Extracting Reusable Knowledge from Portal Activity

Christopher John Hogger and Frank R. Kriwaczek
Department of Computing
Imperial College London
South Kensington Campus, London SW7 2AZ
UNITED KINGDOM
http://www.doc.ic.ac.uk

Abstract: - This paper describes an implemented model of a portal framework through which users in an enterprise may control their activities by following structured plans. The artefacts they manipulate, and the temporal schedules associated with them, are regulated by finite-domain constraints to express and enforce enterprise requirements. A significant feature of the model is a well-defined formulation of experience extracted from the user interface in the form of reusable episodes. Such episodes provide a valuable resource for enterprise users engaged in the development of their plans.

Key-Words: - portal, workflow, multi-agents, planning, constraints, reusability

1 Introduction

This paper describes our design and implementation of an experimental portal framework which helps users to pursue activities satisfying enterprise requirements. Activities are represented as schedules of atomic actions that manipulate artefacts, whilst the requirements are represented as relations desired to hold with respect to a declarative rulebase including finite domain constraints. The actions and requirements determine the attributes, including temporal ones, of the artefacts that evolve as the activities proceed. At any stage, from the experience gained so far in the running of the enterprise, we can extract partial plans and constraint solutions representing reusable knowledge of how aspects of that enterprise can be run.

Section 2 introduces the constructs expressing actions and requirements, and Section 3 describes the framework's operational features. A simple illustrative case study is presented in Section 4 to give an idea of how the system works in practice. Section 5 explains how the experience of a run can be extracted as a reusable entity, whilst Sections 6 and 7discuss related work and conclusions.

2 Enterprise Formulation

In principle, the requirements of an enterprise can be formulated as a purely declarative theory expressing the assumed properties of its various entities, their inter-relationships and their control. Achievable goals of the enterprise can then be identified with logical consequences of the theory, and derivations of those goals can be interpreted as particular runs of the enterprise. In practice such a model is too non-deterministic. A more practical approach is to formulate parts of that theory—particularly those concerned with control and scheduling—as pre-conceived procedural plans, whilst retaining in declarative form only those parts that can be sensibly implemented by inferential mechanisms.

2.1 Expressing Plans

We express a plan as a set of concurrent tasks each consisting of a collection of atomic actions with some associated control regime. An atomic action takes the form *action(args, st, et)* or *pre:action(args, st, et)* in which *action* identifies the species of action, *pre* is an optional precondition for performing it, *st* and *et* are its start and end times and *args* comprises any other arguments upon which it depends. For control purposes, a set of actions required to be performed in sequence is enclosed in <> whilst a set to be performed concurrently is enclosed in {}. Sets of either kind are called *blocks* and may be nested arbitrarily. Conditional branching between one block and another is expressed by a construct *if_then_else(cond, block1, block2)* where *cond* is the branch condition. Other forms of control construct are also implemented, but the above gives a sufficient initial outline of plan structure. A simple but concrete example of an individual task within some plan is the following, named t1:

```
t1 = < { < make(a, st1, et1), test(a, st2, et2) >,
          < make(b, st3, et3), test(b, st4, et4) > },
   assemble(c, [a, b], st5, et5),
   test(c, st6, et6), package(c, st7, et7), dispatch(c, st8, et8) >
```

This expresses that two artefacts a and b are to be made concurrently, each one tested after being made, then jointly assembled to produce artefact c which is then tested, packaged and dispatched, in that order. The action species make, test, assemble, package and dispatch are here specific to the enterprise domain. The terms a, b, c, st1, ..., st8 and et1, ..., et8 are ontological variables that will become bound to concrete values, or sets of possible values, as the plan's various tasks are carried out.

2.2 Expressing Requirements

A requirement is expressed as a logical goal (a *constraint*) of the form *rel(Args)* required to be satisfied by whatever definition of the relation *rel* is given in the enterprise rulebase. The argument vector *Args* typically comprises ontological variables occurring in the users' plans. Such variables are usually ones whose values fall within specifiable finite domains, and so the rulebase is in general a constraint logic program. The declared requirements are desired to be conjointly solvable when the plans have been performed. They ensure that the artefacts manipulated will possess acceptable attribute values and that actions are performed in accordance with a required schedule.

Not all constraints need be declared explicitly. Certain temporal constraints are implicit in the structuring of plans. In the example above, it is implicit that we have, for instance, st1≤et1, et1≤st2 and et2≤st3. The implementation automatically identifies such constraints and adds them to the declared ones. Declared constraints and rules for the example might include the following:

```
constraints = { max_dur(st3, et3, 3), correct_weight(b), ... }
rulebase = {
  max_dur(S, E, D) :- domain([S, D, E], 1, 1000), E-S<D.
 correct_weight(B) :-
     type(B, T), weight(B, W), conforms(T, W).
  conforms(steel, W) :- inrange(W, 3.5, 3.9).
  conforms(alloy, W) :- inrange(W, 1.7, 2.6). ...   }
```

The constraints aim to ensure that the elapse-time between starting the manufacture of item b and ending it is no greater than 3 time units, and that its weight will be in the right range for its metal type. The variables S, D and E here are restricted to a finite domain of time values expressed notionally as integers 1...1000. In practice the domain would comprise serial date numbers.

2.3 Users, Roles and Tasks

Each user of the system is viewed as a role-holder in the enterprise. Their role is to pursue concurrently a set of tasks of the kind described in 2.1 whilst satisfying the constraints. In the simplest arrangement, each user freely devises their plan and associated constraints. Coordination between users arises from the system's treatment of all their constraints as a global set and from allowing a constraint to relate ontological variables occurring in the plans of more than one user. For instance, one user might require that one of their actions be started only after the ending of some other user's action. Each user includes, in their role definition, ontology declarations to stipulate the source of the variables they use. For example, the declarations

```
ontology(ann, own, [a, st1, et1]).
ontology(ann, john, [st10]).
```

declare that ann is using variables a, st1 and et1 devised by herself and also a variable st10 devised by john. Conventions are applied to prevent ambiguity among variables.

The system also enables a user U1 to assign a role, or a role update, to another user U2. This process is implemented as U1 performing, within their own role, a special 'assign' action whose effect is to create and convey to U2 an electronic artefact whose content defines the assigned role or update. This arrangement provides the basis for a hierarchy of responsibilities that is used to control the process by which users revise their constraints if a constraint failure arises while they pursue their various tasks. The details of these mechanisms are not important in this paper but can be found in [1, 2], together with concrete illustrations of how they work in practice.

3 Plan Execution

The system shows to each user, via the portal interface, his progress in performing the plan. At any moment he can see the stage reached in each task. He may scroll back through a task to see when past actions were begun and completed, or scroll forward to review actions still to be done. In particular, the interface highlights the next action (or the next block of concurrent actions) to be done.

Alongside each action the interface displays, in a cellular format, the current domains of the action's start time and end time, as shown in Figure 1. In each case the domain consists of those time points shown as empty white cells. Each of these satisfies (is a solution of) the current constraints, which include an implicit constraint that the cell shall be no earlier than the current time on the portal clock. A white cell containing "*" signifies that the user has already committed to that particular time point, having previously clicked in that cell (when it was white and empty) to indicate that commitment. A grey cell signifies a time point that the user cannot choose because it does not satisfy the current constraints. Thus, at the stage represented by Figure 1 — where the portal clock reads "7" (i.e. July 7th 2004) — the temporal constraints stored internally in the system are such as to imply:

{ st1=4, et1=5, st2=5, et2=6, st3=7, 7≤et3, et3≤10, et3≤st4, st4≤et4, ... etc. }

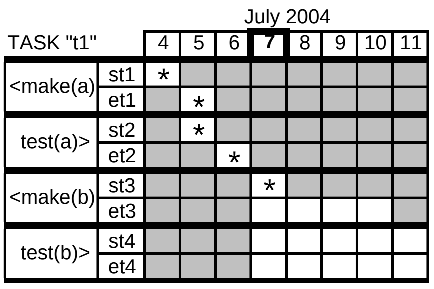

Fig. 1: progress of task in the interface.

These features of the system are driven by two engines, a *Plan Interpreter* and a *Constraint Evaluator*. The former is responsible mainly for displaying the state of progress through the plan and for responding to the user's commitments to begin or end actions. Its response to actions consists not only in recording and displaying the temporal commitments made but also in managing stored abstractions of any artefacts created or manipulated by those actions, as will be described later on.

The constraint evaluator applies algorithms that aim to simplify the current constraint set as much as possible. Initially this set contains all the explicit source constraints contributed by the various users and the implicit constraints (as described earlier) upon the temporal variables, together with predefined initial finite domains for selected variables (including the temporal ones). The evaluator need not operate synchronously with the interpreter (for instance, it may in principle remain dormant until all tasks have ended and then be invoked to check the constraints retrospectively), but in practice it makes sense for it to simplify constraints while the tasks are being performed in order to maximize the informativeness of the interface. The internal form of the constraint set is invisible to the user, but the interface indicates the solutions implied by the set. From Figure 1 the user can see that the current feasible solutions for st4 are {7, 8, 9, 10, 11, ...} but does not know (or need to know) whether the constraint set actually contains the constraint 7≤st4 or merely implies it through other constraints such as 7≤et3, et3≤st4.

As noted earlier, actions in our system are generally concerned with artefacts, these being viewed as the primary deliverables of the enterprise. Each one is represented in the system as a binding of the form Name=art(Attributes), which is created in an internal workspace when its associated *creating action* (such as make or assemble) is performed. The form taken by Attributes will be one of various predefined schemas, according to the category of artefact. A panel, for instance, might have attributes [panel, Type, Weight, Size]. Requirements upon Attributes are elicited from the user by a standard constraint of the form assist(Name, Attributes). When a creating action is performed this constraint, if present, pops up a request for the user to select values for the attributes of the named artefact. By this means the system constructs in the workspace a concrete instance of the artefact binding, such as

c = art([panel, steel, 3.7, (10, 20)])

The "real" activity entailed in actually making or manipulating the artefact is conducted off-line during the interval between the action's start and end times. The system and its interface just provides online guidance and control of the users' activities as they work through their plans, and constructs a symbolic record of what they have achieved by way of constraint solutions and artefact representations.

Actions such as assemble, which presume the prior existence of artefacts, are automatically suspended by the plan interpreter if those artefacts have not yet been made available in the workspace. That is, the user is prevented from starting such an action until they are available: he may have to leave a task, or some block within it, in abeyance until the awaited artefacts have been created, either through acting on some other part of his own plan or through the activity of other users.

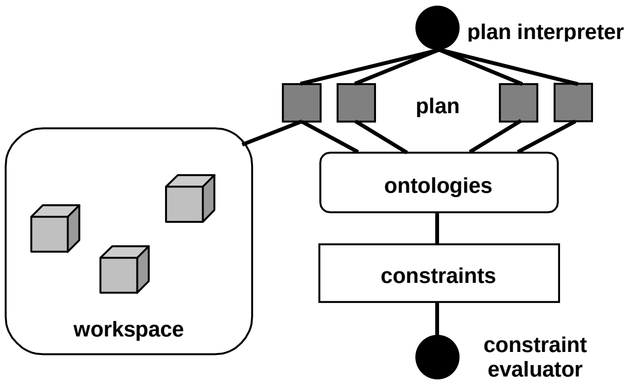

Fig. 2: outline of the system.

Figure 2 gives an outline of the system, showing several users' plans related through their ontological variables and obliged to satisfy conjointly the global constraints. The effect of execution is to determine solutions (or solution-sets) for the variables and to build a symbolic record of the artefacts.

4 Decision-Making Example

In this example a task t2 first assembles an artefact c and then subjects it to painting, drying, testing, packaging and dispatching. Painting and drying can be carried out in two ways. If the weather is warm (ambient temperature > 25^{o}C) then a quick-drying (q) paint is used and the panel is dried in an exterior (e) area. Otherwise a normal (n) paint is used and the panel is dried in an internal (i) oven. So task t2 has a decision-point and appears as follows:

```
t2 = < assemble(c, [a, b], st1, et1),
    if_then_else("temp > 25ºC",
    < paint(c, q, st2, et2), dry(c, e, st3, et3) >
    < paint(c, n, st4, et4), dry(c, i, st5, et5) >),
    test(c, st6, et6), package(c, st7, et7), dispatch(c, st8, et8) >
```

The temporal requirements are: external drying takes $\geq$3 days, internal drying $\geq$2 and all other actions $\geq$1. The task as a whole must take $\leq$11 days.

constraints = {
min_dur(st1, et1, 1), min_dur(st2, et2, 1), min_dur(st3, et3, 3),
min_dur(st4, et4, 1), min_dur(st5, et5, 2), min_dur(st6, et6, 1),
min_dur(st7, et7, 1), min_dur(st8, et8, 1), max_dur(st1, et8, 11),
... and implicit temporal constraints, artefact constraints, etc. }

We will suppose that the user starts the assemble action on July 4th and ends it on July 6th. Figure 3 shows the portal interface at this point.

July 2004

TASK "t2"			4	5	**6**	7	8	9	10	11	12	13	14	15
<assemble(c, [a, b])		st1	*	▨	▨	▨	▨	▨	▨	▨	▨	▨	▨	▨
		et1	▨	▨	*	▨	▨	▨	▨	▨	▨	▨	▨	▨
***temp* > 25^{o} C**	<paint(c, q)	st2	▨	▨				▨	▨	▨	▨	▨	▨	▨
		et2	▨	▨	▨				▨	▨	▨	▨	▨	▨
○ **yes**	dry(c, e)>	st3	▨	▨	▨				▨	▨	▨	▨	▨	▨
		et3	▨	▨	▨	▨	▨	▨				▨	▨	▨
○ **no**	<paint(c, n)	st4	▨	▨					▨	▨	▨	▨	▨	▨
		et4	▨	▨	▨					▨	▨	▨	▨	▨
	dry(c, i)>	st5	▨	▨	▨					▨	▨	▨	▨	▨
		et5	▨	▨	▨	▨	▨					▨	▨	▨
test(c)		st6	▨	▨	▨	▨	▨					▨	▨	▨
		et6	▨	▨	▨	▨	▨	▨					▨	▨
package(c)		st7	▨	▨	▨	▨	▨	▨					▨	▨
		et7	▨	▨	▨	▨	▨	▨	▨					▨
dispatch(c)>		st8	▨	▨	▨	▨	▨	▨	▨					▨
		et8	▨	▨	▨	▨	▨	▨	▨	▨				

Fig. 3: position at July 6th before the decision.

The interface displays the decision-point with two radio buttons, inviting the user to indicate the truth or falsity of "temp > 25^{o}C". If he clicks "yes" then the interface responds as shown in Figure 4.

The white cells indicate that, in order to complete the task within 11 days, the user must start the (quick-dry) paint action no later than July 8th.

July 2004

TASK "t2"			4	5	**6**	7	8	9	10	11	12	13	14	15
<assemble(c, [a, b])		st1	*	▨	▨	▨	▨	▨	▨	▨	▨	▨	▨	▨
		et1	▨	▨	*	▨	▨	▨	▨	▨	▨	▨	▨	▨
***temp* > 25^{o} C**	<paint(c, q)	st2	▨	▨				▨	▨	▨	▨	▨	▨	▨
		et2	▨	▨	▨				▨	▨	▨	▨	▨	▨
[✓ **yes**]	dry(c, e)>	st3	▨	▨	▨				▨	▨	▨	▨	▨	▨
		et3	▨	▨	▨	▨	▨	▨				▨	▨	▨
test(c)		st6	▨	▨	▨	▨	▨	▨				▨	▨	▨
		et6	▨	▨	▨	▨	▨	▨	▨				▨	▨
package(c)		st7	▨	▨	▨	▨	▨	▨	▨				▨	▨
		et7	▨	▨	▨	▨	▨	▨	▨	▨				▨
dispatch(c)>		st8	▨	▨	▨	▨	▨	▨	▨	▨				▨
		et8	▨	▨	▨	▨	▨	▨	▨	▨	▨			

Figure 4: position at July 6th after the decision.

He actually starts it on July 7th, completes it on July 8th, starts the dry action on the same day and completes that action on July 11th. The interface at that point is given in Figure 5 and shows the new temporal options for the remaining actions.

July 2004

TASK "t2"			4	5	6	7	8	9	10	**11**	12	13	14	15
<assemble(c, [a, b])		st1	*	▨	▨	▨	▨	▨	▨	▨	▨	▨	▨	▨
		et1	▨	▨	*	▨	▨	▨	▨	▨	▨	▨	▨	▨
***temp* > 25^{o} C**	<paint(c, q)	st2	▨	▨	▨	*	▨	▨	▨	▨	▨	▨	▨	▨
		et2	▨	▨	▨	▨	*	▨	▨	▨	▨	▨	▨	▨
[✓ **yes**]	dry(c, e)>	st3	▨	▨	▨	▨	*	▨	▨	▨	▨	▨	▨	▨
		et3	▨	▨	▨	▨	▨	▨	▨	*	▨	▨	▨	▨
test(c)		st6	▨	▨	▨	▨	▨	▨	▨			▨	▨	▨
		et6	▨	▨	▨	▨	▨	▨	▨	▨			▨	▨
package(c)		st7	▨	▨	▨	▨	▨	▨	▨	▨			▨	▨
		et7	▨	▨	▨	▨	▨	▨	▨	▨	▨			▨
dispatch(c)>		st8	▨	▨	▨	▨	▨	▨	▨	▨	▨			▨
		et8	▨	▨	▨	▨	▨	▨	▨	▨	▨	▨		

Figure 5: position at July 11th.

The user was relied upon in this example to respond to the ambient temperature condition by making an off-line observation of the weather. More generally, however, we allow the possibility that the system has an on-line connection to a temperature sensor that automatically tests the condition and updates the task structure in the interface accordingly.

We next consider how to extract reusable material representing the experience of pursuing, to any chosen extent, a task like the one just illustrated.

5 Extracting and Reusing Experience

Our conceptual model was originally designed mainly to enable any particular enterprise to set up a simple portal-based interface in which users' activities could be guided and monitored. The model does not presume the level of abstraction at which activities and requirements are formulated. In the simplest case it can be implemented as a light-weight, single-user workplace diary, whilst at the other end of the spectrum it can be implemented as a highly detailed and highly controlling multi-user management system.

The model further enables us to formulate the notion of extracting experience from running the system, experience that might then be stored for subsequent reuse by the same enterprise or exported to other enterprises for adaptation and assimilation into their own workplace contexts.

To put some structure upon this notion of experience we need firstly to give a more precise characterization of a task within a plan. An informal grammar for a simple subset of the plan language is the following:

```
action := atomic action | if_then_else(cond, block, block)
block := action | seq-block | conc-block | pre:block
(where cond and pre are any predicates)
seq-block := <block, ..., block>
conc-block := {block, ..., block}
task := seq-block | conc-block
plan := {task, ..., task}
```

Given a task T, a subtask t of T is then defined as:

```
t := subsequence of a seq-block in T |
     subset of a conc-block in T
```

so that a subtask has the same type as a task. Defining a subtask enables us to formulate the experience of performing any part of a task. However, we must take a further step in this formulation to reflect the fact that when performing a task containing branch-points — that is, *if_then_else* actions — the user (or the system) makes decisions each of which commits to just one branch. The user's experience is restricted to just the branches actually taken.

We therefore define a *linearization* of a task T. A linearization L(T) of T is the result of replacing each branch-point *if_then_else*(cond, block1, block2) in T by either cond:block1 or cond:block2. If the selected block already has its own precondition pre then prefixing it by cond just gives the block an expanded precondition $\text{cond} \wedge \text{pre}$.

An extractable experience in performing part of a task T is called an *episode*. Using the above definitions, an episode E is any linearization L(t: t is a subtask of T).

An episode is tantamount to a trace of the user's traversal through some or all of the actions contained in the original task. However, there is significantly more knowledge to be extracted than the trace alone because, in the course of performing this episode, commitments were also made in order to satisfy the constraints and action preconditions. More precisely, the epsiode E has associated with it an assignment θ_E of values to ontological variables such as time-points and artefact attributes. Some of these variables may be ones declared (in the programs within the constraint-defining rulebase) to be *finite-domain variables* whose values must always lie within prescribed domains. The temporal variables, in particular, are usually of this kind. As constraint evaluation and user activity proceeds, the value of each such variable consists of some non-empty subset of the initially-declared domain. By the time some task has been completed it may be that these domains have become reduced to singletons (i.e. unique solutions), but this need not always be the case.

The assignment θ_E by itself has no meaning without reference to the constraints that it satisfies. Many of the constraints in the enterprise-wide global set may be irrelevant to the episode under consideration. The relevant subset C_E can be identified syntactically as comprising just those constraints that depend directly or (by transitive closure) indirectly upon at least one variable occurring in the episode. However, even these have no meaning without reference to their definitions D_E in the rulebase. So D_E must also be identified in order to make sense of θ_E. The logical relationship between these entities is then that the (relevant) requirements D_E imply all the (instantiated) constraint predicates in $C_E.\theta_E$ (denoting the application to C_E of the relevant bindings in θ_E).

We know also that the episode must satisfy the set P_E of its action preconditions (if any). These may be predicated upon ontological variables, and so the information extracted from them is $P_E.\theta_E$.

The remaining knowledge associated with episode E concerns the set A_E of artefacts created by it. These can be identified from the syntactical content of E alone on the basis of the various species of artefact-creating actions contained in it. By the same means one can identify the set A^*_E of those artefacts that E presumed to pre-exist in the workspace. The episode thus has an associated artefact-mapping $M_E = A^*_E \rightarrow A_E$. The information extracted from this is $M_E.\theta_E$, which captures any contribution made by the constraints to the determination of the artefacts' attributes.

Given all the above, the total package extracted is a tuple $(E, \theta_E, C_E, D_E, M_E)$. In the light of this we now return to the example in Section 4. Via an interface tool the user can at any stage extract such a package by selecting an arbitrary linear subtask as defined above. It is not technically necessary that the selected subtask be restricted to just that part of the task already performed. Thus the package may, in general, consist partly of what has already been done and partly of what is intended to be done in the future. For instance, suppose that on July 11th (Figure 5) the user selects the subtask extending from the assemble action up to the test action (which he has not yet performed). He thereby extracts:

E = <assemble(c, [a, b], st1, et1),
  "temp > 25^{o}C" :
    < paint(c, q, st2, et2), dry(c, e, st3, et3) >,
    test(c, st6, et6) >

θ_E = { st1=4, et1=6, st2=7, et2=8, st3=8, et3=11,
    st6={11, 12}, et6={12, 13},
    and any attribute bindings on a, b, c }

C_E = the relevant constraints;

D_E = the relevant constraint definitions

P_E = "temp > 25^{o}C"

$M_E = \{ \{a, b\} \rightarrow c \} \theta_E$

This can be interpreted as a description of one way of assembling artefact c from pre-existing artefacts a and b and proceeding as far as testing it. Assumed attribute values for c, a and b will be specified somewhere within θ_E. The temporal assignments in θ_E show commitments to specific time-points for the actions prior to testing, but offer a range of time-points for the testing itself. The relevant constraints and their definitions supply the logical explanation (meaning) of these assignments, and P_E informs us that for this episode the weather has to be warm.

Given this characterization of an episode E, an interesting question is how to reuse E when developing some other task. That task can be viewed as a tree whose branch-points, if any, occur wherever decisions are made (as in Section 4). The reuse operation will consist of attaching E as a new twig at a position in this tree. Let B denote the sub-branch that extends from the root of the tree down to the intended attachment position. In attaching the new twig we will want to be assured that this operation is compatible with the assumptions associated with both E and B as regards the satisfiability of constraints and action preconditions as well as the dependencies between artefacts.

Prior to the attachement operation, the task under development already has its own associated constraint definitions D. After the operation the definitions become expanded to $D \cup D_E$, in order to support both the constraints C of the prior state of the task and the constraints C_E of the newly-attached twig. Let V* be the intersection of the ontological variables in C with those in C_E. If V* is non-empty then the values of its variables potentially become further constrained by the attachment operation. Let λ denote the further instantiation or domain narrowing (as appropriate), if any, that these variables experience. Then, if the assimilation of the new twig is to continue to offer feasible and correct solutions for the new state of the task we require, firstly, that $C\lambda$ and $C_E.\theta_E\lambda$ shall be implied by $D \cup D_E$ and contain no variables bound to empty domains. Secondly, we require the preconditions P_E associated with the twig to be logically consistent with those P_B on the branch B. Both these requirements can be easily tested by invoking the constraint evaluator and precondition evaluator already available in the system.

There remains one further condition needed to support the operation, namely that if E depends upon pre-existing artefacts A^*_E then B must already contain the means of producing them. B's syntax alone identifies the set A_B of artefacts it is capable of producing. They will need to have attribute values that are compatible with those that E expects, as determined by the constraints. For this it is sufficient that $A_B\lambda \supseteq A^*_E.\theta_E\lambda$, which is also easy to test.

The simplicity in formulation of the above conditions for the feasibility of assimilating episodic experience into a task under development is owed to the declarative features of the framework, whilst the testing of them is facilitated by the logical machinery already available for supporting the standard operation of the portal.

6 Related Work

Our system has some similarity to the open-source *uPortal* [3] deployed in some US universities and developed by the Java Architectures Special Interest Group. Its ability to reuse past experience to support future portal activity shares motivation with [5], whilst its focus upon user work patterns is similar to that driving the ontology-driven *KA2* system [6].

For finite-domain processing we use the *Sicstus Prolog CLP(FD)* engine. This has enough power to evaluate constraints in the intervals between user actions. It invokes the *Pillow* library to transform the state of the constraints into HTML web pages driving the interface. Other constraint technologies such *ILOG Rules* [7] might have been used instead, but most are more complex linguistically without offering greater expressive power.

We have not detailed our dealings with constraint failure. In [1] we show how its origin can be localized and its remedy sought by conservative belief revision and/or abductive constraint solvers like the *Sicstus*-based *A-system* [8]. These aims arise in other work on coherent workflow management [9, 10, 11, 12], some of which use logic programming but not *CLP(FD).*

7 Conclusion

We presented simple but expressive constructs for the procedural and declarative elements of enterprise activity. Logic programming for the declarative side gives access to many inferential technologies from artificial intelligence. These include finite-domain *(FD)* constraint systems as illustrated here, which in turn support OR-based tools for optimization, scheduling and decision making.

The implementation has been applied in a real-world context, namely managing multiple roles in an academic enterprise. There, artefacts are electronic documents such as text documents, spreadsheets and emails, containing information created and shared by the academic and administrative users, and the portal's abstract artefact-schemas connect directly to real documents residing on servers. In that same context we have also implemented the inverse process of deriving plans, construed as new portal tools, from statistical analysis of user behaviour [4].

Recent years have seen increasing use of project-based organisational structure for delivering bespoke products, sometimes involving multi-firm networks. Such organisations survive on their ability to set up and perform projects, key to which is the integration of project-based learning into the business [13, 14]. Here, projects are treated as the structures we call episodes, enabling learning while performing. We therefore believe that our model contributes a new approach to the devising of flexible knowledge management tools for conveying process-oriented experience and ideas from one project to another.

References:

[1] Hogger, C.J., Kriwaczek, F.R., Constraint-guided enterprise portals, *Proc. of 6th Int. Conf. on Enterprise Information Systems*, 2004, pp. 411-418.

[2] Ahmad, M.S., Hogger, C.J., Kriwaczek, F.R., Implementing a Collaborative Agent System using Prolog, *Proc. of ICIMu-2001, Int. Conf. on Information Technology and Multimedia*, 2001, pp.55-62.

[3] Gleason, B.W., Boston College University-Wide Information Portal — Concepts and Recommended Course of Action, *JA-SIG Portal Framework Project White Paper*, 2000.

[4] Hogger, C.J., Kriwaczek, F.R., Deriving tool specifications from user actions, *Transactions on Information and Systems*, Vol. E87-D, No. 4, 2004, pp. 831-837.

[5] McCallum, A.K., Nigam, K., Rennie, J., Seymore, K., Automating the Construction of Internet Portals with Machine Learning, *Information Retrieval*, Vol. 3, Issue 2, 2000, pp. 127-163.

[6] Staab, S., Angele, J., Decker, S., Erdmann, M., Hotho, A., Maedche A., Schnurr, H.-P., Studer, R., Sure, Y., Semantic Community Web Portals, *Computer Networks*, Vol. 33, 2000, pp. 473-491.

[7] ILOG, Inc., Business Rules, *ILOG Technical White Paper - www.ilog.com*, 2002.

[8] Kakas, A.C., Van Nuffelen, B., A-system: Declarative Programming with Abduction, *Lecture Notes in Artificial Intelligence*, Vol. 2173, 2001, pp. 393-396.

[9] Hwang, G.-H., Lee, Y.C., Wu, B.-Y., A New Language to Support Flexible Failure Recovery for Workflow Management Systems, *Lecture Notes in Computer Science,* Vol. 2806, 2003, pp. 135-150.

[10] Wainer, J., Bezerra, F., Constraint-based Flexible Workflows, *Lecture Notes in Computer Science*, Vol. 2806, 2003, pp. 151-158.

[11] Wainer, J., Barthelmess, P., Kumar, A., W-RBAC - A Workflow Security Model Incorporating Controlled Overriding of Constraints, *Journal of Cooperative Information Systems*, Vol. 12, No. 4, 2003, pp. 455-485.

[12] Dustdar, S., Collaborative Knowledge Flow - Improving Process-Awareness and Traceability of Work Activities, *Lecture Notes in Artificial Intelligence*, Vol. 2569, 2002, pp. 389-397.

[13] Gann, D., Salter, A., Innovation in Project-Based Firms: Constructing Complex Product Systems, *Research Policy, Special Issue on Complex Product Systems*, Vol. 29, No. 2, 2000, pp. 955-972.

[14] Morris, P.W.G., The Validity of Knowledge in Project Management and the Challenge of Learning and Competency Development, *Bartlett School of Construction and Project Management Research Papers*, University College London, 2004.