"Everything Personal, Not Just Business:" Improving User Experience through Rule-Based Service Customization

Richard Hull, Bharat Kumar, Daniel Lieuwen, Peter F. Patel-Schneider, Arnaud Sahuguet, Sriram Varadarajan, and Avinash Vyas

Bell Labs, Lucent Technologies
600 Mountain Avenue
Murray Hill, NJ 07974
{hull,bharat,lieuwen,pfps,sahuguet,sv5,vyasa}@lucent.com
http://db.bell-labs.com/

Abstract. The web and converged services paradigm promises tremendous flexibility in the creation of rich composite services for enterprises and end-users. The flexibility and richness offers the possibility of highly customized, individualized services for the end user and hence revenue generating services for service providers (*e.g.,* ASPs, telecom network operators, ISPs). But how can end-users (and enterprises) specify their preferences when a myriad of possibilities and potential circumstances need to be addressed? In this paper we advocate a solution based on policy management where user preferences are specified through forms but translated into rules in a high-level policy language. This paper identifies the requirements for this kind of interpretation, and describes the Houdini system (under development at Bell Labs) which offers a rich rule-based language and a framework that supports intuitive, forms-based provisioning interfaces.

1 Introduction

One main reason for the tremendous success of the Web is that it managed – since its early days – to establish a personalized relationship with its user. Wasn't the "personal homepage", the first Web killer app? At this point almost all web portals and web-based services offer some form of customization, from letting the end-user select what content is displayed and how, to storing end-user values (such as credit card information) to simplify future interactions, to providing alerts based on end-user requests. The next revolution of the web will be based on service-oriented programming, embracing both the web services paradigm and also *converged* (web and telecom) services, *i.e.,* services based on the evolving integration of the telephony network and the Internet through standards such as SIP, Parlay/OSA and 3GPP/3GPP2. With the next revolution, will the infrastructure in place now be sufficient for customizing and personalizing the web and converged services of the future? This paper argues that the answer is a resounding NO! The paper then introduces the Houdini policy management infrastructure being developed at Bell Labs, a framework that will be sufficient for this purpose.

The customization infrastructure commonly used in today's web is *value-based*: the core logic of an application or service is essentially static, but end-users can insert personalized values to obtain customized behaviors. The prototypical example is a typical on-line newspaper, which lets the user indicate what types of and how many stories should be displayed on her "front page". Consider now a composite web/converged service of the future, which itself is formed from tens or perhaps hundreds of individual services, that gather information from a myriad of sources and potentially invoke numerous side effects to fulfill an end-user's request (or an enterprise's request). Because of the vast richness and variability in composite web/converged services of the future, if only value-based customization were supported, then an end-user would be overwhelmed when inputting all of her preferences. What we need is *rule-based* customization, whereby end-users can express their preferences at a higher, more generalized manner than possible with value-based customization. With rule-based customization, the preferences of the end-user can translate into rules which impact both the values used by the web/converged service and also portions of the *core logic* of the service.

Policy management is not new. A policy management reference architecture that cleanly separates application logic from policy decisions has converged from the data networking and telecom communities [17,11]. But we argue here that current policy management solutions are not yet appropriate for service oriented programming applications for two reasons. The first reason concerns expressive power and performance. In typical IETF-style[17] policy management, the rules have an **if** *conditions* **then** *actions* format, where the actions cause side-effects outside of the rules engine; rule chaining is not supported. We show in this paper that in the context of web/converged services, rule chaining is desirable, to permit substantial succinctness and modularity in rule specifications. In typical production-style rules systems (*e.g.,* ILOG [18], CLIPS [22]) chaining is supported, but also recursion. We argue that this is beyond the needs of customization for future web/converged services, and leads to unnecessary overhead in terms of rules creation, rules maintenance, and runtime space and time consumption. We advocate here the use of production-style rules with chaining but no cycles as the appropriate balance between expressive power and performance. The rules system used in the Houdini framework is based on this balance. This rules system has recently been incorporated into Lucent's MiLife ISG product, a Parlay/OSA gateway that allows internet- and web-based applications to access data and services hosted in telecom networks.

The second short-coming of current policy infrastructures for customizing future web/converged services is the lack of simple, intuitive support for end-users to specify rich preferences. A key part of the Houdini framework is an approach in which end-users provision preferences using a forms-based interface. These preferences are converted into a collection of tables and rules that captures the preferences. End-users are aware of the high-level "flow" of logic that the ruleset will use to interpret the user's preferences, but are insulated from the the explicit rules underneath. Importantly, the use of the Houdini rules engine permits relatively inexpensive modifications and extensions to the provisionable content, and permits radically different provisionable content for different user groups.

The primary focus of this paper is on *rule-based customization*. As such, we do not address the issue of managing end-user profile data, and we touch on the issue of storing user preferences data in a cursory manner. Management of (possibly distributed) profile data is the subject of [25], and is the topic of several industry standards groups (3GPP GUP[1], OMA[3], Liberty Alliance[20]).

Paper organization: Section 2 illustrates the need for rule-based customization using a motivating example from web/converged services and two broad themes (service selection and privacy) that arise in web/converged services. We then (in Section 3) derive from the examples key requirements for customization in service oriented programming. In Section 4 we present the Houdini framework and explain how it satisfies those requirements. We contrast our proposal with related work (Section 5) before offering final remarks and conclusion (Section 6). The emphasis of the current paper is to motivate the use of a policy enablement reference architecture for service oriented programming and to introduce the Houdini framework that supports it. Further information about some aspects of Houdini, including preliminary benchmarks and more details on self-provisioning, are presented in [15,14].

2 Motivating Examples

This section presents three representative areas where service oriented programming will need a high degree of customization. The first, focused on "presence" [9] and "selective-reach-me" [25], is based on a composite of web and converged network services. It involves specifying intricate preferences about when and how an individual can be reached for interactive communication. The last two are more fundamental. The second concerns the selection of atomic services to form a composite service that meets the individual needs of a given user in a given context. The third concerns how end-users can specify policies about who can access personal information, when, and under what circumstances.

2.1 Presence and Selective-Reach-Me

With the emergence of pagers, mobile telephony, and wireless handhelds it is increasingly possible to contact people anytime, anywhere. People sometimes prefer to be contacted on a wireline phone because of voice quality. Instant messaging is also popular as a near-realtime mechanism for communication, *e.g.,* if someone is already in voice communication with someone else. People sometimes prefer not to be contacted at all, instead having messages routed to voicemail, email, or completely blocked. Because of this explosion of contact devices, two crucial emerging services are *Presence* and *Selective-Reach-Me* (SRM). Current products [19,26] support limited forms of presence and SRM capabilities, but the degree of customization provided by these is limited.

Figure 1 illustrates a possible architecture for supporting presence and SRM built around various web services. In particular, the Presence and SRM Web Server acts as a kind of hub/coordinator. It can obtain information from a user's calendar(s), address book, corporate directory, etc. (on the right side of the figure), and it can access raw presence information from various communication networks, and also invoke or re-route calls (bottom of figure). In this example, telecom and network services such as mobile phone presence are "wrapped" to act as web services, *e.g.,* according to the Parlay X [11] standards.

A truly useful Presence [9] service will (a) provide requesters with information about a person's presence across all the devices associated with the person, or simply suggest the best device(s) for communicating with the person, and (b) share the appropriately personalized subset of this information only with authorized requesters at appropriate times in appropriate contexts – to protect privacy. Furthermore, it is sometimes useful to display the person's degree of availability/interruptibility, so that requesters can exercise discretion about whether to attempt live communication, and through what medium.

SRM is an important "dual" of Presence, which allows people to "call people, not devices." More specifically, SRM allows a user to specify preferences about when she is available by what communication medium (wireline phone, wireless phone, IM, etc). This might be based on time of day; day of week; appointments listed in calendars; current presence, location, and availability; whether the person is currently on the phone and with whom; etc.

A tremendously broad variety of data may be relevant for determining a user's Presence, for determining how to share that presence information, and for determining the correct routing for an SRM call. For a class of typical office workers, these decisions can best be described (informally) in terms of a flow of logic, that starts with information about the user's typical schedule (*e.g.,* commute from 8:00 to 8:30, in office from 8:30 to 5:30, etc.), the user's calendar (showing meetings, with whom, and where), and recent device usage; and interprets those based on availability preferences ("if meeting with my boss then route subordinates to voicemail", "if meeting anyone else, then use speech-to-text and route calls to instant messaging and use text-to-speech for my responses (if any)"), hints about location ("if I used my office phone in the last 30 minutes then I'm probably still in my office, unless I have a meeting elsewhere or I typically go home at that time"), and other information (*e.g.,* looming deadlines tend to decrease degree of availability). Different classes of users will have different stereotypical patterns that can be customized. This will minimize their data entry, maximizing ease of use.

2.2 Service Selection

Through the use of standards such as WSDL and SOAP, and repositories such as UDDI for web service descriptions and URLs, the web services paradigm affords tremendous flexibility in dynamically selecting and composing services. Further, it is expected that over time the languages for describing web services will become richer [12], to include, for example, behavioral and other descriptions (*e.g.,* about function call sequencing and pre- and post-conditions). This permits personalization not only of the individual services, but also in selecting what services will be used to support a service request.

To illustrate, we recall one pragmatic approach to service selection and composition, that involves the use of hierarchical composition [6]. For this, it is assumed that a library is available that holds both "base" web services (*e.g.,* to make a flight reservation, to make a hotel reservation) and "template" web services (*e.g.,* that can invoke a flight reservation service followed by a hotel reservation service). A composite web service can be created dynamically by selecting first a top-level web service template (*e.g.,* to create a business or vacation trip itinerary with flights, car rentals, hotels, etc.), and then choosing additional template or base web services to fill in the "gaps" of that template; this process continues iteratively until there are no more "gaps".

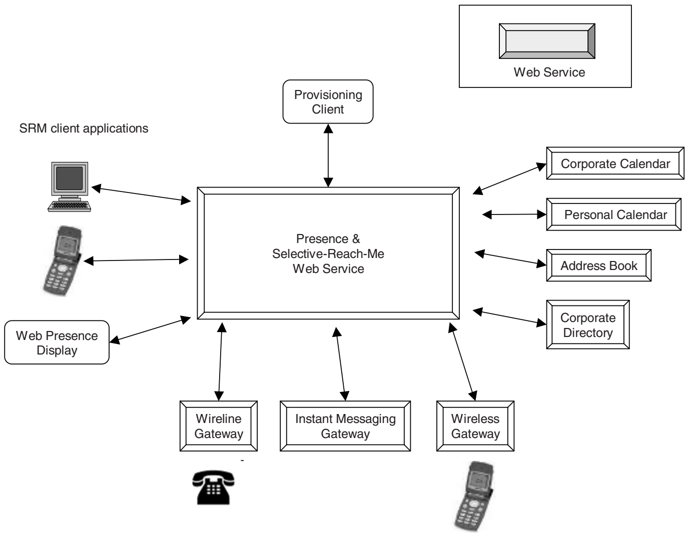

Fig. 1. Presence and SRM Service Architecture

The process of selecting template and base services will need substantial personalization. In the travel itinerary case just described, factors involved in making the selections will include the kinds of trip the person is planning, preferences about who is supplying services (*e.g.,* use Orbitz vs. Travelocity vs. a particular carrier's web service), transactional and quality-of-service guarantees provided by the services, their response times, the kinds of status reporting supported, the costs of the different services, the deals that they can access, etc.

More broadly, personalization in the selection of service will be relevant to any technology that is used for web service selection and composition, not just hierarchical composition. Also, this personalization will be relevant wherever dynamic service creation arises, including, *e.g.,* in the Presence and SRM services mentioned above.

2.3 Privacy Shield

One key aspect of customization is the sharing of personal information (see [25,7]). The end user is willing to share personal information in order to get a better user experience, but she wants to remain in control of what information is used, by whom and for what purpose. We designate this set of preferences as the user *privacy shield.*

As illustrated by the first motivating example, the end user may want to restrict access to information such as presence information (*e.g.,* buddy lists for instant messaging clients). For instance, a user is willing to share traveling information (*e.g.,* favorite airlines, preferred seating, diet, passport number) with authorized Web travel agents, shipping and billing information (*e.g.,* home address, preferred shipping method) with authorized e-retailers. A user is willing to share personal entries from her calendar with friends and family, corporate entries with co-workers, etc.

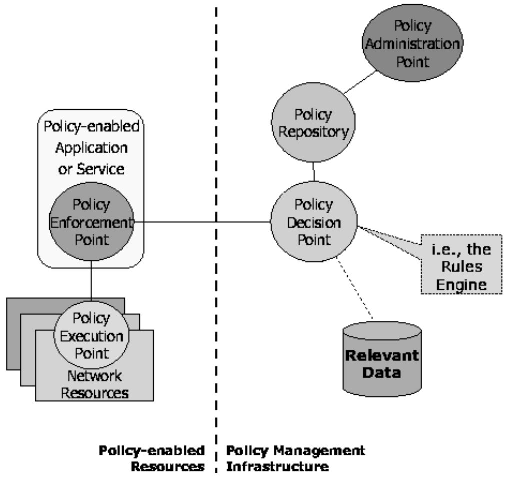

Fig. 2. Policy Management Reference Architecture

Reference [15] applies the Houdini framework to privacy shields stemming from mobility and ubiquitous computing. It also discusses related standards.

3 Requirements on Policy Infrastructure for Services

We now list six requirements on a service oriented programming infrastructure needed to support customization. The first requirement is at a meta-level – we argue that a policy-management infrastructure is needed. The remaining requirements focus on particular characteristics needed by that policy-management infrastructure.

Separation of Concerns: As illustrated in Section 2, there is a need for a high degree of customization in service oriented programming. Furthermore, different user groups will use different kinds of data and paths of reasoning to achieve their customization. Capturing all these variations within services is a daunting, if not impossible, task. Instead we advocate a substantial separation of the core service logic (which should reside in the service code) from the core customization decisions (which should reside in a policy infrastructure). Various standard bodies [17,11] have defined a reference architecture for policy management, identifying some key components and roles (see Figure 2):

- *Policy Enforcement Point*: in charge of asking for a decision and enforcing it (*i.e.,* invoking the required actions). In some cases, the enforcement point uses a *Policy Execution Point*, which is where the invoked actions are actually executed.
- *Policy Decision Point*: in charge of rendering a decision based on a ruleset and a context. The decision point only returns a decision and has absolutely no side-effect on the environment.
- *Policy Repository*: in charge of storing policies, *e.g.,* rulesets.
- *Policy Administration Point*: in charge of provisioning the rules (*i.e.,* letting the user access and modify directly or indirectly the rules that define her preferences) and other administrative tasks (*e.g.,* checking that the rules are valid).

Service-oriented programming should use the standard policy-management reference architecture in order to achieve a separation of concerns between core service logic and personalization.

We now turn to requirements on the policy management infrastructure itself, *i.e.,* the portion of Figure 2 to the right of the dashed line.

Can "scale" to a high degree of complexity: As illustrated by the examples of Section 2, the input data for customization decisions can be voluminous and highly varied in kind and richness (beyond string and numeric types). Customization decisions themselves can be very complex and may require a high-level expressive language to describe individual decisions and/or decision flows.

End-users may find it useful to conceptualize the data into different categories (*e.g.,* for SRM, data indicative of my current and future activity, data indicative of my location, data about the caller), and may conceptualize a flow of decisions based on that data. Users may want to incorporate heuristic and/or probabilistic reasoning, since the information available in the network is incomplete, and human behavior is non-deterministic.

A policy infrastructure must be rich enough to support this myriad of rich input data, and the ability to combine the data in complex ways.

Easy to provision for the end-user: In spite of the implicit variability and complexity of emerging composite services, preference provisioning must be conceptually straightforward for end users. If provisioning is too complex, then the vast majority of users will never bother to provision their preferences, and will lose many of the benefits of web- and converged-services. Although intricate rules might be used inside the policy infrastructure, the end-user must be able to specify his preferences by filling in tables, and making simple selections. One aspect of achieving this will be to support, for a single service such as SRM, different preferences-provisioning interfaces for different classes of users (*e.g.,* office worker, student, emergency worker).

A policy infrastructure must offer user-friendly provisioning interfaces that help the user specify and verify preferences and permit broad variations for different user groups.

Performance: The need for high performance is clear in the case of converged-network applications, such as SRM or mobile commerce, since users expect wireline and wireless telephony network responses at sub-second speeds. Users are more tolerant of services provided over the web, but adding more than a second or two for policy-enablement would be unacceptable. Since many composite web services will involve multiple decisions (either by a single atomic web services, or by multiple web services in a composition), the time for individual decisions must still be well under a second. Further, in order to be economically feasible this response time must be attainable even under heavy load (*e.g.,* 100K or more decisions per hour per processor).

A policy infrastructure must introduce very little overhead when added on top of an existing service.

Ease of Integration: Policy enabling an already existing service (or set of services) must respect the given architecture in terms of domain boundaries, security, etc. The policy component must adapt to the architecture.

*A policy infrastructure must be applicable to a variety of deployment architectures (*e.g., *client/server vs. embedded, centralized vs. distributed).*

Genericity and Extensibility: The proposed framework should not be specific to one application domain but capable of dealing with any kind of preferences. This implies that the framework is extensible in the sense that new data sources can be added and made accessible at the level of the evaluation context.

A policy infrastructure must be generic and extensible to enable it to easily adapt to changing application requirements.

The next two sections provide more detail about tools we are developing to support customization of web and converged services.

4 The Houdini Rules System

This section describes Houdini, which provides the core of our approach for customization in service oriented programming. In particular, we describe here the Houdini language and rules paradigm, and the Houdini engine. For completeness we also touch on the rules engine performance and provisioning infrastructure in Houdini, although these are considered in more detail elsewhere [14,15]. As will be seen below, the Houdini infrastructure has been developed to satisfy all of the requirements identified in Section 3. Houdini is intended primarily to serve as a Policy Decision Point in the sense of IETF/Parlay perspective on policy management. Houdini supports production-style rules and acyclic chaining. The language is side-effect free, i.e., the evaluation of a rule only affects variables inside the ruleset. This allows the solution to be high-performance and scalable with predictable run-times. Note that the rules language syntax is typically hidden from end-users, who are provided a forms-based interface for provisioning their preferences and changing them as desired.

The section begins with a description of the Houdini language, to provide an overview and insight into how it supports modular rule sets, and then moves to the system architecture, performance, and provisioning. A more detailed treatment of the language appears in our earlier work [13].

Rulesets and Variable Typing: Following the paradigm of policy enforcement points and decision points, Houdini provides an explicit notion of *ruleset*, *i.e.,* the set of rules that should be used for a given decision request. As described below, the Houdini rules language is strongly typed. In particular, all variables used by a ruleset have specified types. (We currently support scalars, record of scalar, list of scalar, and list of record). Furthermore, each ruleset has an explicit *input* and *output* signature, corresponding to the parameters passed in and out by decision requests made against the rule set.

All variables used in a ruleset, along with their types, must be specified explicitly during provisioning. The variable types are used for statically checking the rules at provisioning time, as well as validating decision requests at run-time. The types currently offered by the language support traditional types found in relational databases, *i.e.,*, *atomic types* (bool, char, int, float, string, time), *record types* (record of atomic types) and *collection types* (list of atomic or record types).

Houdini supports condition-action rules that take the form:

if (*condition*) **then** $action_1$**;** $action_2$**;** ... $action_n$**; end**

Conditions are arbitrary boolean expressions constructed out of Houdini types and the usual boolean operators. When a rule is invoked, the rule condition is evaluated, and if true, all rule actions are executed in the order in which they are specified. Actions operate only on the variables of the rulesets. There are no side-effects outside of the current evaluation context. In the current version of Houdini there are two kinds of action: variable assignment (=) and list-value assignment (+= to append).

Since Houdini follows the IETF/Parlay PEP/PDP model, the rules engine is not responsible for how the input data is obtained for a decision request. For example, if the rules engine is being utilized by a network service, and some input data values cannot be obtained due to some technical problems (*e.g.,* off-line, out of service, etc.), it is the responsibility of the PEP to generate meaningful data values for the given situation before passing them to the rules engine.

Also, various services may interpret the same raw data in different ways, *e.g.,*, a user's presence and availability may be different based on different applications / contexts. The Houdini rules language is expressive enough to allow different types of data interpretations to be specified in a flexible manner.

Single-valued vs. list-valued variables: Houdini supports operations with both single-valued and list-valued variables during rule evaluation. Houdini provides two mechanisms for working with list-valued variables. The first is through explicit built-in functions, that can manipulate a list as a whole (*e.g.,* aggregate operators such as `count`, `min`, `max`, `sort`, etc.) or combine lists in various ways. The second is to use the production system style of rule evaluation, in which an occurrence of a list-valued variable in a rule condition leads to consideration of the rule for each individual member of that variable. To differentiate between the use of a list-valued variable `listVar` for the first and second mechanism, Houdini uses '`listVar`' to refer to the collection as a whole, and uses '`?listVar`' to indicate that production system semantics should be used. We refer to occurrences of '`?listVar`' in a rule as an *element-wise occurrence*.

We now present a couple of rules to illustrate the Houdini syntax and semantics. The following rule (chosen from a representative SRM ruleset), specifies that if the boss calls while the callee is in a meeting with a subordinate (and possibly others), the callee is *Available* to the boss (in this case represented as 4 on a scale of 1 to 5). The variable `meeting_with` is list-valued, and has an element-wise occurrence here. (The `min` function is used because, for example, some other meeting participant might be providing a different value for `availability`.)

Rule for meetings

```
rule: check_availability_for_boss
if ((caller_relationship == "Boss") && (?meeting_with = "Subordinate"))
then
   availability = min(availability, 4);   /* 4 = Available */
end
```

Note that conditions can operate on both atomic (*e.g.,* `caller_relationship`) as well as record and collection (*e.g.,* `?meeting_with`) variables – they can handle both element-wise occurences and aggregate operators over whole lists.

The above rule explicitly refers to values such as "Boss" and 4. As an alternative, assume now that a list-valued variable `interrupt_table` is holding a family of records, each record corresponding to a category of caller, a category of meeting participant, and an availability level for that combination. In this case the following rule can be used in place of the above rule and other rules analogous to it.

Rule for availability

```
rule: check_availability_for_all
if ((?interrupt_table.caller == caller_relationship) &&
    (?interrupt_table.interacting_with == ?meeting_with))
then
   availability = min(availability, ?interrupt_table.in_meeting_availability);
end
```

Importantly, list-valued variables in Houdini can be used to support capabilities such as white-listing and black-listing. Note that Houdini provides a generic framework to which a particular application (like the one above) adds domain knowledge to specify what data is needed and what rules make sense. The rules interpret the data.

Rule Semantics: The semantics for rule execution follows the typical approach for forward-chaining production systems. We present here only the details of rule execution when element-wise list variables are present.

Definition (Semantics with element-wise variables). Given a production-style rule r, let $prod(r) = \{v_i \mid 1 \leq i \leq n,\ v_i$ is a variable with one (or more) element-wise occurrences in $r\}$. The execution semantics of r are defined as: $\forall(e_1, e_2, ..., e_n) \in (L_1 \times L_2 \times ... \times L_n)$, evaluate r by using e_i as the value associated with each occurrence of $?v_i$, for $1 \leq i \leq n$, where L_i is the list of current members associated with variable v_i for each i, and $\times$ is the cross-product operator. □

In other words, for each tuple of elements, one from each of the L_i's, the rule condition is evaluated, and if the condition is true, the rule actions are executed in the context of the list elements being operated on.

Ruleset Acyclicity: While Houdini does support forward chaining of rules, it restricts the chaining to be acyclic. The specific notion of acyclicity currently used is defined as follows.

Definition (RuleSet Acyclicity). Given a ruleset S, let $rules(S) = \{r \mid r$ is a rule in $S\}$. Also, given a rule r, let (where 'v' ranges over variables)

$$\begin{aligned} def(r) &= \{v \mid v \text{ is defined in an action of } r\} \\ use(r) &= \{v \mid v \text{ is used in condition or RHS of any assignment in an action of } r\} \\ use_c(r) &= \{v \mid v \text{ is used in the condition of } r\} \end{aligned}$$

Let $G(S) = (V, E)$ be a graph defined on a ruleset S where $V = \{r \mid r \in rules(S)\}$ and $E = \{(r_i, r_j) \mid (i \neq j,$ and $def(r_i) \cap use(r_j) \neq \phi)$ or $(i = j,$ and $def(r_i) \cap use_c(r_i) \neq \phi)\}$. Then S is *acyclic iff* $G(S)$ is acyclic. □

Essentially, for ruleset S, we define a graph $G(S)$ where each node in the graph corresponds to a rule in the ruleset S, and there is a directed edge from node r_i to a different node r_j *iff* there is a variable defined in r_i which is used in r_j. Also, there is a self-edge on a node r_i *iff* there is a variable defined in r_i which is also used in the condition in r_i. We say that S is acyclic if there are no cycles in $G(S)$ (including no self-edges).

The acyclicity condition restricts the Houdini rules' expressive power, in contrast with, *e.g.*, general-purpose production system engines like ILOG Rules, CLIPS, OPS5 (see related work in Section 5). However, the decision to restrict to acyclic chaining is based on the focus of Houdini, namely to support reasonably rich policy management for service oriented programming, especially in the time-constrained context of web and converged services. We have built several converged applications with Houdini (including presence, SRM, and location-based services), and lack of recursion was not a problem for any of them. We thus feel it provides a good compromise between performance and functionality. We note that our notion of acyclicity is more restrictive than, but in the same general spirit of, the notion of stratified logic (or datalog) programs.

Conflict Resolution, Priority, and Acyclicity: Conflict resolution is determined by two-levels of ordering of rules in a ruleset. The primary ordering is dictated by the acyclicity restriction on a ruleset S, i.e., rules are evaluated according to a topological sorted order of the graph $G(S)$. The secondary ordering criteria is based on explicit rule priorities, if given (note that in the examples presented above, the optional `priority` keyword has been committed). If explicit priorities are not specified, priorities are assigned based on the order of rules in the rules definition section (rules specified later in the section have higher priority). This secondary ordering is applied to the primary ordering, as long as it still results in the rules being evaluated in *s*ome topological-sort order of $G(S)$.

Note that because of the acyclicity restriction, each rule is considered for execution at most once. For rules affecting a given single-valued variable A, we can say that the “highest priority rule wins”, or more precisely, that if multiple rules have the effect of assigning a value to A, then the rule with highest priority that executes with true condition determines the ultimate value of A. An analogous semantics is given in the case of rules that assign a value to list-valued variables.

Note also that domain experts produce the rulesets, so only they need to understand the conflict resolution—end users will simply fill out forms with their preferences.

Modularity: As mentioned earlier, each ruleset that can be evaluated in response to a decision request has an associated input/output signature. The variables that do not belong to this signature are termed as *intermediate* variables. The use of intermediate variables with forward-chaining allows one to construct modular rulesets, where each module corresponds to the set of rules used to define an intermediate or target variable.

We illustrate this with the SRM service. Based on a callee’s network activity, a rule (or set of rules) can infer his location; based on his location, his calendar and his recent device usage, another rule set can infer his activity and the devices he can be reached at; based on his activity and caller identity, a rule set can decide to (not) try to allow the callee to be reached. Each of these sub-decisions can be thought of as a *module* within the overall ruleset. The combination of intermediate variables and the acyclicity of Houdini rulesets implies that the family of modules associated with a ruleset can be arranged in a directed acyclic graph that represents the flow of decisioning. This modular structure makes it easier to specify and reason about complex rulesets.

Validity Checking: Houdini provides the ability to perform validity checking on rulesets being provisioned. Various levels of validity checking are performed in the following order. First, *syntax checking* is performed to ensure that the ruleset conforms to the rules language grammar. Next, *type checking* is performed to verify that variables are used according to their types (*i.e.,* boolean variables should not be added together).

Acyclicity checking is then performed to make sure that the ruleset is acyclic (and also to generate an appropriate rule ordering for evaluation). Finally, *data dependency checking* is performed to ensure that (taking the optimistic view that all rule conditions will be true at run-time), if the rules are evaluated according to the order generated during acyclicity checking, no variables will be used before being defined (unless they are input variables); moreover after all rules are evaluated, all output variables must have been defined. Note that this obviously need not be true during run-time, but it does help catch some invalid rulesets. Thus, only limited run-time checking is needed.

Self Provisioning As noted above, Houdini provides a framework for self-provisioning of preferences, described in detail in [14,15]. Briefly, that framework includes three layers: (1) A front end that presents a series of forms through which end-users can fill in blanks and mark check-boxes in order to express their preferences; (2) A family of relational tables and rule *templates* are associated with the forms; and (3) The user preferences are mapped into tuples for the relational tables and/or *concrete* rules constructed from the rule templates. Users can also be provided with an image of the overall flow of decisioning, based on the DAG corresponding to (all or some) of the intermediate and target variables that are inferred.

Different families of users (*e.g.,* office working, road warriors, field service technicians, students, home-makers) can be provided with different sets of provisioning forms. Different sets of forms can also be provided for "naive" users vs. "power" users. Importantly, all of the provisioning interfaces can be supported by the same underlying policy management infrastructure, *i.e.,* there is no need to re-compile the policy engine or services when modifying or extending the family of data values that preferences are based on. Experts produce the forms and write the corresponding rules. End users simply fill out intuitive forms as described in [14,15].

Toolkit-based Architecture: The Houdini implementation has been structured around a *toolkit* approach, where the toolkit provides a collection of C++ classes containing the core functionality of loading/parsing/executing rulesets, with the application-specific portions (scheduling of events/processes/threads, communication with internal/external components) being layered on top of the toolkit to create the complete application. The Houdini framework is also extensible, allowing additional types and functions to manipulate them to be incorporated easily (the functions can be incorporated either at compile- or run-time).

Applications that wish to interact with a rules engine can incorporate the Houdini toolkit into the application itself, and call the Houdini API methods as necessary. Conversely, different protocols (*e.g.,* SOAP, CORBA, etc.) can be layered very easily on top of the Houdini API to create a stand-alone rules engine that can interact with various clients. Indeed, interfaces for SOAP and the CORBA-based Parlay/OSA standard have already been created for Houdini, in addition to a more direct TCP/IP interface. There are plans for a SIP interface as well.

Performance: Preliminary performance results on a Sun UltraSPARC-IIe with 1GB of main memory for the Houdini engine that we have developed at Bell Labs show that Houdini can execute on the order of 35K to 40K rules per second for the SRM service, and can render decisions in under 3 milliseconds [14,15]. This performance is clearly within the acceptable range for customization of web- and converged-services. We expect that this performance is substantially better than rules engines supporting full production system rules (with cycles), because of the substantial data structures needed in connection with the RETE algorithm [8] and its descendants. More performance details can be found in our related work [14,15].

Work is underway on performance improvements for Houdini. Some particular areas we are focusing on include decision caching, data caching, rule caching, taking advantage of common rule structures, etc. We also expect further optimizations will be possible for specific application contexts, *e.g.,* for a web services host or a telephony service provider.

5 Related Work

There are several other technologies that are potential solutions for the requirements described in Section 3, but all are missing something important that our solution offers.

Hard-coded solutions: The simplest solution is to just write code that implements the user preferences, but that can be customized by reading data provided by end users (usually in the form of tables). For example, end users could specify their whereabouts by entering information into a table of days and times. This solution has several advantages. First, it does not need a system to execute any rules. Second, it is relatively easy to write an interface for users to enter the table data. However, this solution is not sufficiently flexible for our purposes. It does not allow the computation of different information or differing control flow for different kinds of users. In certain cases though (*e.g.,* packet routing applications where extreme performance is needed), hard-coding is the only way.

Decision trees: Another solution is to use a form of decision tree [21], where the decision tree uses branching conditions to determine the action to take. Decision trees can be quite general, but decision trees of any reasonable size are difficult for end users to create, as it is difficult to keep track of the context of internal nodes and, moreover, it is hard to determine the correct structure of the tree.

"Simple" rule formalisms: Yet another solution is to use a simpler rule formalism, such as the no-chaining rule formalism in the IETF standards [17]. In this rule formalism, the effect of running a rule is to trigger some action external to the rule system, thus preventing one rule from affecting the behavior of another. This potential solution is again too limited as it prevents the computation of intermediate information and using this in later rules, such as computing whereabouts and using this to determine call acceptability in our SRM examples. This "simple rule formalism" approach has been embraced by some Web preference languages such as P3P/APPEL [27], PICSRules [28], and more recently XACML [23]. However, those languages are primarily designed to address privacy issues, and thus have either restricted vocabulary for rule conditions and actions (*e.g.,* APPEL), or no chaining (*e.g.,* PICSRules, XACML), and are not designed to be general purpose personalization languages.

More powerful formalisms: It would also be possible to use a more-powerful rule formalism. Many systems for such rule formalisms exist, such as OPS5 [5], CLIPS [22], ILOG [18]. We compare Houdini directly with ILOG (however, some of the arguments can be applied to the other systems as well).

The core of the ILOG Rules engine [18] consists of a fast and robust implementation of the RETE[8] algorithm, which is recognized as the fastest and most efficient algorithm for production style rules with cycles. The ILOG rule language supports a rich set of data types, permitting the use of (C++ or Java) objects on both sides of the rules. Object properties and methods (possibly with some side effects) can be invoked. ILOG rules execution consists of rules, stored in an *agenda*, which execute against objects, stored in a working memory. Rule execution may modify working memory objects, therefore new rules may become eligible for execution and thereby added to the agenda.

The problem with this style of rules is that as soon as cycles are allowed it becomes dramatically harder to write correct rule bases or to estimate run-time – crucial in real-time, converged services. This would be a particularly difficult problem to overcome in our situation, as the actual creation of the rule bases is done by end users, with developers only providing rule templates.

In Houdini there is also a working memory to hold the "context", or set of variables being accessed and manipulated by the ruleset execution. However, because of the restriction to acyclic rulesets, a variable is not used or read until all writes (and overwrites) to it have completed. As a result, the "agenda" in Houdini is much less dynamic than in ILOG; in particular, the sequencing of rules is computed by the acyclicity condition and can be determined in advance. This reduces the possibility of unexpected interactions between rules, simplifies debugging, and simplifies the Houdini internal execution algorithms considerably.

We believe that there is little benefit to cycles in rules in service customization, so we do not feel that paying the performance penalty for going to a more-powerful rule formalism is justified.

6 Conclusions

This paper's core observation is that intelligent e-services arising from the web services paradigm and the converged network will need rule-based customization. Further, for these applications, the typical IETF-style rules language is not sufficient; rather, a production-style language that supports chaining but not recursion is appropriate. More broadly, the paper identifies requirements on the policy infrastructure needed in the realm, and describes a framework and the high-performance Houdini system under development at Bell Labs, that satisfies these requirements. It also discusses privacy and provisioning aspects.

This paper raises several areas for further investigation; we mention three here. An obvious topic is further optimization of rules systems based on Houdini-style rules. Some natural places to look are pre-"compilation" and caching of rulesets, and exploiting commonality across rulesets. The trade-off between storing preferences in rules, or in tables and using rules to interpret the tables, should be explored. It will also be useful to develop techniques to avoid "unnecessary" testing of conditions based on (design-time or run-time) analysis of the rulesets (as done for the Houdini rules language [16]; note that it was referred to as the Vortex rules language in that paper).

Research on personalization has focused mainly on automated discovery of user profile and preference data. Important recent work (*e.g.,* [2]) has focused on using data mining to discover user preferences in the form of association rules (i.e., no chaining). If a Houdini-based language is adopted for specification of user preferences in web and converged services, then it would be useful to extend the techniques of that work, to discover rulesets over the richer rules language.

The current paper has focused largely on providing a single rules engine that is used by a single web service, where that service typically plays the role of coordinator of other web services. In a composite web service, several of the individual services may be policy-enabled, perhaps using the same rules paradigm or perhaps using different ones. In spite of this distribution and possible heterogeneity, end-users should be able to specify a single set of "global" preferences, that are mapped appropriately to the participant e-services. Preliminary work in this direction is found in [24,4,13].

Acknowledgements. The authors would like to thank Prasan Roy for fruitful discussions on this paper.

References

1. 3GPP. Generic User Profile, 2001. http://www.3gpp.org.
2. G. Adomavicius and A. Tuzhilin. User profiling in personalization applications through rule discovery and validation. In *Proc. Fifth ACM SIGKDD Intl. Conf. on Knowledge Discovery and Data Mining*, 1999.
3. Open Mobile Alliance. http://www.openmobilealliance.org.
4. X. Ao, N. Minsky, and T. D. Nguyen. A hierarchical policy specification language, and enforcement mechanism, for governing digital enterprises. In *Proc. of IEEE 3rd Intl. Workshop on Policies for Distributed Systems and Networks (Policy2002)*, 2002.
5. L. Brownston, R. Farrell, E. Kant, and N. Martin. *Programming Expert Systems in OPS5: An Introduction to Rule-Based Programming*. Addison-Wesley, Reading Massachusetts, 1985.
6. V. Christophides, R. Hull, and A. Kumar. Querying and splicing of XML workflows. In *Proc. of Intl. Conf. on Cooperating Information Systems (CoopIS)*, 2001.
7. C. Clifton, I. Fundulaki, R. Hull, B. Kumar, D. Lieuwen, and A. Sahuguet. Privacy-enhanced data management for next-generation e-commerce. In *Proc. VLDB*, 2003. To appear.
8. C. L. Forgy. Rete: A Fast Algorithm for the Many Pattern/Many Object Pattern Match Problem. *Artificial Intelligence*, 19:17–37, 1982.
9. PAM Forum. Presence and availability forum home page. http://www.panforum.org/.
10. Apache Foundation. Module mod_rewrite URL Rewriting Engine. http://httpd.apache.org/docs/mod/mod_rewrite.html.
11. Parlay Group. The Parlay Group – specifications. http://www.parlay.org/specs/index.asp.
12. R. Hull, M. Benedikt, V. Christophides, and J. Su. E-services: A look behind the curtain. In *Proc. ACM Symp. on Principles of Database Systems (PODS)*, pages 1–14, 2003.
13. R. Hull, B. Kumar, and D. Lieuwen. Towards federated policy management. In *Proc. IEEE Policy 2003*, 2003.
14. R. Hull, B. Kumar, D. Lieuwen, P. Patel-Schneider, A. Sahuguet, S. Varadarajan, and A. Vyas. A policy-based system for personalized and privacy-conscious user data sharing. Technical report, Bell Labs, 2003. http://db.bell-labs.com/project/e-services-customization/personal-data-sharing-2003-TM.pdf.

15. R. Hull, B. Kumar, D. Lieuwen, P. Patel-Schneider, A. Sahuguet, S. Varadarajan, and A. Vyas. Enabling context-aware and privacy-conscious user data sharing. In *Proc. IEEE Intl. Conf. on Mobile Data Management*, 2004. To appear.
16. R. Hull, F. Llirbat, B. Kumar, G. Zhou, G. Dong, and J. Su. Optimization techniques for data-intensive decision flows. In *Proc. IEEE Intl. Conf. on Data Engineering*, 2000.
17. IETF. Policy Framework, 2001. http://www.ietf.org/html.charters/policy-charter.html.
18. ILOG. ILOG Rules. http://www.ilog.com.
19. iMerge Enhanced Business Services (EBS). http://www.agcs.com/aboutv2/iMergeEBS/.
20. Liberty Alliance. http://www.projectliberty.org.
21. Tom Mitchell. Decision tree learning. In Tom Mitchell, editor, *Machine Learning*, pages 52–78. McGraw-Hill, 1997.
22. NASA. CLIPS. http://www.ghg.net/clips/CLIPS.html.
23. OASIS. XML Access Control Language. http://www.oasis-open.org/committees/xacml.
24. L. Pearlman, I. Foster, V. Welch, C. Kesselman, and S. Tuecke. A community authorization service for group collaboration. In *Proc. of IEEE 3rd Intl. Workshop on Policies for Distributed Systems and Networks (Policy2002)*, 2002.
25. A. Sahuguet, R. Hull, D. Lieuwen, and M. Xiong. Enter Once, Share Everywhere: User Profile Management in Converged Networks. In *Proc. Conf. on Innovative Database Research(CIDR)*, January 2003.
26. Appium Technologies. Fuzion-UC: Unified communications, October 1 2002. http://www.appium.com/pdf/fuzion_uc.pdf.
27. W3C. A P3P Preference Exchange Language (APPEL). http://www.w3.org/TR/P3P-preferences.
28. W3C. PICSRules. http://www.w3.org/TR/REC-PICSRules.
29. W3C. Platform for Internet Content Selection (PICS). http://www.w3.org/PICS.
30. W3C. Platform for Privacy Preferences Project (P3P). http://www.w3.org/TR/P3P.