Formalizing (and Reasoning About) the Specifications of Workflows

Goce Trajcevski[1] and Chitta Baral[2] and Jorge Lobo[3]

[1] Univ. of Illinois at Chicago, Dept. of EECS, gtrajcev@eecs.uic.edu
[2] Arizona State Univ., Dept. of CSE, chitta@asu.edu
[3] Bell Labs, jlobo@research.bell-labs.com

Abstract. We address the problem of *workflow requirements specifications* under realistic assumptions that it involves experts from different domains (different business policies), where not all the possible execution scenarios are known beforehand. Using recent results on *reasoning about actions*, we formalize the notion of the specifications' correctness. To address this, we propose a high level language $\mathcal{A}_{\mathcal{W}}$ as a basis of our prototype tool for process specification. We go "one step" before actual analysis and design, and offer a formalism which enables the experts to specify their knowledge in terms of the effects that the activities have on the workflow environment. Our methodology allows expressing not only the knowledge, but also the "ignorance" (the semantics allows unknown values to reflect a realistic situation of agents dealing with incomplete information) and the possibility of exceptional situations. We define an entailment relation which enables reasoning about the correctness of the specifications, in terms of achieving a desired goal, and testing about consequences of modifications in the workflow descriptions.

1 Introduction and Motivation

A *workflow* (WF) is a *process* which executes various cooperative and coordinated *activities* in order to achieve a desired *goal* [22]. Workflow Management Systems (WFMS) provide tools for modeling, executing and monitoring workflows [27, 40] and they need well – defined correctness/reliability criteria [1, 17, 44] and the ability to adapt to changes in a flexible manner. Recent works have addressed several of these issues [20, 25, 30, 32, 41, 50] and identified solutions of many problems of interest in WFMS. Among the other contributions, several formalisms have been proposed (e.g. OGWL (Opera Graphical Workflow Language) [30]; State and Activity Charts [50]; Concurrent Transaction Logic (CTR) [25]; Transaction Datalog (TD) [14]) which enable representation, reasoning and execution of workflows.

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 [13, 46, 47]. This is an important aspect in workflow/ process specification in many applications (e.g. Virtual Enterprises) [2, 24]. The specification stage involves experts from different domains, representatives of different organizations and policies (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* ...") [49]. Typically, not all the possible executional scenarios are known; participants need not know the details of how other partners are implementing certain (business) policies. Also, a participant may not be willing to fully reveal a 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 extent it should be "opened"). When it comes to cooperation and synchronization, a particular domain expert typically states the "weakest preconditions" and the "strongest effects" expected.

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

1.1 An Overview of Our Approach

In this work, we view workflows as a collection of cooperative agents and 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 [43] and transactions [15, 16].

Our main contributions 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 visual) 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* (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 situations of dealing with incomplete information (essential during dynamic modifications of a workflow/construction of *ad-hoc* workflows); – we give 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, we have a formal notion of a *correctness.* of a workflow description. The rest of the paper is structured as follows. Section 2 introduces the basis of our formalism, describes the basic aspects of the language $\mathcal{A}_{\mathcal{W}}$ and our current 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. In Section 5 we summarize, compare with the relevant literature 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 $\mathcal{A}$ [26] (the description of the language is based on $\mathcal{A}_K$ in [11]; $\mathcal{L}_{active}$ in [7]; 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]). The later use makes it more appropriate than the traditional approach of program correctness [23] which has been designed for standard programming languages and lacks the flexibility of defining new operators/ activities. 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 [21]. 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".

For the implementation of our tool, we followed the principles of process languages design surveyed in [47](e.g.: – pictures help in improving intuition and communication/ cooperation; – they work best when they depict modest ammount of information; – the use of semantically deep formalism supports many desirable properties). 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 to convert between textual and visual parts of the specification.

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 1 illustrates the specification of a rule in a control module (defined below) of a particular workflow[4]. The detailed description of the functionality is specified in [35].

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

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

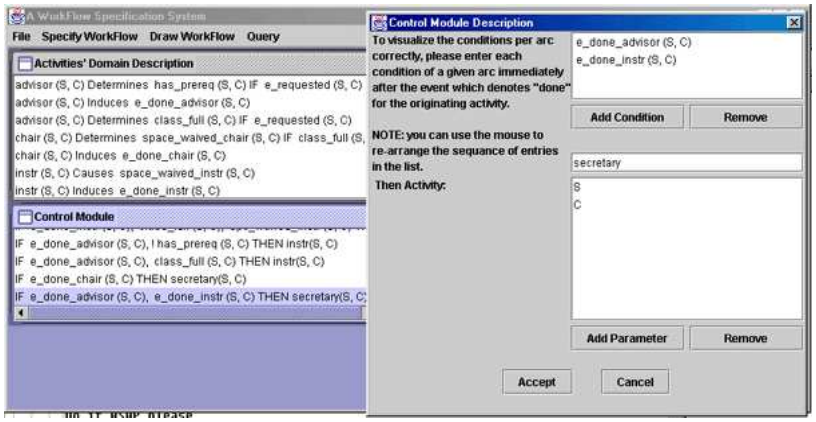

Fig. 1. Specification of a workflow

context of reasoning about actions. A *literal* is a fact or event, possibly preceded by $\neg$.

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

a **causes** f **if** $p_1, \ldots, p_n, e_1, \ldots, e_m$ (an "ef-proposition")
a **determines** f **if** $p_1, \ldots, p_n, e_1, \ldots, e_m$ (a "k-proposition")
a **induces** e **if** $p_1, \ldots, p_n, e_1, \ldots, e_m$ (an "event definition")

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

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].

Example 1. 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 (simplified from [33]) reveals a subtle issue. When specifying the workflow, all that the business partners need to know is that there is 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.

To represent input/output parameters of the activities, we use variables. Hence, strictly speaking, the syntax of an "ef-proposition" looks like:
$a(\overline{X_{a_1}}, \ldots, \overline{X_{a_n}})$ **causes** $f(\overline{X_f})$ **if** $p_1(\overline{X_1}), \ldots, p_n(\overline{X_n}), e_1(\overline{Y_1}), \ldots, e_m(\overline{Y_m})$
If some variable in $(\overline{X_f})$ or negated literal in the preconditions, does not appear in the positive literal in the precondition (*unsafe* negation), 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. Events are useful when enforcing *order constraints* [25] on the execution of the workflow, as we'll 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 them; (ii) external events – generated by other workflow agents, which notify the main workflow about the occurrence of "something of interest" which 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 [12, 28]. However, as we will discuss shortly, we need to slightly modify the "k-propositions" if they are to be used in workflow specification.

Example 2. Observe a simple workflow which automates a course registration process of a graduate student. Some of the activities description are:
$advisor(S, C)$ **determines** $has\_prereq(S, C)$ **if** $e\_requested(S, C)$
$advisor(S, C)$ **determines** $class\_full(S, C)$ **if** $e\_requested(S, C)$
which illustrate that it is the advisor's responsibility to check if the student is eligible to register.

Once everything is completed, the secretary should grant the registration password of a particular course to the student:
$secretary(S, C)$ **causes** $passwd(S, C)$ **if** $has\_prereq(S, C), \neg class\_full(S, C)$

Of course, the effect of the activity "secretary" may be different, depending on the status of the environment in which it is executed. For example, if the student does not have the prerequisites and the course instructor did not agree to waive them, then the secretary will deny the registration password to the

student:
$secretary(S, C)$ **causes** $\neg passwd(S, C)$ **if**
$\neg has\_prereq(S, C), \neg prereq\_waived(S, C)$

In the example[5] above, although the activity *advisor* 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 deadline). Similar to activity *welcoming_message* in the context of Example 1 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 2 may be a template, or a workflow itself.

2.2 Control Modules

The set of activities' description specifies the effects of executing a particular action in a particular state. But we need to allow the users of our tool to specify their knowledge about *how* activities execution is coordinated in order to achieve a desired goal. Recall that in the specification stage the complete information about the environment may not be available and not all the possible execution scenarios are known. To define the *execution* of workflow agents, we rely on a *control module* of reactive rules. Such control modules, similar to production rule systems, have been used for real-time robot control [10, 42].

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 reflects the user's knowledge about the *data* and *control flow* (i.e. how to sequence the activities, what are pre–conditions on their execution; the input/output parameters). The module can be in several possible states: *active/running* in which it 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. termination of some cooperative activity). 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 condition in the LHS of any rule in the module

[5] The complete specification of the registration workflow is given in [9].

(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. 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. Formally, for a given workflow specification (domain description of each activity and the control module), the state is a triplet $s = [\Sigma, \mathcal{E}, \mathcal{R}]$ where: $\Sigma = \langle T, F \rangle$ 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*); $\mathcal{E} = \mathcal{E}_i \cup \mathcal{E}_e$ denotes the set of events (internal and external) which are known to be true in a particular state; and $\mathcal{R}$ 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 $\mathcal{E}$). The semantics is given by defining a (partial) transition function Ψ_w which specifies how the state changes by executing rules from the control module (see [9] for a complete specification of the semantics of $\mathcal{A}_{\mathcal{W}}$).

For example, a particular instance of the module "wakes – up" (is triggered) when an external event is generated. In the context of Example 2 this happens when a particular student requests a registration (e.g. 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 that they complete by making requests to other workflow agents (i.e. remote procedure calls or consults to the external world). The "unknown" value may very well be implemented with having a *null* value for an attribute in a database. However, 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 1, 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. The set of propositions in the activities' domain description and the set of rules in the control module provide enough flexibilities for specifying workflows (as prescribed by [22]: *sequencing* ; *AND/OR–splits and joins*; *iteration nesting*).

Figure 2 illustrates a case of AND – join in context of Example 2 (if a section is full, and both the department head and the particular course instructor decide if the space should be waived, the two "activities" can be executed concurrently).

Ramifications A description using "ef-propositions" and "k-propositions" can only describe the direct effects of an activity in a given state. However, executions of actions may have indirect effects in a particular state (e.g. a database update 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). Thus, we introduce the following ramification propositions

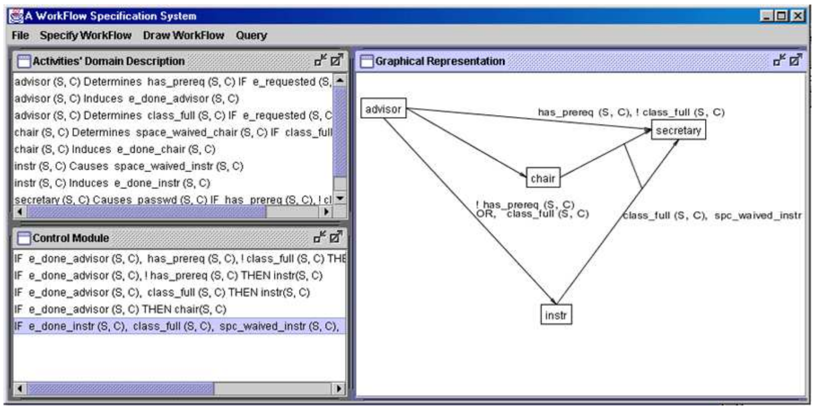

Fig. 2. An example of a registration workflow

in $\mathcal{A}_{\mathcal{W}}$:
• $p_1, \ldots, p_n, e_1, \ldots e_m$ **suffice_for** f – 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 f is also true. Observe that f can be either a fact or an event. Although we do not address the issue of composite events in this work, a ramification proposition can be readily used to specify a conjunction of events.

3 Cooperation of Activities and Exceptions

Assume that a new customer enters the virtual enterprise of Example 1. In order to determine if (s)he should be granted a credit (and the limit value), it is very likely that the workflow will make a "cooperative" request to another workflow agent – credit history agency. All that the main workflow agent *knows* is that the request for the service of credit history check has been sent. 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 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 2. 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 cooperative agent. Let us reiterate the flip-side of the coin: although the activity *advisor(S,C)* is a cooperative for the main workflow, for the advisor workflow agent the request to act on behalf of registration workflow is an "interference". It is 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 main 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 of a particular service being 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 records that he had completed the request (internal event). The notification to the registration workflow that its request has been served is an external event (the truth value of *has_prereq(S,C)* is known). The work in [31] uses *exported event list* and *imported event queue*[6].

To formalize the discussion above, we modify the "k-proposition" in the activity descriptions as follows. The cooperative activity like the *advisor(S,C)* in the registration workflow description consists of two parts[7]:
$advisor(S,C)^{\rightarrow}$ **induces** $e\_advisor\_asked(S,C)$ **if** $\neg e\_advisor\_responded(S,C)$
$advisor(S,C)_{\leftarrow}$ **determines** $has\_prereq(S,C)$ **if** $e\_advisor\_responded(S,C)$
In the particular setting of the registration example, we may want to add the following rule to the control module:
if $e\_advisor\_asked(S,C), \neg e\_advisor\_responded(S,C)$ **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 particular state. We have the following definition:

Definition 1. 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. Also note that Definition 1 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 1 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.

[6] The authors also discuss different event revocation policies for exceptional cases.
[7] This, of course, would be appropriately reflected in the specification of the rules in the control module

3.1 Defeasibility and Exceptions

The workflow research community has only recently tackled the problem of exceptions [18, 19, 30] and identified several types of failures (c.f. [18]): 1. basic (hardware/network/DBMS); 2. application failures (WMFS invokes an application which returns an error code or does not return any value). For the most part, these types of failures are handled by relying on the recovery mechanisms of the underlying DBMS (e.g. *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 [19]. Expected exceptions correspond to the executions which are not desirable but may happen (hopefully, very rarely). This approach is similar to the standard exception handling in programming languages like Smalltalk and C++.

Let us reiterate that in a real situation the domain experts may not be aware of all the possible executional scenarios during the specification. 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 2, the agent *advisor* is considered a cooperative one, but it 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 assumed at specification time that any advisor will always respond to a request from a 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 recognizes and reacts to the exceptional case in the future; 2. modify the schema with rules to "repair" the current failure instances. 3. modify the schema so that the workflow instances "beyond repair" are aborted (possibly re–started). These 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 are 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:

$p_1, \ldots, p_n, e_1, \ldots, e_m$ **exceptionally_suffice_for** f
a **exceptionally_causes** f **if** $p_1, \ldots, p_n, e_1, \ldots, e_m$
a **exceptionally_induces** e **if** $p_1, \ldots, p_n, e_1, \ldots, e_m$
a **exceptionally_determines** f **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:

$advisor(S,C)_{\leftarrow}$ **exceptionally_causes** $advisor\_no\_available(S,C)$
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 can create a new state for the workflow (say, by "cleaning" some events and facts from the state before the exception was detected). Although it is not obvious from the preceding discussion, there is a noteworthy observation regarding the flexibility of the extended $\mathcal{A}_{\mathcal{W}}$. Namely when iterating, say, over a collection of elements, if one element is exceptional we need not terminate the entire iteration. The repair policy (exceptional ramification propositions) may specify a state in which that particular cycle of the iteration is ignored, but yet the conditions for iterating over the rest of the collection are still valid (i.e. why throwing the entire bucket of apples if we can remove the rotten one). 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. Here 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 a "current" version of the domain description D as a set of ef–propositions, k–propositions and event–definitions.

Definition 2. 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) $(\forall s' \in S')(\exists s \in S)$ such that s' is reachable from s by executing legal sequence of actions described in D.

The formula φ in Definition 2 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 (c.f. [9]).

Definition 3. 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 3 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 4. 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, for a given activities (domain) description D. Definition 4, as opposed to Definition 2, 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 3 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) $(\forall s' \in S')(\exists s \in S)$ such that s' is reachable form s by executing legal sequence of actions from M described in D.

Definition 5. 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 [12] to actually *generate* the control module from a given domain description and a given goal[8]. (in [12] control modules are referred as *plans*).

[8] The conditions of soundness and completeness for sequential workflows are given in [8].

5 Concluding Remarks, Related Literature and Future Work

In this work we have extended previous results in formalizing reactive control using action theories [12] 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. Also note that the semantics of $\mathcal{A}_\mathcal{W}$ incorporates the events as a part of the state unlike, for example, process algebra – based approaches where transitions are modeled explicitly and the states in between consecutive transtions are implicit (c.f. [48]). 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.

There are several formalisms for workflow representation. Graphic – based ones like OGWL (Opera Graphical Workflow Language) [30] (which is subsequently converted to internal textual representation) similar to IBM's FDL [38] and State and Activity Charts [50] (which uses ECA rules for describing transitions among states), which is close to the UML standard[9]. The works [25, 24] use CTR (Concurrent Transaction Logic) to design and reason about workflows (in presence of a rich set of constraints) and [14] presents TD (Transaction Datalog) as a concurrent programming language and uses it to determine computational complexity of workflows. All these formalisms, 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 what 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.

The closest approach to ours is CONGOLOG [36], where correctness of a concurrent agent language is formulated. Our language is simpler that CONGOLOG – we do not have non-deterministic activities, but 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 also 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 [45], but the problem of requirements

[9] Both OPERA and Mentor projects have addressed other very important problems [3, 41].

specification stage is not treated as a separate aspect, which is the core of our approach.

Another formal approach to modeling and analysis of workflows is based on Petri Nets [48]. 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 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 [18, 19] defining the Chimera–Exc language (in FAR system). However, the main difference in 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. 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 Vortex based design of workflows.

Observe that one of the salient features of $\mathcal{A}_{\mathcal{W}}$ is 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. We would like to formalize the treatment of composite events (contexts and consumption policies) and we are planning to investigate the possibility of translating between $\mathcal{A}_{\mathcal{W}}$ based specification and other workflow representation formalisms. Also, we are looking into making the visual part of our tool close/ compatible with the existing visual modeling tools, like UML. Currently, we are incorporating the unexpected exception handling and full correctness analysis in the prototype implementation, and we are developing a collaborative version of it.
Acknowldgements: The authors would like to thank Mr. Maxim Kondratyev for implementing part of our formalism in the prototype specification tool. We are grateful to the anonymous referees for their valuable comments and suggestions.

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, J. Lobo, and G. Trajcevski. Formalizing (and reasoning about) the specification of workflows. Technical Report UIC–EECS–00–3, University of Illinois at Chicago, Dept. of EECS, 2000.
10. 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.
11. 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.
12. C. Baral and T. Son. Relating theories of action and reactive control. In *Linkoping Electronic Articles in computer and Information Science*, volume 3. 1998.
13. S. Bergamaschi and C. Sartori. On taxonomic reasoning in conceptual design. *ACM Transactions on Database Systems*, 3(17), 1992.
14. A. Bonner. Workflow, transactions and datalog. In *Principles of Database Systems (PODS)*, 1999.
15. A. Bonner and M. Kifer. Transaction logic programming (or a logic of declarative and procedural knowledge). Technical report, Univ. of Toronto, 1995.
16. A. Bonner and M. Kifer. Concurrency and communication in transaction logic. In *Joint Intl. Conference and Symposium on Logic Programming*, September 1996.
17. 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.
18. F. Casati. *Models, Semantics and Formal Methods for the Design of Workflows and their Exceptions.* PhD thesis, Politecnico di Milano, 1999.
19. F. Casati and G. Pozzi. Modeling exceptional behavior in commercial workflow management systems. In *4th Intl. Conf. on Cooperative Information Systems (CoopIS)*, 1999.

20. 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.
21. A. Cichocki and M. Rusinkiewicz. Migrating workflows. In *NATO–ASI, Advances in Workflow Management Systems and Interoperability*. 1997.
22. The Workflow Management Coalition. Terminology and glossary. Technical Report WFMC-TC-1011, The Workflow Management Coalition, June 1996.
23. P. Cousot. Methods and logics for proving programs. In *Handbook of theoretical computer science*, volume B, pages 841–994. MIT Press, 1990.
24. 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.
25. 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.
26. M. Gelfond and V. Lifschitz. Representing action and change by logic programs. *Journal of Logic Programming*, 17:301–321, 1993.
27. 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.
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. C. Mohan. Tutorial: State of the art in workflow management system research and products, March 1996.
41. 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.
42. N. Nilsson. Teleo-reactive programs for agent control. *Journal of AI research*, pages 139–158, 1994.
43. R. Reiter. On specifying database updates. *Journal of Logic Programming*, 19,20:1–39, 1994.
44. 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.
45. M.P. Singh. Formal semantics for workflow computations. Technical report, TR – 96 – 08, North Carolina State University, 1996.
46. I. Sommerville. *Software Engineering.* Addison – Wesley, 1992.
47. 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.
48. W.M.P van der Aalst. The application of petri nets to workflow management. *The Journal of Circuits, Systems and Computers*, 8(1), 1998.
49. B. Whorf. *Language, thought and reality.* MIT Press, 1956.
50. D. Wodtke and G. Weikum. A formal foundation for distributed workflow execution based on state charts. In *6th Intl. Conf. on Database Theory (ICDT 97)*, 1997.

This article was processed using the LaTeX macro package with LLNCS style