Formalizing and Reasoning About the Requirements Specifications of Workflow Systems

Goce Trajcevski
Department of EECS
Univ. of Illinois at Chicago
gtrajcev@eecs.uic.edu

Chitta Baral*
Department of CSE
Arizona State University
chitta@asu.edu

Jorge Lobo†
Bell Labs
jlobo@research.bell-labs.com

Abstract

This work addresses the problem of *workflow requirements specifications* considering the realistic assumptions that: – it involves experts from different domains (i.e. representatives of different business policies); not all the possible execution scenarios are known beforehand, during the early stage of specification. In particular, since the main purpose of a workflow is to achieve a certain (bussiness) *goal*, we propose a formalism which enables the users to specify their requirements (and expectations) and test if the information that they have provided is, in a sense, sufficient for the workflow to behave "as desired", in terms of the goal. Our methodology allows domain experts to express not only their knowledge, but also the "ignorance" (the semantics allows for unknown values to reflect a realistic situation of agents dealing with incomplete information) and the possibility of occurence of exceptional situations. As a basis for formalizing the process of requirements specifications, we are using the recent results on *reasoning about actions*. We propose a high level language $\mathcal{A}_{\mathcal{W}}$ which enables specifying the effects that activites have on the environment and how they should be coordinated. We also describe our prototype tool for process specification. Strictly speaking, in this work we go "one step" before actual analysis and design, and offer a formalism which enables the involved partners to see if the extent to which they have expressed their domain knowledge (which may sometimes be subject to a proprietary restricions) can satisfy the intended needs and behaviour of their product_to_be. We define an entailment relation which enables reasoning about the correctness of the specification, in terms of achieving a desired goal and, also testing about consequences of modifications in the workflow descriptions.

Keywords: Worfklow Systems; Reasoning about Actions; Process Specification; Information Management; Knowledge Representation and Reasoning;Business Process Modeling;

1 Introduction and Motivation

A *workflow* (WF) is most commonly viewed as a *process* which executes various cooperative and coordinated *activities* in order to achieve a desired *goal* [21]. Worlkflow systems combine the data – centric view of applications, typical for information systems, with the process – oriented (behavioral) view of operating systems. Workflow Management Systems (WFMS), on the other hand provide tools for modeling, executing and monitoring workflows. Throughout the last few years, the development of workflow management systems has gained increasing attention and have

*Support was provided by the National Science Foundation, under grant Nr. IRI-9501577.

†Support was provided by Argonne National Laboratory under contract Nr. 963042401.

been used in wide range of applications [26, 41, 47]. It has already been observed that, in order to be useful for the enterprises, WFMSs need well – defined correctness/reliability criteria and the ability to adapt to changes in a flexible manner [1, 16, 48]. Recent works have addressed quite a few of these issues [19, 24, 30, 32, 43, 57] and identified solutions of many problems of interest in Workflow Management Systems. Among the other contributions, several formalisms which enable representation, reasoning about important run-time properties (e.g. concurrency and deadlocks, distributed execution) and execution of workflows, have been proposed: OGWL (Opera Graphical Workflow Language) [30]; State and Activity Charts [57]; Concurrent Transaction Logic (CTR) [24]; Transaction Datalog (TD) [13] – to name a few.

On the other hand, since the beginning of eighties, the software engineering research has been pointing out the importance of *knowledge representation* being thoroughly captured during the requirements specifications stage, before moving on with analysis/design and implementation [12, 50, 51]. This aspect is very important for process specification in many applications (e.g. E – commerce, Virtual Enterprises) for the workflow systems [2, 23]. Namely, the specification stage involves experts from different domains, who are representatives of different organizations and bussiness policies, and as Whorf's hypothesis from psycholinguistics says: " *... the language a person uses to describe his or her environment is a lens through which he or she views that environment...*". As an "off the context" extreme example, there are escimo tribes which are known to have up to 40 different expressions for the word *snow*, which is normal for them, given the importance that the snow has on their everyday life [55]. Typically, when it comes to workflows, not all the possible executional scenarios are known during the requirements specification stage; participants need not know the details of *how* other partners are implementing certain (business) policies. On the flip–side, a participant may not be even willing to fully reveal a complicated decision – making process and (s)he should have the ability to choose to what extent a process will be a "black box" for the other partners (or to what level of granularity it should be "opened"). Moreover, when it comes to cooperation and synchronization, a particular domain expert typically states the "weakest preconditions" and the "strongest effects" expected.

Consequently, during the early specification stage, the main problems are: has there been enough domain *knowledge* collected; given the information, can we *reason* about its *correctness* (which, at this stage is mostly concerned with being able to achieve a desired (business) *goal*); given the tendency of domain experts to specify mostly *plausible* scenarios, can we handle the occurrence of *exceptional* cases.

Example 1 *A common framework for representing a workflow (as an abstract process) is an annotated* control flow *graph in which vertices represent the activities (steps of execution) and arcs represent the flow of control/data, where the annotations on the arcs express the execution logic. A natural abstraction of the activity performed by an agent, on the other hand, is the execution of a (control) program. A program defining an activity is a sequence of finer activities which, when executed, achieve a desired* goal *(in AI parlance, programs are called* plans*). The control of flow in the program is given by conditions about the workflow* environment[1] *that can be tested by the program.*

Figure 1 (based on an example from [24]) gives an example of a graph – based representation of a workflow with many basic constructs (AND splits and joins; OR splits and joins; preconditions; transition constraints) [21]. It also illustrates a corresponding conditional program which, when

[1]Throughout the paper we will use the terms environment and state interchangably, and we will formaly define the notion of *state* in Section 5. In terminology of WF Management Coalition (WFMC) the environment includes "workflow relevant data", "WF control data" and "audit data".

```
a; case:
        e_a,F1 -> b; [(e_b,F3 -> d; e_d,F4 -> h; e_h ->j) | (e_b ->e; e_e,F5 -> j)]
        e_a,F2 -> c; [(e_c,F6 -> f; e_f -> i) |(e_c -> g)]
 endcase;
 case:
    e_j,e_i,F7 -> k
    e_j,e_g,F8 ->k
 endcase
```

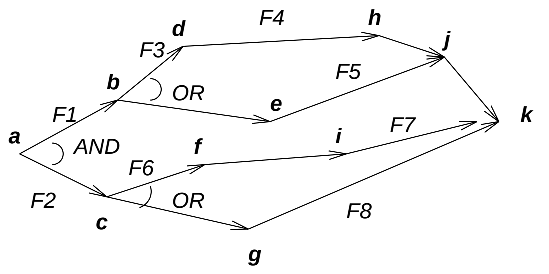


Figure 1: Control flow graph and its conditional program

executed by the workflow engine, will achieve the same behavior. The conditional program uses literals (facts and events – formally defined in Section 2) which describe the effects of activities on the workflow environment, where the meaning of each e_x *is the event that the activity x has completed its execution.*

The example above illustrates possible representations of a workflow. The conditional program is a formalism often used in AI community. On the other hand, the graph in its original form, was used in [24] to demonstrate that a set of operators in Concurrent Transaction Logic (based on Transaction Logic [14, 15]) is sufficient to express various properties of workflows.

However, the problem that we tackle in this article comes from a slightly different perspective. Namely, none of the representations in Example 1 explains *how* it was decided that it is indeed what corresponds to the users' needs. On the other hand, we are offering an approach which will enable the users to reason if the representation corresponds to the intended meaning/purpose of the workflow.

1.1 An Overview of Our Approach

Throughout this work, we view workflows as a collection of cooperative agents and we use recent results on reasoning about actions [4, 6, 29, 36] to formalize the process of their specification and test their correctness. Our approach, in a philosophical sense, resembles formalization of database updates [45] and transactions [14, 15]. In cooperative design applications' terms [40], we give a formal characterization at the activities coordination level ("level–4" in [46]).

Our main contribution are as follows. We present a very simple, high level language $\mathcal{A}_{\mathcal{W}}$ and, based on it, a prototype tool, which facilitates the process specification (i.e. the domain where the workflow activities will execute). The tool enables dual (textual and graphical) representation of workflow specification AND enables the user to toggle between the representations at any time. The domain experts are enabled to specify the constraints on valid states and valid transitions among them, and can specify not only their knowledge but also their "ignorance" (e.g. incomplete state specification) and the possibility of abnormal (exceptional) cases. The users can express behavioral and control aspects in terms of *reactive module* and, once again, both textual and visual representations are available and an user can flip between the two at any time. With all these flexibilities: – the language allows *unknown* values to reflect the realistic situations of dealing with incomplete information (essential during dynamic modifications of a workflow/construction of *ad-*

hoc workflows); – an *entailment* relation for querying the correctness of the specifications, and reasoning about consequences of modifications to its description ("what – if" scenarios)), $\mathcal{A}_\mathcal{W}$ is still based on a strict logical foundation and has a formal semantics. Hence, given the workflow description, we have a formal notion of its *correctness.*

The rest of this article is structured as follows. Section 2 introduces the basis of our formalism, describes the foundation aspects of the language $\mathcal{A}_\mathcal{W}$ and the basic version of our prototype tool. In Section 3 we specify how we address the issues of cooperativity and exceptions and Section 4 presents the formalization of the notion of correctness. The formal declarative semantics of $\mathcal{A}_\mathcal{W}$ is given in Section 5. In Section 6 we position our work and compare it with the relevant literature Finally, in Section 7 we summarize and propose the directions for the future work.

2 Basic Aspects of Formalizing the Workflow Specifications

Specification and reasoning about activities in our approach is done using a tool based on a high level language $\mathcal{A}_\mathcal{W}$, developed in the spirit of the action description language $\mathcal{A}$ [25]. The description of the language is based on $\mathcal{A}_K$ in [10]; and $\mathcal{ADC}$ in [6]). Action theories have been successfully used in reasoning about robot control programs [37] and in the logical formalization of active databases [5, 7] (reasoning about parameterized actions, qualification and ramification constraints, concurrent execution of actions [29]). and facts exempt from commonsense law of inertia [29]. The later use makes it more appropriate than the traditional approach of program correctness [22] which has been designed for standard programming languages and lacks the flexibility of defining new operators/ activities.

Example 2 *Observe an E–commerce scenario where, upon login, a customer should be presented with a "welcome" promotional message. If its an old customer, the content of the message should be based on several criteria like: – credit status; – history of purchases reflecting his/her preferences; – new "hot" products which may be of interest for the particular customer. On the other hand, a new customer should "view" a more broad welcoming menu which shows general categories and enables him to specify particular interest(s).*

The example above (simplified from [33]) reveals a very subtle issue. When specifying the workflow, all that the business partners need to know is that there will be a template activity, say *welcoming_message*, which requires certain input/output parameters. It should further be decided, upon negotiations among the partners involved, which details (if any) of *how* that activity is implemented should be given.

Recall that during the requirements specification of workflows, we do not want to generate "fully correct" but *sound* and *"mostly" complete* descriptions, similar to the incomplete specifications in [20]. This enables avoiding "full blown" theorem provers and using planners to generate modules. Levesque et al. [37] discuss the difference between the "action" approach and traditional approach of "program correctness".

2.1 Activities' Domain Description

$\mathcal{A}_\mathcal{W}$ has three disjoint nonempty sets of symbols, called *facts, events* and *activity names.* Facts are the data items which describe the environment of the workflow (i.e. a tuple in a relational database, or an attribute value of an object). They correspond to the notion of *fluents* in AI parlance, used by McCarthy [39] in the context of reasoning about actions. A *literal* is a fact or event, possibly preceded by $\neg$.

Definition 1 *An* activity description *in the language $\mathcal{A}_{\mathcal{W}}$ consists of a collection of three kinds of "propositions":*

$$a \textbf{ causes } f \textbf{ if } p_1, \ldots, p_n, e_1, \ldots, e_m \quad (\textit{an "ef-proposition"}) \tag{1}$$

$$a \textbf{ determines } f \textbf{ if } p_1, \ldots, p_n, e_1, \ldots, e_m \quad (\textit{a "k-proposition"}) \tag{2}$$

$$a \textbf{ induces } e \textbf{ if } p_1, \ldots, p_n, e_1, \ldots, e_m \quad (\textit{an "event definition"}) \tag{3}$$

• An "ef-proposition" describes the effect of an activity on the truth value of a fact. a denotes an activity, whereas each of $f, p_1, \ldots, p_n$ ($n \geq 0$) is a (possibly negated) fact, and each of $e_1, \ldots, e_m$ ($m \geq 0$) is a (possibly negated) event.

Note that the *facts* can have **unk** values (i.e. are evaluated wrt 3-valued logic), but the events cannot (negation of an event indicates that it is not a part of the environment). Intuitively, the literals p_i, e_j are *preconditions* on the effects of the activity a, meaning, if a is executed when the preconditions are true then f becomes true after the execution of a. Observe that an action may have different effects on the environment, depending on the state in which it is executed (one can specify "simultaneous" effects too) similar to Vortex [32].

To represent input/output parameters of the activities, we use variables. Hence, strictly speaking, the syntax of an "ef-proposition" should look like:
$a(\overline{X_{a_1}}, \ldots, \overline{X_{a_n}})$ **causes** $f(\overline{X_f})$ **if** $p_1(\overline{X_1}), \ldots, p_n(\overline{X_n}), e_1(\overline{Y_1}), \ldots, e_m(\overline{Y_m})$

Note that if some variable in $(\overline{X_f})$ or negated literal in the preconditions, does not appear in the positive literal in the precondition, our tool will automatically warn the user of the possibility of unsafe evaluation and the need for the variable to be instantiated at the invocation time of the activity.

• An "event definition" is a proposition which describes the occurrence of an *entity of interest* during the course of execution of a workflow. Besides "book–keeping", the events are useful when enforcing *order constraints* [24] on the execution of the workflow, as we will explain shortly. Also, they are essential for inter–process communication [31]. Heterogeneous workflow tools may be integrated/synchronized by a common event notification mechanism (instead of using common API and/or fully shared database). Events are also used to denote which is the current state of the workflow. In our implementation of $\mathcal{A}_{\mathcal{W}}$ [35], upon specifying an "ef–proposition", the tool automatically generates an "event definition" signaling the end of the particular activity. We distinguish among two types of events: (i) internal events, given by the definitions, because at specification time we know which action could induce their occurrence; (ii) external events – the ones generated by other workflow agents, which notify the main workflow about the occurrence of "something of interest". However, that occurrence is not controlled by the main agent.

• A "k-proposition" stipulates that if a is executed in a state in which its preconditions are true, then in the resulting state the truth value of f becomes known. However, the value can not be predicted and it will be known only at run time. The activities in "k-proposition"s are what we have referred to as *sensing* activities (we also call them *knowledge producing activities*), and they have been studied formally in the context of planning problems [11, 28]. Their main purpose in a context of workflow specifications is to represent cooperative agents. As we will discuss shortly, we need to slightly modify the original syntax "k-propositions". to be used in workflow specification.

Just like we mentioned for the "ef-proposition", both "k – propositions" and "event definitions" need variables to represent the various input/output parameters.

Example 3 *Observe the problem of an automated paper trail involved in a students registration process, as depicted on Figure 4. Intuitivelly, in order to register for a given course, the student*

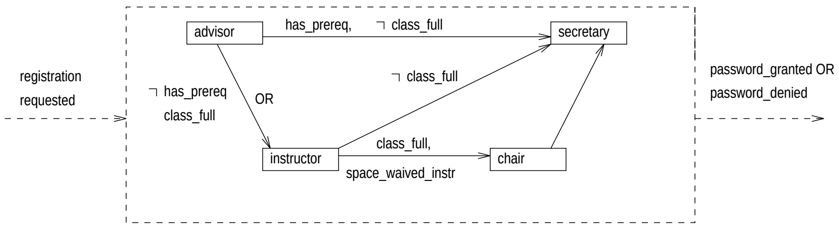


Figure 2: An example of a registration workflow

must have his advisor verify that he has the prerequisites for it and that the section is not full. If any (or both) conditions fail, the student may ask the instructor to waive the prerequsites and/or waive the space which, in turn, must be approved by the departmens's chair. Based on the outcome, the secretary determines if the student should be given a registration password or not.

We are using the simple example above so that we can concentrate on the main aspects of our formalism, without having to go into details of domain – specific issues. Below, we present a partial activity description for the registration process. The variables S and C correspond to variables from the domains of *Courses* and *Students* respectively. A particular instance of the workflow will, of course, have ground values for them from their respective domain. Recall that we have a notiont of *event*, representing an occurence of "something" of interest. In the description below, we use *e_requested(S,C)* to denote that the event of requesting a registration for the course C by the student S is present (i.e. the student has requested a registration). Throughout the rest of the work, we will use the similar notation – prefixing with *e_* the predicates which denote event type.
$advisor(S,C)$ **determines** $has\_prereq(S,C)$ **if** $e\_requested(S,C)$
$advisor(S,C)$ **determines** $class\_full(S,C)$ **if** $e\_requested(S,C)$
$instr(S,C)$ **determines** $prereq\_waived(S,C)$ **if** $\neg has\_prereq(S,C)$
$instr(S,C)$ **determines** $space\_waived\_instr(S,C)$ **if** $class\_full(S,C)$
$instr(S,C)$ **induces** $e\_seen\_instr(S,C)$
$chair(S,C)$ **determines** $space\_waived\_chair(S,C)$ **if** $class\_full(S,C)$
$chair(S,C)$ **induces** $e\_seen\_chair(S,C)$
$secretary(S,C)$ **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** $class\_full(S,C), \neg space\_waived\_instr(S,C)$
$secretary(S,C)$ **induces** $e\_secretary(S,C)$

Note, in particular, that we are not concerned with the issue of *roles* in this work. Strictly speaking, we should not be mixing an activity name (like *advisor*), with the notion of *who* (i.e. which agent) is responsible for executing it. Hence, in reality we may have an activity like
$check\_prereq(A,S,C)$ **determines** $has\_prereq(S,C)$ **if** $e\_requested(S,C), advises(A,S)$.
There is another observation regarding the *advisor in Example 3.* Although the activity is a

cooperative one, the registration workflow has no influence/impact on *how* the advisor workflow agent operates. Moreover, the main registration workflow agent is someone that is "harrasing" the advisor by asking him/her to execute an activity on its behalf (especially when the advisor has a grant–proposal or paper submission deadline). Similar to the activity *welcoming_message* in the context of Example 2 where the business partner decides to which extent it will be unfolded or kept as a black box, as far as $\mathcal{A}_{\mathcal{W}}$ is concerned, an activity like the *advisor(S,C)* in Example 3 may be a template, or a workflow itself.

2.2 Control Modules

The set of activities' description will specify the effects of executing a particular action in a particular state. However, we need to allow the users of our formalism to specify their knowledge about *how* activities execution is sequenced/coordinated in order to achieve a desired goal. Recall that during the specification stage the complete information about the environment may not be available and not all the possible execution scenarios are known. To express or define the *execution* of workflow agents, we rely on a *control module* of reactive rules. Such control modules, which are similar to production rule systems, have been used for real-time robot control [9, 44].

Definition 2 *A control module is a collection of rules of the form:*
if $e_1, \ldots, e_n, f_1, \ldots, f_k,$ **unk** $f_{k+1}, \ldots,$ **unk** f_m **then** a
where:
• *each* e_i *is an event literal (possibly negated)*
• *each* $f_i (1 \leq i \leq k)$ *is a (positive or negated) fact*
• *each* **unk** f_j *specifies a particular fact whose truth value is* unknown *at a given state.*
• a *is the action to be executed in a state in which the conjunction on the left-hand side (LHS) of the rule is true.*

The set of rules in the control module reflects the user's knowledge about the *data* and *control flow* (i.e. how to sequence the activities, what are the pre–conditions on their execution, what are the input/output parameters). Strictly speaking, the module can be in several possible states: *active/running* in which the module executes actions of the RHS of some rule which LHS conditions are satisfied in the given state of the environment; *HALT* which is reached upon a "successful termination"; *SUSPEND*, which is a state that the module has entered because it has to "wait" (e.g. waiting for some cooperative activity to terminate). However, this state of the control module illustrates the "knowledge" of the user about the particular state of the workflow environment; *failure* state of the control module illustrates the case where the workflow environment does not match any of the conditions in the LHS of any rule in the module (which, in cooperative terms could mean a "deadlock"). The difference between *failure* and *SUSPEND* is that the former illustrates a possibility which can not be "predicted" at specification stage. Note that the "state" of the control module is a part of the overall state of the workflow and it can be captured using the set of facts and events. For example, a particular instance of the module "wakes – up" (is triggered) when an external event is generated. In the context of Example 3 this happens when a particular student requests a registration, like for example, the event *e_requested(john,eecs361)* becomes true in the workflow state. We introduce the **unk** connector because agents are constantly dealing with incomplete information which they complete by making requests to other workflow agents (i.e. remote procedure calls or consults to the external world). The notion of the "unknown" value may very well be implemented with having a *null* value for a certain attribute in a database. However, during the early specification stage it is unlikely that the domain/ business expert is familiar with the meaning of *null* values (or any implementation details). This implies reasoning in 3-valued logic. In the context of the Example 2, if a new customer applies for a credit line with the on-line shopping enterprise, the goal of the workflow is to determine the eligibility for credit and the credit line. However, evaluated under 3-valued logic, the formula: *grant_credit(Customer, Limit)* $\vee\neg$*grant_credit(Customer, Limit)* need not be a tautology, which makes the goal of the workflow a non – simple one. Similarly, the registration workflow of Example 3, the goal is complex – to make the formula *passwd(S,C)* $\vee\neg$ *passwd(S,C)* true at the end of its execution. The set of propositions in the activities' domain description and the set of rules in the control module provide enough flexibilities for specifying the basic primitives of workflows (as prescribed by [21]): *sequencing* ; *AND/OR–splits and joins*; *iteration* and *nesting.*

Example 4 *A set of rules for the control module of the activities involved in the student's registration can be described as follows:*

if $e\_requested(S,C)$, **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 $has\_prereq(S,C), class\_full(S,C)$, **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 $e\_secretary(S,C), passwd(S,C)$ **then** $HALT$
if $e\_secretary(S,C), \neg passwd(S,C)$ **then** $HALT$

2.2.1 Ramifications

A description using "ef-propositions" and "k-propositions" can only describe the direct effects of executing an activity in a given state of the workflow. However, executions of actions may have indirect effects in a particular state (e.g. a database update, when executed in a particular state, may cause an integrity constraint violation). Ramification effects are usually derived from the set of facts and events, and they can be used to specify which are the valid states of the workflow evolution (as well as the valid transitions among those states). For that purpose, we introduce the following ramification propositions in $\mathcal{A}_{\mathcal{W}}$:

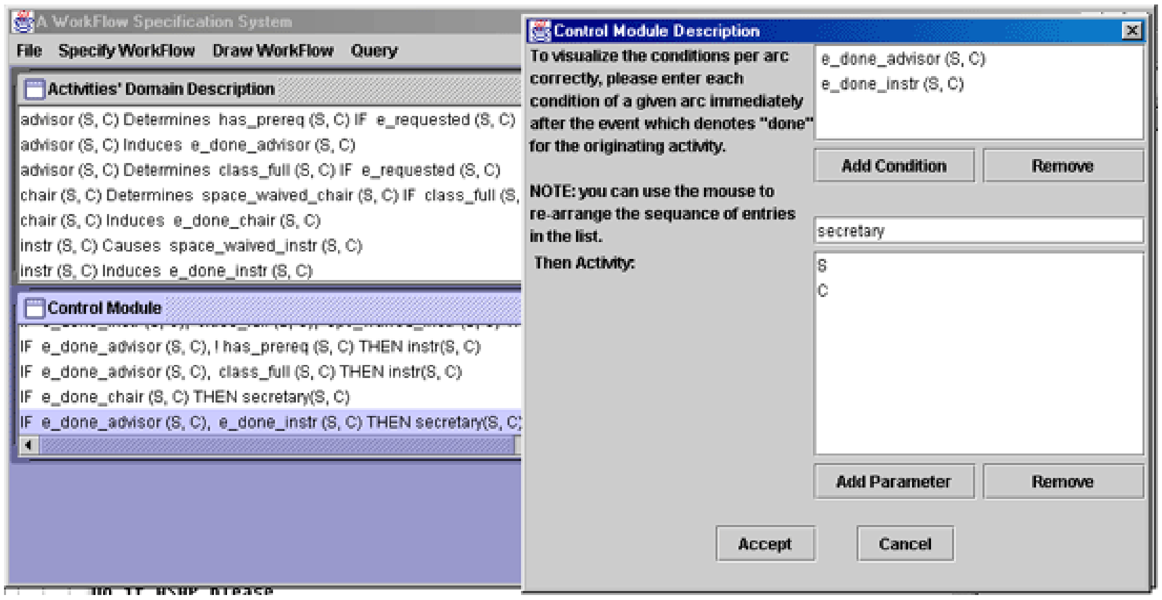
- $p_1, \ldots, p_n, e_1, \ldots e_m$ **suffice_for** l

with the intuitive meaning that in any state in which $p_1, \ldots, p_n, e_1, \ldots e_m$ are true, we can infer that l is also true. The set of ramification propositions is similar to the rules for defining intentional predicates in deductive databases [52]. Observe that l can be either a fact or an event. Although we do not address the issue of composite events (c.f. [42]) in this work, a ramification proposition can be readily used to specify a conjunction of events.

2.3 Prototype Implementation of $\mathcal{A}_{\mathcal{W}}$

For the implementation of our tool, we followed the principles of process languages design surveyed in [51], which can be summarized as follows:
– pictures help in improving intuition and communication/ cooperation;
– pictures work best when they depict modest ammount of information;
– the use of semantically deep formalism, behind the picture, may support many desirable properties.

In a nutshell, there are 3 main clasess: 1. *MainFrame.java* – the "brain" which creates the actions listeners that invoke appropriate methods to handle user input; 2. *WorkflowArea.java* which is responsible for the graphical representation of the workflow being specified (e.g. redrawing an obscured area, dragging the nodes of activities around so that they are arranged in a most appealing manner); 3. *Mediator.java* is responsible for generating an internal representation and using it when converting between textual and visual counterparts of the workflow specification.



```
advisor (S, C) Determines  has_prereq (S, C) IF  e_requested (S, C)
advisor (S, C) Induces  e_done_advisor (S, C)
advisor (S, C) Determines  class_full (S, C) IF  e_requested (S, C)
chair (S, C) Determines  space_waived_chair (S, C) IF  class_full (S,
chair (S, C) Induces  e_done_chair (S, C)
instr (S, C) Causes  space_waived_instr (S, C)
instr (S, C) Induces  e_done_instr (S, C)
```

```
IF  e_done_advisor (S, C), ! has_prereq (S, C) THEN instr(S, C)
IF  e_done_advisor (S, C),  class_full (S, C) THEN instr(S, C)
IF  e_done_chair (S, C) THEN secretary(S, C)
IF  e_done_advisor (S, C),  e_done_instr (S, C) THEN secretary(S, C
```

To visualize the conditions per arc correctly, please enter each condition of a given arc immediately after the event which denotes "done" for the originating activity.

NOTE: you can use the mouse to re-arrange the sequance of entries in the list.

```
e_done_advisor (S, C)
e_done_instr (S, C)
```

```
secretary
```

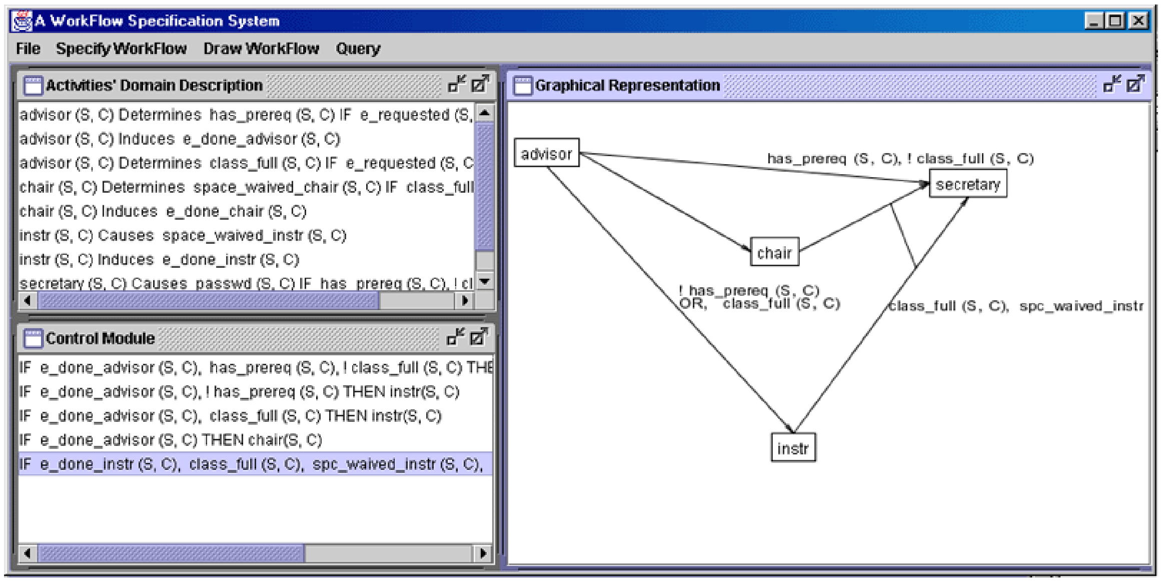
```
S
C
```

Figure 3: Specification of a rule in a control module

Our tool will cooperate and guide the users through specifications of each proposition of the language in both textual and visual representations, with a set of pulldown menus. Figure 3 illustrates the specification of a rule in a control module of a particular workflow. There is a similar menu–driven part which can be used for specifying the proposition in the activities domain description. The user has a choice of selecting if he/she wants to describe an ef–proposition, a k–proposition or an event–definition. As another example of the tool's cooperative aspects (not illustrated in Figure 3), if *unsafe* negation is used in some proposition, the user will be warned of it and reminded that the particular variable must be instantiated at the invocation time of the activity[2]. The detailed description of the functionality is specified in [35].

Figure 4 illustrates a case of AND – join in the context of the registration workflow in the Example 3. Recall that if a section is full, and both the department chair and the particular

[2]The current version of $\mathcal{A}_{\mathcal{W}}$ based prototype tool is available via anonymous ftp at *www.eecs.uic.edu/ ∼ gtrajcev/workflowtool*.



```
advisor (S, C) Determines  has_prereq (S, C) IF  e_requested (S,
advisor (S, C) Induces  e_done_advisor (S, C)
advisor (S, C) Determines  class_full (S, C) IF  e_requested (S, C
chair (S, C) Determines  space_waived_chair (S, C) IF  class_full
chair (S, C) Induces  e_done_chair (S, C)
instr (S, C) Causes  space_waived_instr (S, C)
instr (S, C) Induces  e_done_instr (S, C)
secretary (S, C) Causes  passwd (S, C) IF  has_prereq (S, C), ! cl
```

```
IF  e_done_advisor (S, C),  has_prereq (S, C), ! class_full (S, C) THE
IF  e_done_advisor (S, C), ! has_prereq (S, C) THEN instr(S, C)
IF  e_done_advisor (S, C),  class_full (S, C) THEN instr(S, C)
IF  e_done_advisor (S, C) THEN chair(S, C)
IF  e_done_instr (S, C),  class_full (S, C),  spc_waived_instr (S, C),
```

Figure 4: An example of a registration workflow

course instructor need to decide if the space should be waived, the two “activities” can be executed concurrently). The figure shows a textual of the activities domain description and the control module to the left and the visual counterpart to the right. Note that in order to see what are the effects of a parcitular node (activity) in the graph, the user may either look in the domain description or select that activity in the graph. Moreover, when drawing the graph, if a particular node is labeled with an activity which has not been “described” so far, the user is warned and (s)he is offered a menu with textual fields just like the one for the regular, textual – based specifications of the propositions in the activities’ domain description.

3 Cooperation of Activities and Exceptions

Observe the scenario where a new customer enters the virtual enterprise of Example 2. In order to determine if the customer should be granted a credit (and the limit value), it is very likely that the workflow will make a “cooperative” request to some other workflow agent representing credit history agency. All that the main workflow agent *knows* is that the request for the service of credit history check has been sent to the cooperative agent. It cannot know beforehand when the request will be actually served. However, this should not prevent the main workflow agent of the enterprise from guiding the user through the list of products and their prices/availability. Periodically, the main workflow agent may re–send the request to the credit agency workflow, until it gets a response. In case the request has not been served after some predefined time period, the workflow may decide to inform the new user that the decision about granting him/her a credit line will be determined later. The situation is more extreme in the Example 3. Here, the registration workflow cannot continue its execution until the response from the *advisor* agent has been received and the truth value of *has_prereq(S, C)* is determined.

However, in general there may be several other processes running concurrently and it is not practical to suspend the entire execution in a given state, waiting for a response from a particular (cooperative) agent. Let us reiterate the flip-side of the coin: as far as the registration workflow is concerned, the activity *advisor(S, C)* is a cooperative one, but for the advisor workflow agent, the request to act on behalf of registration workflow is an "interference". It is very likely that the advisor has his own policy on how to respond to external requests, not known to the registration workflow agent.

This is why we assume the existence of two types of events, internal and external. Namely, when the registration workflow asks the advisor to determine *has_prereq(S, C)* it generates its *internal* event so that it is "aware" that the request for service has been made. Within the world of the advisor workflow this generates an *external* event notifying him that a particular service has been requested. Once the advisor finds out if the student has the prerequisites, it may send the value back to the registration workflow. Upon that, he may record that he had completed the request (internal event). The notification to the registration workflow that its request has been served is an external event (indicating that the truth value of *has_prereq(S, C)* is known). The work presented in [31] uses *exported event list* and *imported event queue* (the authors also discuss different event revocation policies for exceptional cases).

To formalize the discussion above, we introduce the following modification of the "k-proposition" in the activity descriptions. We view the cooperative activity like the *advisor(S, C)* in the registration workflow description as if it consists of two parts:

$$advisor(S,C)^{\rightarrow} \textbf{ induces } e\_advisor\_asked(S,C) \textbf{ if } \neg e\_advisor\_responded(S,C) \tag{4}$$

$$advisor(S,C)_{\leftarrow} \textbf{ determines } has\_prereq(S,C) \textbf{ if } e\_advisor\_responded(S,C) \tag{5}$$

This, of course, needs to be properly reflected in the control module: In the particular setting of the registration example, we may want to add the following rule:

$$\textbf{if } e\_advisor\_asked(S,C), \neg e\_advisor\_responded(S,C) \textbf{ then } SUSPEND(S,C)$$

to specify that the main workflow agent should wait for the response from the advisor. Since, in general, there may be several other activities that could be executed in a given state, even though we may have a *SUSPEND* rule in the control module, the workflow instance itself need not enter that suspended state. We have the following definition:

Definition 3 *Let $A = [a_1, a_2, \ldots a_n]$ be a sequence of activities executed by a workfow agent. We call the sequence legal iff for every $a_j = action_{\leftarrow}(X)$ there exists at least one $a_i = action^{\rightarrow}(X)$ such that $i < j$.*

Observe that each a_k will have ground values for the variables, corresponding to a particular workflow instance. More over, it can also be observed that Definition 3 does not preclude a sequence of actions in which several occurrences of a particular instance $action^{\rightarrow}(m, n, \ldots, r)$ will not have a matching $action_{\leftarrow}(m, n, \ldots, r)$. In the setting of Example 2 this corresponds to the case where the main enterprise workflow agent has repeated the request for a credit history check but, for whatever reasons, it has not obtained an answer from the cooperative agent yet. Clearly, this is not the desired executional scenario and it may lead to an *exception*, formalized below.

3.1 Defeasibility and Exceptions

The workflow research community has only recently tackled the problem of exceptions [17, 18, 30]. Several types of failures have been identified (c.f. [17]): 1. basic failures (hardware/network/DBMS); 2. application failure (WMFS invokes an application which returns an error code or does not return any value). For the most part, these type of failures are handled by relying on the recovery mechanisms of the underlying DBMS. For example, *Exotica* project offers tools for specification of compensating tasks, which are subsequently translated into *FDL* (FlowMark Definition Language).

Since we are concerned in formalizing and testing the correctness of workflow specification, we are interested in *expected* and *unexpected* categories of failures [18]. Expected exceptions correspond to the executional scenarios which are not desirable but may happen (hopefully, very rarely). This approach is very similar to the standard notion of exception handling in programming languages like Smalltalk and C++.

We would like to reiterate that in a real situation the domain experts may not be aware of all the possible executional scenarios during the specification stage. This may yield some cases of unexpected exceptions, for which the main workflow agent may not know how to react. In the context of Example 3, although the agent *advisor* is considered a cooperative one, may not respond to the messages from the main (registration) workflow agent. This is an exceptional situation for which the main workflow may not know how to react (i.e. it was assumed at specification time that any particular advisor will always respond to a request from a given workflow instance). Once a workflow instance encounters an unexpected situation, there are several steps that need to be taken: 1. modify the schema (workflow definition) so that it can recognize and react to the exceptional case in the future; 2. modify the schema so that it has the rules to "repair" the current failure instances. 3. modify the schema so that the workflow instances which are "beyond repair" are aborted (and possibly re–started). These three steps correspond to managing a workflow definition with a so–called progressive case policy. Clearly, steps 2. and 3. will not be needed after the "infected" workflow instances have been managed. The modifications done in step one, however, will become propositions which remain in the schema for the future–expected exceptions. The constructs of $\mathcal{A}_{\mathcal{W}}$ that we introduced so far are sufficient to specify the repair policies in steps 2. and 3.

Hence, we must allow for some activities (i.e. the consequences of their executions) and some ramifications to be *defeasible*, in a sense that their effects to not apply in a particular state s because that state is considered to be exceptional. To handle this exceptional behavior, we extend $\mathcal{A}_{\mathcal{W}}$ by introducing exceptional causality and ramification with the following propositions:

$$p_1, \ldots, p_n, e_1, \ldots, e_m \textbf{ exceptionally\_suffice\_for } f$$
$$a \textbf{ exceptionally\_causes } f \textbf{ if } p_1, \ldots, p_n, e_1, \ldots, e_m$$
$$a \textbf{ exceptionally\_induces } e \textbf{ if } p_1, \ldots, p_n, e_1, \ldots, e_m$$
$$a \textbf{ exceptionally\_determines } f \textbf{ if } p_1, \ldots, p_n, e_1, \ldots, e_m$$

In the exceptional situations, the effects of the activities are determined by the exceptional propositions and the effects of the defeasible propositions are ignored. To illustrate the concept, in the registration example we could add:

$$advisor(S,C)_{\leftarrow} \textbf{ exceptionally\_causes } advisor\_no\_available(S,C) \textbf{ if } e\_advisor\_timeout$$

in case a student request for the advisor's response has reached a deadline marked by an event and the workflow is still waiting for the effects of the activity $advisor(S,C)$. In this situation the exceptional effect is assumed. Now we can have *corrective* actions that will execute in an environment in which *advisor_no_available(S,C)* is true. Moreover, the set of exceptional ramification propositions will induce new facts/ events that will create a new state for the workflow (say, by "cleaning" some events and facts from the state before the exception was detected).

The main benefit of extending $\mathcal{A}_{\mathcal{W}}$ with exceptional propositions is that we allow very early, in the specification stage, to separate the failure semantics and exception handling, from the control logic. The formal treatment of the defeasible and exceptional propositions is given in [6], based on the language $\mathcal{ADC}$.

4 Checking the Correctness of the Workflow Specifications

We allow two types of testing of the workflow specifications. The first type corresponds to "off–line" deliberation on the data and control flow. In this type of tests we assume that each of the experts involved in the specifications has provided (what (s)he believes is) the sufficient information about the effects of the activities. In other words, we assume that we have a "current" version of the domain description D as a set of ef–propositions, k–propositions and event–definitions.

Definition 4 *Given a domain description D and a legal sequence of activities $A = [a_1, a_2, \ldots, a_n]$, a sequence_query is an expression of the form φ **after** A **at** s, where where φ is a formula (evaluated in 3-valued logic) and s is a state.*

Given a set of states S and a domain description (action theory) D, let *S' = Closure(S,D)* denote the minimal set of states (with respect to the $\subseteq$ ordering) such that: (i) $S \subseteq S'$; (ii) $(\forall s \in S)$ if s' is reachable from s by executing legal sequence of actions described in D, then $s' \in S'$; and (iii) $(\forall s \in S)$ any state s' reached during the execution of legal sequence of actions starting at s is in S'.

The formula φ in Definition 4 will denote a *goal* that the workflow should achieve. We say that a domain description D entails a sequence_query $q \equiv \varphi$ **after** $[a_1, a_2, \ldots, a_n]$ **at** s, $(D \models q)$ if q is true in all the models of D.

Definition 5 *A domain description D is sufficient with respect to a goal G iff there exists a state s and for every $s' \in$ Closure(s,D) there exists a legal sequence of actions $A = [a_1, a_2, \ldots, a_n]$ such that $D \models G$ **after** A **at** s'.*

Definition 5 describes the means which the domain experts can use to test if they have specified enough of the domain description. Moreover, by assigning a cost value to the actions, it can be used as a tool to check the cost of executing a particular sequence of actions which achieves the desired (business) goal.

If the experts involved in the workflow specification are confident, there may be no need for any off–line deliberation: they may complete the workflow specification at a single "negotiation", both the domain description D and the control module M. In any case, once the complete workflow description is available, we have the other type of query for testing its correctness:

Definition 6 *Given an activity description D and a control module M, a workflow_query is an expression of the form φ **after** M **at** s where φ is a formula (evaluated w.r.t. the 3-valued logic) and s a state.*

Workflow_queries enquire about the consequences of executing a control module M, based on a given activities (domain) description D. Note that Definition 6, as opposed to Definition 4, implicitly requires that the legal sequence of actions consists only of the actions specified as a RHS of some rule in the control module. In order to have a definition similar to Definition 5 above, we need to slightly modify the notion of a closure. Given a set of states S, a control module M and a domain description (action theory) D, let *S' = Closure(S,M,D)* denote the minimal set of states (with respect to subset ordering) such that: (i) $S \subseteq S'$; (ii) $(\forall s \in S)$ if s' is reachable from s by executing legal sequence of actions from M described in D, then $s' \in S'$; and (iii) $(\forall s \in S)$ any state s' reached during the execution of M at s is in S' . Now we have the following:

Definition 7 *A workflow specification is said to be correct with respect to a control module M, a set of states S, a set of activities description D, and a goal G if, $S = Closure(S, M, D)$ and for all states $s \in S$, $D \models G$* **after** *M* **at** *s.*

as a tool to formally state the correctness of the workflow control module (and verify it for a given goal).

There is a special subclass of workflows for which we can do even more during the specification stage. The class of *sequential* workflows consists of all the workflows in which the set of rules in their control modules have the property that all the rules whose LHS are satisfied at a given state, have the same RHS. Informally, sequential workflows rule out the AND–joins (OR–rules do not result in concurrent execution of activities). For the class of sequential workflows we can directly apply the results in [11] to actually *generate* the control module from a given domain description and a given goal (in [11] control modules are referred as *plans*). The conditions of soundness and completeness for sequential workflows are given in [8].

5 Declarative Semantics of $\mathcal{A}_{\mathcal{W}}$

The semantics of an activity description is given by defining a (partial) transition functions Ψ called a *causal interpretation*, which specifies how the execution of (possibly empty) sequence of activities modifies a particular state $s = [\Sigma, \mathcal{E}]$ of the workflow. Here $\Sigma = \langle T, F \rangle$ where T denotes the set of facts which are known to be *true* and F denotes the set of facts which are known to be *false*. The rest of the facts are assumed to be *unknown*. On the other hand, $\mathcal{E} = \mathcal{E}_i \cup \mathcal{E}_e$ denotes the set of events (internal and external) which are known to be true in a particular state, and the rest of the events are assumed to be false.

We say that a literal f is an *effect* of executing an activity a in a state s if there is an "ef-proposition" in the domain description, of the form a **causes** f **if** $p_1 \dots p_n$ and all the p_i's are true in s. Let:
$C_a^+(s) = \{f : f \text{ is an effect of } a \text{ in } s\}$
$C_a^-(s) = \{f : \neg f \text{ is an effect of } a \text{ in } s\}$
• The *direct effects* of executing an activity a in a state $s = [\langle T, F \rangle, \mathcal{E}]$ are:
$T_d(s, a) = (T \cup C_a^+(s)) \setminus C_a^-(s)$, and
$F_d(s, a) = (F \cup C_a^-(s)) \setminus C_a^+(s)$

Similarly, an event literal e is *signaled* by executing an activity a in a state s if there is an "event-definition" in the domain description, of the form a **induces** e **if** $p_1 \dots p_n$ and all the p_i's are true in s. Let:
$E_a^+(s) = \{e : e \text{ is signaled by } a \text{ in } s\}$
$E_a^-(s) = \{e : \neg e \text{ is signaled by } a \text{ in } s\}$
• The *directly induced* events by executing an activity a in a set $s = [\langle T, F \rangle, \mathcal{E}]$ are:
$\mathcal{E}_i^d(s, a) = (\mathcal{E} \cup E_a^+(s)) \setminus E_a^-(s)$

We must take into consideration the external events too (although the main worfkow agent has no control over their appearence). Let $\mathcal{E}_e^d(s, a)$ denote the set of external events that were generated by some cooperative agent(s) for a given state of the main workflow. Now we have: $\mathcal{E}^d(s, a) = \mathcal{E}_i^d(s, a) \cup \mathcal{E}_e^d(s, a)$

Yet another thing that we have to consider is the set of ramification propositions in the domain description. They can generate some "new" facts which are true (or make some false) and also "generate" new events. Let $Ram_F^+([\Sigma, \mathcal{E}])$ (resp. $Ram_F^-([\Sigma, \mathcal{E}])$) denote all the positive (resp. negative) facts which are deduced/inferred in a given state and, similarly, let $Ram_E^+([\Sigma, \mathcal{E}])$ (resp. $Ram_E^-([\Sigma, \mathcal{E}])$) stand for the deduced events. Clearly, this may require a fixpoint computation. One last thing that we need to consider is the possibility of *exceptions*. Executing an action that will cause an exceptional effect (or induce an exceptional event) will create a state in which both defeasible and exceptional ramifications constratins may be valid. Let $Ram_F^{+EXC}([\Sigma, \mathcal{E}])$ (resp. $Ram_F^{-EXC}([\Sigma, \mathcal{E}])$) denote all the positive (resp. negative) facts deduced by an exceptional ramification rule. Also, let $Ram_E^{+EXC}([\Sigma, \mathcal{E}])$ (resp. $Ram_E^{-EXC}([\Sigma, \mathcal{E}])$) stand for the exceptionaly deduced events. Since the exceptional effects should take precedence over the regular (defeasible) ones, we have the following to describe the effects of ramification propositions in a given state:
•$R_F^+([\Sigma, \mathcal{E}]) = Ram_F^{+EXC}([\Sigma, \mathcal{E}]) \cup (Ram_F^+([\Sigma, \mathcal{E}]) \setminus Ram_F^{+EXC}([\Sigma, \mathcal{E}]))$.
•$R_F^-([\Sigma, \mathcal{E}]) = Ram_F^{-EXC}([\Sigma, \mathcal{E}]) \cup (Ram_F^-([\Sigma, \mathcal{E}]) \setminus Ram_F^{-EXC}([\Sigma, \mathcal{E}]))$
•$R_E^+([\Sigma, \mathcal{E}]) = Ram_E^{+EXC}([\Sigma, \mathcal{E}]) \cup (Ram_E^+([\Sigma, \mathcal{E}]) \setminus Ram_E^{+EXC}([\Sigma, \mathcal{E}]))$
•$R_E^-([\Sigma, \mathcal{E}]) = Ram_E^{-EXC}([\Sigma, \mathcal{E}]) \cup (Ram_E^-([\Sigma, \mathcal{E}]) \setminus Ram_E^{-EXC}([\Sigma, \mathcal{E}]))$.

Given the notation above, we have:
$Res(a, [\langle T, F\rangle, \mathcal{E}]) = [\langle T', F'\rangle, \mathcal{E}']$ where
$T' = (T_d(s,a) \cup R_F^+([\langle T_d(s,a), F_d(s,a)\rangle, \mathcal{E}^d(s,a)])) \setminus R_F^-([\langle T_d(s,a), F_d(s,a)\rangle, \mathcal{E}^d(s,a)])$
$F' = (F_d(s,a) \cup R_F^-([\langle T_d(s,a), F_d(s,a)\rangle, \mathcal{E}^d(s,a)])) \setminus R_F^+([\langle T_d(s,a), F_d(s,a)\rangle, \mathcal{E}^d(s,a)])$
and
$\mathcal{E}' = (\mathcal{E}^d(s,a) \cup R_E^+([\langle T_d(s,a), F_d(s,a)\rangle, \mathcal{E}^d(s,a)])) \setminus R_E^-([\langle T_d(s,a), F_d(s,a)\rangle, \mathcal{E}^d(s,a)])$

Observe that one can vary the preferences of insertion versus deletion by interchanging the order among the $\setminus$ and the $\cup$ operators (c.f. [27]).

For illustration, assume that a particular state of the registration workflow has $has\_prereq(s_1, c_1) \in T$ and $class\_full(s_1, c_1) \in F$. If we execute the activity $secretary(s_1, c_1)$ in this particular state we have that $Res(secretary(s_1, c_1), [\langle T, F\rangle, \mathcal{E}]) = [\langle T \cup \{passwd(s_1, c_1)\}, F \setminus \{passwd(s_1, c_1)\}\rangle, \mathcal{E} \cup \{e\_secretary(s_1, c_1)\}]$. On the other hand, for a cooperative activity such as $advisor(s_1, c_1)_{\leftarrow}$, with respect to determining if a student has the prerequisites for the course, there are two possible functions. One which adds $has\_prereq(s_1, c_1)$ to T and the other one which adds $has\_prereq(s_1, c_1)$ to F. However, both of them should leave $\Sigma = \langle T, F\rangle$ unmodified, if they are executed in a state in which $e\_advisor\_responded(s_1, c_1) \notin \mathcal{E}$. We refer to Res as transition function (similar to ones presented in [10]). Now we have:

Definition 8 *A causal interpretation Ψ is a model of a domain description D iff for any state $s = [\Sigma, \mathcal{E}]$ and a sequence of actions α*
1. If $\alpha = []$ **then** $\Psi(\alpha, s) = s$
2. If $\alpha = [a, \beta]$ **then** $\Psi(\alpha, s) = \Psi(\beta, Res(a, s))$
3. Ψ is undefined otherwise.

Given a *sequence_query* $q \equiv \varphi$ **after** A **at** s we say that it is true in a model Ψ of a domain description D iff φ is true in the state $\Psi(A, S)$. Similarly, a domain description D *entails* q ($D \models q$) iff φ holds in all the models Ψ of D.

For a complete workflow specification (both domain description and control module), the state is defined as a tripplet: $s = [\Sigma, \mathcal{E}, \mathcal{R}]$ where the meaning of Σ and $\mathcal{E}$ is as before, and $\mathcal{R}$ denotes the set of the *enabled* rules from the control module (i.e. the set of ground instances of rules whose LHS evaluates to *True* for the given Σ and $\mathcal{E}$). Now the causal interpretation Ψ_w is defined by using an additional function $Sel(\mathcal{R})$ which, in a given state selects (the action in the RHS of) one of the enabled rules, to be applied as an argument to Res function. $Sel(\mathcal{R})$ can be implemented using the "choice" operator of Zaniolo, or by a predefined priority – based policy among the rules (similar to Starburst and Chimera) [56]. Hence:

$\Psi_w(M, [\Sigma, \mathcal{E}, \mathcal{R}]) = \Psi_w(M, [(Res(Sel(\mathcal{R})[\Sigma, \mathcal{E}])), \mathcal{R}'])$,
where $Res(Sel(\mathcal{R})[\Sigma, \mathcal{E}])), \mathcal{R}][\Sigma', \mathcal{E}', \mathcal{R}']$ and $\mathcal{R}'$ consists of all the rules that are enabled for the Σ' and $\mathcal{E}'$. The termination condition for Ψ_w is that in any state s in which a rule with a RHS $HALT$ is enabled, we have that: 1. $Sel(\mathcal{R})$ will always select that rule and 2. $\Psi_w(M, [Res(HALT, [\Sigma, \mathcal{E}]), \mathcal{R}]) = [\Sigma', \mathcal{E}', \mathcal{R}']$ where $\Sigma' = \Sigma$; $\mathcal{E}' = \mathcal{E} \setminus \{e_t\}$ where e_{ti} is the event which activated the particular workflow instance; and $\mathcal{R}' = \mathcal{R} \setminus M_{ti}$ where M_{ti} is the set of ground instances of rules in the controle module, corresponding to the particular workflow instance activated by e_{ti}.

6 Related Work

There are several formalismsthat have recently been proposed for workflow representation. One of the graphic – based ones is OGWL (Opera Graphical Workflow Language[3]) [30] (which is subsequently converted to internal textual representation) similar to IBM's FDL [38]. State and Activity Charts [57] is another one, close to the UML standard, and it uses ECA rules for describing transitions among states. The works [24, 23], not concentrating too much on the visual representation, use CTR (Concurrent Transaction Logic) to represent the design and reason about properties of workflows, in presence of a rich set of constraints. [13] presents TD (Transaction Datalog) as a concurrent programming language and uses it to determine computational complexity of workflows. All these methodoligies, in a sense, do not strictly formalize the specification stage per se. It is somehow assumed that through intellectual negotiations, the representation indeed corresponds to whatever the users wanted to specify. On the other hand, we go one step behind in the workflow design, and propose a methodology to formalize the specification phase. $\mathcal{A}_{\mathcal{W}}$ for example, offers more flexibility than CTR or TD, in a sense that none of the two languages allows expressing multiple effects of a same action, or takes a clear account for exceptions.

With respect to the foundations used in the formalism, the closest approach to ours is CONGOLOG [36], where correctness of a concurrent agent language is formulated. However, our language is simpler that CONGOLOG, in a sense that we do not have non-deterministic activities. On the other hand, we consider exogenous/cooperative activities, which are not considered in CONGOLOG and are essential to model workflows.

The problem of planning with incomplete information has been addressed in the Description Logic [28], however it is too specialized formalism to be used in workflow specification. A formal approach for the workflow computation is given in [49], but the problem of requirements specification stage is not treated as a separate aspect, which is the core of our approach.

Another existing formalism which has been used as a basis for modeling and analysis of workflows is the Petri Nets. In [53], the author presents a detailed description of the dimensions/ aspects of the Worflow Management Systems and how they can be mapped into a Petri Net – based specification (actually, there is a formal definition of a *WF–net*) and presents construction of all the routing constructs of interest, like AND/ OR spits and joins, iteration, etc. A benefitial aspect of this approach is that Petri Nets (and their higher – level versions: colored, timed ...) are very well studied and have formal foundation for investigating various properties of interest to process – based

[3]Both OPERA and Mentor projects have addressed other very important problems [3, 43] in workflow execution and WFMS in general.

systems (e.g. liveness, boundedness) and there are many Petri Nets tools available. Note that in our formalism, we are much more "liberal" for the specification stage (not necessarily complete) and we also have a tool which can toggle between visual and textual representation. We do not require that the user is familiar with a specialized formalism and we allow incorporating the exceptional scenarios. Let us point that the comparison between logic programs and high – level Petri nets presented in [34] (recall that the specifications in $\mathcal{A}_{\mathcal{W}}$ can be translated to logic program).

Some recent works, addressing the issue of exceptions are [31] (Opera) and [17, 18] defining the Chimera–Exc language (in FAR system). However, the main difference with our work is that we tackle the exceptions handling in the context of specification stage, so that a domain expert can express its knowledge about exceptional situations and their handling according to a given (business) policy.

The programming paradigm Vortex [32, 33] provides a choice-based execution of "attribute centered" workflows. One of the main contributions is that the authors provide a form of incremental decision-making in collecting the values of specific attribute. We view our work as something that can be used as a pre–processor of a Vortex based design of workflows.

7 Concluding Remarks, Related Literature and Future Work

In this work we have extended previous results in formalizing reactive control using action theories [11] and applied it to cooperative workflow agents. The main advantage of the language $\mathcal{A}_{\mathcal{W}}$ is that it allows an expert to express his/her domain knowledge in terms of the effects that an activity or task may have on the environment, without any specific knowledge about *how* the activity is implemented. This is an important aspect during the specification stage, where experts from different domains are involved. We allow the users to specify the control logic AND reason if the specifications that they have provided (so far) are sufficient to achieve a desired goal. More importantly, given the description of the effects of the activities, we can help the users "generate" enough control-logic information to achieve their goal. We have implemented a prototype tool which enables both dual (textual and graphical) representations of the workflow specifications. Observe that the constructs in $\mathcal{A}_{\mathcal{W}}$ are sufficient to express all the routing constructs, as specified by the Workflow Management Coalition [21]:
1. *sequencing:* If the action a_i precedes the execution of the action a_j, we need to include $e\_a_i\_end$ in the LHS of the rule(s) which have a_j on the RHS.
2. *OR–splits:* If, depending on the effect/outcome of executing certain action a_i the workflow should take one of the possible routes, we need to include $e\_a_i\_end$ in the preconditions of every first action along every route AND ensure that the rest of the preconditions are mutualy exclusive.
3. *AND–splits:* If we can have 2 or more sequences (of activities) execute concurrently (after completion of certain a_i), then we need to ensure that the LHS of each "first–rule" in each of the sequences is the same (and contains $e\_a_i\_end$ in it).
4. *OR–joins:* If an activity a_j should follow a completion of one of the possible sequences of activities (as a result of an OR–split), we need a collection of rules with a_j on the RHS (one for each route). In the LHS of each rule we need $e\_a_i\_end$ where a_i is the last activity of a particular sequence/route.
5. *AND–joins:* If an activity a_j should wait for all the concurrently executing sequences (due to AND–split) to complete their execution, then in the preconditions of the rule with a_j on the RHS, we need to include the conjunction of all $e\_a_i\_end$ (i.e. one for each last action in each route/sequence).
6. *iteration:* the control module will execute continuously, until $HALT$ is being executed.

7. *nesting:* Any collection of activities can be grouped into a higher–level activity. As far as the outside clients are concerned, it's details will appear like a black box. All we need to know is what are the preconditions and effects of executing it in a particular state.
This is one of the salient features of $\mathcal{A}_{\mathcal{W}}$ – its modularity as a specification formalism. New activities descriptions and control module rules can be added independent of the rest of the domain description.

There are few immediate extensions of our work. At present, we are considering how we can incorporate the treatment of composite events (contexts and consumption policies). Also, we are investigating the possibility of translating between $\mathcal{A}_{\mathcal{W}}$ based specification and other workflow representation formalisms. Notably, if we could have a translation to a UML based specification, it should be straightforward to use the State and Activity Charts [57], which in turn, can be used for distributed workflow execution. Currently we are developing a collaborative version of our prototype implementation. The work in [5] shows how we can translate the set of propositions writen in $\mathcal{A}_{\mathcal{W}}$ into a generalized logic program. In this regard, we are also working on the output formatting so that it can generate a logic program that can be run on the XSB Engine [54].

Acknowldgements: The authors would like to thank Mr. Maxim Kondratyev for implementing part of our formalism in the prototype specification tool.

References

[1] G. Alonso, D. Agrawal, A. El Abadi, and C. Mohan. Functionalities and limitations of current workflow management systems. *IEEE Experts – Special Issue on Cooperative Information Systems*, 12(5), 1997.

[2] G. Alonso, U. Fiedler, C. Hagen, A. Lazcano, H. Schuldt, and N. Weiler. Wise: Business to business e-commerce. In *Research Issues on Data Engineering (RIDE)*, 1999.

[3] G. Alonso, C. Hagen, and A. Lazcano. Processes in electronic commerce. In *ICDCS Workshop on Electronic Commerce in Web-Based Applications*, 1999.

[4] C. Baral, M. Gelfond, and A. Provetti. Representing Actions: Laws, Observations and Hypothesis. *Journal of Logic Programming*, 1996.

[5] C. Baral and J. Lobo. Formal characterization of active databases. In *International Workshop on Logic in Databases (LID'96)*, 1996.

[6] C. Baral and J. Lobo. Defeasible specifications in action theories. In *Intl. Joint Conference on AI*, 1997.

[7] C. Baral, J. Lobo, and G. Trajcevski. Formal characterization of active databases: Part ii. In *5th Intl. Conf. on Deducive and Object – Oriented Databases (DOOD'97)*, 1997.

[8] C. Baral, J. Lobo, and G. Trajcevski. Formalizing workflows as collection of condition – action ruls. Technical report, UIC – EECS – 1998 – 2, Univ. of Illinois at Chicago, 1998.

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

[10] 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.

[11] C. Baral and T. Son. Relating theories of action and reactive control. In *Linkoping Electronic Articles in computer and Information Science*, volume 3. 1998.

[12] S. Bergamaschi and C. Sartori. On taxonomic reasoning in conceptual design. *ACM Transactions on Database Systems*, 3(17), 1992.

[13] A. Bonner. Workflow, transactions and datalog. In *Principles of Database Systems (PODS)*, 1999.

[14] A. Bonner and M. Kifer. Transaction logic programming (or a logic of declarative and procedural knowledge). Technical report, Univ. of Toronto, 1995.

[15] A. Bonner and M. Kifer. Concurrency and communication in transaction logic. In *Joint Intl. Conference and Symposium on Logic Programming*, September 1996.

[16] U.M. Borghoff, P. Bottoni, P. Mussio, and R. Parechi. Reflective agents for adaptive workflows. In *2nd Intl. Conf. on Practical Applications of Intelligent Agents and Multi – Agent Technology (PAAM'97)*, April 1997.

[17] F. Casati. *Models, Semantics and Formal Methods for the Design of Workflows and their Exceptions.* PhD thesis, Politecnico di Milano, 1999.

[18] F. Casati and G. Pozzi. Modeling exceptional behavior in commercial workflow management systems. In *4th Intl. Conf. on Cooperative Information Systems (CoopIS)*, 1999.

[19] F. Cassati, S. Ceri, B. Pernici, and G. Pozzi. Deriving active rules for workflow enactment. In *7th Intl. Conf. on Database and Expert Systems Application*, 1996.

[20] A. Cichocki and M. Rusinkiewicz. Migrating workflows. In *NATO–ASI, Advances in Workflow Management Systems and Interoperability.* 1997.

[21] The Workflow Management Coalition. Terminology and glossary. Technical Report WFMC-TC-1011, The Workflow Management Coalition, June 1996.

[22] P. Cousot. Methods and logics for proving programs. In *Handbook of theoretical computer science*, volume B, pages 841–994. MIT Press, 1990.

[23] H. Davulcu, M. Kifer, R.L. Pokorny, C. Ramakrishnan, and S. Dawson. Modeling and analysis of interactions in virtual enterprises. In *Research Issues on Data Engineering (RIDE)*, 1999.

[24] H. Davulcu, M. Kifer, C.R. Ramakrishnan, and I.V Ramakrishnan. Logic based modeling and analysis of workflows. In *ACM Principles of Database Systems*, 1998.

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

[26] 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.

[27] S. Ghandeharizadeh, R. Hull, and D. Jacobs. Heraclitus: Elevating deltas to be a first–class citizens in database programming language. Technical Report USC–CS–94–581, Univ. of S.C., 1995. revised 1995.

[28] G. De Giacomo, L. Iocchi, D. Nardi, and R. Rosati. Descriptoin logic-based framework for planning with sensing action. In *Description Logic*, 1997.

[29] E. Giunchiglia and V. Lifschitz. An action language based on causal logic. In *AAAI – 98*, 1997.

[30] C. Hagen and G. Alonso. Flexible exception handling in the opera process support system. In *18th Intl. Conf. on Distributed Computing Systems (ICDCS 98)*, April 1998.

[31] C. Hagen and G. Alonso. Beyond the black box: Event-base inter-process communication in pss. In *19 International Conferrence on Distributed Computing Systems (ICDCS)*, 1999.

[32] R. Hull, F. Llirbat, E. Simon, J. Su, G. Dong, B. Kumar, and G. Zhou. Declarative workflows that support easy modifications and dynamic browsing. In *Intl. Joint Conference on Work Activities Coordination and Collaboration (WACC)*, 1999.

[33] R. Hull, F. Llirbat, J. Su, G. Dong, B. Kumar, and G. Zhou. Efficient support for decision flows in e–commerce applications. In *2nd Intl. Conf. on Telecommunications and Electronic Commerce (ICTEC)*, 1999.

[34] J. Jeffrey, J. Lobo, and T. Murata. A high – level petri net for goal – directed semantics of horn clause logic. *IEEE Transactions on Knowledge and Data Engineering*, 8(2), 1996.

[35] M. Kondratyev. Dual representation of workflow specifications. Master's Project Report, March 2000. University of Illinois at Chicago.

[36] Y. Lesperance, H. Levesque, F. Lin, D. Marcu, R. Reiter, and R. Scherl. Foundations of a logical approach to agent programming. In *Intelligent Agents - II*. 1995.

[37] H. Levesque, R. Reiter, Y. Lesperance, F. Lin, and R. Scherl. Golog: A logic programming language for dynamic domains. *Journal of Logic Programming*, May 1997.

[38] F. Leymann and W. Altenhuber. Managing business processes as an information resource. *IBM Systems Journal*, 2(32), 1994.

[39] J. McCarthy and P. Hayes. Some philosophical problems from the standpoint of artificial intelligence. In *Machine Intelligence*, volume 4, pages 463–501. Edinburgh University Press, 1969.

[40] B. Mitschang, T. Harder, and N. Ritter. Design management in concord: Combining transaction management, workflow management and cooperation control. In *6th Intl. Workshop RIDE-NDS*, 1996.

[41] C. Mohan. Tutorial: State of the art in workflow management system research and products, March 1996.

[42] I. Motakis and C. Zaniolo. Formal semantics for composite temporal events in active database rules. *JOSI*, pages 1–37, 1997.

[43] P. Muth, J. Weissenfels, M. Gillman, and G. Weikum. Integrating light-weight wfms with existing business environments. In *International Conferrence on Data Engineering (ICDE)*, 1999.

[44] N. Nilsson. Teleo-reactive programs for agent control. *Journal of AI research*, pages 139–158, 1994.

[45] R. Reiter. On specifying database updates. *Journal of Logic Programming*, 19,20:1–39, 1994.

[46] N. Ritter. An infrastructure for cooperative applications based on conventional database transactions. In *CSCW Infrastructure Workshop*, 1994.

[47] 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.

[48] A. Sheth, D. Georgakopoulos, S.M.M. Joosten, M. Rusinkiewics, W. Scacchi, J. Wileden, and A. Wolf. Report from the nsf workshop on workflow and process automation in information systems. *ACM SIGSOFT – Software Engineering Notes*, 22(1), 1997.

[49] M.P. Singh. Formal semantics for workflow computations. Technical report, TR – 96 – 08, North Carolina State University, 1996.

[50] I. Sommerville. *Software Engineering*. Addison – Wesley, 1992.

[51] S. M. Sutton, P. L. Tarr, and L. J. Osterweil. An analysis of process languages. Technical Report 95 – 78, Dept. of Computer Science, University of Massachusetts, Amherst, 1995.

[52] J. D. Ullman. *Principles of Database and Knwoledge – Base Systems*. Computer Science Press, 1989.

[53] W.M.P van der Aalst. The application of petri nets to workflow management. *The Journal of Circuits, Systems and Computers*, 8(1), 1998.

[54] D. S. Warren. Programming in tabled prolog (draft). Department of CS, Suny at Stony Brook, July 1999.

[55] B. Whorf. *Language, thought and reality*. MIT Press, 1956.

[56] J. Widom. The starburst active database rule system. *IEEE Transactions on Data and Knowledge Engineering*, 8(4), 1996.

[57] D. Wodtke and G. Weikum. A formal foundation for distributed workflow execution based on state and activity charts. In *6th Intl. Conf. on Database Theory (ICDT 97)*, 1997.