Specification of Workflow Processes Using the Action Description Language C

Pinar Koksal, Nihan Kesim Cicekli, I. Hakki Toroslu

Department of Computer Engineering,
M.E.T.U., Ankara, Turkey
{koksal, nihan, toroslu}@ceng.metu.edu.tr

Abstract

We propose the use of the action description language C to formally specify workflow processes. The Workflow Management Coalition (WfMC) describes a set of all possible transitions among the activities in a workflow. We show how these transitions can be specified within the language of C. CCALC, the causal calculator, is used to run the examples and to generate the execution plans. In addition we also show how agents, duration and the cost of the workflow activities can be incorporated to the current framework.

1. Introduction

Workflow management is a fast evolving technology that has been exploited in a variety of industries. Its primary characteristic is the automation of processes involving combinations of human and machine-based activities, particularly those involving interaction with IT applications and tools. There are many commercial products to model and execute workflows but the current state of the art lacks clear theoretical basis, defined correctness criteria, support for consistency of concurrent workflows and support for reliability in the presence of failures and exceptions [8]. A formal definition is also required to prevent interpretation ambiguities and to enable formal reasoning about the model.

There have been several proposals to provide a formal framework to model workflow processes [2,3,7,11]. In this paper, we propose to use action description languages in the specification of workflows [5]. The contribution of this study is twofold: First, we aim to show the use of the action description language C in a realistic application. Second, the action description language can be used as a tool for the formal specification of workflow processes. The expressive power of the language provides us with a high-level abstraction of the workflow processes. It can also be used in generating execution plans for workflow instances.

Our work is closely related to [1] in the sense that we also use an action description language. The language A_k used in [1] is based on the language A of Gelfond et al. [4] which is a predecessor of the language C. C extends A by incorporating features to model non-deterministic and concurrent actions. These features can be very helpful in the specification of workflow processes. Therefore we have chosen C to be our framework.

The paper is organized as follows. Section 2 summarizes the transitions required to define a workflow process, defined by Workflow Management Coalition (WfMC). Section 3 gives a brief introduction to action description language C and its computational environment CCALC. The translation from workflow process definition to the action description language is presented with examples in Section 4. The programming environment CCALC is used to run the examples and to generate the plans. Section 5 presents some extensions to the specifications. We discuss how agents can be incorporated into the proposed framework. We also present the modelling of workflow activities with durations. Finally we discuss the calculation of the total cost of a workflow execution. Section 6 presents an example of a workflow process definition and its formal specification in C. We conclude the paper in Section 7 by discussing the future work.

2. Workflow Process Definition by WfMC

Workflow Management Coalition (WfMC) defines a 'reference model' which describes the major components and interfaces within a workflow architecture [8,12]. In a workflow, activities are related to one another via flow control conditions (transition information). According to this reference model we identify four routings among the activities:

1. *Sequential*: Activities are executed in sequence (i.e. one activity is followed by the next activity.)

2. *Join:* Two building blocks are identified: (a) AND-join and (b) XOR-join. The AND-join synchronizes the parallel flows, one activity starts only after all activities in the join have been completed. In XOR-join no synchronization is required. The activity at the

join can start when any one of the incoming activities has been completed.

3. *Split:* The AND-split enables two or more activities to be executed concurrently after another activity has been completed. If the transitions have conditions the actual number of executed parallel activities depends on the conditions associated with each transition which are evaluated concurrently. In *XOR-SPLIT* the decision as to which single transition route is selected depends on the conditions of each individual transition as they are evaluated in the sequence specified in the list.

4. *Loop*: It may sometimes be necessary to execute an activity or a set of activities multiple times.

In the rest of the paper, we discuss how these features of a workflow management system can be modeled in the framework of the language `C`. We first give a brief summary of the action description language `C`.

3. Action Description Language C

The action description language `C`, is first defined in [6], based on the theory of causal explanation proposed by McCain and Turner [10]. Non-deterministic actions and concurrent execution of actions can be conveniently described in `C`. Inertia is not a 'built-in feature' in the semantics of `C`; the user is free to decide for each fluent whether or not to postulate inertia for it. The propositions of the action language `C` are classified into static laws and dynamic laws. Static laws are of the form

caused F **if** G

and dynamic laws are of the form

caused F **if** G **after** H

where, F and G stand for propositional combinations of fluent names and H is propositional combination of the fluent names and action names. The syntax of the language `C` includes some short and convenient notation in order to specify some frequently used expressions. Here we give a summary of only some these expressions. An expression of the form

U causes F if G

where *U* is a propositional combination of elementary action names and *F*, *G* are state formulas, stands for the dynamic law

caused F if True after G Λ U.

An expression of the form

nonexecutable U if F

where *U* is a propositional combination of elementary action names and *F* is a state formula, stands for the dynamic law

caused False after F Λ U.

The Causal Calculator [9], CCALC, is a model checker for the language of causal theories. Using rewrite rules, propositions in `C` can be translated into the language of causal theories. Therefore `C` is used as an input language for query answering and planning with the Causal Calculator. Finding a satisfying interpretation shows that there exists a causally possible world history in which the plan is executed (in the initial state of the interpretation) and the goal is achieved. It does not, however, guarantee the plan is valid, i.e., that the plan will necessarily achieve the goal if the plan is executed in the initial state of the interpretation, or that it can be so executed.

4. Workflow Process Specification with C

Any workflow process can be defined using the transitions defined by WfMC, given in Section 2. In this section, we give the translations of these transitions into propositions in `C`. In order to define a workflow process using `C` propositions, we should first decide about the *constants*, *actions* and *fluents* of the system. Each activity in the workflow process is represented by a constant name (say, A_i). The action which denotes the execution of an activity is represented by: *execute(activity)*. In addition we define the following (inertial) fluents:

- *commit(activity)* : It is false initially. It becomes true after the workflow activity commits (i.e. after the action `execute(activity)` is done).
- *cond(activity)* : This fluent is used to define the necessary conditions in AND-SPLIT, XOR-SPLIT, and LOOP transitions.
- *count(number)* : It is used as a loop counter in the specification of loop transitions.

4.1 Sequential Transition

In this transition, actions are executed in sequential order. After a workflow activity commits, the next activity in the specification can start execution. The activity diagram of a sequential transition is given in Figure 1.

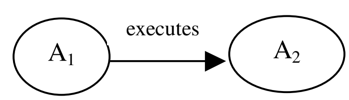

Figure 1. Activity Diagram

In Ccalc, states are described by the values of the fluents. The execution of an action causes a change from one state to another. The following state diagram in Figure 2, shows the states of the example sequential transition whose activity diagram is given above. This transition can be defined in `C` as:

execute(A_1) causes commit(A_1).
execute(A_2) causes commit(A_2)
if commit(A_1).
nonexecutable execute(act1)
if commit(act1).

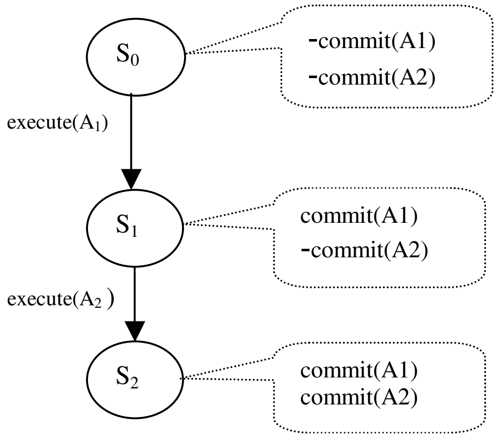

Figure 2. State Diagram

where `act1` is a variable of `activity` sort and A_1 and A_2 are the activities given in the example. With these rules, we represent that, the activity A_2 can be executed only if the activity A_1 commits. In addition, we ensure that once the activity A_1 or A_2 commit, they are not executed again.

4.2 AND-SPLIT transition

In the AND-SPLIT transition after an activity commits, one or more activities can start executing concurrently. Their execution depends on the truth value of their associated conditions. Here, we need to use the concurrent actions in modeling the execution of concurrent activities.

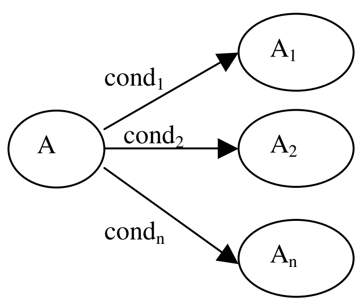

Figure 3: AND-SPLIT transition

As shown in Figure 3, when the activity A commits, the activities (A_1, A_2, …, A_n) for which the associated condition holds, are executed concurrently in the next state. This transition is represented by n rules of the form:

```
execute(Aj) causes commit(Aj)
  if commit(A) && cond(Aj).
```

One such rule is written for every activity A_j such that $1 \leq j \leq n$. The other rule of the sequential transition definition (nonexecutable) remains the same.

4.3 XOR-SPLIT Transition

XOR-split transitions are similar to the AND-SPLIT transitions graphically, but only one of the alternative activities is executed. In order to represent such transitions, the following rules are added to the AND_SPLIT transition definition:

```
nonexecutable execute(A1) &&
   execute(A2) && .. && execute(An).
nonexecutable execute(act1)
  if commit(act2) && -(act2=A).
```

where `act1` and `act2` are variables of `activity` sort. With the first rule, we state that, although there may be more than one activity for which the associated condition is true, one and only one of them can execute in the same state. With the second rule we say that, if any one of the activities A_1 … A_n is committed in one state, the other activities cannot be executed in the next states.

4.4 AND-JOIN Transitions

The execution of the activity at the join depends on the synchronization of the preceeding concurrently running activities A_1…A_n (Figure 4). The activity A can be executed only after all of them have committed.

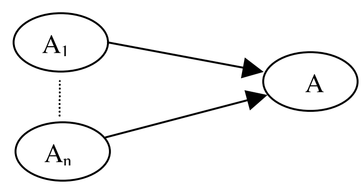

Figure 4. AND-JOIN transition

This transition can be described in C as follows:

```
execute(A) causes commit(A)
  if commit(A1) && .. && commit(An).
```

4.5 XOR-JOIN Transitions

The difference between an AND-JOIN and XOR-JOIN is that the activity at the join does not need to wait for the commit of all incoming activities. Instead, it can be executed if any one of them commits:

```
execute(A) causes commit(A)
  if commit(Aj).
```

One such rule is written for every activity Aj such that $1 \leq j \leq n$. This set of rules represents that if any one of the activities A_1 … A_n has committed, the activity A can execute.

```
nonexecutable execute(act1)
  if commit(act2) && -(act1=A).
```

where `act1` and `act2` are variables of *activity* sort. With this rule we state that, if any one of the activities A_1 through A_n is committed in one state; in the next states, the other activities, other than the activity `A`, cannot start execution.

4.6 LOOP Transitions

In a LOOP transition, we can include any one of the transitions described so far. Here we give an example of a LOOP that includes sequential transitions only.

In the LOOP transition, the execution of the activities in the loop is repeated until the loop condition is false. The value of the condition can be changed by the execution of activities in the loop, or by changing the value of a counter. In Figure 5, an iteration of a set of sequentially ordered activities is illustrated. Assume that the loop is executed for a given number of times.

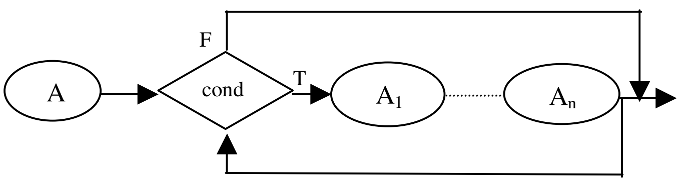

Figure 5. A LOOP transition

The modeling of A_1 through A_n is similar to the sequential transition. However, since this sequential transition appears in a LOOP transition, the activities A_1 to A_n, should be executed several times. In order to achieve this, the execution of the last activity of the LOOP transition, A_n, causes the `commit` fluents to be false again. We describe this behaviour by the following rule:

```
execute(An)causes -commit(act1)&&count(J)
  if count(I) && sum(J,I,1) && -(A=act1).
```

where `sum(i, j, k)` is a macro definition that calculates `i = j + k`.

The rule:

```
caused -count(I) if count(J) && -(I=J).
```

is to state that only one of the `count(number)` fluents can be true any time.

5. Extensions to the Specifications

In this section we discuss several possible extensions to the specification of a workflow within the current framework. These extensions include agent assignment to the activities, modeling activities with duration and calculation of the execution cost of a workflow.

5.1 Agents

When there are agents, each qualified to execute some or all of the workflow activities, additional rules must be included to describe their behaviour. The following requirements should be supported:

- Activities can be executed only by those agents that are qualified to do so.
- The agents can run activities if they are not occupied by another activity. In other words, one agent can execute only one activity at a time.
- In an AND-SPLIT transition, although it is desired, the activities in the split may not be executed concurrently if all the qualified agents are occupied. This may happen when the number of qualified agents is less than the number of activities in the split.

To support these requirements, the action `execute(activity)` is modified as:

```
execute(agent, activity) :: action
```

to explicitly specify which agent executes the activity. Also, the fluent:

```
qualified(agent, activity)::
                         inertialFluent
```

is added to describe whether the agent is qualified to execute the activity or not.

As an example, a sequential transition including three activities and three agents can be defined in CCALC as the following:

```
execute(ag1, A1) causes commit(A1)
  if qualified(ag1, A1).

execute(ag1, A2) causes commit(A2)
  if qualified(ag1, A2) &&
     commit(A1).

execute(ag1, A3) causes commit(A3)
  if qualified(ag1, A3) &&
     commit(A2).
```

One activity (`act1`) cannot be executed by more than one agent at the same time:

```
nonexecutable execute(ag1, act1) &&
  execute(ag2, act1) if -(ag1 = ag2).
```

One agent (`ag1`) cannot execute more than one activity at the same time:

```
nonexecutable execute(ag1, act1) &&
  execute(ag1, act2) if -(act1 = act2).
```

One activity cannot be executed again once it commits:

```
nonexecutable execute(ag1, act1) if
  commit(act1).
```

In addition to these rules, the facts to specify which agents are qualified to execute which activities needs to be included to the specification.

5.2 Activities with Duration

We have so far modeled workflow activities as instantaneous actions. However, in real world the activities have durations for their execution. Thus we need to formalize different states of activities. In other words we need to be able to model an activity to be executing, committed, not started, etc. In order to model activities with duration, we introduce the action *start(agent, activity)* in addition to the action *execute(agent, activity),* four fluents: *executing(agent, activity, number1), committed (activity), duration(agent, activity, number1)* and *free(agent)*. We add the following rules:

```
start(ag1, A1) causes
  executing(ag1, A1, 0) && -free(ag1)if
  qualified(ag1, A1) && free(ag1).

execute(ag1, A1) causes
  executing(ag1, A1, I) if
  executing(ag1, A1, J) && sum1(I, J, 1)
  && duration(ag1, A1, K)&&(K > I).

execute(ag1, A1) causes
  committed(ag1, A1) && free(ag1) if
  executing(ag1, A1, J) && sum1(I, J, 1)
  && duration(ag1, A1, K)&&(K=I).
```

An action `execute(agent, act)` causes an increment on the execution time of the activity (third argument of the fluent `executing(ag1, act, I)` denotes the current execution time).

5.3 Cost of a Workflow Execution

Each workflow activity has an execution cost (e.g. time) that sums up to the total cost of the workflow execution. In order to calculate the total cost of executing a workflow instance, we add the fluents, *totalcost(number), cost(agent, activity, number)* to our framework and we modify the rules above as follows:

```
execute(ag1,act1) causes
  committed(act1) && totalcost(I)
  if qualified(ag1, act1) &&totalcost(K)
  && cost(ag1, act1, J) && sum(I, K, J).
```

We are planning to use this calculation to find the plan with minimum cost.

6. An Example Workflow Definition

In this section, we give an example workflow and describe how it is translated to the action description language `C`. Figure 6 shows the example workflow process which is to be executed by two agents X and Y. The first activity in the workflow is A1. When this activity commits, the next activity A2 executes. After A2 commits, activities A3 and A4 run concurrently. When both A3 and A4 commit, activities A5 and A6 execute in a sequential manner. Then the workflow process terminates. The figure shows the activity diagram and the qualified agents for each activity.

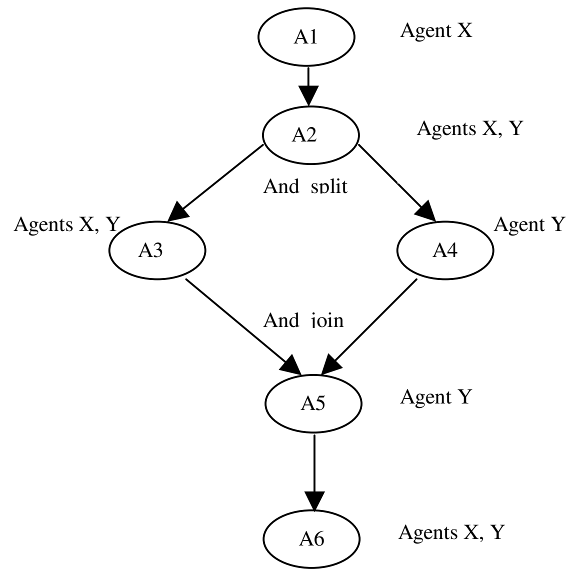

Figure 6. An example workflow process

The CCALC definition of this workflow process is given as follows:

```
start(agent1, A1) causes
  executing(agent1, A1, 1) &&
  -free(agent1) if
  qualified(agent1,A1) &&
  free(agent1).

start(agent1, A2) causes
  executing(agent1, A2, 1) &&
  -free(agent1) if
  qualified(agent1, A2) &&
  free(agent1) &&
  committed(A1).

start(agent1, A3) causes
  executing(agent1, A3, 1) &&
  -free(agent1) if
  qualified(agent1, A3) &&
  free(agent1) && committed(A2).
```

```
start(agent1, A4) causes
  executing(agent1, A4, 1) &&
  -free(agent1) if
  qualified(agent1, A4) &&
  free(agent1) &&
  committed(A2).

start(agent1, A5) causes
  executing(agent1, A5, 1) &&
  -free(agent1) if
  qualified(agent1, A5) &&
  free(agent1) &&
 committed(A3) && committed(A4).

start(agent1, A6) causes
  executing(agent1, A6, 1) &&
  -free(agent1) if
  qualified(agent1, A6) &&
  free(agent1) &&
 committed(A5).

execute(agent1, activity1) causes
   executing(agent1, activity1, I)
   if executing(agent1, activity1, J) &&
   sum(I,J,1) &&
   duration(agent1, activity1, K) &&
   (K>I).

execute(agent1, activity1) causes
   committed(activity1) &&
   free(agent1) && totalcost(L)
   if executing(agent1, activity1, J) &&
   sum(I, J, 1) &&
   duration(agent1, activity1, K) &&
   (K = I) && totalcost(M) &&
   cost(agent1, activity1, N) &&
   sum(L, M, N).

nonexecutable start(agent1, activity1)
   if executing(agent2, activity1).

nonexecutable start(agent1, activity1)
   if committed(activity1).

nonexecutable execute(agent1,activity1)
   if committed(activity1).

nonexecutable start(agent1, activity1)
   && start(agent2, activity1) if
   -(agent1=agent2).

caused -executing(agent1,activity1,I)
   if executing(agent1,act1,J) &&
   -(I=J).

caused -executing(agent1,activity1,I)
   if committed(act1).

caused -totalcost(I2) if totalcost(J2)
&& -(I2=J2).
```

7. Conclusion and Future Work

In this paper we have presented our preliminary results on the use of the action description language C in the specification of workflow processes. We have implemented all the rules presented in this paper using CCALC and observed that the action description language C can be used as a specification tool to model simple workflow processes. It is also observed that CCALC generates correct plans for the execution of a workflow instance. However, CCALC uses satisfiability solvers, *sato* [13] or *rel-sat*, which do not generate alternative plans. On the other hand we need to be able to generate alternative workflow executions in order to do verification of a workflow specification. Therefore *'smodels'* must be studied further to see whether any modifications can be made to satisfy our needs.

We plan to continue on this topic in several directions including the problem areas like constraint programming, optimum schedule development, verification of the specifications, developing a high level workflow specification language on top of C and a visual programming environment. Among these research directions we have already started to work on the development of the high-level workflow specification language and visualization of transition systems.

References

[1] Baral, C., Lobo, J., 'Formalizing Workflows as Cooperative Agents', Dynamics '97, Workshop in ILPS, 1997.

[2] Cicekli-Kesim, N. A Temporal Reasoning Approach to Model Workflow Activities, Lecture Notes in Computer Science, no. 1649, ed. R.Y. Pinter and S. Tsur, NGITS'99, Zikhron-Yaakov, Israel, July 1999.

[3] Davulcu, H., M. Kifer, C.R. Ramakrishnan, and I.V. Ramakrishnan. Logic Based Modeling and Analysis of Workflows. In *ACM Symposium on Principles of Database Systems*; Seattle, Washington, ACM Press, 1998.

[4] Gelfond, M., Lifschitz, V., 'Representing Action and Change by Logic Programs', Journal of Logic Programming, Vol.17, pp. 301-321, 1993.

[5] Gelfond, M., Lifschitz, V., 'Action Languages', in Electronic Transitions on AI, Vol.3. No:16, 1998.

[6] Giunchiglia, E., Lifschitz, V., 'An Action Language Based on Causal Explanation: Preliminary Report', Proc. of AAAI-98, pp. 623-630, 1998.

[7] Harel, D. StateCharts: A Visual Formalism for Complex Systems. Science of Computer Programming, 8:231-274,1987.

[8] Hollingsworth, D., 'Workflow Management Coalition – The Workflow Reference Model', Document No: TC00-1003, January 1995.

[9] McCain, N., 'Using the Causal Calculator with the **C** Input Language', 1999.

[10] McCain, N., Turner, H., 'Causal Theories of Action and Change', Proc. of AAAI-97, pp. 460-465, 1997.

[11] Singh, M., Meredith, G., Tomlinson, C., Attie, P., 'An Event Algebra for Specifying and Scheduling Workflows', 4th Intl. Conf. On Database Systems for Advances Applications (DASFAA), Singapore, April 1995.

[12] WfMC, 'Workflow Management Coalition – Interface 1 : Process Definition Interchange Process Model', Document No: TC-1016-P, November 1998.

[13] Zhang, H., 'SATO: An Efficient Propositional Prover', Proc. of the Intl. Conf. on Automated Deduction (CADE-97), 1997.