CONSTRAINT-GUIDED ENTERPRISE PORTALS

Christopher J. Hogger, Frank R. Kriwaczek
Department of Computing, Imperial College London, London SW7 2AZ, United Kingdom

Keywords: enterprise portal, finite-domain constraints, multi-agent system, plan revision, role management.

Abstract: It is shown how an enterprise portal, supporting a community of users discharging roles expressed as combinations of plans and constraints, can be usefully guided by a constraint processor. In particular, constraint logic programming on finite domains provides the users with useful insights regarding their possible work schedules. Constraints assist also in shaping the electronic artefacts created and transmitted by the users. The implementation is supported by mechanisms for assigning and updating roles and for assisting the search for remedies in the case of constraint failure

1 INTRODUCTION

This paper describes recent work in the *e-DoC Project* which investigates the applicability of computational logic to enterprise portal architectures. Its main testbed exemplar is an academic department of computing. It shares some aims of the Java Architectures Special Interest Group whose open-source *uPortal* is used in several universities (Gleason, 2000). Our interest, however, is in exploiting the power of logic programming and finite-domain constraints to imbue portals with intelligence and flexibility. Portals able to acquire, sift and interpret knowledge can construct and self-embed new tools into the portal interface (McCallum, 2000; Hogger, 2003) or re-structure that knowledge to suit their users' interests, as does the ontology-driven *KA2* system (Staab, 2000). Portals able to control the logical cohesion of users' activities enhance the collaborative functions of the enterprise (Ahmad, 2001).

We have built and implemented a conceptual model that uses *CLP(FD)—constraint logic programming over finite domains*—to guide and regulate the actions of a community of portal users. As they follow their various individual plans, the constraint processor helps determine their possible work schedules and the shaping of the electronic artefacts that they create and share. This form of constraint processing can contribute significantly to the development of predictive and anticipatory enterprises, owing to its generality of application and the powerful algorithms on which it relies.

The model employs formulations of user roles comprising plans mediated by constraints. It is designed to be simple, transparent, of general applicability and uncluttered by implementation details. It can be used either as a standalone application, driving its own web-based portal interface, or as a tool embedded in an existing portal. It can be operated in various modes according to whether constraints are evaluated lazily, on-the-fly or eagerly.

Section 2 describes, with an example, our representation of a role. Section 3 explains how the implementation interprets plans, manages artefacts and evaluates constraints. Section 4 presents a more detailed example, in which several users with inter-dependent roles experience but recover from a constraint failure. Section 5 discusses the work undertaken so far and the further work required.

2 ROLE STRUCTURE

The core construct in the model is a user's *role*, which has two main components: a procedural component — the *plan* — expresses *actions* that a role-holder (user) intends to perform, whilst the declarative component — a *constraint set* — expresses requirements upon the timing and the consequences of those actions. Together, they capture the essential logic of the role; aggregating such components over all users gives the essential logic of the enterprise. The procedural commitments in the plans can be viewed as compiled outcomes of

J. Hogger C. and R. Kriwaczek F. (2004).
CONSTRAINT-GUIDED ENTERPRISE PORTALS.
In *Proceedings of the Sixth International Conference on Enterprise Information Systems*, pages 411-418
DOI: 10.5220/0002599804110418
Copyright © SciTePress

some portion of that logic, leaving the residual portion in the form of declarative constraints.

The actions can be designed for any purpose, but in the *e-DoC* project they deal chiefly with the manipulation of electronic *artefacts*, such as text documents, emails and database tables. These reside variously in private (local) user workspaces or in public (enterprise-wide) repositories.

An important species of artefact is the *script*, which describes a role or role update. It enables users to create and transmit roles just as they can with other kinds of artefact.

Figure 1 outlines a script for a user ttr working as a timetabler, as viewed in the portal interface. Written in a logic programming syntax, it is arranged into several groups: ontology declarations, action declarations, constraint declarations and *CLP(FD)* programs. This particular script describes how ttr assembles a timetable artefact named timtab from a set of requests and stores it in a database tt_db, whilst aiming to satisfy various constraints upon timing and attribute values.

2.1 Ontology Declarations

The arguments timtab, requests, st1, et1, etc. occurring in Figure 1 are examples of *ontological variables*. The script for a role R must declare each such variable within some *ontology declaration* of the form

ontology(R, *origin*, *list_of_variables*).

This declares that R makes use of the variables in the given list, and also declares their origin. If the origin is own then R devised them. If it is public then the variables are enterprise-wide, that is, accessible to all users. If it is the identifier of some other user R2 then the variables were devised by R2. Any number of such declarations can occur in R's script.

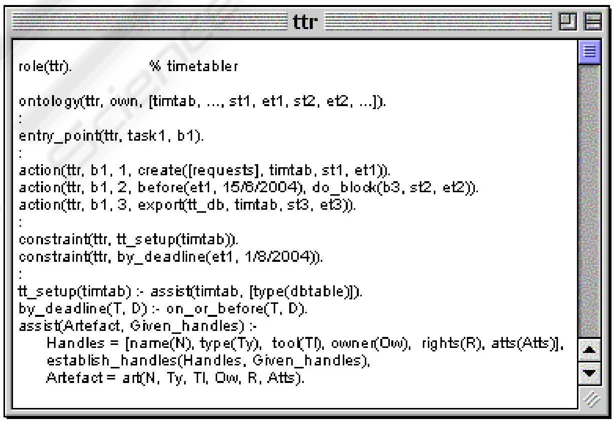
Ontological variables mostly denote time-points, artefacts and artefact attributes. Each one is named by a constant, e.g. st1, to ensure that its occurrences in separate script statements share the same denotation. (By contrast, logical variables in separate script statements would be unrelated, whether quantified or not.) In the implementation, each declared name *var* is mapped to an underlying logical variable *Var* that is uniformly associated with all occurrences of *var* in any script, so that correct referencing is established throughout all scripts. Then, if *Var* is subsequently bound to some value *val*, *var* also is effectively bound to it by making a binding-pair (*var*, *val*), as in (et1, 7/6/2004) or (name, 'c:/MyDocs/timtabs/timtabs-2004'). Typically, such a binding arises either as a direct response to a user selection (e.g. from a menu) in the portal interface or as a result of constraint evaluation.

2.2 Plan Structure

A role typically contains a number of separate, though possibly related, *tasks*, each one consisting of *action declarations* arranged into *blocks*. The block structure serves only to facilitate the expression of some simple control constructs. Given this, we can view each task as a program, and the aggregate of all these programs as the overall plan for the role.

For each task in a role, the script must contain a unique *task entry-point declaration* of the form

entry_point(*role_id, task_id, block_id*).



```
role(ttr).            % timetabler

ontology(ttr, own, [timtab, ..., st1, et1, st2, et2, ...]).
:
entry_point(ttr, task1, b1).
:
action(ttr, b1, 1, create([requests], timtab, st1, et1)).
action(ttr, b1, 2, before(et1, 15/6/2004), do_block(b3, st2, et2)).
action(ttr, b1, 3, export(tt_db, timtab, st3, et3)).
:
constraint(ttr, tt_setup(timtab)).
constraint(ttr, by_deadline(et1, 1/8/2004)).
:
tt_setup(timtab) :- assist(timtab, [type(dbtable)]).
by_deadline(T, D) :- on_or_before(T, D).
assist(Artefact, Given_handles) :-
     Handles = [name(N), type(Ty),  tool(Tl), owner(Ow),  rights(R), atts(Atts)],
     establish_handles(Handles, Given_handles),
     Artefact = art(N, Ty, Tl, Ow, R, Atts).
```

Figure 1: Outline view of the timetabler's script

Its purpose is to identify some particular block as that in which execution of the task is to begin. In Figure 1 the entry-point for a task task1 is declared to be the block b1. All the action declarations appearing there belong to this block, although one refers to some other block b3 not revealed in the figure.

Each action declaration has the basic form

```
action(role_id, block_id, index, action_predicate).
```

which identifies the role to which it belongs, the block that contains it, its index within that block and its *action predicate*.

Every action predicate contains two time-point arguments *st* and *et* denoting the action's start-time and end-time. The species of action intended is determined by the predicate symbol, most often create, import or export. These three are used for creating and transmitting artefacts. A slightly more elaborate form of declaration is

```
action(role_id, block_id, index, pre, action_predicate).
```

in which pre is a precondition The action declaration having index 2 in Figure 1 contains a precondition requiring that the previous action shall have been completed by June 15th 2004. Only if this holds will the (control) action do_block(...) be executed, in which event control will branch to some other block b3 within task1. Other kinds of control action are available to specify conditional branching and while-iteration.

Although such action-based plans have a procedural character, they can be viewed as partial solutions of declarative constraints relating their start and end times, as explained in the next subsection.

2.3 Constraint Declarations

Each user's role is generally subject to constraints expressing requirements. Some of these constraints may relate his own variables to other users' variables. Communal cohesion therefore requires that, however constraints be devised and whatever they may constrain, in operation they must be treated as applying globally across the enterprise.

The expression of a constraint consists of two parts. One is a *call* to a relation required on some tuple of ontological variables and/or constants. The other is a program for evaluating that call. The call is declared in a *constraint declaration* of the form

constraint(*origin*, *call*)

where *origin* identifies the user who devised it. Figure 1 shows two such declarations, one constraining attributes of the timtab artefact and the other constraining the completion-time et1 of task1's first action. In the lower region of the script are various *CLP(FD)* programs defining the called relations. In practice, many such programs only test inequalities of time-point variables, but constraint-supporting programs may in principle be arbitrarily complex.

Figure 1 is a limited view of the script, revealing only the constraints devised by ttr. In the full portal interface a user R can apply options to a script window to reveal just those constraints devised by R or all those that depend upon R's own variables or all constraints in the enterprise. The script itself consists, technically, of the ontologies and plans used by R together with a pointer to the enterprise's current constraint set.

Besides the user-declared constraints, the system uses further constraints implicit in the 'physics' of the model. Thus, the end-time of an action must succeed its start-time; its start-time must succeed the end-time of any action with lower index in the same block; an artefact cannot be acted upon until it has been created. These simple conditions are established automatically as constraints whenever a plan is created or revised. Expression of intended control flow in the plan would be unnecessary if the user declared all these constraints explicitly and allowed the constraint engine to determine possible schedules for the actions. However, expressing some control flow avoids the need to declare large numbers of these physics-based constraints and enables the user to commit *a priori* to particular sets of scheduling solutions.

3 OPERATIONAL MODEL

Our implementation deploys two closely-coupled engines, one being a *plan interpreter* and the other a *constraint evaluator*. These, together with the users' interactions through their portal interfaces, ultimately determine the bindings made to the ontological variables, and hence the character and scheduling of the artefacts manipulated. Besides this, there are also system components for managing artefacts, script assignments and constraint failures.

3.1 Plan Interpreter

The function of the *Plan Interpreter* is to exhibit to the users, via selected views in their interfaces, their states of progress through their plans. For instance, a *task view* window highlights actions recognized by the interpreter as ready to begin or (if begun) waiting to be completed.

Users pursue their tasks concurrently, following initiation from the declared entry-points. Within any block, actions are required to be performed in order of increasing index. A group of actions may, however, have a common index, indicating that they can be performed concurrently, that is, with no plan-imposed ordering upon their time-points.

Once an action has been indicated as ready to begin, the user can begin it at a time of his choosing. He commits to this by clicking in a displayed cell for the relevant start-time variable; the interpreter then binds the action's start-time argument *st* to the current time on the portal clock, and updates the interface accordingly.

Work entailed in performing the action itself is invisible to the interpreter except to the extent that it may bind ontological variables; in effect, it is treated as off-line activity. For example, activity to create a text document may involve the user entering textual content, but the details of that content are of no concern to the portal; however, activity that binds ontological variables denoting the document's attributes is recognized and acted upon by the portal.

When the user considers that an action has been completed, he formally signals this by another click in the interface; this binds the action's end-time *et* to the clock value and again updates the interface and the state of the interpreter.

In Section 4 we will present a more detailed view of these interface episodes and their consequences, using a multi-user example. That example will also show how the plan's execution affects, and is affected by, the concurrent operations of the constraint evaluator.

3.2 Artefact Manager

A new artefact is produced whenever a user performs a create action. This takes the form

create(*As, A, st, et*)

Here, *A* names the artefact to be created. The argument *As* supplies the names of any other artefacts that the user expects to use in preparing *A*. The plan interpreter requires these to be already residing in the user's local workspace when the create action is begun; otherwise, it suspends at that point in the plan and informs the user of the reason.

The first action in Figure 1 requires ttr to create an artefact named timtab that depends on the availability of another named requests. The latter could have been imported by an earlier action in this task, but our present example assumes it to be created or imported by some other task in ttr's plan.

The portal's *Artefact Manager* constructs abstract terms representing such artefacts, and maintains pointers between them and their concrete instances. Each abstract artefact term has the form

art(*Name, Type, CrTool, Owner, Rights, Atts*)

whose arguments hold the primary attributes — the artefact's name (with full location), type, creator tool, owner, the owner's rights and a set *Atts* of any secondary attributes appropriate to the type. The main types supported in our implementation are script, txt (text document), email and dbtable (database table). Examples of artefact terms are

```
art('c:/MyDocs/ timtabs/timtabs-2004', dbtable,
    'MSExcel', ttr, [read, write], atts(schema1, view1))

art('c:/MyDocs/ notice', txt, notepad, ttr, [read], null)
```

The attribute values are determined, in general, partly by the user in the course of creating the artefacts and partly by the evaluation of constraints upon ontological variables. Knowing these values, the artefact manager invokes appropriate mechanisms in the host system to store each artefact in an appropriate workspace substructure, integrating it with the correct infrastructure for its use. A text document, for instance, will be stored in some standard file-oriented directory, whilst an email will be stored in some email-client mailbox.

Once created, copies of artefacts may be transmitted between local workspaces and communal repositories. An action import(*Rep, A, st, et*) requires the user to import into his workspace a copy of artefact *A* from repository *Rep*. An action export(*Rep, A, st, et*) transmits a copy of *A* in the converse direction. The plan interpreter will suspend either action if *A* is not currently available in its specified source.

3.3 Constraint Evaluator

As users pursue their plans, the *Constraint Evaluator* attempts to solve conjointly the calls cited in all constraint declarations, testing or determining values for the ontological variables occurring as arguments within them. It behaves effectively as an assistant, displaying to the users their future possibilities and alerting them to constraint failure.

The evaluator can be operated under various regimes controlling the timing of constraint-checking. It is possible to simulate, prior to real-time user activity, the future performing of plans: eager constraint-checking can then compute, for instance, feasible schedules (if any) that the users might adhere to in practice. Another possibility is that users perform their plans but with inessential constraint-checking deferred until all plans have been completed; in that case they discover only afterwards whether their activities satisfied their intentions. The standard regime, however, checks constraints incrementally in step with the plan interpreter, informing the users of their remaining scheduling opportunities and readying those tools they will need to manipulate their artefacts. Under this regime the evaluation of temporal constraints is similar to conventional critical path analysis.

The *CLP(FD)* formalism has the flexibility to support these various regimes. For an ontological variable that is declared (or is by default) a *CLP domain variable*, the evaluator holds at any instant a current subset of that domain as the potential solution-set for that variable, as determined by the given constraint-defining programs and the current solution-sets of any other variables upon which it depends. User actions in the portal interface that themselves bind any variables cause the algorithms to restrict further the solution-sets. Restriction also arises from each advance of the portal clock, which deletes the previous clock value from the domains of all temporal variables. If the domain of any variable becomes void this implies that the constraints upon that variable have no feasible solution. The solvability or otherwise of the constraints is independent of when the constraints are checked.

In our implementation *CLP(FD)* is applied chiefly to the temporal variables, though it is applicable in principle to finite domains of artefact attribute values. In practice, however, we find it is usually more practical to deal with these by a form of *Query-the-User* driven by an interactive conventional logic program. This program is designed to evaluate a call of the form

assist(*A*, *Given_handles*)

which associates with artefact *A* a set *Given_handles* specifying already-chosen attribute values for *A*. This call may be placed in the body of any user constraint program. When it is executed, the user is invited by dialogues in the portal interface to choose values for any so-far-undetermined attributes. The net result of this interaction is to bind *A* to some ground term art(...) that the *Artefact Manager* then associates with the concrete stored artefact.

Figure 1 contains an example of this call being used to help ttr complete the attribute determination for timtab, whose type has been already prescribed in the given handles as dbtable. The assist program will invite ttr to choose the full path-name, the creator application tool and so forth. Although viewable in ttr's script window, this program is not devised by ttr but is instead one of many supportive resources shared by all users.

3.4 Script Manager

Like any other artefact, a script can be created, transmitted or modified by a user. A user performing a role has a copy of its associated script in his own local workspace and can view it in his interface.

Although some roles may be entirely self-defined, our system enables a user U1, having appropriate authority, to assign a role (or a role-update) R to another user U2. (We impose a simplifying restriction whereby no user has more than one role.) This is achieved by U1 performing within his own plan an action of the form

assign(*R*, *U2*, *st*, *et*)

This assumes that R is a script artefact already residing in the workspace of U1. If U2 currently has no role then the assign action invokes a *Script Manager* to create for U2 a new interface whose workspace contains a copy of R. Alternatively, if U2 already has a role then R is some update to it, in which case the *Script Manager* copies R to U2's existing workspace and alerts U2 to the need to incorporate the update into his existing script.

A key presumption is that U2 cannot reject the requirements in any script content assigned to him, but may further detail it on his own initiative and may later (if suitably authorized) re-assign it to someone else. So in general the script for a user may contain accumulated contributions from other users besides himself. With these arrangements we can model the circumstance whereby requirements posed by some level of authority can be elaborated but not revoked by lower levels of authority.

3.5 Failure Manager

It is easy to detect when a constraint violation occurs but by no means always easy to determine the best way to explain or resolve it. A *CLP(FD)* failure occurs when some domain variable's feasible solution-set is empty. Unless the constraint engine holds an accessible history of the internal constraint simplifications it has made in the course of reaching this situation, there is relatively little information on which to isolate a specific cause of the failure. In particular, the variable(s) now found to have no solution may be common to multiple source constraints variously sensitive to multiple roles. Thus, the responsibility for the failure may in general be communal, rather than individual.

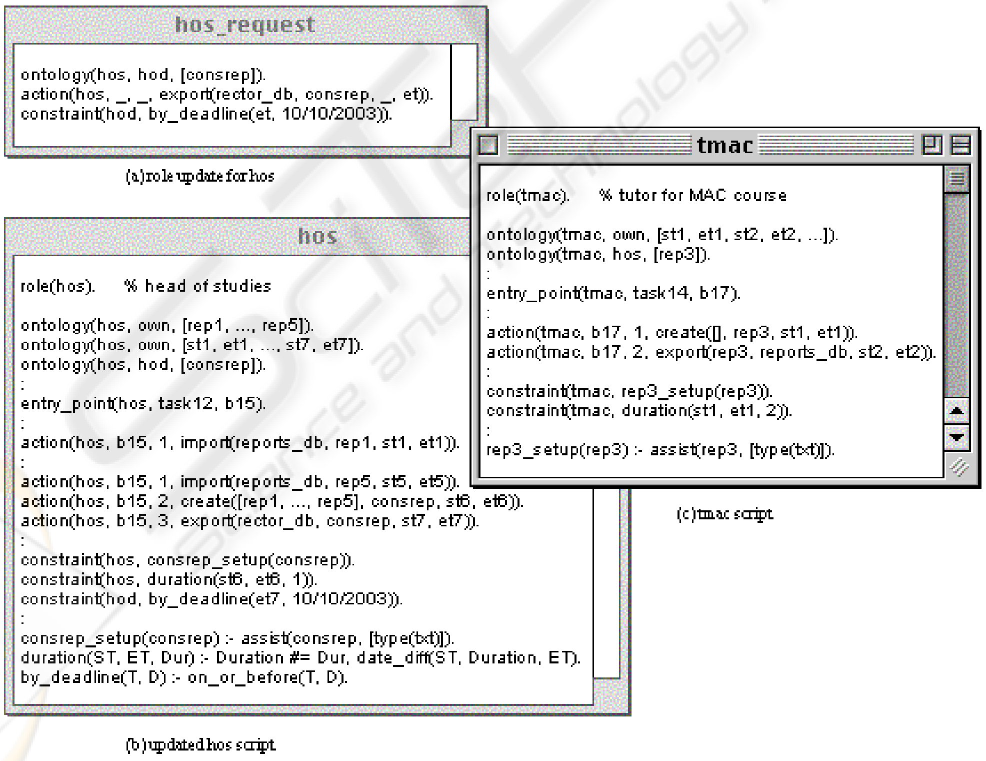
Sometimes it may be possible to detect that a particular individual's recent activity, such as starting an action too soon or choosing an artefact attribute inappropriately, is the sole cause of the problem. If they can backtrack on that action and try again then the problem may be remedied, demanding no revision of plans or constraints. It will, however, require garbage-collecting artefact operations undertaken by them prior to backtracking.

Otherwise, the *Failure Manager* tracks down those constraints that impact, directly or indirectly, upon the problematic variables, and further identifies their originators, that is, those who have the authority to revise that material. They are alerted to their need to review, in the light of the current failure, the requirements they posed and to agree upon suitable revisions sufficient to alleviate the failure — for instance, by extending deadlines. The revisions are conveyed to the affected role-holders for assimilation into their scripts.

4 CASE STUDY

This example illustrates an episode involving academic role-holders in a college's Computing department. The college's Rector wishes to be supplied, by October 10th 2003, with a report on this department's student drop-out rate from its Head (hod). The latter requests his Head of Studies (hos) to supply that report directly to the Rector. He poses this request by assigning to her the role update shown in Figure 2a, whose anonymous slots ("_") she instantiates when assimilating this material into her role.

The department runs five degree courses each with its own tutor, and hos requests these tutors to supply reports on drop-out rates for their courses but sets no deadlines for them. She believes it will take her just 1 day to consolidate their reports into her full report consrep for the Rector. Each tutor estimates of how long it will take to produce his report. The tutor (tmac) for the MSc in Advanced Computing course estimates it will take 2 days to create his report rep3.



```
ontology(hos, hod, [consrep]).
action(hos, _, _, export(rector_db, consrep, _, et)).
constraint(hod, by_deadline(et, 10/10/2003)).
```

(a) role update for hos

```
role(hos).     % head of studies

ontology(hos, own, [rep1, ..., rep5]).
ontology(hos, own, [st1, et1, ..., st7, et7]).
ontology(hos, hod, [consrep]).
:
entry_point(hos, task12, b15).
:
action(hos, b15, 1, import(reports_db, rep1, st1, et1)).
:
action(hos, b15, 1, import(reports_db, rep5, st5, et5)).
action(hos, b15, 2, create([rep1, ..., rep5], consrep, st6, et6)).
action(hos, b15, 3, export(rector_db, consrep, st7, et7)).
:
constraint(hos, consrep_setup(consrep)).
constraint(hos, duration(st6, et6, 1)).
constraint(hod, by_deadline(et7, 10/10/2003)).
:
consrep_setup(consrep) :- assist(consrep, [type(txt)]).
duration(ST, ET, Dur) :- Duration #= Dur, date_diff(ST, Duration, ET).
by_deadline(T, D) :- on_or_before(T, D).
```

(b) updated hos script

```
role(tmac).     % tutor for MAC course

ontology(tmac, own, [st1, et1, st2, et2, ...]).
ontology(tmac, hos, [rep3]).
:
entry_point(tmac, task14, b17).
:
action(tmac, b17, 1, create([], rep3, st1, et1)).
action(tmac, b17, 2, export(rep3, reports_db, st2, et2)).
:
constraint(tmac, rep3_setup(rep3)).
constraint(tmac, duration(st1, et1, 2)).
:
rep3_setup(rep3) :- assist(rep3, [type(txt)]).
```

(c) tmac script

Figure 2: Role scripts in the computing department

Figures 2b and 2c show the scripts of hos and tmac after assimilating hos's requests. In her script, hos has put the assigned export action into a new task12 and can view its deadline constraint in her script window. She has substituted her own variable et7 in place of the et variable used in the role-update request that hod assigned to her. She has elaborated that request with constraints to help set up consrep and to restrict the duration of its creation process.

Figure 3a shows snapshots from the interface of tmac taken on October 6th, 7th and 8th. Each of his variables has a row of cells variously filled grey or white. White cells indicate the current domain, i.e., the time-points (dates) that the constraint engine has calculated as admissible solutions. So on the 6th tmac sees that he can start his create action only on that day or the next, and complete it only on the 8th or 9th. However, his teaching obligations prevent him creating his report that day. By the 7th the portal clock has advanced one day and so eliminated the 6th from the domain of st1, whose value tmac can now choose only as the 7th. Unfortunately he misses that opportunity also, so that by October 8th no choice at all remains to him. The domains of st1 and et1 are now void, signifying constraint failure. Their cells have turned black, signifying that they are implicated in that failure.

The *Failure Manager* must now locate the failure. It identifies the constraint calls referring to st1 and et1 and constructs the transitive closure of this set under the relation of two calls sharing a domain variable. It thereby traces the dependencies between variables. It extracts the originators of the calls in this closure, being those in whose hands it lies to remedy the failure. Here, they are hod, hos, tmac and the other tutors.

tmac may reduce his report-creation's expected duration, or hos may decide she can complete her consolidation on the same day she begins it, but her deadline is assumed to be rigid. It is decided that tmac must create rep3 faster than he had estimated. He accordingly revises his own call to

```
constraint(tmac, duration(st1, et1, 1)).
```

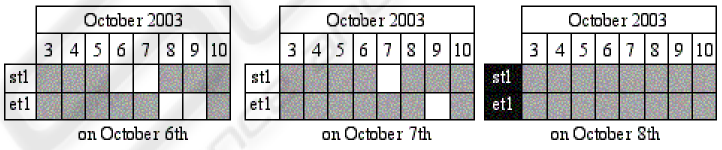
so that his interface on the 8th changes to that in Figure 3b. He clicks in st1's white cell for the 8th and begins working on his report. He finishes it the next day and so clicks in et1's white cell for the 9th, then immediately exports the report. This leaves hos enough time to consolidate all five reports, provided the other tutors supply their reports on time too.

We showed how the constraint engine presents feasible schedules as users commit to time-points. However, the interface also supports a "what-if?" mode permitting experimentation with non-committed time-points to further explore the options.

The episode above need not have entailed a constraint failure, as tmac could foresee on October 6th his scope for completing rep3. The displayed domains of his variables enabled him to *anticipate* solutions. Moreover, because *all* solutions were visible, it was *predictable* that failure would occur if he did not begin his report before October 8th.

5 CONCLUSION

The model has applied experimentally in the *e-DoC* domain for simulating real roles among college staff, as a stand-alone portal. As a prototype under development, quantitative evaluation of it in a context of serious scale is not yet available.



	October 2003							
	3	4	5	6	7	8	9	10
st1	▨	▨	▨	□	□	▨	▨	▨
et1	▨	▨	▨	▨	▨	□	□	▨

on October 6th

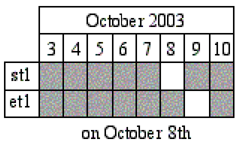
	October 2003							
	3	4	5	6	7	8	9	10
st1	▨	▨	▨	▨	□	▨	▨	▨
et1	▨	▨	▨	▨	▨	▨	□	▨

on October 7th

	October 2003							
	3	4	5	6	7	8	9	10
st1	▨	▨	▨	▨	▨	▨	▨	▨
et1	▨	▨	▨	▨	▨	▨	▨	▨

on October 8th

Figure 3a. Interface snapshots for "tmac".



	October 2003							
	3	4	5	6	7	8	9	10
st1	▨	▨	▨	▨	▨	□	▨	▨
et1	▨	▨	▨	▨	▨	▨	□	▨

on October 8th

Figure 3: Updated interface for the "tmac"

We employ a *Sicstus CLP(FD)* engine fast enough to solve realistic constraints in the intervals separating user actions. It uses the *Pillow* package to construct web pages displaying the users' interfaces. *CLP(FD)* was chosen for its flexibility in knowledge representation, its support of user-defined evaluation schemes and, in particular, its simplicity compared with other proprietary systems such as ILOG Rules (ILOG, 2002).

In our academic domain we use finite-domains mainly to compute temporal variables. However, actions and artefacts can have attributes measuring resources such as diskspace allocations or administrative overheads. These can be processed likewise by *CLP(FD)* if they are discrete-valued. For other domains, action and artefact types can be implemented without altering the core model. In particular, domains concerned with assembling industrial artefacts, with constraints upon component supply and workflow, can be represented.

A key priority is to enhance the intelligence of the *Failure Manager* which currently identifies only source constraints upon void-domain variables. It cannot trace back its evaluation to identify where failure-disposing events occurred. Standard logic programming engines can deliver traces showing which instances of source statements led to a computed outcome, so facilitating debugging and showing how the computation might be backtracked to try other instances. The constraints within a *CLP(FD)* engine, by contrast, are some arbitrary transformation of the source ones and may have attained that state using a mix of several algorithms. Using the record of users' instantiations in their plans could compensate to some degree for these difficulties, whilst the treatment of purely temporal failures could usefully exploit scheduling analysis. Recent development of abductive constraint solvers such as *A-system* (Kakas, 2001), which is itself written in *Sicstus Prolog*, may point to ways of enhancing the ability of the *Failure Manager* to further localize the origins of constraint failure and to infer and rank suitable constraint revisions for the relevant originators, assisting their collaborative recovery. These aims share common ground with work on constrained workflow management, as in (Hwang, 2003) and (Wainer, 2003a) and (Wainer 2003b) who variously employ procedural constraints or standard logic programming integrity constraints but not *CLP(FD)*. We also intend partly to re-implement our model in the multi-agent formalism *Go!* (Clark, 2003) whose multi-threaded communication inherently reduces the number of temporal constraints required.

REFERENCES

Ahmad M.S., Hogger C.J. and Kriwaczek F.R., 2001. Implementing a Collaborative Agent System using Prolog. In *ICIMu-2001, Int. Conf. on Information Technology and Multimedia*, Kuala Lumpur.

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

Hogger C.J. and Kriwaczek F.R., 2003. Abstracting Wizards from Portal Observations. In *WITSE-2003, Workshop on Intelligent Technologies for Software Engineering, 9th European Software Engineering Conference and 11th ACM SIGSOFT Symposium on the Foundations of Software Engineering*, Helsinki.

McCallum A.K., Nigam K., Rennie J. and Seymore K., 2000. Automating the Construction of Internet Portals with Machine Learning. *Information Retrieval*, Vol. 3, Issue 2.

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

Hwang G.-H., Lee Y.C. and Wu B.-Y., 2003. A New Language to Support Flexible Failure Recovery for Workflow Management Systems. In *CRIWG-2003, 9th International Conference on Groupware*, Grenoble.

Wainer J. and Bezerra F., 2003a. Constraint-based Flexible Workflows. In *CRIWG-2003, 9th International Conference on Groupware*, Grenoble.

Wainer J., Barthelmess P. and Kumar A., 2003b. W-R-BAC - A Workflow Security Model Incorporating Controlled Overriding of Constraints. To appear in *Int. Journal of Cooperative Information Systems.*

Clark K.L. and McCabe F.G., 2003. Go! for Multi-Threaded Deliberative Agents. In *DALT-03, AAMAS Workshop on Declarative Agent Languages and Technologies*, Melbourne.

ILOG, Inc., 2002. Business Rules: ILOG Technical White Paper. At www.ilog.com.

Kakas, A.C. and Van Nuffelen, B., 2001. A-system: Programming with Abduction. In *Logic Programming and Nonmonotonic Reasoning*. Lecture Notes in Artificial Intelligence, Vol. 2173, Springer Verlag.