Wizard Building in an Abstract Portal Framework

by

Christopher J. Hogger, Frank Kriwaczek and Myrto Serafetinidou

Department of Computing,
Imperial College of Science, Technology and Medicine,
London SW7 2AZ.

Departmental Technical Report: 2000/18
ISSN 1469-4174

December 2000

Abstract

This paper investigates a particular way in which enterprise portals can support the working lives of their users. Through monitoring users' behaviour and observing archetypal sequences of actions, the system constructs abstract templates of tools that assist them to carry out such sequences efficiently and easily in the future. We call these new tools "wizards" after the utilities introduced by Microsoft into their applications and operating systems, designed to guide users through potentially complex tasks such as constructing graphs in Excel or setting up the computer to access the Internet.

At the heart of the process lies a re-programmable rule-base encoding the principles by which action records are to be assembled into intelligible generic patterns of working practice, and thence converted into templates of new wizard tools providing direct support for those patterns.

1. Introduction

In the context of Internet technology the use of the term portal is relatively recent. There is, as yet, no precise and universally accepted technical definition of a portal. Indeed, it has come to denote many ostensibly different things [F99], [WSHJ00]. There is, nevertheless, general acceptance that a portal—consistent with its literal meaning—should serve as some kind of gateway providing access to certain types of resources in a manner personalized to the requirements of the user.

Amongst the various types of portal, the most ambitious is the “workspace portal”, currently not really much more than a vision: a single coherent integrated portal which presents its users with all the information-related resources they will ever need to carry out their jobs. Following [M99] such desktop environments are often called Enterprise Knowledge Portals. They aim to provide personalized information and information-processing tools as well as collaborative computing capabilities of all kinds, to connect users with other people according to their abilities, expertise and interests, and to proactively deliver links to such content and people as are relevant to whatever the users may be working on in real time.

In this paper we concentrate upon a single feature of the workspace portal that we anticipate could be beneficial in making the portal owner’s working life both easier and more efficient. In the course of a working day a user will carry out a large number of information-processing tasks: opening email messages and examining any attachments, creating new text documents, updating database records, and so on. In this stream of activity there are likely to be repetitive sequences of actions, often involving different operations on the same object—for example, opening a text file received as an email attachment, editing this text file and inserting into it a table copied from a spreadsheet file, and finally sending the text file as an email attachment back to the original sender.

By monitoring the user's actions and identifying commonly-occurring action sequences, the portal will be able to produce tools to assist the user through such action sequences in the future. Each tool will be an interactive utility presenting a sequence of dialogue boxes through which the user can navigate while filling in the details required. In accordance with the terminology first introduced by Microsoft, we call such tools "wizards". At present, our system creates platform-independent abstract templates from which such wizards can be suitably crafted to take account not only of the requirements of the host operating system but also of principles considered essential for their usability [C98]. Some work has been done on automating such wizard building [K98], albeit within a more limited context and with a far narrower focus.

The aim of our work on recognising sequences of actions, based upon that of [S00], is distantly related to [DH98], in which the authors propose an algorithm which, when applied to sequences of user actions, allows a user interface to adapt over time to an individual's pattern of use. There is also a strong interest in the fields of active databases and of workflow in concatenating recurring sequences of primitive events or actions.

An active database system monitors situations of interest and triggers appropriate responses in the form of production rules which are stored within the database. This arrangement provides a uniform approach for dealing with a range of common database tasks including enforcing integrity constraints and maintaining derived data values. Since some production rules may be fired only after particular combinations of primitive database events have occurred, it is vitally important to be able to specify composite events flexibly [MZ95], [MZ97], as well as detect them efficiently [UUMN98].

Workflow is concerned with the coordination of people, information and events that cause work to flow, such as deadlines, and the status of work required to meet organisational objectives. Using specifications and process definitions, systems can be built which help the user to accomplish tasks and to be more effective in achieving the objectives, through presenting required information at the right moments in the process whilst automating those aspects of the work that are strictly repetitive. Workflow researchers are interested in reusing business process models, including templates (models that have been especially adapted for reuse) [JC00]. Such templates are essentially complex composite actions. In [AGL98], the authors apply data mining techniques to construct process models from workflow logs. Their aim is to create new workflow systems that conform to previous behaviour and are therefore easier to introduce.

2. Portal Responses

A portal might evolve by many different mechanisms. The present work focuses on mechanisms instigated by the observation of events, specifically of actions taken by the portal's user. This notion can be formulated as a condition-action rule

```
happens(E) ---> portal_response(E)
```

or, equivalently, as a Prolog rule

```
portal_response(E) :- happens(E)
```

We shall prefer to use the Prolog notation, primarily because the underlying procedures and data structures are themselves implemented in Prolog in our framework. One significant reason for this is that Prolog can operate upon partially-instantiated structures, using pattern-matching either to access or to create concrete data instances. The rule above can be read as saying that, it in order for the portal to respond to an event E that event must happen.

Portal responses may be of many kinds. Some may have the effect of revising priorities in the deployment of existing resources, for instance to reflect that some application is used particularly often or that some species of document appears to have greater significance than others. Responses of this kind would be expected to adjust the portal so as to elevate the visibility, readiness or accessibility of high priority resources. A different kind of response is one concerned with process optimization. For instance, observing that the user undertakes a variety of disparate searches for data, the portal may discern a more uniform, comprehensive or efficient means of doing the searches, and perhaps take the

initiative to identify or acquire appropriate new tools. Yet another kind of response is relation-building, that is, the discernment of potentially useful connections between resources. Observing that a user imports records from two databases and prepares from these a document using data from each, the portal may recognize a meaningful relation between the relevant attributes and propose other data sources supplying instances of that relation.

Here we focus on yet another kind of response which we refer to as action composition, of which typical exemples are wizard-building and toolbar customization. A particularly simple form of action composition is that of concatenation, that is, aggregating selected actions into some ordered sequence which is then regarded as a single composite action. This new action can then be implemented as a self-contained tool, or wizard, capable of marshalling and coordinating the underlying atomic actions on the user's behalf. The instance of the rule above that we are interested in here is therefore

```
wizard_build(E) :- happens(E)
```

This formulation does not suppose that the process of building a wizard is premissed upon a single event E, rather that E contributes to the further evolution of whichever wizard (or wizards) can make use of it. In general, this evolution requires access to the portal's history of events. This is because the decision-making entailed in the aggregation of events needs to know not only which events are potential candidates but also their temporal order and their frequencies of occurrence. Given this, we assume that every event instigates an update to a stored event history, that is, another kind of portal response formulated as

```
update_history(E) :- happens(E)
```

For practicality there clearly have to be policies in effect that govern the granularity, relevance and temporal baseline of the events recorded in the history.

3. Characterizing Events

The characterization of an event wholly determines what can be done in response to it. We represent any event as a term `event(U, T, A, R, Ts)` whose arguments are as follows:

user (U)	the initiator of the event
tool (T)	the tool employed
action (A)	the action applied using the tool
resources (R)	the resources so entailed in the action
timestamp (Ts)	the time when the event was completed

An event history is a set of such terms. The representation inherently imposes—besides some degree of uniformity—some degree of granularity upon the observable events. As one instance of this, we have chosen not to observe what is happening during the interval between the initiation and completion of an event. Moreover, we have chosen to focus mainly upon those events whose actions apply tools to resources.

Deciding suitable types for the arguments is—like our deciding what an event shall be in broad terms—a matter of making practical choices. There is no formally optimal formulation for the underlying ideas. Such choices must, on the one hand, be sufficiently general to match realistically observable events whilst, on the other hand, be sufficiently specific to enable realistically useful responses to flow from those events. The choices we have made have been applied so far in one medium-scale, real-world context to test their suitability. Later in this paper we will present some results from that exercise.

3.1 User (U) Types

We take the simplest case only in which U is an atom serving as a user identifier. In most cases U identifies the portal's user, but in some cases U identifies some other user. This flexibility enables our portal to observe certain kinds of event instigated elsewhere.

3.2 Tool (T) Types

A tool T is represented as a compound term

tool(Name, Exec, Location, Rights, Params)

All five arguments are represented by atoms whose interpretations are as follows:

Name: the internal system name for the tool, appropriate to the underlying operating system. In Windows, for example, 'MSWord' could be such a name. A tool name in Unix might be 'emacs'.

Exec: the internal system name for the executable file. It may take the form 'Filename.ext' in cases where the file has an associated extension.

Location: the location of the executable file. This could be, for instance, a path on a local drive—such as '/etc/bin/emacs/'—or a URL to a remote computer.

Rights: a member of {true, false} denoting whether or not the user has access to the executable file.

Params: the values of any parameters the tool expects to be entered in the user's command line invoking the tool. In the special case where no such parameters are needed, Params takes the value null.

3.3 Action (A) Types

The framework supports twelve conceptually distinct kinds of action, each denoted by some atom. The available atoms are

created	copied
moved	pasted
renamed	received
deleted	sent
opened	logged_in
updated	logged_out

Thus, a term of the form event(U, T, Created, R, Ts) signifies an event in which the user U used tool T and thereby Created resources R at time Ts.

It is unnecessary for an action to have arguments of its own. All the material upon which it bears is contained instead in the other top-level arguments—primarily T and R—of the event term. It is for this reason that actions can be denoted simply by atoms.

Not all actions involve tools or resources. Actions which do involve them must be compatible with them. Implicitly, then, there are further constraints upon the type characterizations of events as a whole in order to render them meaningful. As an example, the constraints upon event(U, T, logged_in, R, Ts) require U to be the portal's user, T to be the atom null (no tool is employed in logging-in), R to be null (no resources are involved) and Ts to be not null. By contrast, the constraints upon event(U, T, created, R, Ts) require U to be the portal's user, T to be not null, R to denote precisely one resource and Ts to be not null. The full set of assumed constraints can be found in [S00].

When a null atom appears in an argument position within an event term, its purpose is to signify that the argument is undefined, that is, can have no concrete meaningful value. This provision must be clearly distinguished from that which permits an argument to be undetermined. In the latter case the argument is the underscore symbol. For example, a term of the form event(U, _, moved, R, Ts) represents an event in which resources R were moved but in which the tool used (if any) was not observed, or not recorded or not of interest. This abstract term can be viewed as standing generically for all the concrete terms obtained by choosing various values for the tool argument.

3.4 Resource (R) Types

Deciding upon suitable resource types is perhaps the hardest aspect of characterizing events. In the real world there is a wide diversity of resources, each having its own particular kind of content, handles, and so on. In our framework we have concentrated on just three primary types of resource—document, database table and email, denoted by the atoms doc, dbtable and email respectively. Each resource has the following attributes:

Type: its primary type, being any member of {doc, dbtable, email}

Name: its current internal system name (an atom).

Loc: its current location, such as a local pathname or a URL (an atom).

Rights: its rights, being a list whose membership is any subset of {read, write}.

U_cr: the identifier of the user who created it (an atom).

T_cr: either null or the local pathname of the tool used to create it (an atom).

OldN: for the renamed action this is the resource's prior name (an atom), but in all other cases is null.

OldL: for the moved action this is the resource's prior location, being of the same type as Loc (an atom), but in all other cases is null.

Atts: further attributes specific to the primary type: for a document it is null; for a database table it is a term atts(Spec, Schema) where Spec is an atom encoding a query or view definition, and Schema is an atom encoding the table's schema; for an email it is a term atts(Sender, Recipients, Subject, Attachment) where Sender is a user identifier (an atom), Recipients is a list of user identifiers, Subject is the message's subject line (an atom) and Attachment is a list of atoms each encoding the local pathname or URL of an attached file.

More: a place-holder for any other information about the resource that the portal might be required to exploit.

The representation of a resource is then a term of the form

```
resource(Type, Name, Loc, Rights, U_cr, T_cr, OldN, OldL, Atts, More)
```

3.5 Timestamp (Ts) Types

A timestamp is represented by a term whose arguments are chosen according to the temporal granularity with which we wish to record events. We have chosen a term of the form

```
ts(Year, Month, Day, Hour, Min, Sec)
```

Whenever an event's action is completed, the arguments of the event's timestamp are instantiated with values drawn from the system clock.

A complete example of an event drawn from the real-world data we have experimented with is the following:

```
event('u146', tool('Outlook', 'outlook.exe', '/ProgramFiles/Outlook', true, null), opened,
[resource(doc, 'agenda.agd', '/users/u146/Outlook/', [read, write], 'u146',
'ProgramFiles/Outlook/outlook.exe', null, null, null, null)], ts(2000, 9, 5, 9, 10, 44))
```

This represents the completion at time 9.10.44 am on September 5th 2000 of the action by user u146 of opening with the Outlook tool a single resource, namely a document agenda.agd—drawn from the local directory /users/u146/Outlook/—which that same user had previously created with that same tool.

4. Event Patterns

Events are real and tangible manifestations of an individual's work. We can identify them not only correctly but also unobtrusively. More difficult is the discernment of meaningful work patterns. A sequence of temporally-ordered events may be recognizable as a whole to the user in relation to their defined role. However, the flexibility of working arrangements to achieve overall objectives may be such that some sequences, whilst logically consistent with the role, may not be directly recognizable as such. Moreover, the environment may be one in which no individual roles have been formally defined.

Given this, our framework incorporates a customizable rulebase which, for any given event sequence, decides whether that sequence is a significant work pattern for the user that would merit wizard support in the portal. This allows us the flexibility of determining such patterns without regard to institutional roles and without the need to consult the user, although it does not preclude the assistance of either of these. The rulebase recognizes such patterns within the portal's event history solely on the basis of their internal content.

The event history is conceptually a set of stored events. In general, however, a pattern will have some natural temporal order over the timestamps in the events contained in it. In the absence of such an order a set of events may not amount to a set of logically coherent actions. For example, a resource cannot be sent as an email attachment before it has been created. A coherent set will normally require that its timestamps satisfy at least some given partial order, and in some cases may even need to be totally ordered. Given two particular events e1 and e2 we might require that e1 occurred before e2 and/or that the interval between them were less than some given bound.

A system routine within the portal's infrastructure generates the event history as a stream of event terms. The portal's wizard-building operation proceeds concurrently, reading terms from this stream as they arise. Each new term is checked for structural integrity and, if validated, added to a database representing all the events observed so far since some selected (and resettable) temporal baseline. This database is effectively indexed by timestamp in order of decreasing recency. From the new state of the database the wizard-builder now elicits some of the more recent events—starting with the one just added—and stores them in a list called Events, from which patterns will subsequently be sought. For subsequent convenience, Events is arranged in order of increasing recency. Its membership (and hence its length) is constrained by a modifiable bound in the rulebase upon the elapsed time between the most recent member and every other member, effectively limiting how far back in history the search for viable wizards will be taken.

Once the list Events has been constructed, any pattern identified within it is has to be some not-necessarily-contiguous subsequence of it. For instance, if Events has the form [e1, e2, e3, e4, e5, e6, ...] then the subsequence [e1, e3, e4, e6] might be a pattern. Whether it is accepted as a pattern depends upon whether it is recognized as such by customizable *coherence rules* held in the rulebase. These rules stipulate the detailed relationships between events that should hold in order that they have sufficient functional coherence to be regarded collectively as a significant work pattern.

In general the coherence of a subsequence S of events might depend upon relationships defined upon S as a whole, or upon arbitrary parts of it. In practice it is easiest to test for coherence only between adjacent members in S. This reduces somewhat the discernibility of meaningful connections between events, but we believe this is not a very significant reduction and that it is outweighed by the practicality of defining and evaluating pairwise constraints.

Each coherence rule takes the form

cohere(C, Cnext) :- *conditions*

A concrete pattern [e1, e3, e4, e6] must then satisfy cohere(e1, e3), cohere(e3, e4) and cohere(e4, e6). The concrete pattern is not, however, the structure we wish ultimately to extract. A coherence rule of the form just shown is typically such that the arguments C and Cnext in its head are non-ground, that is, contain variables. When such variables appear, this signifies that C and Cnext are defined to cohere whatever values those variables might take. For instance, it might be enough that C has an opened action and Cnext a copied action, without regard to the particular tools employed for those actions. In that case the tool arguments within C and Cnext would be anonymous variables. Moreover, the

anonymity would prevent them from being bound by the evaluation of the rule's conditions. What we wish to extract is not the concrete input pattern, but rather an abstract template of that pattern having no greater specificity than that required by the coherence rules. Given the concrete pattern [e1, e3, e4, e6], one extracted template may be [t1, t3, t4, t6]. Then, e1, ..., e4 must be instances of t1, ..., t4 and the rulebase must contain satisfied rules whose heads are cohere(t1, t3), cohere(t3, t4) and cohere(t4, t6).

Prolog is well-suited to the handling of such partially-instantiated structures. Even so, the coding of a procedure capable of efficiently extracting all templates from an Events list is still not entirely easy. The procedure currently used in our framework is as follows:

```
find_templates(Events, Templates) :-
    findall(Template, template(Events, Template), Templates).

template(Events, Template) :-
    scan(Events, [], [], Template), length(Template, N), N>1.

scan([], _, T, Template) :-
    reverse(T, Template).
scan([_ | Es], B, T, Template) :-
    scan(Es, B, T, Template).
scan([E | Es], [], T, Template) :- !,
    scan(Es, [E], T, Template).
scan([Enext | Es], [E], [], Template) :-
    cohering_pair(E, Enext, C, Cnext), !,
    scan(Es, [Enext], [Cnext, C], Template).
scan([Enext | Es], [E], [Cprev | T], Template) :-
    cohering_pair(E, Enext, C, Cnext),
    C=Cprev, !,
    scan(Es, [Enext], [Cnext, Cprev | T], Template).

cohering_pair(E, Enext, C, Cnext) :-
    clause(cohere(A, Anext), Conditions),
    copy_term((A, Anext), (C, Cnext)),
    E=A, Enext=Anext,
    Conditions.
```

The principal case in the algorithm is the final scan clause, whose logic is such that a partially-constructed list T of template events in reverse temporal order is extended by a term Cnext if (a) a pair (E, Enext) of concrete events satisfies a coherence rule whose head's argument tuple is a variant copy of (C, Cnext) and (b) C unifies with the term Cprev most recently put in T. Requirement (b) is needed to ensure that each abstract term in the template coheres with both its predecessor and its successor, as is identically required of the corresponding concrete terms in the input pattern. The final state of T is reversed by the first scan clause to yield a completed template arranged in order of increasing recency.

There is scope for further optimizing the above procedure so that each time it is invoked in response to an update of the events history it does not need to test for coherence of any event pair that has already been tested during a prior invocation. This could be done by indexing the concrete events in the history and by consulting and maintaining an auxiliary database of index pairs corresponding to event pairs already shown to cohere.

The returned list Templates is then pruned to eliminate any members not satisfying a modifiable length constraint defined in the rulebase. In our experimentation we have set the constraint to reject templates whose length is not in the range 3-10. A template of length below 3 might represent so little activity that a corresponding wizard would confer little realistic benefit, whilst for one of length above 10 the wizard might be overly specialized or unwieldy to use.

5. Coherence Rules

The following is an example of a coherence rule.

```
cohere(event(U, _, received, Rs, Ts1), event(U, _, opened, [R], Ts2) :-
   portal_user(U),
   R=resource(email, _, _, _, _, _, _, _, atts(Sender, _, _, _), _),
   U\==Sender,
   member(R, Rs),
   within_limit(Ts1, Ts2).
```

It identifies coherence between the receipt of an email by the portal's user U and their subsequent opening of it, conditional upon the email having been sent by someone other than U. The intuitive additional requirement that U shall be among the recipients of the email will have been already checked during the validation of whichever concrete input event is matched with the rule-head's first argument.

An example of a concrete event pattern is:

```
[event(u146, null,
           received,
           [resource(email, 'inbox.dox', '/Win/ApplicationData/Identities/u146/Outlook/',
            [read, write], null, null, null, null, atts(u416, u146, 'Book Meeting', null), null),
            resource(email, 'inbox.dox', '/Win/ApplicationData/Identities/u146/Outlook/',
            [read, write], null, null, null, null, atts(u811, u146, 'Request', '~pn105'), null)],
           ts(2000, 9, 5, 9, 2, 52)),

 event(u146, tool(Outlook, outlook.exe, '/ProgramFiles/Outlook', true, null),
           opened,
           [resource(email, 'inbox.dox', '/Win/ApplicationData/Identities/u146/Outlook/',
            [read, write], null, null, null, null, atts(u416, u146, 'Book Meeting', null), null)],
           ts(2000, 9, 5, 9, 10, 20)),
...]
```

The first event records that u146 received two emails, one from u416 about the booking of a meeting and the other from u811 issuing a request, and the second event records that about seven minutes later u146 opened the first of those emails. Provided that this elapse time is within the prescribed limit, the coherence of these events as defined by the rule above will result in a template for u146's portal having the form

```
[event(_, _, received, _, _), event(_, _, opened, [_], _), ...]
```

In itself this template reflects no more than the pattern of receiving resources and then opening a single resource, not necessarily among those received. A portal wizard constructed from it might support just this level of abstraction and enforce nothing further. This would depend upon the logic of whatever further software was employed to construct wizards from templates. If the coherence rule were more specific, say:

```
cohere(    event(U, _, received, [resource(email, N, L, _, _, _, _, _, Atts, _)], Ts1),
           event(U, _, opened,    [resource(email, N, L, _, _, _, _, _, Atts, _)], Ts2) :-
   portal_user(U),
   Atts=atts(Sender, _, _, _),
   U\==Sender,
   within_limit(Ts1, Ts2).
```

then we could extract a more specific template reflecting the receipt of just one resource, comprising an email, and the subsequent opening of that same email:

```
[event(_, _, received, [resource(email, N, L, _, _, _, _, _, Atts, _)], _),
 event(_, _, opened,    [resource(email, N, L, _, _, _, _, _, Atts, _)], _),
...]
```

The expressive power of our templates is limited to what can be expressed solely by the form of the arguments appearing in their abstract event terms. Logical properties of those arguments tested within the bodies of the coherence rules cannot form part of the templates. In the example just shown, the coherence condition that the email's sender shall not be the portal's user is not enforced by the template. For the same reason, we cannot express in a template that the opened email must belong to some longer list of received resources.

In principle we could base our wizard-building upon extended templates which contained not only the abstract events but also logical conditions upon them selected from the coherence rules' bodies. These extra conditions could be used to constrain the compile-time construction of the wizards, or even be incorporated into their run-time operations. A further possibility would be to produce templates consisting only of lists of pointers pointing to the coherence rules satisfied during the identification of a pattern. Such alternatives, however, effectively presume the wizard-builder to be capable of translating arbitrary Prolog programs into system-level code driving the portal interface. It appears to us more practical to supply to the wizard-builder only the simple, non-logical templates of the kind first proposed, and to assume that separately-formulated software decides how best to implement them.

6. Frequency Analysis

The treatment described so far identifies wizard templates on the basis of the coherence rules provided. If this were the sole basis then no explicit account would be taken of the frequency with which concrete events and concrete patterns occurred. It would amount to declaring, *a priori*, that some pattern was inherently worth instituting as a wizard provided only that it could be observed to occur at least once in the event history.

A more rational basis is to require that a pattern p shall also have a sufficiently high frequency of occurrence, over some allotted time period, in relation to the frequencies of occurrence of its constituent events. One simple way of measuring this is by the *frequency fraction*

$$f_p / (f_1 + ... + f_n)$$

where n (>1) is the length of p, f_p is the number of occurrences so far of p and each f_i is the number of occurrences so far of event e_i (ignoring its timestamp). Then the nearer this fraction approaches to 1, the more significant is p within the totality of observed events. For each n we can stipulate a threshold t_n above which the fraction is taken to indicate that p justifies the building of a wizard; the least value for such a threshold is (1/n) but a more practical value might be, say, 0.8. A less discerning but more readily computable scheme is one which defines f_i as the number of occurrences so far of events having just the same action component of event e_i, ignoring all its other components.

Methods such as this fall within the more general topic of behaviour mining, itself just one aspect of enterprise modelling. They may rely solely upon frequentist analysis or may be additionally informed by other considerations such as the user's defined institutional role. In this present work we have ignored defined user roles and instead resorted to the coherence rules described previously. The coherence rules effectively filter the patterns that have been identified on the basis of frequency, by selecting those whose low-level actions are considered to cohere functionally within any user role.

In our framework, then, each new event arriving on the event history entails first updating the number of times that this species of event has occurred, then deriving the new candidate patterns it yields according to the coherence rules, and then updating the number of times that these patterns have occurred. Wizard templates are retained only for patterns satisfying the fraction threshold described above. There remain various further optimizations that could be implemented, though we have not yet explored these in detail. They include rejecting patterns which are smaller subpatterns of patterns having higher frequency fractions, and garbage-collecting stored events and patterns whose frequencies have not been updated for some specified long period of time.

7. Conclusion

Wizard-building offers useful economies to the user. Although working practices and host platforms are subject to change, the general logical principles underlying evolutionary portals are less volatile. We believe that those principles should be encoded explicitly as a transparent and easily-modifiable rulebase playing an active role in the portal framework. In particular, this rulebase should employ an ontology sufficiently abstract to enable portal events to be expressed and related in a platform-independent manner, whilst sufficiently concrete to enable practical association between the rulebase and the portal's real observables and actions. We have outlined above how this can be done in a simplified context dealing with a limited range of resource types, and we have tested it in a prototype implementation.

8. References

[AGL98] Agrawal, R., Gunopulos, D. and Leymann, F. “Mining Process Models from Workflow Logs”. Proc. of the Sixth International Conference on Extending Database Technology (EDBT). 1998.

[C98] Carliner, S. “Designing wizards”. Training and Development. 1998.

[DH98] Davison, B.D. and Hirsh, H. “Predicting Sequences of User Actions”. Proc. of the 1998 AAAI/ICML Workshop “Predicting the Future: AI Approaches to Time-Series Analysis”.

[F99] Firestone, J.M. “Defining the Enterprise Information Portal”. Executive Information Systems, Inc. 1999.

[JC00] Jørgensen, H. and Carlsen, S. “Writings in Process Knowledge Management: Management of Knowledge Captured by Process Models”. ISBN 82-14-01928-1. SINTEF Telecom and Informatics. 2000.

[K98] Kipp, N.A. “Hyperwizards: XML over CGI”. Proc. of XML'98. 1998.

[M99] Murray, G. "The Portal is the Desktop". Group Computing. May-June 1999, p. 22.

[MZ95] Motakis, I. and Zaniolo, C. “Composite Temporal Events in Active Database Rules: A Logic-Oriented Approach”. Proc. of the 4th Intl. Conference on Deductive and Object-Oriented Databases. 1995.

[MZ97] Motakis, I. and Zaniolo, C. “Temporal aggregation in active databases rules”. SIGMOD Rec. 26(2), pp.440-451. 1997.

[UUMN98] Urban, S.D., Unruh, A., Martin, G. and Nodine, M. “Expressing Composite Events in InfoSleuth”. Microelectronics and Computer Technology Corporation, Technical Report #MCC-INSL-131-98. 1998.

[S00] Serafetinidou, M. “Wizard Building in an Abstract Portal Framework”. MSc Dissertation, Department of Computing, Imperial College of Science, Technology and Medicine, University of London. 2000.

[WSHJ00] Wells, D., Sheina, M. and Harris-Jones, C. “Enterprise Portals: New Strategies for Information Delivery”. Ovum Report. 2000.