Formalizing workflows as cooperative agents

Chitta Baral*
Department of Computer Sc.
Univ. of Texas at El Paso
El Paso, TX, 79968, USA
chitta@cs.utep.edu
915-747-6952

Jorge Lobo†
Department of EECS
Univ. of Illinois at Chicago
851 S. Morgan St
Chicago, IL 60607, USA
jorge@eecs.uic.edu

Abstract

In this paper we present a formal model to characterize cooperative agents in a workflow setting in terms of activities, description of effect of activities, and definition of activities as execution of programs and control modules. We define the correctness of a sequential control module, give sufficiency conditions about the rules in the module, and use the sufficiency conditions to give a naive algorithm to construct sequential control modules. Finally, we extend our notion of correctness to non-sequential control modules.

1 Introduction

Among the most important challenges that will drive research in databases and information systems in the future is the development of robust workflow management systems. Collaborative design, Health-care information systems and data processing on the Internet are examples of applications that will demand automation of management of information workflow (see [GHS95, Moh96, She96] for examples). There are many commercial products to model and execute workflows but the current state of the art lacks clear theoretical basis, defined correctness criteria, support for consistency of concurrent workflows and support for reliability in the presence of failures or exceptions [WS96]. *In this paper we propose treating workflow as a collection of cooperative agents and use recent results on reasoning about actions [LLL+94, San92, GL93, BGP96] to formalize correctness of a workflow. We also discuss automatic verification and construction of reactive rules that specify the workflow control.*

1.1 Registration Example

Throughout the paper we will consider the example of automating the paper trail involved in the graduate advising process of a graduate student at the University of Texas at El Paso. In this process, a student fills an advising form with his/her name, student id, the current semester and the details of the course he would like to take (course number, course name, course time).

Next he takes the form to the graduate advisor during the advising hours (or at a prearranged time - normally prearranged using email) and gets his authorization. (If he has a thesis topic he can get advised by her thesis advisor, in the absence of the graduate advisor.)

The advisor determines if he has already taken the prerequisites of the courses he intends to take and if the class in not full. If these conditions are met and there is no extraneous circumstances (for example, he may be in probation – below 3.0 GPA, or some of the courses did not appear in the published schedule, etc.) then the advisor authorizes the registration form by her signature. The student then takes the form to the

*Support was provided by the National Science Foundation, under grant Nr. IRI-9211662 and IRI-9501577.
†Support was provided by Argonne National Laboratory under contract Nr. 963042401.

graduate secretary who gives the student the registration password which the student can use to register for the courses.

If the student does not have the prerequisite of a course he would like to take he needs to contact the instructor of that course and get his permission. If the class is full the student needs to get permission from the instructor and the chairperson of the department to allow over-subscription to that course.

If the advisor is out of town then a substitute advisor can advise the student. Similarly, the substitute chair can fill the role of the chair. If the instructor is out of town then the student has to wait, unless time is critical. In that case the advisor can make a decision about the prerequisites.

Workflows are usually depicted using some kind of graph representation, where nodes are associated with activities and arcs with control or data flow between tasks. Sometimes arcs are labeled with conditions that represent under which circumstances the control flow will follow the arc after the active where the arc originates is completed. A simplified version of our registration example is represented in Figure 1.

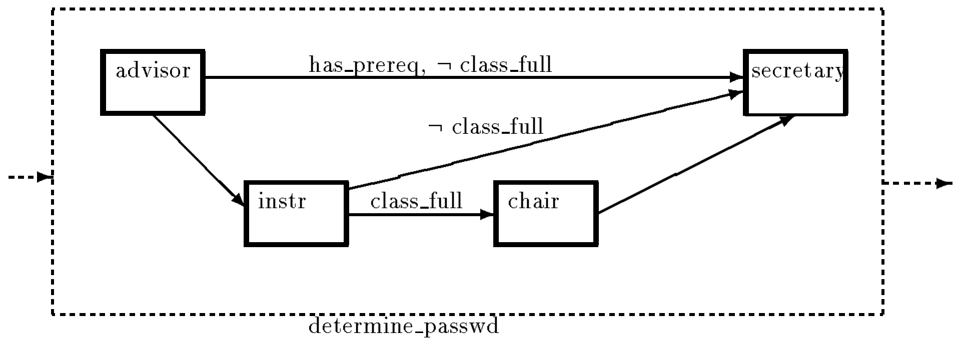

Figure 1: Registration authorization workflow

2 Workflows and its constituent agents

A workflow always involves several processes or activities[1], each performed by an agent. The main activity can be considered to be performed by the workflow agent itself whose job is to maintain the control flow of information, monitor the other agents, recover from failures, answer queries about the status of the workflow agent, etc. The other activities receive commands from the workflow agent and proceed as requested. The other agents could be automated agents, or a human being, or an automated agent monitored by a human being. In our motivating example, the secretary, the advisor, the chair and the instructors perform other activities. The advisor may decide to further automate his job and develop an automated agent to act on his behalf.

The activity performed by an automated agent is an execution of a program. In other words, we can say that an agent (or the activity it performs) is *defined* by a control program. A program could be made of finer activities. A *simple program* is a sequence of activities, which when executed achieves a goal. In AI parlance it is referred to as a *plan*. For any activity P, its *environment* consists of a local database (possibly a set of databases), the messages it has received (these messages can be passed through changes in the local database), and the databases of other activities that can be accessed by P.

We refer to the program executed by the workflow as the workflow program. However, simple programs (i.e. a sequence of activities) are rarely appropriate since, normally, workflows are not generated in run time. They are written beforehand when exact information about the environment is not available.

[1] We use the term activities, as it is the one used in the workflow community. In the 'agents' community and the 'reasoning about actions' community the equivalent word is an 'action'. Similarly, the terms *simple programs* and *conditional prtograms* in this paper are referred to as *plans* and *conditional plans* in the AI and planning community.

If we assume that *complete* information about the environment becomes available when the workflow is executed then we can specify the workflow program using case statements. The case statements correspond to the different paths the workflow may take depending on the environment on hand during its execution. Such a program is called a *conditional program*. But it is also the case that even during the execution time, the workflow does not have complete information about the environment. In that case it needs to perform activities that collect values of certain parameters from sources outside the workflow during the execution of the workflow. We call these activities *sensing* activities because of their similarity to robot sensing where a robot, using special devises such as a camera, gathers information from its surrounding to plan its navigation. Thus, when specifying the effects of activities we may need to distinguish between sensing and non-sensing activities.

We define conditional programs as follows.

Definition 1 Conditional program

(i) If a is an activity then a is a conditional program.

(ii) If c_1 and c_2 are conditional program, then $c_1; c_2$ is a conditional program.

(iii) If $c_1, \ldots, c_n$ are conditional programs and $p_{i,j}$'s are test conditions on the status of the workflow environment then the following is a conditional program.

case

$p_{1,1}, \ldots, p_{1,m_1} \rightarrow c_1$

$\vdots$

$p_{n,1}, \ldots, p_{n,m_n} \rightarrow c_n$

endcase

(iv) Nothing else is a conditional program. □

The conditions in a case statement do not need to be mutually exclusive. If two or more conditions in a case statement become true during the execution of a conditional program this indicates that the corresponding sub-programs will run in parallel or concurrently.

Example 1 *An example of a conditional program P for the registration domain is given below. This conditional program corresponds to the workflow in Figure 1, but is little more detailed than it.*

$advisor(s_1, c_1);$

case:

$has\_prereq(s_1, c_1), \neg class\_full(s_1, c_1) \rightarrow secretary(s_1, c_1)$

$\neg has\_prereq(s_1, c_1), \neg class\_full(s_1, c_1) \rightarrow instr(s_1, c_1); secretary(s_1, c_1)$

$has\_prereq(s_1, c_1), class\_full(s_1, c_1) \rightarrow instr(s_1, c_1);$

   case:

   $space\_waived\_instr(s_1, c_1) \rightarrow chair(s_1, c_1); secretary(s_1, c_1)$

   $\neg space\_waived\_instr(s_1, c_1) \rightarrow secretary(s_1, c_1)$

   endcase

$\neg has\_prereq(s_1, c_1), class\_full(s_1, c_1) \rightarrow instr(s_1, c_1);$

   case:

   $prereq\_waived(s_1, c_1), space\_waived\_instr(s_1, c_1) \rightarrow$

      $chair(s_1, c_1); secretary(s_1, c_1)$

   $\neg space\_waived\_instr(s_1, c_1) \rightarrow secretary(s_1, c_1)$

   endcase

endcase

In the above conditional program, the activity $advisor(s_1, c_1)$, is a sensing activity which determines whether the student s_1 has met the requirements with respect to the course c_1 or not and if the class c_1 is full. The intended meaning of the conditions in the case statements is the following:

> $requested(s_1, c_1)$: *Student s_1 has requested course c_1.*
>
> $has\_prereq(s_1, c_1)$: *s_1 has prerequisites for c_1.*

$class\_full(s_1, c_1)$: Course c_1 has been determined to be full when s_1 requested (We are assuming for the time being no drops in requests are allowed.).
$prereq\_waived(s_1, c_1)$: Prerequisites for course c_1 are waived for s_1.
$space\_waived\_instr(s_1, c_1)$: Space requirement for course c_1 has been waived for student s_1 by the instructor.
$space\_waived\_(s_1, c_1)$: Space requirement for course c_1 has been waived for student s_1 by the Chair.
$passwd(s_1, c_1)$: Student s_1 is granted a password to register for c_1.

The effect of all the activities in this program will be described in Section 3. □

Although conditional programs will be our building blocks for showing the correctness of workflow specifications, they are not enough to define activities. This is because activities do not happen in isolation. i.e., while an activity (say the workflow activity) is executed by an agent, another activity (performed by another agent) may change the environment which may require the workflow activity to change its course. (For example, if the student is allowed to change his request and he decides to drop one of his requests while it is waiting to be processed by the instructor, the workflow should not proceed with the processing of the registration request.) Also, the act of sensing will be usually a complex process. The request of external information may require the intervention of another agent that supplies the information requested by the workflow agent. Hence, the workflow agent may need to wait for the other agent to respond. This can not be expressed in a conditional program. Furthermore, the workflow agent, while waiting for the other agent to respond, may want to do something else, such as attending the request for another registration. This also can not be expressed in a conditional program. To express or define the control program of agents that allow these flexibilities we need the continuous execution of a control module of rules that are triggered by changes in the workflow environment.

A simplified version of a workflow control module corresponding to the registration example is the following:

A sequential control module

REPEAT
if $requested(S, C)$, **unk** $has\_prereq(S, C)$, **unk** $class\_full(S, C)$ **then** $advisor(S, C)$
if $has\_prereq(S, C)$, $\neg class\_full(S, C)$, **unk** $has\_passwd(S, C)$ **then** $secretary(S, C)$
if $\neg has\_prereq(S, C)$, **unk** $prereq\_waived(S, C)$ **then** $instr(S, C)$
if $class\_full(S, C)$, **unk** $space\_waived\_instr(S, C)$ **then** $instr(S, C)$
if $class\_full(S, C)$, $has\_prereq(S, C)$, $space\_waived\_instr(S, C)$, **unk** $space\_waived\_chair(S, C)$
then $chair(S, C)$
if $class\_full(S, C)$, $\neg has\_prereq(S, C)$, $space\_waived\_instr(S, C)$, **unk** $space\_waived\_chair(S, C)$
$prereq\_waived(S, C)$ **then** $chair(S, C)$
if $\neg has\_prereq(S, C)$, $\neg class\_full(S, C)$, $seen\_instr(S, C)$, **unk** $has\_passwd(S, C)$
then $secretary(S, C)$
if $class\_full(S, C)$, $seen\_instr(S, C)$, $seen\_chair(S, C)$, **unk** $has\_passwd(S, C)$ **then** $secretary(S, C)$
if $passwd(S, C)$ **then** $SUSPEND$
if $\neg passwd(S, C)$ **then** $SUSPEND$
FOREVER

This control module is started when the predicate $requested(S, C)$ becomes true in its local environment (this will be its initial state), and it is supposed to run continuously until SUSPEND is executed. It wakes up when changes happen to the environment as, for example, when a new request for registration is submitted. Conditions in the **if** - **then** statements (also referred to as *rules* with the **if** part of a rule referred to as the LHS of the rule and the **then** part referred to as the RHS of the rule) are tested with respect to the environment of the workflow agent. These conditions (as in the conditional programs) are queries about the environment. We will distinguish between queries whose answer is true vs. queries whose answer is false or unknown. For example, in the registration control module above, we expect the answer for $requested(S, C)$ to be true, and the answer for $has\_prereq(S, C)$ and $class\_full(S, C)$ to be unknown (denoted by **unk** in the control module) to make the whole condition in the LHS of the first rule true and then execute the *advisor* activity in the RHS. The activity $instr(S, C)$ in the third **if** statement will be executed if $has\_prereq(S, C)$ is false and $prereq\_waived(S, C)$ is unknown. The full version of this paper describes how this could be represented. If more than one LHS evaluates to true then the activities in the RHS of the rules can be ran concurrently.

A control module is said to be *sequential* for a state s if for any state s' that can be reached from s executing the control module, all rules whose LHS is satisfied by s' have the same RHS.

Although, we prefer activities (performed by an agent) to be defined as control modules, it should be noted that in a particular environment s of a particular activity, if we assume that no other agents are going to affect the environment, besides the agents explicitly invoked by the control module, then the execution of the control module will be the same as the execution of a particular conditional program. We say that the control module *unfolds* to that particular conditional program in environment s. We will describe how the process of unfolding is done later in Section 5. Nevertheless, we can not dispense with the control module, because we will not know the environment until run time.

The control module we described earlier is a sequential control module. In general, workflow systems should allow us to concurrently execute several activities. For example, for the registration workflow, if the class is full, and the student needs to get permission from both the instructor and the chair (independently); the workflow control module should allow these two activities to be executed concurrently. (*We assume that concurrency control is managed by the database manager.*) Such control modules are referred to as *non-sequential control modules*. Such a non-sequential control module is graphically described in Figure 2 and specified as a continuously executing module as follows:

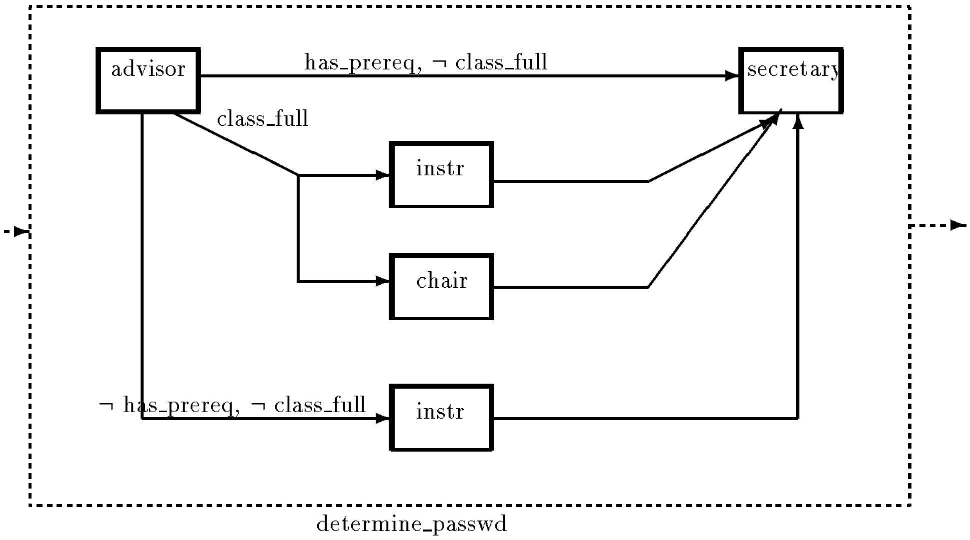

Figure 2: Non-sequential control module for registration

A non-sequential control module

REPEAT

if $requested(S, C)$, **unk** $has\_prereq(S, C)$, **unk** $class\_full(S, C)$ **then** $advisor(S, C)$
if $has\_prereq(S, C), \neg class\_full(S, C)$, **unk** $has\_passwd(S, C)$ **then** $secretary(S, C)$
if $\neg has\_prereq(S, C)$, **unk** $prereq\_waived(S, C)$ **then** $instr(S, C)$
if $class\_full(S, C)$, **unk** $space\_waived\_instr(S, C)$ **then** $instr(S, C)$
if $class\_full(S, C)$, **unk** $space\_waived\_chair(S, C)$ **then** $chair(S, C)$
if $\neg has\_prereq(S, C), \neg class\_full(S, C), seen\_instr(S, C)$, **unk** $has\_passwd(S, C)$
then $secretary(S, C)$
if $class\_full(S, C), seen\_instr(S, C), seen\_chair(S, C)$, **unk** $has\_passwd(S, C)$ **then** $secretary(S, C)$
if $passwd(S, C)$ **then** $SUSPEND$

if $\neg passwd(S, C)$ **then** $SUSPEND$
$FOREVER$

In the above specification if the class is full, and neither the chair and nor the instructor have yet acted upon the waiver, the LHS of Rules 4 and 5 above will be simultaneously true and as a result the above control module will concurrently execute the activities $instr(S, C)$ and $chair(S, C)$, by invoking the instructor agent and the chair agent.

3 Formal Characterization of Activities and Programs

We reiterate that an agent continuously executes a control module, which can be abstracted as an activity and would be referred to as the agent activity. For example, the activity performed by the workflow agent by continuously executing the workflow control module is referred to as the workflow activity. A control module itself is specified using finer activities.

We specify and reason about activities in a high level language $\mathcal{A}_K$[2], which is in the spirit of the action description language $\mathcal{A}$ [GL93]

This language has two disjoint nonempty sets of symbols, called *fluents* and *activity names*. A *fluent literal* is either a fluent or a fluent preceded by $\neg$. For a fluent, we abbreviate $\neg\neg l$ to l.

Fluents are data items that describe the state of the workflow and they are modified or added to the workflow description by the activities. Activity names will have an arity associated with them and will indicate the inputs required to run the activity. In the examples we will assume that values of fluents are initially unknown, unless it is indicated otherwise.

An *activity description*[3] in the language $\mathcal{A}_K$ consists of "propositions" of two kinds: "ef-propositions" and "k-propositions". An "ef-propositions" describes the effect of an activity on the truth value of a fluent. An *ef-proposition* is an expression of the form

$$a \textbf{ causes } f \textbf{ if } p_1, \ldots, p_n \tag{1}$$

where a is an activity , and each of $f, p_1, \ldots, p_n$ $(n \geq 0)$ is a fluent literal. The set of fluent literals $\{p_1, \ldots, p_n\}$ is referred to as the *precondition* of the ef-proposition and f is referred to as the *effect* of this ef-proposition. *All the arguments that appear in f, if they are free variables, they must also appear as arguments of a or one of the p_i.* About this proposition we say that it *describes the effect of a on f*, and that $p_1, \ldots, p_n$ are its *preconditions.*

A *k-proposition* is an expression of the form

$$a \textbf{ determines } f \textbf{ if } q_1, \ldots q_n \tag{2}$$

where a is a sensing activity, f a fluent and each $q_1, \ldots, q_n$ $(n \geq 0)$ is a fluent literal. The set of fluent literals $\{q_1, \ldots, q_n\}$ is referred to as the *precondition* of the k-proposition and f is referred to as the *k-effect* of this k-proposition. This proposition stipulates that if a is executed in a situation where $q_1, \ldots, q_n$ are true, then in the resultant state the truth value of f becomes known. However, the value can not be predicted and it will be known only at run time. We will refer to activities in k-propositions as knowledge producing activities. We will assume that knowledge producing activities appear in k-propositions only.

A *proposition* is an ef-proposition, or an k-proposition. An *activity description* is a set of propositions which does not contain contradictory ef-propositions.

Example 2 *The following is partial activity description D for the student registration example.*
$advisor(S, C)$ **determines** $has\_prereq(S, C)$ **if** $requested(S, C)$
$advisor(S, C)$ **determines** $class\_full(S, C)$ **if** $requested(S, C)$

[2]Due to lack of space we do not formally describe this language [BS97] but briefly describe its salient features and give examples of specifications in this language. The formal description of the language is available through http://cs.utep.edu/chitta/chitta.html.

[3]Note that the activity description is different from activity definition. The former specifies the effect of performing the activity. The later spells out what exactly needs to be executed when performing an activity.

$instr(S,C)$ **determines** $prereq\_waived(S,C)$ **if** $\neg has\_prereq(S,C)$
$instr(S,C)$ **determines** $space\_waived\_instr(S,C)$ **if** $class\_full(S,C)$
$instr(S,C)$ **causes** $seen\_instr(S,C)$
$chair(S,C)$ **determines** $space\_waived\_chair(S,C)$ **if** $class\_full(S,C)$
$chair(S,C)$ **causes** $seen\_chair(S,C)$
$secretary(S,C)$ **causes** $passwd(S,C)$ **if** $has\_prereq(S,C), \neg class\_full(S,C)$
$secretary(S,C)$ **causes** $passwd(S,C)$ **if** $has\_prereq(S,C), class\_full(S,C), space\_waived\_instr(S,C), space\_waived\_chair(S,C)$
$secretary(S,C)$ **causes** $passwd(S,C)$ **if** $\neg has\_prereq(S,C), \neg class\_full(S,C), prereq\_waived(S,C)$
$secretary(S,C)$ **causes** $passwd(S,C)$ **if** $\neg has\_prereq(S,C), class\_full(S,C), prereq\_waived(S,C), space\_waived\_instr(S,C), space\_waived\_chair(S,C)$
$secretary(S,C)$ **causes** $\neg passwd(S,C)$ **if** $\neg has\_prereq(S,C), \neg prereq\_waived(S,C)$
$secretary(S,C)$ **causes** $\neg passwd(S,C)$ **if** $class\_full(S,C), \neg space\_waived\_chair(S,C)$
$secretary(S,C)$ **causes** $\neg passwd(S,C)$ **if** $class\_full(S,C), \neg space\_waived\_instr(S,C)$

□

The semantics of an activity description is given by defining transition functions Φ, that describes how the execution of an activity a modify the state σ of the workflow. *Here, a state*[4] *describes the environment of the workflow at a particular moment,* and it is represented by a pair of disjoint sets of fluents $\sigma = \langle T, F\rangle$. Fluents in T represent statements about the current properties of the workflow that are true. F represent statements that are false. The transition functions Φ, are called models of the activity description. Models of activity description in $\mathcal{A}_K$ are formally defined in [BS97].

A query is an expression of the form

$$\varphi \textbf{ after } c \textbf{ at } \sigma \tag{3}$$

where φ is a formula made with fluents, c is a conditional program, and σ a state.

We say a query φ **after** c **at** σ is true in a model Φ if φ is true in all states that may be reached by executing the program c in the state σ. (The effect of the execution of c in σ is obtained by iterative applications of Φ.)

We say that a description D entails a query q (denoted by $D \models q$) if q is true in all models of D. In our registration example, if D is the domain description in Example 2 and P the program in Example 1, it can be shown that

$$passwd(S,C) \vee \neg passwd(S,C) \textbf{ after } P \textbf{ at } < \{requested(S,C)\}, \emptyset >$$

Note that because of our three valued semantics the formula $passwd(S,C) \vee \neg passwd(S,C)$ is not a tautology. If the truth value of $passwd(S,C)$ is unknown then the formula is not true.

For our simple language (where we do not have non-deterministic activities), every description has a unique model.

4 Correctness and Automatic Construction of Sequential Workflow control modules

We mentioned before that in an workflow environment activities do not happen in isolation. During the execution of the main workflow activity (and the activities invoked by the workflow) other agents may act (say, corresponding to the bursar's office, or other activities performed by the student) simultaneously. For example the student could run an activity to drop one of his/her classes. This is an activity that will cause the regular route of the registration workflow to change. We will also consider the activity that changes the control module status from suspended to running one of these external activities, since the workflow does not have control of when this activity will happen. For that reason (to take into account changes in the

[4] In the rest of the paper we refer to state and environment interchangeably.

environment made by other activities), and since the workflow agent needs to execute continuously, it is defined by a control module, which we refer as the workflow control module, or simply the workflow.

In formalizing sequential workflow control modules, we consider all the agents outside the main workflow as *exogenous* agents. Later in Section 5, when we formalize non-sequential workflow control modules, we consider some of the agents outside the main workflow as cooperative agents, and the others as exogenous agents. A cooperative agent of the workflow agent differs from an exogenous agent in the sense that the workflow agent can communicate with its cooperative agents and request them to perform some activities, while it has no control over the exogenous agents. Although, the workflow agent can request its cooperative agents to perform some task, due to communication delays, it can not exactly predict how long it will take for the cooperative agent to complete its assigned activity. In case of sequential workflow control modules the workflow agent waits for any delegated activity to be done. In case of non-sequential workflow control modules, the workflow agent may concurrently execute other activities, after sending a request to a cooperative agent, without waiting for the completion of the assigned activity by the cooperative agent.

In our registration example we have two exogenous activities controlled by the student (agent). These activities are described by the following ef-propositions:

$$request(S, C) \textbf{ causes } requested(S, C).$$

$$drop(S, C) \textbf{ causes } \neg requested(S, C).$$

In this section we describe the correctness of sequential workflow control modules that tolerate the intrusion of exogenous activities, and discuss how they can be constructed automatically.

The workflow control module could be either automatically constructed or verified before hand. In both cases the activity descriptions (regular and exogenous) and the *goal* of the workflow is given to a constructor-verifier system. The goal describes the purpose (or purposes) of the workflow. For example, in our registration workflow the goal is to process registrations requested by students. This system may then generate a workflow control module, or if a workflow control module is given to it, it can verify its correctness with respect to its goal.

Goals will be described by formulas built with fluents. In our registration example, the goal is to make the formula $registered(S, C) \Rightarrow passwd(S, C) \vee \neg passwd(S, C)$ true. This is what we use in this paper. But in general a goal [BK96] could be a temporal formula that the workflow is trying to make (or maintain) true. This will allow us to not only state what needs to be true at the end of the execution of a program but what needs to be true or needs to be maintained during the execution of the program.

When a workflow is constructed, to keep the control module within reasonable size certain assumptions are made and certain states are ruled out. But if during the execution the workflow control module gets into such a state then we have an *exception*. In that case the workflow control module will either refer to its owner (a person) or if it is in auto mode then it may directly invoke the constructor-verifier system to generate additional case statements corresponding to the unaccounted for states. Even when the workflow refers to its owner, the owner may use the constructor-verifier system to generate or verify additional case statements corresponding to the unaccounted for states.

4.1 Correctness of sequential workflow control modules

Let S be a set of states, M be a control module and A be a set of exogenous activities. By, $Closure(S, M, A)$ we denote the the least[5] (w.r.t. subset ordering) set of states that contains S, and that satisfies the conditions: **(i)** if $s \in S$ then any state that can be reached by the execution of a sequence of exogenous activities at s is in $Closure(S, M, A)$; and **(ii)** if $s \in S$ then the states reached during the execution of M in s are in $Closure(S, M, A)$.

Let P be a sequential workflow control module and s be the state corresponding to the initial input given to the workflow. $Unfold(P, s)$ is the workflow program obtained by unfolding the control module P with respect to s, by assuming that no activities other than the ones dictated by P happen. (Note that the occurrences of any other activity will be captured by the Closure definition.) $Unfold(P, s)$ will be formally defined in the full paper.

[5] We can formally prove that such a set exists.

For deterministic workflow control modules (i.e., only one rule can fire at any state, and all activities are deterministic), if s is a complete state then $Unfold(P, s)$ is a sequence of activities, otherwise it is a conditional program.

Definition 2 A workflow control module P is said to be correct w.r.t. a set of states S and a set of activities A defined in description D, and a goal G if, $S = Closure(S, P, A)$ and for all states s in S, and for all f in G, $D \models f$ **after** $Unfold(P, s)$ **at** s. □

Note that, even though when we unfold P w.r.t. s we do not consider exogenous activities (i.e., the activities outside the workflow), they are considered when defining closure and hence are taken into account.

Intuitively, the correctness of a workflow control module P w.r.t. S, A and G means that if we can guarantee that if we start executing the workflow control module with initial data corresponding to some state in S, then when the workflow control module completes its execution, G will be true. Moreover, during the execution of the workflow control module other processes can interact as long as they execute activities only from A. Note that the workflow control module may execute infinitely if it is continuously harassed (or interfered in a negative way) by outside activities. But given a sufficient window of non-interference it will complete its execution.

4.2 Sufficiency Conditions

Definition 3 Soundness and Completeness
A workflow control module P is said to be *complete* w.r.t. goal G and a set of states S, if for each s in S there exists at least one rule in P which is applicable in s.

A simple rule r is said to be *sound* w.r.t. goal G, a set of states S, and a description A if for all $s \in S$ such that r is applicable in s there exists a **minimal cost** [6] **program** (in the language of A) that achieves G from s and has the RHS of r as its prefix. □

Theorem 1 Let P be a deterministic workflow control module, S be a set of states, and G be a goal.
If P is sound and complete w.r.t. G, S, and A then P is correct w.r.t. G, S and A. □

We now give the sketch of an algorithm that uses Theorem 1 to construct a workflow control module that achieves a goal G from a set of states S.

Algorithm 1 *Input*: Goal G, a set of states S, and a set A descriptions for the activities in the workflow and other interacting activities.

Output: A control module P that is correct w.r.t. G, S and A.

Step 1 $M = \emptyset$, $S_{in} = \{s' : s'$ is reachable from some state s in S by applying sequences of activities in $A\}$ and $S_{out} = \emptyset$.

Step 2 While $S_{in} \neq S_{out}$
  2.1 Pick a state s from $S_{in} \setminus S_{out}$
  2.2 *Find the minimal cost program*[7] P from s that achieves G. Let a be the first activity in P.
  2.3 If P is a null plan (i.e., the goal G is true in S then $M = M \cup \{$ **if** s **then** $HALT\}$;
    else $M = M \cup \{$ **if** s **then** $a\}$;
  2.4 $S_{out} = S_{out} \cup \{s\}$.

□

5 Formalizing cooperation in a non-sequential workflow

The formalization in the previous sections was with respect to sequential control modules only - where we do not exploit the concurrency inherent in a cooperative agents setting. In this section we formalize non-sequential workflows where the workflow agent after requesting a cooperative agent to perform some activity, can concurrently pursue other activities.

[6] The cost of a program is defined by assigning costs - a positive integer - to each activity in the program
[7] We will need a modification of the planners in [EHW+92] for constructing minimal conditional programs.

The approach we will take to model cooperation is based on breaking the message-passing activities such as the activity $advisor(S, C)$ into two parts: the *first part* (that we will denote by "$advisor(S, C)_($") sends a message to the advisor agent and the *second part* (that we will denote by "$advisor(S, C)_)$") is a cooperative activity that sends the determination of the advisor back to the main workflow agent. When $advisor(S, C)$ is executed the first part is immediately executed. The second part is expected to (asynchronously) happen anytime after that.

For defining correctness, we need to change the definition of $Closure$, as the occurrence of the second part is not completely known. Similarly, we need to change the definition of Unfold.

More precisely. We will extend $\mathcal{A}_K$ with two new types of effect propositions, called *message effect* propositions. Given a message activity m, an *outgoing message effect* proposition is an expression of the form:

$m_($ **causes** $sent\_m$ **if** $p_1, \ldots, p_n, \neg sent\_m$

where $p_1, \ldots, p_n$ are fluent literals and $sent\_m$ is a new fluent added to $\mathcal{A}_K$. Message activities can only appear in outgoing effect propositions. An *incoming message effect* proposition is a expression of the form:

$m_)$ **causes** f **if** $p_1, \ldots, p_n, sent\_m$

or

$m_)$ **determines** f **if** $p_1, \ldots, p_n, sent\_m$

where $m_)$ is a cooperative activity, called the *complementary* activity of $m_($, and f and $p_1, \ldots, p_n$ are fluent literals (f in the sensing proposition must be positive). To define closure we need to restrict the sequence of exogenous activities that can be interleaved between the activities controlled by the agent. For this, we define when a sequence of activities is legal.

1. an empty sequence [] is legal.
2. if α is a legal sequence of activities and a is a non cooperative exogenous activity then $\alpha \circ a$ is a legal sequence of activities.
3. If $[a_1, \ldots, a_n]$ is a legal sequence of activities and if the LHS of a rule in the monitor P is satisfied in $\Phi(a_n, \Phi(a_{n-1}, \ldots \Phi(a_1, s) \ldots)$ and its RHS activity is a then $[a_1, \ldots, a_n] \circ a$ is a legal sequence of activities.
4. If α is a legal sequence of activities and for some message activity $m_($ there are more $m_($ in α than $m_)$, then α followed by $m_)$ is a legal sequence of activities.

Thus, $Closure(S, P, A)$ is the set of states that can be reached from any state s in S through a legal sequence of activities in that state with respect to the control module P and set of activities A.

The unfolding of a control module must now consider the occurrences of cooperative activities, since achieving a goal may only happen with the collaboration of cooperative agents if the sequence of activities that achieve the goal involves message activities. Since exogenous activities are not controlled by the workflow agent, the agent must be prepared for their occurance at any (possible) time. Thus, there will be different ways of unfolding the control module, one for each possible alternative. The new notion of unfolding also covers the presence of concurrent activities. We define an unfolding function $\mathcal{U}$ to be any partial function that maps a control module, a state and a sequence of activities α (that led to the state s from the initial state) into an activity a. An unfolding function $\mathcal{U}$ is correct with respect to a control module P if and only if for every state s reached through a sequence of activities α one of the following conditions holds:

1. $\mathcal{U}(P, s, \alpha) = a$ and there exists a LHS of a rule in P that is true in s and whose RHS is a.
2. $\mathcal{U}(P, s, \alpha) = m_)$, for some message activity m, and there are more $m_($ in α than $m_)$.
3. If there is no rule in P whose LHS is true in s and for any message activity $m_($ the number $m_($ in α is equal to the number of $m_)$, or if the last activity in α is $SUSPEND$ then $\mathcal{U}(P, s, \alpha)$ is undefined.

Beginning with a sequence of activities α and a state s , the unfolding of P based on a correct unfolding function $\mathcal{U}$, $Unfolding_{\mathcal{U}}(P, s, \alpha)$ is defined as follows:

1. If $\mathcal{U}(P, s, \alpha) = SUSPEND$ then $Unfold_{\mathcal{U}}(P, s, \alpha) = \alpha$.

2. If $\mathcal{U}(P, s, \alpha) \neq SUSPEND$ and is not a sensing activity then $Unfold_{\mathcal{U}}(P, s, \alpha) = [\mathcal{U}(P, s, \alpha); Unfold_{\mathcal{U}}(P, \Phi(\mathcal{U}(P, s, \alpha), s), \alpha \circ \mathcal{U}(P, s, \alpha))]$.

3. If $\mathcal{U}(P, s, \alpha) \neq SUSPEND$ and is a sensing activity then, let $\mathcal{U}(P, s, \alpha)$ **determines** f_i **if** Q_i for some i, $0 \leq i \leq n$ be all the k-proposition for the activity, $\mathcal{U}(P, s, \alpha)$ for which Q_i is true in s. If $i = 0$ then $Unfold_{\mathcal{U}}(P, s, \alpha) = \alpha$. Otherwise, $Unfold_{\mathcal{U}}(P, s, \alpha) =$
   $[\mathcal{U}(P, s, \alpha);$
   case
   $\neg f_1, \ldots, \neg f_{n-1}, \neg f_n \rightarrow Unfold_{\mathcal{U}}(P, s_0, \alpha \circ \mathcal{U}(P, s, \alpha))$
   $\neg f_1, \ldots, \neg f_{n-1}, f_n \rightarrow Unfold_{\mathcal{U}}(P, s_1, \alpha \circ \mathcal{U}(P, s, \alpha))$
   $\neg f_1, \ldots, f_{n-1}, \neg f_n \rightarrow Unfold_{\mathcal{U}}(P, s_2, \alpha \circ \mathcal{U}(P, s, \alpha))$
   $\neg f_1, \ldots, f_{n-1}, f_n \rightarrow Unfold_{\mathcal{U}}(P, s_3, \alpha \circ \mathcal{U}(P, s, \alpha))$
   $\vdots$
   $f_1, \ldots, f_{n-1}, f_n \rightarrow Unfold_{\mathcal{U}}(P, s_{2^n}, \alpha \circ \mathcal{U}(P, s, \alpha))$
   endcase]
   where s_k is the model in $\Phi(P, s_k)$ in which the k precondition in the case statement (i.e. the precondition of $Unfold_{\mathcal{U}}(P, s_k, , \alpha \circ \mathcal{U}(P, s, \alpha)))$ is true.

Definition 4 An agent control module P is said to be correct w.r.t. a set of states S and a set of activities A defined in description D, and a goal G if, $S = Closure(S, P, A)$ and for all states s in S, for every correct unfolding function $\mathcal{U}$ of P, and for all f in G, $D \models f$ **after** $Unfold_{\mathcal{U}}(P, s, [])$ **at** s. □

6 Final remarks

In this paper we extended our earlier work on formalizing reactive control using action theories [BS96] to cooperative agents in a workflow setting. We have implemented the workflow described in this paper [8] and it is undergoing extensive testing before it will be used at UTEP.

Although there has been a lot of work on multi-agent systems ([Sin94] and papers in earlier ATAL workshops), the closest work to ours is CONGOLOG [LLL+95], where correctness of a concurrent agent language is formulated. Although, our language is simpler that CONGOLOG – we do not have non-deterministic actions, we consider exogenous actions, which are not considered in CONGOLOG. While our control modules are reactive, programs in CONGOLOG are not.

Due to lack of space we have not been able to give detailed formalization of many notions in this paper. We are also unable to give a more detailed comparison with related work. We hope to include this in the full paper.

References

[BGP96] C. Baral, M. Gelfond, and A. Provetti. Representing Actions: Laws, Observations and Hypothesis. *Journal of Logic Programming (to appear)*, 1996.

[BK96] F. Bacchus and F. Kabanza. Planning for temporally extended goals. In *In Proc. of AAAI '96 Thirteenth National Conference on Artificial Intelligence*, pages 1215–1222, Portland, OR, 1996.

[BS96] C. Baral and T. Son. Relating theories of actions and reactive robot control. In *Proc. of AAAI 96 Workshop on Reasoning about actions, planning and robot control: bridging the gap.*, 1996.

[8] The interface of our system can be seen at http://cs.utep.edu/chitta/workflow.

[BS97] C. Baral and T. Son. Approximate reasoning about actions in presence of sensing and incomplete information. Technical report, Dept of Computer Science, University of Texas at El Paso, 1997.

[EHW+92] O. Etzioni, S. Hanks, D. Weld, D. Draper, N. Lesh, and M. Williamson. An approach to planning with incomplete information. In *Proc. of Principles of Knowledge Representation and Reasoning*, pages 115–125, 1992.

[GHS95] D. Georgakopoulos, M. Hornick, and A. Sheth. An overview of workflow management: From process modeling to workflow automation infrastructure. *Distributed and Parallel Databases*, 3:119–153, 1995.

[GL93] M. Gelfond and V. Lifschitz. Representing action and change by logic programs. *Journal of Logic Programming*, 17:301–321, 1993.

[LLL+94] Y. Lesperance, H. Levesque, F. Lin, D. Marcu, R. Reiter, and R. Scherl. A logical approach to high level robot programming – a progress report. In *Working notes of the 1994 AAAI fall symposium on Control of the Physical World by Intelligent Systems, New Orleans, LA*, November 1994.

[LLL+95] Y. Lesperance, h. Levesque, F. Lin, D. Marcu, R. Reiter, and R. Scherl. Foundations of a logical approach to agent programming. In M. Woolridge, J. Muller, and M. Tambe, editors, *Intelligent Agents - II*, pages 331–346. 1995.

[Moh96] C. Mohan. Tutorial: State of the art in workflow management system research and products. In *Proc. 5th International Conference on Extending Database Technology*, Avignon, France, March 1996.

[San92] E. Sandewall. Features and fluents: a systematic approach to the representation of knowledge about dynamic systems. Technical report, Institutionen for dataventeskap, Universitetet och Tekniska hogskolan i Linkoping, Sweden, 1992.

[She96] A. Sheth. Proc. of the NSF workshop on workflow and process automation in information systesm, 1996. URL: http://lsdis.cs.uga.edu/activities/NSF-workflow.

[Sin94] M. Singh. *Multiagent systems - a theoretical framework for intentions, know-how, and communications*. Springer-Verlag, 1994.

[WS96] D. Worah and A. Sheth. What do advanced transaction models have to offer to workflows? In *Proc. of the International Workshop on Advanced Transaction Models and Architectures (ATMA)*, Goa, India, August 1996.