Tucupi: a flexible workflow system based on overridable constraints

Jacques Wainer
Institute of Computing
UNICAMP, Brazil
wainer@ic.unicamp.br

Fabio Bezerra
Centro de Ciencias Exatas e Tecnologia
Universidade da Amazonia, Brazil
ra012111@ic.unicamp.br

Paulo Barthelmess
Department of Computer Science
University of Colorado, USA
barthelm@colorado.edu

ABSTRACT

This work presents the idea and a prototype of workflow systems whose definition is based on constraints. The flexibility is reached through the less rigid definition of workflow definitions - the workflow is defined as a set of pre and post conditions of activities, which are selected dynamically as the process instance unfolds. The workflow system besides dispatching activities that have all their preconditions fulfilled to be executed, also helps users to decide which activity to chose through what if scenarios. The system also includes an access control model which not only represents which users have the authority to chose and execute the activities but also the authority to override the constraints. In particular, overriding constraints is itself an activity and thus may have pre and post conditions defined in other constraints. The paper present Tucupi, a prototype of such constraint based WFMS.

Categories and Subject Descriptors

H.4.1 [**Office Automation**]: Workflow management

Keywords

Workflow, Fexibility, Constraints, RBAC

1. INTRODUCTION

Workflow management systems (or WFMS) are systems that allow for the specification, execution, and monitoring of business processes. In execution, an instance of a process, or a **case** goes through a series of **activities** performed either automatically or by people, according to a pre-specified **process definition**. In usual business domains the process definition is created well in advance of its use, and although most process definition formalisms allow for processes that have some flexibility, this flexibility is defined in advance as conditional execution of different (pre-defined) paths based on the case data.

In health care processes, as well as others, such as software design, there is the need for what we call as **partial workflows**, that is, workflows that are only partially defined and are conditional. When a patient is admitted to a hospital, one does not have information to define, at that time, all the activities that will be performed to/on behalf of the patient. Clearly, after a patient is discharged, one can look back and see the patient's stay as the execution of a workflow: activities ordered in time were performed to/on behalf of the patient, but at no moment there was a template (or a workflow specification) of which the patient's case was an instance.

Patient care workflows can be characterized as **dynamic planning** workflows. The particularities of a case as it unfolds defines which activities should be performed in the future. On the other hand, whoever is in control of the case does not have total freedom to decide which activities must be performed and which should not be performed. There are **rules** or **constraints** that must be followed, which impose the execution of unrequested activities (pre-conditions), or that forbid the execution of some activities. Furthermore dynamic planning workflows should help the decision maker to select which activity should be scheduled, specially given that different choices of activities may have to satisfy different constraint and thus may have to be scheduled in different ways.

Finally, it may be the case that some of these constraints, although reasonable in general, should not be applied to a particular case. Thus it is desirable that some of the constraints could be overridable by people with sufficient authorization.

The paper is organized as follows: section 3 presents the constraint specification of workflow definition; in section 4 we present the implementation of our proposed ideas; in section 5 we present a discussion about related work; in section 6 we conclude this paper with a brief analysis and we propose some future work that can be done as extension to this paper.

2. SUPPORT FOR PLANNING AND EXECUTION

Workflow systems, as used in business environments, usually only allow for well defined processes. The workflow definition for such processes is total and complete: the workflow

SAC '04, March 14-17 2004, Nicosia, Cyprus

defines all the activities that will be executed in a case and in which order. Once activity A has finished the system can compute exactly which activities must start then. In such defined processes, the workflow system works as a **dispatcher** – at the end of an activity it computes which activities become enabled and dispatch the case to the executors of such activities. The control of what will be executed is in the WFMS, and some control of when the activity will start (once enabled) is left to the executors themselves (if the WFMS uses the concept of inboxes).

2.1 Workflow as a helper

In domains with less defined processes, usually there is a person, or persons, which we will call the **controller** who decides which activities must be performed to the case. We call those activities that the controller explicitly decides to schedule a **target** activity. In the hospital example, a physician (the controller) decides that a surgical intervention (the target activity) is needed, and thus schedule it.

In the workflow as a helper scenario, the workflow helps the controller decide on the implications of scheduling the target activity. They include the need to execute activities both before and after the target activity (which we call **forced** activities). The controller may then decide that some of the forced activities cannot or should not be executed, and thus may query the workflow/helper if she has enough authority to override some of the constraints, or who has it.

Constraints are used to represent the rules that create the forced activities - a constraint can specify that if a target activity A is to be executed, then B must happen before it (a precondition) and C must happen after it (a post-condition). Constraints are a particular useful representation for this problem because of its declarative nature, that is, they express what should happen in a way that is independent on how this information is used. The rule above could be represented as an ECA (event-condition-action) rule, but such representation is more suited for forward reasoning purposes - the post condition for A can be represented as an ECA rule (in the event of A's termination, under all conditions, activate C), but a rule that specifies that B is a pre-condition is more cumbersome. Furthermore such ECA rules are less convenient when reasoning in a what-if scenario which is the central aspect of a workflow as a helper.

2.2 Workflow as helper/dispatcher

The Tucupi architecture we developed is really a helper/dispatcher system. Each workflow case has a set of **future** activities, that is activities that must be executed for this case, either because such activity was explicitly **scheduled** by the controller or because they are forced pre-and post-conditions of those scheduled ones (and the other future activities).

All future activities that have no open preconditions (described below) are considered **enabled**, that is ready to be executed. The Tucupi as a dispatcher places all enabled activities in the appropriate user's inboxes. When the server receives information that an activity is finished, all future activities that depended only on that one become enabled and are dispatched to their executors.

New future activities are created by the actions of the controller. She may schedule a new target activity, which likely create new forced activities, which themselves may also spawn forced activities. Future activities may also change because of constraints that are overridden: an overridden constraint does not spawn the forced activities an thus the set of future activities will likely be reduced.

2.3 WRBAC: a support component to overriding of constraints

WRBAC [11] is an extension of the Role-Based Access Control (RBAC [8, 5, 4]) mechanism to workflows. RBAC defines the classes *user*, *role*, *privilege* and relations among them: *can-play* $\subset$ *user* $\times$ *role*, and *hold* $\subseteq$ *roles* $\times$ *privilege*. A user's privileges are computed indirectly through the role the user can play (or is playing depending on the implementation). RBAC seems to be a balance between the flexibility of a discretionary access control mechanism and the organizational-wide control but lack of flexibility provided by a mandatory access control mechanism. At the organizational level, policies of privileges are defined through the hold relation: what are the privileges that a database administrator should have, and what are the privileges that a surgeon should have. At a more local level users are assigned to one or more roles, thus Mary is a surgeon, which allows her some privileges, but she may also be a database administrator, which endow her with a different set of rights.

To start, WRBAC makes the connection between the concept of role in workflows (an indirect way of specifying the possible executors of an activity) and roles in RBAC. Also, the right to execute an activity is one of the privileges in the RBAC hierarchy. But more importantly, WRBAC adds to RBAC the concepts of a *workflow case*. Using the concept of a workflow case it is possible to model *dynamic* rights and non-rights in extension to the RBAC *static* modeling of rights. Thus, by using constrains in addition to the RBAC relations one can model rules such as: the same person that received a complain from a customer must answer the customer (binding of duties), the execution of a request for reimbursement and the approval of the request must be executed by different people (separation of duties), and if person A approved B's request then B cannot be the one to approve A's request (mutual separation of duties), and so on. We call the WRBAC constraint as **executor constraints**, to distinguish them from the order constraints which are the main topic of this paper.

The inclusion of the concept of case in WRBAC solves a problem of specifying dynamic constraints in standard RBAC. In RBAC dynamic constraints are modeled through an entity called session, which is a temporally limited binding of a user to a role, modeled after the idea of "login on" to a data base. With sessions, one can model such constraints as "a same person cannot be the pilot and the navigator on the same flight" by stating that there can be no sessions binding the same user to both the pilot and the navigator roles. But sessions do not capture the needs of workflow constraints: in the separation of duties example above, users can be at the same time both the requester and the approver of some request. What is important is not the temporal dimension but the case dimension: the same person cannot be the approver of his own request.

The WRBAC allows queries such as *who?(A,C)* that is, who are the users that can execute activity *A*, for case *C*. The WRBAC will verify which users have static rights to execute *A*, and remove from that set all users that violate some dynamic constraint for case *C*. WRBAC answers to this query with a set of users that can perform the activity,

but to whom the task is assigned is a decision made outside the WRBAC. The workflow system may offer the activity to all potential executors and as soon as one accept the task, the offer is withdraw from the other users. Or the workflow system may keep a list of tasks attributed to each user and assign the activity to the user with the least load among the potential executers. The WRBAC must then be be informed on which user gets to execute the particular activity for a case.

3. CONSTRAINT-BASED WORKFLOW SPECIFICATION

We defined a restricted language to represent constraints. The language define both the preconditions and post conditions of a target activity.

The simplified version of a constraint is illustrated in the following example:

```
rule: c1
     target:  A
     precondition:  B
     precondition: C
     postcondition:  D
```

We call such rules above **order constraints**, to be distinguished from other constraints explained in the previous section.

The order constraint `c1` specifies that to perform the target activity A, activity B must have ended (`precondition: B`), activity C must also have ended, and activity D must happen after A ended (`postcondition:  D`).

The full expressivity of the constraint language is reached by adding the following constraints constructs:

- the precondition can be a disjunction of activities.
- also in precondition there can be a constraint that an activity should *not* have happen before the target activity.
- the postcondition can be a disjunction of activities.
- the construct `parcondition` (after parallel condition) states that an activity must happen before or after the execution of the target activity.

3.1 Formal definitions and algorithms

- For each defined activity A there must a privilege *execute(A)* in the WRBAC hierarchy which represents the privilege to execute the activity.
- If A is a target activity, there must be a privilege *schedule(A)* to schedule the activity
- For each order constraint r there may be a privilege *override(r)* to override the constraint
- For each privilege *override(r)* there may be one or more rules in which *override(r)* is the target activity
- For each case c there is a set of users defined as the **controllers** of c.

For a constraint of the form:

```
rule: r
   target: A
   precondition:  B  or C
   precondition:  D
   postcondition: E  or F
   postcondition: G
   precondition:  not H
   parcondition: I
```

we will say that:

$$\begin{aligned} pre_r(A) &= \{choice(B,C), D\} \\ neg_r(A) &= \{H\} \\ pos_r(A) &= \{choice(E,F), G\} \\ par_r(A) &= \{I\} \end{aligned}$$

Notice that the formalism above defines a new activity *choice*(E,F), which is the activity of choosing between activities E and F. For the kind of applications we envision, for example health care, we feel that choosing between alternatives is a central task, unlikely to be executed automatically. Thus we decided that such activities (which are not represented in the WRBAC structures - there is no privilege to execute *choice*(E,F)) can only be executed by the controllers of the case.

The current state of the execution of the case is represented by three sets:

- *done*(c) is the set of activities that has already terminated for case c.
- *sched*(c) is the set of target activities already scheduled for case c.
- *over*(c) is the set of constraints that has been overridden for the case c. If R is the set of all order constraints, we will define $rules(c) = R - over(c)$ the set of constraints (rules) that are still valid for the case c.

For a current state of execution for a case c, the set of **future** activities (*fut*(c)) is defined by the following fixed point equation:

$$\begin{aligned} fut(c) = sched(c) \ \cup \ \{override(x) \mid x \in over(c)\} \ \cup \\ \bigcup_{r \in rules(c)} [pre_r(a) \cup par_r(a) \cup pos_r(a) \text{ for all } a \in fut(c)] \\ - \ done(c) \end{aligned}$$

The set of **enabled** activities *enab*(c), is the set of future activities whose preconditions have been satisfied, and is defined as:

$$enab(c) = \{a \in fut(c) \mid (\bigcup_{r \in rules(c)} pre_r(a)) \subseteq done(c)\}$$

A new target activity A can be **scheduled** by user U in the current execution context for case c if:

- user U has (through some role) the privilege *sched*(A)
- user U is a controller of the case c
- $\bigcup_{r \in rules(c)} neg_r(A) \cap done(c) = \emptyset$ that is, there is no negative preconditions for A among the activities that have already terminated.

If the activity is really scheduled, it is included into the set *sched*(c) and thus the current execution context of case c changes; the sets of future and enabled activities must be recomputed.

A rule r can be **overridden** for case c by user U if:

- user U has the right *override*(r)
- user U has the right *sched*(*override*(r))
- the target *override*(r) can be scheduled

If the rule is indeed overridden, then *override*(r) must be included in *sched*(c). If *override*(r) $\in$ *enab*(c) then the rule r can be overridden immediately and thus r must be included in *over*(c). This changes the current execution state of the case, and thus the set *fut* and *enab* must be recomputed.

4. THE TUCUPI SERVER

The core of the system is the Tucupi server, which answers queries and accept commands and updates from a client. The client is the software that interact with the users and in particular the client is responsible for authenticating users. At the time of this writing, the client has not been written or even specified.The server stores:

- the WRBAC hierarchy and relations
- the set of executor constraints
- a set of order constrains for different processes (workflow definitions)
- a relation between a case and the process the case is an instance of.
- the current state of each case.
- the set of controllers for each case.

The server answers to the following queries:

who(A,C) This query returns the users that can perform activity A for the case C. This is a pure WRBAC query;

done(C) This query returns the set of activities already executed for case C.

enabled(C) This query returns the list of enabled activities for case C.

enabled(C,R) This query returns the list of enabled activities for case C, if the rules in the list R where overridden.

not-ready(C) This query returns the list of activities that are not yet enabled for the case C. This is the set of future activities minus the set of enabled activities for case C.

not-ready(C,R) This query returns the list of activities that would not yet be enabled if the constraints in the list R where overridden.

what-if(A,C) This query returns the set of enabled and not ready activities that *will be added* to case C if target A would be scheduled. This list is annotated with the corresponding constraint identification that forces each of the activities.

what-if(A,C,R) Same as above but returns the set of enabled and not-ready activities to be added if target A would be scheduled and if all constraints in R were overridden.

The server accept the following updates:

started(U,A,C) This command informs the server that activity A for case C started now and that U is the executor of that activity. The executor information is also passed to the WRBAC component.

ended(A,C) This command informs that activity A for case C of process W ended at this time;

Finally the server accepts the commands below. Each command has an entry X, which indicates the user that is issuing the command (and whose identity has to be verified by the client). Upon receiving the command the server will first check if the user X has the right to execute the command, and if she does execute it. If X does not have the right the server answers with an error code.

create(X,C,W,U) create a case C as an instance of process W and define the users in the list U as the controllers of the case.

add(X,A,C) This command informs the server to schedule the target activity A for case C.

add-controller(X,C,U Adds users in the list U as controllers of the case C

chosen(X,Y,Z,C) If Y is the activity *choice(A,B)* for case C and it is enabled, then *choice(A,B)* is removed, and the activity Z $\in$ {A,B} is scheduled for case C.

override(X,C,R) Override the constraints in the list R for case C.

The Tucupi server implements both the workflow as a helper/dispatcher and the WRBAC component. Tucupi is implemented in Prolog, with an interface to a data-base to store the more volatile information (*started* and *ended* updates). The WRBAC hierarchies, the executor constraints, and the workflow constraints are represented as Prolog clauses that are loaded when the server starts. The server listen to a fixed IP port where queries, updates, and commands are accepted.

5. RELATED WORK

To the authors knowledge there is very few works on constraint or even logic based workflow representation formalisms. The closest work to this research is the one reported in [7, 6], in which a flexible workflow is defines as a set of workflow segments (in a graphic notation). A full workflow is constructed by "gluing" these workflow segments.

The work reported in [10] also has similarities with this one. There, a temporal logic is used to represent the temporal ordering among activities in a workflow. Our `precondition` construct is similar to the temporal operator **before** used in that work, it state that the precondition must happen before the target activity but not *immediately* before it. In that work, a non-monotonic logic is used to infer which activities are enabled and thus which ones can start immediately. In our work we use a simpler rule that all activities

that have all preconditions fulfilled are enabled and thus can start immediately.

The work reported in [9] share some of the goals Tucupi: a case oriented workflow-like support system. Their work (the FLOWer system) is centered on data constraints - preconditions of activities are the existence of value for certain data fields; this work is centered on activities. In FLOWer, the unit of overridability is an activity - an activity can be *skipped* if a user form the appropriate role requests it - in Tucupi, the unit is a constraint. FLOWer allows for multiple executions of an instance of activity through the *redo* mechanism, which is not allowed in Tucupi.

Paul Dourish et al. [3] describes *Freeflow*, a prototype workflow system that uses constraints as a workflow definition language. In Freeflow the user has total control over actions in system - the constraints are seen as guides to the user - the user is informed that scheduling some activity will violate some constraint but may schedule it anyway.

Models that tolerate inconsistencies such as divergence in order of execution, execution by some user different from the one anticipated and so on, are addressed e.g. by [2, 1]. In Cugola et al [2], a dynamically established *deviation handling policy* determines classes of constraints that can be violated. The dynamic aspect allows one to establish different policies for different process phases, or for different users - an expert may be trusted to violate more constraints than a novice user. Borgida and Murata [1] propose reifying activities and workflows, storing related information in classes that are accessible at execution time, e.g., ordering of tasks and constraints. Constraint violations are flagged and handling can be performed either by an automated handler or by users, by modifying the reified information (e.g. changing the order of steps). Deviations can be tolerated through the use of *excuses*, objects that record, e.g., the authorizing agent, reason for deviation and so on.

6. CONCLUSION AND FUTURE WORK

The constraint language and formal definitions presented above have two limitations which are being addressed. The first one is that they do not allow for repetitions of instances of activities, which is needed in case of loops or periodic repetitions of activities: after a surgical intervention the incision must be examined and dressed daily, until the physician decides it is no longer necessary. The lack of repetition of instances makes this constraint language less expressive than standard workflow definition languages.

The other limitation is the lack of temporal references in the constraints. As the example above made clear, health care examples usually have temporal constraints: the examination of the incision must be done *daily*, the patient fast for at least 12 hours before a surgery, and so on. By adding temporal constraints, the workflow as a helper becomes much more useful: the physician in charge can know the earliest she could schedule a surgery given the patient was served a meal three hours ago. The system can also tell the latest the surgery can start, given a constraint that the cardiac evaluation must happen in less than two days before the intervention. In [12] we discuss the solution for a simplified version of the full temporal constraint problem, where pre- and post-conditions of all activities are organized as a forest, that is, there are no two paths relating different activities.

7. REFERENCES

[1] A. Borgida and T. Murata. Tolerating exceptions in workflows: A unified framework for data and processes. In *Proc. of the WACC'99*, San Francisco, CA, February 1999.

[2] G. Cugola. Tolerating deviations in process support systems via flexible enactment of process models. *IEEE Transactions on Software Engineering - Special Section on Managing Inconsistency in Software Development*, 24(11):906–907, November 1998.

[3] P. Dourish, J. Holmes, A. MacLean, P. Marqvardsen, and A. Zbyslaw. Freeflow: Mediating between representation and action in workflow systems. In *Proceedings of the ACM 1996 Conference on Computer Supported Work*, pages 190–198, New York, Nov. 16–20 1996. ACM Press.

[4] D. Ferraiolo, J. Cugini, and R. Kuhn. Role based access control: Features and motivations. In *Annual Computer Security Applications Conference*, IEEE Computer Society Press, New Orleans, Louisianna, December 1995.

[5] S. I. Gavrila and J. F. Barkley. Formal specification for role based access control user/role and role/role relationship management. In *ACM Workshop on Role-Based Access Control*, pages 81–90, 1998.

[6] P. Mangan and S. Sadiq. On building workflow models for flexible processes. In *The Thirteenth Australasian Database Conference ADC2002*, Melbourne, Australia, 2002.

[7] P. J. Mangan and S. Sadiq. A constraint specification approach to building flexible workflows. *Journal of Research and Practice in Information Technology*, 2002.

[8] R. Sandhu, E. Coyne, H. Feinstein, and C. Youman. Role-based access control models. *IEEE Computer*, 29(2):38–47, 1996.

[9] W. van der Aalst and P. Berens. Beyond workflow management: Product-driven case handling. In C. Ellis, T. Rodden, and I. Zigurs, editors, *ACM Conference on Supporting Group Work (GROUP 2001)*, pages 42–51. ACM Press, 2001.

[10] J. Wainer. Logic representation of processes in work activity coordination. In *Proceedings of the ACM Symposium on Applied Computing, Coordination Track*, volume 1, pages 203–209. ACM, ACM Press, March 2000.

[11] J. Wainer, P. Barthelmess, and A. Kumar. W-RBAC: a workflow secutiry model incorporating controlled overriding of constraints. *International Journal of Cooperative Information Systems*, page to apperar, 2003.

[12] J. Wainer and F. Bezerra. Constraint-based flexible workflows. In *Proceedings of the CRIWG International Workshop in Groupware*, page to appear, 2003.