' scripts. As described in section 3.4 the effects of having such model is that it greatly simplifies integration, in particular towards scaling to new data sources and types, and encourages script portability by reducing dependence on the source representation. In addition to this model, AM's action language, consisting of Javascript with extra classes and operators is more general than what is generally provided by the visual dataflow interfaces of data mashups such as Pipes. Finally, unlike most data mashups, AM supports the integration of private data sources such as e-mail, and sources on the user's desktop, such as the user's local filesystem.

2.2 Rule-based reactive systems

AM can be considered a type of rule-based reactive behavioural system for end user information management. A previous system which employed reactive production rules towards a similar goal was the Information Lens [9], an end-user rule-based system designed to help members of an organization cope with the large number and variety of electronic messages they received via their new enterprise messaging system each day. Today, the Web and e-mail have extended the reach of information far beyond the walls of corporate enterprises, this problem has become much greater and more general. Another important similarity surrounded the fact that this paper also concluded that rich "semi-structured metadata" could reduce the need for natural language processing techniques to enable this automation. Sadly as described later, many sources of information on the web are designed for human consumption and not richly structured; AM takes advantage of what structure is available.

2.3 Context awareness

As stated in the introduction, AM enables applications to be "context-aware" by letting users leverage context information about a user's activity available in the various data sources on the web. Much work has surrounded making computers more context aware, in particular for handling input from sensors connected to the environment. Out of the proposed architectures for facilitating the creation of such applications, AM most closely resembles blackboard architectures, proposed by Winograd for use in context-aware applications, due to its pattern-based nature and centralized common representation [16].

3. ATOMSMASHER ARCHITECTURE

In this section we briefly give an overview of AM's design goals, and describe each of the architectural components of AM in detail.

3.1 Objectives

As with many EUA systems, our primary goal was to build a framework that grants users enough flexibility in script creation to create scripts that achieve a wide variety of tasks. The overarching design criteria that we therefore sought were versatility, scalability, and openness that would ensure that users could extend, appropriate or modify the system to do things other than what we as system designers could have conceived of. To this end, AM uses open data formats and public APIs to encourage code re-use and integration with 3rd party applications. To encourage appropriation and co-modification we added two additional design goals: system transparency and simplicity achieved through uniformity, which we believed to be important for encouraging end-user system extension and appropriation.

Since we wished to target an audience that would be enthusiastic and creative in identifying opportunities for automation in their lives, we targeted "life hackers", a term that connotes a person who takes pride in techniques for optimising aspects of their lives. In this paper we focus on the language and toolkit to provide this value, aiming at a class of users who are comfortable with computers and have basic programming experience, particularly with scripting for the web in Javascript, a similar audience to that of most web mashup tools and Chickenfoot. In future work we discuss how we plan to extend this first prototype with considerations of a UI accessible to non-programmers.

With these design goals in mind we built AM, a reactive behavioural end-user scripting environment driven by web and personal data sources. Figure 1 details an architecture diagram, the highlighted numbers represent:

1) AM periodically retrieves external information via web feeds (RSS/ATOM), web service api calls (e.g., weather), e-mail and IM.

2) Feed Prisms - process each source item decoding source encodings, extracting information from source schemata, and constructing a generic instance in RDF aligned to the AM ontology (Section 3.6.1)

3) Feed rules - reconcile new items produced by prisms with entities already in the KB, resolving references to entities mentioned in the new entity's properties

4) State rules - drive the state model, by analyze incoming entries and setting state variables based on patterns in these items

5) Behaviour rules - execute reactive behaviours based on incoming items and state variable values, and causes actions to occur.

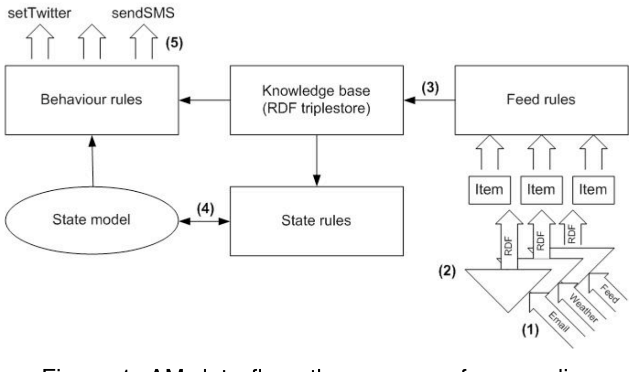


Figure 1. AM data flow: the process of responding to new incoming information

In the following section, we describe the core *external* data model of the system in terms of how users express and represent automation. Our primary design goal throughout in designing this external data model is to provide the simplest possible programming model that would be sufficiently expressive to encompass desired use cases while retaining familiarity to web (Javascript) developers.

3.2 Representations in AtomsMasher

The basic unit of AtomsMasher (AM) is the user-contributed script, consisting of instructions on what to do and when to do it. In the language of rule-based systems, each of these scripts can be considered a rule, with the conditions for execution forming the *antecedent* (which may be empty for scripts meant to be manually triggered), and the actions to take the rule *consequent*. In AM, writing a rule is as easy as writing a simple Javascript if-statement. The next section describes how AM allows succinctly expressing antecedents in a syntax familiar to Javascript programmers.

3.2.1 Rules

Rules are used in AM to represent end-user reactive behaviours, update the user's state model described in section 3.3, and process new, incoming data items as described in section 3.6. A rule is triggered when its antecedent can be *satisfied*. AM can satisfy the antecedent for a rule if and when it can find a means to make its antecedent true. If a rule's antecedent consists of query variables which represent wildcards standing for entities such as people, places and things in AM's KB, this problem translates to finding suitable entries in the KB such that the substitution of these entries into the expression yields true. AM executes the consequent for each such set of satisfying values. We describe how AM computes the satisfaction of query variables next.

3.2.2 Query Variables

Rule antecedents in AM can consist of query variables, which are wildcards that represent some set of entities in AM's knowledgebase. To resolve query variables to satisfying entities, AM's query variables start by representing all (or a set of) entities in the underlying KB simultaneously. For example, suppose person is declared to be a query variable over all entities representing people in the KB. The expression person.name would represent all names of people in the KB. The expression person.name.equals("John Smith") would then correspond to the value true for all person entities whose names matched "John Smith" in the KB.

When an operator is applied to a query variable, it yields a new derived query variable that represents the values resulting from mapping the operator over each of the items individually. Operators that represent tests (e.g., ==, <, >) correspond to a filter operation; they result in a new variable containing only values corresponding to the objects of the values that satisfy the source variable's values. Whenever an operator is applied, the resulting query variable maintains pointers back to the original database entry or entries that were the source(s) of each value. This makes it possible to re-identify at the end of a series of operator applications the set of entities in the KB that met the criterion.

Using this model, binary operators involving two query variables become slightly more complex. They are handled by considering all consistent combinations of variable values, and the results of applying the desired operator to each such combination. Therefore, the result of a binary operator is a new query variable whose values have multiple source objects apiece. To keep these bindings straight, AM maintains in the aforementioned source object field of a query variable the list of *variable id-to-value* bindings corresponding to the interpretation that was in effect when the value was computed. These variable ids are created when a top level query variable is created by user code and are carried over to new derived query variables, such as when an operator is applied. Variables with different ids are considered "free" to bind to values independently of one another. A consistent set of bindings then comprises the set of values for which source bindings agree, that is, the values for which the source values represent the same value for the same variable, or any values for different variable.

The successive application of binary operators to unique query variables causes this number of combinations to grow exponentially in the number of such unique variables. However, based upon our scenarios, we believe that in general, the number of unique variables involved in successive binary operator applications in typical use should be very small (2-3) in practice. Such exponential explosions can be made more controlled by using an *and* operator (described next) over query variables with domains that have already been narrowed, such as with one of the aforementioned filter operators. Narrowing the set of values for such variables amounts to reducing the base of the exponent.

Under the semantics of consistency and binary operators just discussed, the *and* operator acts as a "gateway" that returns a derived query variable representing all of the query variables across all its clauses, admitting only non-false values whose source bindings were consistent across clauses. The *or* operator, in contrast, consolidates all values across all its clauses, admitting all (and removing dominated) bindings. These semantics yield "sensible" results visible below:

```
   // creates 2 unique query variables

   var a = person(); var b = person();

   // returns all bindings of a with a bound to
all people whose name starts with Max AND are
over age 25

   and(a.name.startsWith('Max'),
a.age.greaterThan(years(25)));

   // returns all |a|x|b| combinations of bind-
ings (a,b) for a is the subset of all people
whose name starts with 'Max',

   //  b is the subset of all people over 25

   and(a.name.startsWith('Max'),
b.age.greaterThan(years(25)));

   // returns a new query variable with a bound
to UNION of

   //   the set of all people whose name starts
with 'Max', and the

   // and the set of all people older than 25
(with duplicates removed)

   or(a.name.startsWith('Max'),
a.age.greaterThan(years(25)));
```

One might notice that the syntax in our above examples to be slightly peculiar due to the use of functions instead of Javascript's built-in operators. Although we wished to overload the default implementation of Javascript's operators with our query variable and RDF-type aware (see section 3.5) implementation, this goal was thwarted by the lack of support for operator overloading in the current (1.7) version of the language specification. Javascript 2.0 is currently slated to support operator overloading, at which point we will leverage that to make the syntax more natural; for example `and(a.name.startsWith('Max'), b.age.greaterThan(years(25)))` will appear as `a.name.startsWith('Max') && b.age > years(25)`

The above-described design of AM query variables was inspired by object relation mappers (ORMs) such as SQLObject [12] and Hibernate [5], which make it easier for program code to create and manipulate data stored in databases by creating proxy objects in the language which represent the items in the databases. Using an ORM, a complex JOIN of tables in an underlying database could appear as a simple field access on an object instance. However, because ORMs typically establish a one-to-one mapping between proxy objects and items in the underlying database, programs still use special query constructs to find and select among elements in the database. Our goal was to see if we could let the user express queries over sets of items without having to use query terms or higher order predicates (map/filter/reduce).

3.2.3 Special query variable: 'New'

As illustrated throughout section 4 a special reserved query variable called New can be used in rules to represent an item that has just been imported into the knowledgebase. New items added to AM's knowledgebase get bound to New exactly once in their lives. Although it is intended for use in input processing and state model rules (described in section 3.6) New is occasionally useful for non-idempotent behaviours that need to be executed only once. For example, if Bob only wants his Facebook/Jabber status once per new incoming Twitter message; for this he should use the New query variable to check for incoming twitters.

3.3. State variables

Certain sources of information publish updated observations of some dynamically changing state of the world. Examples of such data sources include Plazes [11] which reports a user's most recently identified location, a user's Twitter state, or current weather. Unlike regular entities in the KB, for these types of information, only the latest (e.g., most recent) entry is ultimately important. To make it convenient for users to employ such data in rule antecedents, AtomsMasher supports a second type of variable known as a state variable which work simply by being assigned to by a rule (which we then call a state rule). AtomsMasher then holds state variables' values until they are explicitly reset by another rule triggering or expire. An example of a rule setting such a state variable is as follows:

Antecedent:

```
  and(New.type.equals('plaze'),
New.location.nearAddress("Central Square Apart-
ments", miles(0.01)))
```

Consequent:

```
 my.location = Location("Home", {geo:New.geo});
```

This simple state rule is responsible for setting the user's location state to an RDF Location entity called "Home" when he or she is reported (via a Plazes entry) to be very close to his or her apartment. As can be seen in this example, the state model is identified in a global Javascript object called my. Rules can create new state variables simply by assigning to my; values assigned to the state model can be of any arbitrary type (typically RDF objects or strings). To set an expiry time, users can use the special wrapper autoExpiring, which takes the new value, a decay time and a post-decay value as follows: `my.location = autoExpiring(Location("Home", {geo:New.geo}), hours(24), Location("Unknown"));` This specifies that my.location should assume the user is home until up to 24 hours have passed since the state variable was last assigned; after which the point the value should be reset to Location("Unknown"). Besides my, a secondary special state variable, now, maintains the current time, to facilitate time-conditional antecedents.

3.4. Simplifying access to RDF Resources

As described in section 3.6, AtomsMasher internally uses an RDF data model as its entity knowledgebase or KB. This KB is kept in a persistent triplestore using a MySQL backed Jena model in Java, which are loaded on demand into Javascript over XML-RPC. To make accessing RDF properties "feel" like accesing regular Javascript object properties, AtomsMasher creates wrapper proxies for every entity it loads from the triplestore with accessor functions for every property on the original resource. AtomsMasher also augments each resource with all operators that are applicable to the resouce, so that they can be directly invoked as if they were a Javascript object method.

One small difficulty with mapping access to RDF properties using Javascript object property names is that RDF properties (like resources) typically identified by a full-length URI (e.g., http://AtomsMasher.csail.mit.edu/2006/01/am#Person). Typing such a full-length URI is, first, too cumbersome, and cannot be used directly using dotted field access notation, since URIs contain characters which are not allowed within Javascript identifiers. To make access convenient given these constraints, AtomsMasher creates additional accessors that use only the local name of a property's URI, which, although not guaranteed to be globally unique, are often unique enough to be useful.

When a set of proxied RDF entities are assigned to a query variable, the query variable wraps all of the operators and methods found on all of the entities represented by the query variable up to the query variable itself. Since items may have different properties and supported operators, AtomsMasher ensures that when one of these accessors are called, it only considers the values for which that operator or property exists. If more than one outgoing edge for an item, the set of all values for the property are collected, and wrapped in a single returned query variable. This way, the same query filter mechanism can be used to fully navigate the RDF graph and select nodes with minimal syntactic overhead.

For example, if the query variable person is initially bound to the set of all entities corresponding to people in AM's knowledgebase, the simple expression `person.email` would correspond to the set of all email addresses for all people. Similarly, finding the person in the KB with a particular e-mail address can be expressed simply by narrowing this set; e.g., `person.email.equals('max@mit.edu')` would query all such entries for that had that email address. Note that this syntax is identical to Java syntax of checking to see if a particular object's email matches a particular string

3.5 Comparison Operators

Comparison operators in AM play a large part in defining the expressiveness of rule antecedents because they determine the ways in which entities can be compared with one another, and values for which a rule will trigger. Three considerations make the design of operators challenging. First, many types of operators need to be specific to the type of the entity; however since RDF does not mandate what properties must exist for a given type, examining an entity's type exclusively is insufficient to tell whether an operator applies. AM handles this by identifying the operators that are compatible with a given resource (corresponding to an entity) in two different ways; by type and by topology. For the former, AM looks up the RDF types of the resource (comprising its declared and entailed types, which are computed by Jena and included in the object proxy), and for each, consults its registry of operators. For the latter (topology), AM similarly consults a separate registry indexed by property name. This latter strategy is employed for operators such as nearTo(), which supports any entity that has either a geo property (which indicates a latitude and longitude), or a streetAddress.

Second, the surface type of an object might not be the actual type; for example, a string could designate a time or a location. For this, AM uses a simple strategy of maintaining a list of string-constructors that parse strings into an RDF type, and attempts to apply these string constructors if an operator lookup on a string argument to an operator fails. AM only currently supports this runtime coercion for operator arguments; therefore, datatype constructors should be called explicitly if beginning an expression that needs to be coerced from string. This strategy resembles "sloppy programming" [8], which searches over the space of function applications (which could be type conversions); adding such functionality would greatly enhance the system's ability to coerce types, but may also increase computational complexity.

The final challenge surrounds the need for operators to contain some robustness to noise -- for example in comparing variations on string renderings of a person's name. AM approaches this problem by adding liberal comparison operators for several low-level types with optional tolerance threshold parameters. For example, the string liberal string equality operator .resembles() first strips arguments of padded whitespace, ignores case, checks for containment, and compares the edit distance with an optional maximum edit distance parameter (expressed as either number of characters or percentage of the original length of the string). We are currently working on making these thresholds more adaptive by using a Bayesian likelihood computation approach that can be trained to be sensitive to different prior probabilities for entities being the same or different.

3.6 Acquiring and Representing external information

Having a rich common RDF representation for entities in AM creates an abstraction barrier between data sources and end-user reactive behaviours, shielding query variables from the source of information. This allows the system to scale to new data sources, and encourages the re-usability and sharability of behaviours by preventing authors from writing their behaviours specific to a particular source. From a reactive behavioural-based systems standpoint, having a common representation makes the system's model of the world concrete. This was important for many aspects of the system, including the lazy rule scheduler described in Section 3.7, which relies on knowing how the system's view of the world has not changed to determine which rules it can ignore.

In this section, we describe how this intermediate representation is built from external data sources.

3.6.1 Data Prisms: Low-level data extraction

We quickly discovered that despite standardization in data schemas for feeds, e.g., RSS 0.95, 1.0, 2.0, ATOM, there was much variation among content providers regarding how and what information was conveyed in feeds. That is, while the base syntax and schema was standardized, different sources on the web used fields in these schemas for different purposes. As a result, it was necessary for some data sources (mostly web-feeds) to create feed-specific import filters, which we call data prisms, to distil information from packed and misappropriated source schema fields into RDF. A yet additional common problem was that feeds included linkback URLs in feed fields instead of the actual data; under such circumstances, several data prisms retrieve the indicated page and grabs the value using Chickenfoot.

Since prisms are rather onerous to create and require substantially more programming experience than writing rules, we wanted to ensure that most people would not have to worry about writing them. Fortunately prisms perform source-specific transformations that are rather user-agnostic, they are ideal types for being redistributed and shared among users. Although we have currently a centralized infrastructure to do that (e.g., a single repository of prisms we have created), we are moving towards a more community-sharing oriented model (see section 6).

3.6.2 Feed rules: Reconciling and personalizing incoming items

As just described, the output of the data prisms in the first phase consists of new RDF descriptions of entities such as news stories, updates from the local weather service, facebook and Twitter; personal e-mails, or upcoming events, as obtained directly from particular information sources such as RSS feeds, mail servers, and web services. There are two problems with putting this new description directly into the entity KB; first, there might already be a description corresponding to the same entity that came in from previously, possibly from another information source. In such a situation, the two descriptions may or may not have exactly the same information; either description may have been incomplete or incorrect. Thus, there is a need to reconcile and merge descriptions of entities to create a coherent view based on incomplete or redundant sources.

The second problem surrounds resolving references to entities within an incoming entity description. For example, an event may list an organiser, a location, and attendees. Each of these entity references needs to be resolved to the appropriate entity description in the knowledgebase in order for AM to be able to service query variable expressions via this new entity. For example, if an event's location is successfully resolved, then all information pertaining to that event becomes available through the query variable expression, e.g., `event.location.streetAddress`; this additional information might be important because it might be required for comparison via operators such as nearTo, described in the next section.

If such an entity to be resolved has a globally agreed-upon unique identifier, (as proposed by proponents of the Semantic Web), then entity resolution corresponds to a database lookup. In general, however, this is virtually never the case with Web 2.0 data sources which tend to be highly heterogeneous. Thus, AM must rely on comparing available information, often consisting of noisy and ambiguous identifiers -- to entities in its KB. Since fully automatic approaches to entity resolution in an open (personal) domain is an open unsolved problem, AM takes a purely pragmatic approach: use a greedy strategy that might work most of the time, and keep this strategy transparent (easily modifiable) by the user.

In order to do this, AM uses a special set of rules called feed rules which operate like other rules in the system but are privileged because they get first access to incoming data items - before these items have been added to the KB. This gives the feed rules an opportunity to modify the incoming item and to declare that it is a duplicate of an existing item. AM allows feed rules an extra operator, sameAs() on New which takes a resource as an argument. This establishes an OWL sameAs relation [15] between the New and specified items, effectively merging these two resources. Feed rules can also freely modify fields on the item, such as for resolving embedded entities. After all triggered feed rules have been applied, the changes to the New item are committed to the KB. Examples of such feed processing rules are given in section 4.

3.7 Scheduling rules

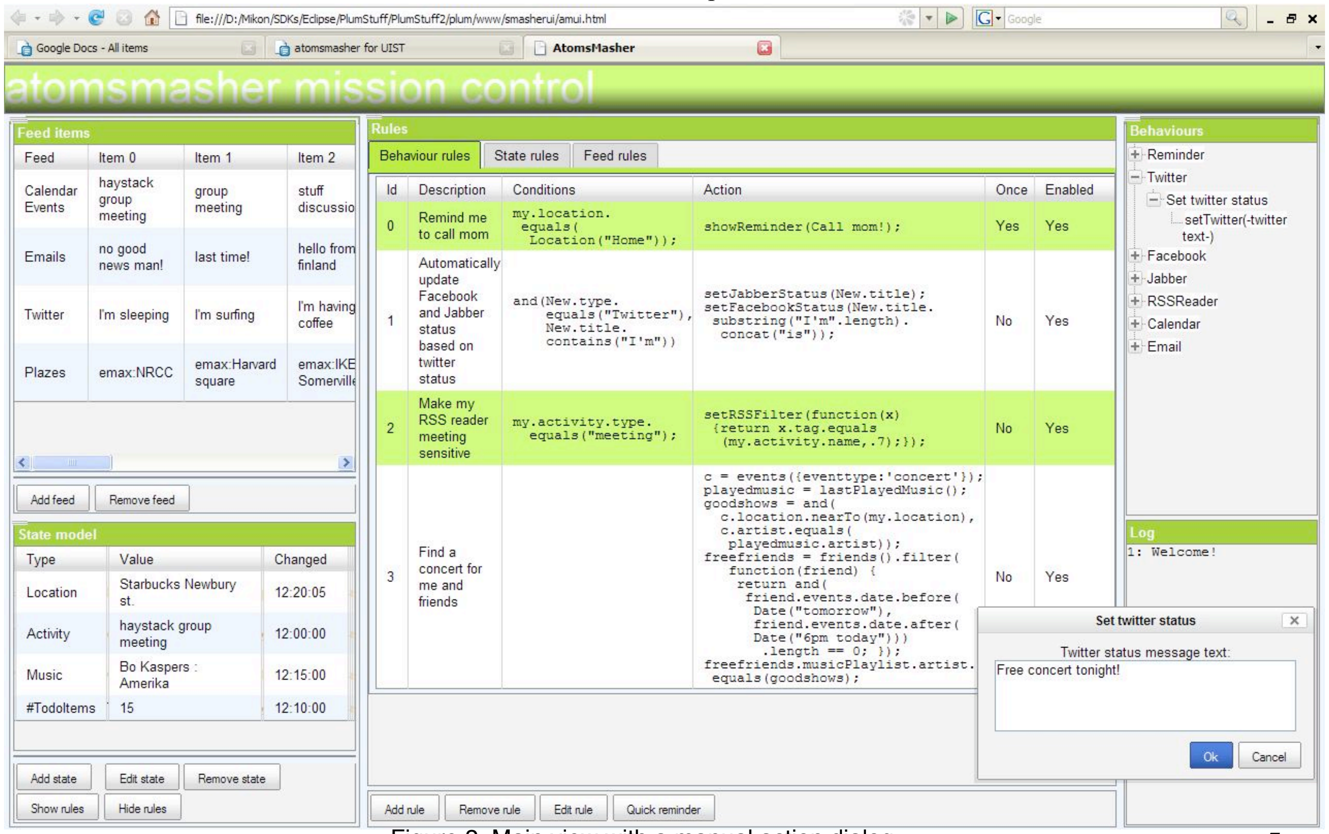
Since evaluating a rule's antecedent can involve a complex set of queries over the KB, AM's rule engine attempts to conserve computational cycles by postponing the consideration of a rule until an event occurs that could cause that rule to trigger. Such an event could include the retrieval/arrival of a new data item, the changing of a state variable, or merely the passing of time. For example, when a particular state variable changes, it considers all the rules whose antecedents depend on it; similarly, when new entities of a particular type are added to the KB, AM considers the rules whose antecedents rely on query variables over entities of that type. In addition, rules that condition on New are considered whenever a new entity is introduced to the system. Antecedents involving Now (the variable representing the current time) can cause significant problems with this approach, because it might suddenly become true when the clock strikes a particular value (without any external change). Thus, AM handles such expressions specially. For time expressions involving comparing Now to an absolute time (e.g. "Wednesdays at 3pm"), AM determines the soonest moment that the expression could become satisfied, and sets a system callback alarm for that moment. Antecedents that somehow relate Now to a state or query variable require more delicate consideration; AM determines the soonest moment the rule could trigger by evaluating the expression involving Now over all the (current) values of that variables in the expression, setting a wakeup alarm for the soonest such time. AM also re-evaluates such rules if the relevant state or query variables experience updates, since this could result in a yet sooner trigger time.

Note that AM does not yet employ logic for detecting conflicts or feedback when considering rules or their actions; rules are simply considered and triggered one at a time. Since conflicts are likely indicators of problems with user rules, we are considering strategies to try to detect and reveal such conflicts.

3.8 User Interface

The user interface of the AM prototype is shown in Figure 2. It consists of five main views: feed items (top left), the state model (bottom left), behaviours (middle), and actions (top right). The log view (bottom right) displays a detailed record of rule triggers and actions taken by the system. This default view was chosen to give users a complete "eagle's-eye" high level overview of the state of the system in one glance, to easily inspect what the system as a whole was doing. From this view, the UI is designed to facilitate drilling down into the details of any particular aspect of the system.

For example, the feed item view by default displays only the titles of items of all types, with the most recently acquired items displayed most prominently. If one wishes to further inspect any item, a complete summary (of all fields) is displayed when the mouse cursor is hovered over it. Clicking on an item displays the item fully, and provides simple editing facilities for the item. If the item view is clicked, it becomes expanded, which reveals keyword-based search facilities across items. Feeds can be added and removed by clicking on the corresponding button, and providing a URL to a web feed. Note, however, that every feed requires a suitable prism to be available to it for AM to be able to extract information out of it. AM has rudimentary facilities for inspecting feeds to determine whether a prism it has already installed may be applicable. This is used to suggest a prism when adding a URL to a feed; users can override their choice by providing a path to an alternative prism.



file:///D:/Mikon/SDKs/Eclipse/PlumStuff/PlumStuff2/plum/www/smasherui/amui.html
Google
Google Docs - All items
atomsmasher for UIST
AtomsMasher
atomsmasher mission control

Feed items

Feed	Item 0	Item 1	Item 2
Calendar Events	haystack group meeting	group meeting	stuff discussio
Emails	no good news man!	last time!	hello from finland
Twitter	I'm sleeping	I'm surfing	I'm having coffee
Plazes	emax:NRCC	emax:Harvard square	emax:IKE Somervill

State model

Type	Value	Changed
Location	Starbucks Newbury st.	12:20:05
Activity	haystack group meeting	12:00:00
Music	Bo Kaspers : Amerika	12:15:00
#TodoItems	15	12:10:00

Rules

Id	Description	Conditions	Action	Once	Enabled
0	Remind me to call mom	`my.location. equals( Location("Home"));`	`showReminder(Call mom!);`	Yes	Yes
1	Automatically update Facebook and Jabber status based on twitter status	`and(New.type. equals("Twitter"), New.title. contains("I'm"))`	`setJabberStatus(New.title); setFacebookStatus(New.title. substring("I'm".length). concat("is"));`	No	Yes
2	Make my RSS reader meeting sensitive	`my.activity.type. equals("meeting");`	`setRSSFilter(function(x) {return x.tag.equals (my.activity.name,.7);});`	No	Yes
3	Find a concert for me and friends		`c = events({eventtype:'concert'}); playedmusic = lastPlayedMusic(); goodshows = and(c.location.nearTo(my.location), c.artist.equals(playedmusic.artist)); freefriends = friends().filter(function(friend) { return and(friend.events.date.before(Date("tomorrow"), friend.events.date.after(Date("6pm today"))) .length == 0; }); freefriends.musicPlaylist.artist. equals(goodshows);`	No	Yes

```
1: Welcome!
```

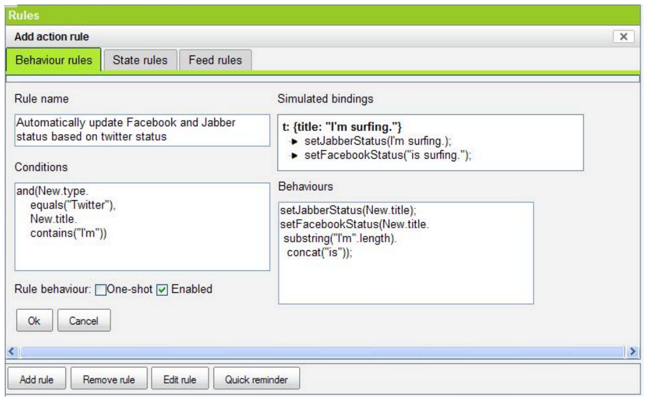
Set twitter status

Twitter status message text:

Free concert tonight!

Figure 2. Main view with a manual action dialog.

The three tabs of the rule view correspond to the three different rule sets: behaviour rules, state rules, and feed rules, and shown in Figure 1. In Figure 2 the behaviour rules tab is visible. For each rule, the view shows a description, conditions, actions, and two properties specifying whether the rule should be fired only once, and whether the rule is currently enabled. All of these can be edited in place by clicking on them, or a rule editing view can be opened by selecting a row and clicking the 'Edit rule' button. The edit view is shown in Figure 3. This view provides the user with direct feedback for the rule authoring by displaying the bindings of the query variables against current knowledgebase contents in the 'Simulated bindings' box.



Rule name: Automatically update Facebook and Jabber status based on twitter status

Conditions:
```
and(New.type.
   equals("Twitter"),
   New.title.
   contains("I'm"))
```

Simulated bindings:
```
t: {title: "I'm surfing."}
  ▸ setJabberStatus(I'm surfing.);
  ▸ setFacebookStatus("is surfing.");
```

Behaviours:
```
setJabberStatus(New.title);
setFacebookStatus(New.title.
 substring("I'm".length).
  concat("is"));
```

Rule behaviour: ☐ One-shot ☑ Enabled

Figure 3. Editing a rule.

As discussed previously, the state model describes time-varying aspects of the user's situation. In Figure 2 the state variable view is condensed to display only the type of the state variable, its current value, and a timestamp. This table can be expanded to also show the associated rule of each state variable by clicking the 'Show rules' button. Clicking the 'Edit state' button replaces the rules view with a state editing view. The state editing view is similar to the behaviour rule editing view, except for the action part which always assigns a value to a state variable.

The actions view allows the user to browse the available actions for rules and to manually fire behaviours. Figure 2 shows the manual action view for setting Twitter status message.

3.9 Implementation

AM is implemented partially in Java as part of PLUM [14], our user modelling framework, and partially in Javascript. The Java components of AM consist of components responsible for retrieving and transforming information into RDF, including code for parsing web feeds (using ROME), interfacing with e-mail (via POP/IMAP using JavaMail), and IM (using Muse). Data prisms which call these APIs are also implemented in Java. We built plug-ins for ROME to handle special RSS schema extensions such as XCAL which were not previously supported. RDF items are persisted in Java by Jena using an OWL-reasoning enabled MySQL-backed model. The Java components of AM start an XML-RPC server which allows AM's Javascript components with retrieve entities and save and load state.

All remaining parts of AM are written in Javascript and are currently designed to run within Firefox. The rule and query variable engine make heavy use of functional programming patterns, which was greatly facilitated by the MochiKit functional programming API [10]. This API let us make the query variable code closely resemble a textbook example of a rule-based systems often presented in introductory AI texts in Scheme, which made it compact and elegant. AM employs jsolait [6] for asynchronous XML-RPC2, to communicate with the Java components, and Yahoo's JSON parser to validate communications. The UI components were developed in parallel with the engine, using HTML, CSS, JavaScript, and the Yahoo! User Interface Library (YUI).

Currently both Java and Javascript components need to be running in order for AM to be reactive. We are currently working to get around this limitation two ways: by porting the Javascript code to run under Rhino [12] for those who want to install and have AM running on their machines in the long term; and second, make AM entirely self-contained within a Firefox extension that launches a Java subprocess for casual users who want to try AM out without having to perform an installation.

4. DISCUSSION

In this section, we revisit the scenarios from the Introduction and illustrate how they are implemented using AM. A description and code example for each shows how suitable feed or state rules, and behaviour or query rules can be written in AM syntax to implement each scenario.

Scenario 1: *remind me to call my mother when I get home*. The state rule instructs AM to look for such incoming items of type "plaze" whose name equals "Central Sq Apts". Having found such an item, AM creates a Location object called "Home", and assigns the geo-spatial coordinates from the incoming item into this object. This object is stored to the state model as the value of location variable. Note that the variable *my* always refers to the state table. The behaviour rule of 1) is satisfied when the state variable location equals the home-object the creation of which we just described.

```
If //state/feed rule
  and(New.type.equals('plaze'),
        New.location.name.equals("Central Sq
Apts"));
then
  my.location = Location("Home", {geo:New.geo});
------
if //query/behaviour rule
     my.location.equals(Location("Home"));
then
     showReminder("Call mom!");
```

Scenario 2: *When I send an update to Twitter, update Facebook and Jabber too*. This requires only a single behaviour rule. The antecedent of the rule is instantiated when the title of a new Twitter feed item contains the string "I'm". Note that this antecedent always has at most one binding, because there can be only one Twitter feed item bound to New at a time. As an example, let the Twitter message be "I'm working". In this case the Facebook status would be set to “is working” (Facebook prepends the message automatically).

```
If //just a behaviour rule
    and(New.type.equals("Twitter"),
New.title.contains("I'm"))
then
     setJabberStatus(New.title);
     setFacebookStatus(
New.title.substring("I'm".length).concat("is"));
```

Scenario 3: *make my RSS reader meeting sensitive, to serve as a easy to get e-mails and messages pertaining to the meeting I am currently in*. This is slightly more complex. The feed processing rule used to implement this scenario looks for new emails whose sender or subject contain “haystack” (project name). When such an item is met, the consequent of the rule assigns a tag “haystack” to it. This allows the tag to be easily used in the associated behaviour ruleThis rule triggers whenever Christine’s current activity, a state variable, constitutes a meeting; the result is that it sets the RSS filter to display all items tagged with a word that appears in the active meeting’s name.

```
Filters that appropriately tag incoming data
items with meeting names. For example:
If //state rule
     and(New.type.equals('email'),
        or(New.recipient.contains("haystack"),
            New.subject.contains("[haystack") );
then
     New.tag = "haystack";
--
if //behaviour rule
    my.activity.type.equals("meeting");
then
    setRSSFilter(function(x) {
      return my.activity.name.contains(x.tag);
    });
```

Scenario 4: *incoming items from multiple sources consolidated and redundant entries eliminated to view in y calendar*. Here only feed processing rules are involved. This rule checks a new item against existing items and asserts them as the same item, if the set of specified fields have identical values. We have borrowed the sameAs-relation from OWL[15] for this purpose. Note that asserting this relation between the two items (RDF resources), means that the fields of them become a union of their fields. This rule also demonstrates how easily one can incorporate the JavaScript else-statement in a rule. In the else-branch, the consequent turns the new item into an event by simply assigning to it a new field “eventtype” and adding it to the person’s events calendar.

```
Incoming item processor:
If //state/feed rule
    m = events({ eventtype: 'film' });
    and(New.type.equals('event'),
          New.location.equals( m.location ),
          New.name.equals( m.name ),
          New.start.equals( m.date.start ))
then // auto-reconcile two entities
    New.sameAs(m);
else  // turn into an event; add to our Events
calendar under "films"
    New = newEvent(New);
    New.eventtype = 'film';
    add(events, New);
```

Scenario 5: *who is playing in my area tonight, and which of my friends that like similar music are free to come?* This illustrates an “extreme” use of AM to query across information obtained from potentially hundreds of data sources – all of her friends’ online social calendars. This rule, which for simplicity we assume is meant to be manually triggered, starts by isolating a set of concerts she might want to attend, by finding the intersection between concerts in her area happening on the particular day in question, and artists on her recently played (last.fm) list. Then, the script selects her friends who have no appointments scheduled that evening, and determines whether each have recently listened to any of the artists featured in the evenings concerts. The list of all such people and the concerts for which this final criterion is satisfied are returned.

```
if (none) // manually triggered query
then
   c = events({eventtype:'concert',
dtstart:Now.day()});
   playedmusic = recentlyPlayedMusic();
   goodshows = and(c.location.nearTo(my.location,
miles(2)),

c.artist.equals(playedmusic.artist));
   freefriends =
friends().filter(function(friend) {
         return
and(friend.events.date.before(Date("tomorrow"),
      friend.events.date.after(Date("6pm to
day"))).length == 0; });
freefriends.musicPlaylist.artist.equals(goodshows
);
```

5. FUTURE WORK

5.1 End-User Interface

It was our aim in this iteration of AM to target a similar audience to mashups, users familiar with scripting, to demonstrate the feasibility of creating reactive behaviours from previously passive sources. We are currently undertaking studies in other types lay-user automation to examine how we could develop a user interface that truly supports all types of end-users. This involves work in simplifying both the specification of rule antecedents and the actions that should be taken. For example, in integrating AM more closely with our user modelling framework PLUM [14], we can use a form of query-by-example to look back in your history and say, in future, 'when something like this happens, I want this to happen'. We are also considering other visual programming metaphors and programming-by-demonstration, to simplify the initiation, understanding and completion of actions, and scrutability of behaviours. In addition, part of this work is designing and evaluating AtomStasher, a new component described next.

5.2 Extensibility and sharing

We encourage the re-usability and sharing of behaviours by shielding query variables from direct access to the information sources, preventing authors from writing their behaviour specific to a particular source, and allowing the system to scale to new data sources. As we have elaborated elsewhere [1], the social community data that inspired AM is part of a wider social evolution on the Web. By establishing the "AtomStasher" (similar to the Co-Scripter wiki [7]), we aim to make prisms and actions shareable, encouraging an active community, and allowing less experienced users to download more complex rules that others have written. As a further social aspect of the system, we aim to allow publishing state variables as feeds, to provide the user with a way of exposing some of their state to their friends, and their applications.

5.3 Privacy and Security

There are obvious concerns in blending personal and Web data, though by running AM client-side we hope to retain control over any potential problems. These dangers include exposing the unwary to any behaviours that may engage with one's personal data in potentially nefarious ways, and as we mention in 5.1, part of the UI challenge is the scrutability of the effects and actions taken by behaviours. In a broader sense, AM may even create its own privacy implications. By increasing the ease of combining multiple sources of data about a friend (twitter updates, facebook actions, last.fm feed, flickr photos), AM highlights how much personal information is being broadcast to the Web, and enables inference and reactive behaviours based on that information. It remains to be studied what users' major privacy concerns regarding AM are.

5.4 Rule Language, Engine Design, Fine-Grained Context

It is our ongoing work to identify the most useful type of rules for AM, and to design an easily comprehensible syntax for the constructs needed by those rules. We intend to explore how to support rule validity duration and reverting rule consequences. For example when checking location and setting a twitter status to 'at home', AM could suggest a rule that states when location is not 'home', unset the status, to avoid leaving the house and still appearing to be at home. There may also be need for a 'while/afterwards' construct, for example to filter e-mails while in a meeting to only those relevant, but remove the filter after the meeting. This also requires subtleties in book-keeping of other actions that may have fired. We also intend to further explore handling uncertainty, (we currently support approximate matching of strings), and the most feasible way of propagating uncertainty and how this should be displayed to a user. A simple feature that was found to be desirable in early test drives was an "ask user" tag to either ask a user for confirmation about an automated action, or to ask for some additional action parameter that cannot be automatically detected or derived.

As we integrate our user capture framework PLUM, as mentioned above, we have the potential of gaining fine-grained, frequently updated context such as currently running applications, visited websites, WLAN positioning, even web camera images. It will be interesting to see, for example, whether users with to publish this information as a feed through their state variables, and whether the rules become proportionally more fine-grained.

6. CONCLUSION

In this paper we have presented AM, a browser-based desktop tool that explores the blending of increasingly 'microblogged' personal, public and social data to drive context-aware reactive behaviours. We offer evidence that the design and implementation make it feasible to use these sources of information to automate our repetitive, tedious tasks. The core design problem we addressed is that of providing a suitable rule language for specifying the reactive behaviours, as well as a consistent data model and representation over which it is easy to write behaviours. With these contributions, others can start creating these blends of data, and sharing them as we discuss in ongoing work, and we can begin to explore the interesting user interface issues of how to present this time-saving automation for end-users, not just coders.

REFERENCES

1. André, P., schraefel, m., Wilson, M. L. and Smith, D. A. The Metadata is the Message. Web Science Workshop at WWW'08.
2. Bolin, M., Webber, M., Rha, P., Wilson, T., and Miller, R. C. Automation and customization of rendered web pages. UIST '05.
3. Ennals, R. J. and Garofalakis, M. N. 2007. MashMaker: mashups for the masses. SIGMOD'07.
4. Hartmann, B., Wu, L., Collins, K., and Klemmer, S. R. Programming by a sample: rapidly creating web applications with d.mix. UIST'07.
5. Hibernate: http://hibernate.org
6. Jsolait: http://jsolait.net/
7. Leshed, G. and Haber, E. and Lau, T. and Cypher, A. CoScripter: Sharing 'How-to' Knowledge in the Enterprise. GROUP'07.
8. Little, G., and Miller, R. C. Translating Keyword Commands into Executable Code. UIST 2006.
9. Malone, T. W., Grant, K. R., Lai, K., Rao, R., and Rosenblitt, D. A. 1989. The information lens: an intelligent system for information sharing and coordination. In Technological Support For Work Group Collaboration, Lawrence Erlbaum Associates, Mahwah
10. MochiKit: http://www.mochikit.com/
11. Plazes: http://plazes.com
12. Rhino – JavaScript for Java: http://www.mozilla.org/rhino/
13. SQLObject: http://sqlobject.org
14. Van Kleek, M., Shrobe, H. A Practical Activity Capture Framework for Personal, Lifetime User Modeling. UM2007.
15. W3 Web Ontology Language: http://www.w3.org/TR/owl-features/
16. Winograd, T. Architectures for Context. Human-Computer Interaction, 16(2, 3 & 4).
17. Wong, J. and Hong, J. I. Making mashups with marmite: towards end-user programming for the web. CHI'07.