A Prolog-based language for workflow programming

Steve Gregory and Martha Paschali

Department of Computer Science
University of Bristol, BS8 1UB, England

steve@cs.bris.ac.uk

Abstract

Workflow management systems control *activities* that are performed in a distributed manner by a number of human or automated *participants*. There is a wide variety of workflow systems in use, mostly commercial products, and no standard language has been defined in which to express workflow specifications. In this paper we propose Workflow Prolog, a new extension of Prolog. The language allows workflow systems to be implemented in a novel declarative style, while preserving the existing properties of Prolog, such as its familiarity and efficiency. We then demonstrate the expressiveness of the language by showing how it can express each of the *workflow patterns* that have previously been identified as the requirements of a workflow language.

1. Introduction

1.1 Workflow languages

Workflow management systems (WFMSs) automate the flow of tasks, information, and data between people or other entities within an organization. A WFMS is controlled by a description of the business processes to be automated, where a *process* comprises a number of primitive *activities* performed by *participants*. Activities may be *manual* (performed by human participants) or *automated* (performed by software, etc.). Like processes in computer systems, a business process can have many *instances*, which run concurrently and might need to synchronize and communicate with each other. Differences from computational processes are that (a) the activities are generally outside the control of the WFMS, and (b) business processes might "run" for a very long time.

In a WFMS, a business process is described by a *process definition* or *workflow specification*. In general, this describes the control flow (the order of activities in the process), the data flow (how documents, etc., are passed between participants), the resources needed for each activity, and possibly the detailed actions in each activity. The process definition is written in a *workflow language*.

There are many different WFMSs, most of which are commercial products. A sample of 15 of them have been evaluated by van der Aalst *et al* in [1]. There is also a wide variety of workflow languages: most products use a different proprietary workflow language, and independent workflow languages have also been proposed, for example [2, 4, 25]. These

often provide a graphical syntax and sometimes have an underlying formal semantics based on (for example) Petri nets. The languages differ fundamentally in major ways.

Although there is no standard workflow language, there have been some useful attempts to analyse the various existing languages and identify common characteristics. One is the standardization effort by the Workflow Management Coalition [28, 29]. Another is the survey in [1], which identifies 20 *patterns* that abstract the control flow features found in the various languages and lists which languages support each pattern. The authors observed that none of the WFMSs surveyed supports all of the patterns. See also [30]. These workflow patterns are a good guide to the kind of features that a workflow language should support, and so we return to them in Section 4.

1.2 Logic for workflow

Many workflow languages are based on a formal framework, the most popular of which is probably Petri nets or some high-level variant; for example [2, 18]. However, a few systems have been proposed that are based on some form of logic.

Bi and Zhao [3] use propositional logic to model workflow problems and present a "process inference" method for verifying them.

Davulcu *et al*. [10] propose Concurrent Transaction Logic as a vehicle for specifying workflow problems and for verifying and scheduling them. This was extended by Senkul *et al*. [23] to handle resource constraints, using Concurrent Constraint Transaction Logic.

Pokorny and Ramakrishnan [22] argue that Generalized Linear Temporal Logic is a more suitable basis for workflow than transaction logic because of its ability to express temporal and fairness properties. Ma [19] also proposes a system for modelling workflow using linear temporal logic, and extends it in [12] to provide "flexibility": the ability to change a specification over time. Wang and Fan [27] propose the use of Lamport's Temporal Logic of Actions (TLA) for modelling and analysing workflow, and show how it can be used to prove the absence of deadlock.

All of the above use logic to model and reason about workflow, and they have much in common with other work on modelling and reasoning about concurrent and distributed systems. In contrast, we are interested a logic *programming* approach: i.e., using logic to *implement* workflow management systems. In principle, this should offer a high-level declarative way to develop such systems. For this, we need a logic programming language that can conveniently express workflow problems.

Concurrent logic programming languages (e.g., [6, 24]) have long been used to implement concurrent applications, so they are obvious candidates for workflow programming in logic. However, they have the following drawbacks with respect to workflow:

- A WFMS needs to interact with activities outside the system, which are performed by external participants: either humans or existing programs. This need is not addressed by concurrent logic programming languages.
- Communication by incrementally binding logical variables — a key feature of concurrent logic programming — is not a natural way to communicate with participants. In general, it requires shared memory.
- Processes in a workflow management system are persistent, in that they might run for days or even years, but a concurrent logic program runs (at best) only as long as its host machine is switched on.

Prolog systems with coroutining provide some of the expressive power of concurrent logic programming languages, and have additional advantages. However, they suffer from the same drawbacks as concurrent logic programming languages.

1.3 Workflow Prolog

We propose a new logic programming language, Workflow Prolog, which is designed to enable the implementation of workflow management systems in logic. Key features are:

- Workflow Prolog is an extension of Prolog, not a completely new language. A program deviates from Prolog only where necessary for workflow purposes. This means that users do not need to learn a new language, and can benefit from the availability of efficient and well-supported Prolog implementations.
- A *workflow variable*: an asynchronous bidirectional communication variable, allowing communication with workflow participants. This is used together with an asynchronous RPC (remote predicate call), which represents the effect of a workflow activity in the form of a goal: a relation on a workflow variable.
- A coroutining mechanism, using guards, to receive results from activities on workflow variables and suspend when necessary. Deep and nested guards are possible, for expressiveness.
- Time constraints, for a declarative treatment of time.
- A garbage collector that automatically cancels workflow activities.
- Persistence, to handle long-lived workflow processes.

1.4 This paper

In Section 2 we describe the Workflow Prolog language, justify its design, and illustrate its use with example programs. Section 3 discusses some of the key implementation issues. In Section 4 we attempt to demonstrate the expressiveness of Workflow Prolog by showing how each of the workflow patterns of van der Aalst *et al.* [1] can be expressed in the language. Conclusions and related work are described in Section 5.

2. Workflow Prolog

2.1 Overview

We view a WFMS (or "workflow system") as one component of a distributed system which comprises the workflow system and an arbitrary number of participants. A workflow programming language is needed to implement the workflow system but not the participants, which in general may be existing applications or even humans. Therefore, a workflow language need not be a complete distributed programming language; it just needs the ability to create and manage concurrent (business) processes and assign activities to participants.

Workflow Prolog is a workflow programming language based on logic. There are already hundreds of logic programming languages, most of which have been little used, so one of our design aims has been to avoid designing a completely new language. Instead, we have added to an existing language (Prolog) only the features necessary for workflow

programming. Workflow Prolog is essentially the same as Prolog but with a few extensions, which are described and justified below.

2.2 Workflow variables

An *activity* is the smallest unit of work scheduled by a workflow system as part of a process [29]. After being invoked by the WFMS, the activity is performed independently by a participant; when completed, the participant returns the activity's results to the workflow system. We assume that there is no communication between the participant and the workflow system while performing the activity.

A workflow language therefore needs a way to:

- invoke an activity, identified by the name of the participant,
- pass input data to the activity,
- wait for the activity to complete,
- receive output data when the activity is complete.

In general, this seems to require some kind of (bidirectional) message passing feature. In traditional concurrent logic programming languages, message passing is achieved by incrementally instantiating variables to lists [13], but we reject this method because:

- It is more general than needed. For example, messages are allowed to contain variables, and variables may be incrementally bound to structures other than lists, such as trees. Moreover, stream communication is not even needed, because each activity is defined to receive only one message and send one reply.
- In its general form, it requires shared memory, which is not available between the workflow system and participants.
- All variables are potentially communication channels, so the language would need special unification or matching features that allow goals to suspend on variables.

Another method which has been included in logic programming languages, as well as others, is to add explicit "send" and "receive" message primitives or Linda-like blackboard primitives. But these are more general than we need for workflow purposes, and sometimes have no declarative meaning.

Instead, in Workflow Prolog we introduce the concept of a *workflow variable*. This acts as an asynchronous bidirectional communication channel between a Workflow Prolog program and a workflow participant. A workflow variable contains:

- *Destination*: the identity of the participant.
- *Message content*: data to be sent to the participant.
- *Message time*: absolute time when message was sent to the participant.
- *Reply content*: data returned by the participant.
- *Reply time*: absolute time when reply was sent by the participant.

As usual in logic programming languages, workflow variables are single-assignment: each component is set only once. The destination and message content are assigned by the Workflow Prolog program; the reply content is assigned by the participant; the message time and reply time are assigned by the implementation. The advantage of this method is that each workflow variable captures all available information about an activity.

2.3 Remote predicate call

In imperative languages, such as Ada, bidirectional message passing is sometimes achieved using a procedure call syntax, in a remote procedure call (RPC). Input parameters of a procedure call are sent in a message to a server and output parameters are taken from the reply when it is received. The RPC is usually synchronous: it returns only after the reply is received.

In Workflow Prolog, also for bidirectional communication, we introduce an *asynchronous* form of RPC (remote predicate call):

```
rpc(Dest, MC, WV)
```

This:

- creates a new workflow variable `WV`,
- sets the destination of `WV` to `Dest`,
- sets the message content of `WV` to `MC`,
- sends a message containing `MC` to `Dest`,
- sets the message time of `WV` to the time at which the message was sent.

The `rpc` goal succeeds immediately after sending the message. Later, when the participant replies, the reply content and reply time can be extracted from `WV`, as explained in Sections 2.4 and 2.7.

The `rpc` goal is used in Workflow Prolog to invoke an activity and pass data to it. To wait for the activity to be complete, and retrieve results from it, the program needs to wait for the reply from the participant on the workflow variable; see Section 2.4.

An advantage of this method is that the `rpc` goal (representing an activity) has a declarative reading as a relation between the activity's input and output data, and possibly the times of these. E.g., if `M` is a person assessing a project proposal `P`, then `rpc(M,P,WV)` might mean ("`M` approves `P`" $\wedge$ `WV.reply=accept`) $\vee$ ("`M` rejects `P`" $\wedge$ `WV.reply=reject`).

Program 1 shows part of an example program to receive and check project proposals submitted by students. The query creates a process to solicit a proposal from a student named `zz1234` and have it checked by two "markers" named `smith` and `jones`. The `rpc` goal here represents an activity `send_proposal` (writing a proposal) performed by participant `zz1234`. When `zz1234` completes the activity, the proposal will be available as the reply on workflow variable `P`, for checking by the markers.

```
:- proposal(zz1234, send_proposal, [smith,jones]).

proposal(Student, Request, Markers) :-
  rpc(Student, Request, P),
  check(P, Student, Request, Markers).
```

Program 1: Proposal/approval example

Restrictions. In `rpc(Dest,MC,WV)`, `Dest` and `MC` must be ground and `WV` unbound. Because the effect of `rpc` is immediate, it is not allowed to appear in a context that might fail: it must appear at the Prolog top level (i.e., there must be no choicepoints) and it must not appear in a guard (see Section 2.4).

2.4 Suspendable goals and guards

Our remote predicate call is asynchronous, to allow activities to be active concurrently. Therefore, a Workflow Prolog program needs the ability to wait until an activity completes, and to get the results from it. To do this we introduce a special coroutining mechanism.

Coroutining is a long established feature of Prolog (e.g., [7]) and is available in modern Prolog systems, such as Sicstus and SWI-Prolog. In these Prologs, a goal can be suspended until a specified variable is bound, ground, equal to another variable, or a nested conjunction or disjunction of those conditions; this is controlled by the use of `freeze` and `when` built-in predicates.

In Workflow Prolog, goals need to suspend, but only on workflow variables, not regular variables. Instead of using features like Prolog's `freeze` or `when` predicates to determine which variables to suspend on, we use guards [11], which are now a standard feature of concurrent programming languages: they were used in CSP [16] and were introduced into logic programming by Clark and Gregory [5].

In Workflow Prolog, goals are executed sequentially from left to right with the exception of *suspendable goals*, which might suspend on workflow variables. A suspendable goal is a goal for a suspendable predicate, which is either:

- A built-in suspendable predicate:

  ```
  replycontent(WV, RC)
  ```

  which suspends on workflow variable `WV` if it has no reply yet, and otherwise succeeds, unifying `RC` with the reply content of `WV`. There are also some other suspendable primitive goals, described in Section 2.7.

- A user-defined suspendable predicate. This is a predicate defined by clauses at least one of which has a non-empty guard.

  A clause with a guard `G` and body `B` is written `H :- G : B.` where `G` and `B` are both Prolog conjunctions. If a guard is empty, the clause is written `H :- true : B.` or simply `H :- B.`

If all clauses for a predicate have empty guards, the predicate is not considered suspendable. If a suspendable predicate only has guards that never suspend, the '`:`' operators could be replaced by '`!`' and it would become a (non-suspendable) Prolog predicate.

For a suspendable goal, Workflow Prolog searches for a clause by executing each clause's guard in turn. Each guard may succeed, fail, or block (on a set of workflow variables). A guard *blocks* on a set of variables *Vs* if it contains a suspendable goal that is blocked on *Vs*; this means that the guard cannot succeed or fail until at least one of the variables in *Vs* has a reply value. (If `replycontent` is the only suspendable goal used in guards, each guard will block on at most one variable.)

If a successful guard is found, the goal *commits* by reducing to the corresponding body. If all guards fail, the goal fails. If no guard succeeds but one or more block, the goal suspends on the union of the sets of variables on which these guards are blocked.

If a goal *suspends* on some set of workflow variables *Vs*, it is delayed until there is a reply on one of these variables from the corresponding participant, and the computation proceeds with the next goal to the right. The work done in searching for a clause will be repeated later when the goal is woken (see Section 2.5).

Program 2 defines the `check` predicate used in Program 1. This is a suspendable predicate because its clauses contain guards. If `check` is called before a reply arrives on workflow variable `P`, the goal will suspend on `P` because both guards block on `P`. Otherwise,

the guards will extract the content of the reply, `Proposal`, and check whether it is valid. If so, the `approve` stage will be performed; otherwise, the `proposal` stage will be repeated, invoking a new `send_proposal` activity.

```
check(P, Student, _, Markers) :-
  replycontent(P, Proposal), valid(Proposal) :
  approve(Proposal, Student, Markers).
check(P, Student, Request, Markers) :-
  replycontent(P, Proposal), \+ valid(Proposal) :
  proposal(Student, Request, Markers).
```

Program 2: Proposal/approval example (contd.)

Program 3 defines the `approve` predicate (not suspendable). Its `rpc` goals invoke two concurrent `approve_proposal` activities, one performed by each marker. The `decide` goal (suspendable) waits for the markers to reply. Initially, the goal `decide([R1,R2],…)` suspends on both `R1` and `R2` because its first two clauses are each blocked on `R1` and its third clause is blocked on `R2`. If *both* markers reply '`ok`', the proposal is accepted (first clause). However, if *one* marker replies with comments (second or third clause), the comments and proposal are sent to the student to `revise`. In this case, the other marker's reply is ignored.

```
approve(Proposal, Student, Markers) :-
  rpcs(Markers, approve_proposal(Proposal), Rs),
  decide(Rs, Markers, Proposal, Student).

rpcs([], _, []).
rpcs([Dest|Dests], Message, [WV|WVs]) :-
  rpc(Dest, Message, WV),
  rpcs(Dests, Message, WVs).

decide([R1,R2], _, Proposal, Student) :-
  replycontent(R1, ok),
  replycontent(R2, ok) :
  rpc(database, accepted(Student,Proposal), D),
  replycontent(D, ok).
decide([R1,_], [M1,M2], Proposal, Student) :-
  replycontent(R1, c(C)) :
  proposal(Student, revise_proposal(Proposal,C), [M1,M2]).
decide([_,R2], [M1,M2], Proposal, Student) :-
  replycontent(R2, c(C)) :
  proposal(Student, revise_proposal(Proposal,C), [M1,M2]).
```

Program 3: Proposal/approval example (contd.)

Restrictions. All suspendable goals must be ground unless they appear in a guard, to avoid problems with dependent goals on the right. Suspendable goals in guards need not be ground because, if they block, goals to the right are not executed. For example, a `replycontent` goal can only occur in a guard unless its second parameter is ground. Suspendable goals must always appear at the Prolog top level (i.e., there must be no choicepoints). This prevents premature commitment, in the case of a cut following a suspended goal.

2.5 Persistence and waking goals

As described in Section 2.4, each top-level goal in a Workflow Prolog query may succeed, fail, or suspend. If a goal fails, the query fails. If all goals succeed, the query succeeds. If some goals suspend, the query will terminate with goals suspended. Each suspended goal may be suspended on one or more workflow variables; each workflow variable may have one or more goals suspended on it.

Goals are never woken up during a query, because it is impossible for a Workflow Prolog program to give a workflow variable a value. They are woken only when participants reply on workflow variables as the result of completing an activity. Because activities typically take days or years to complete, Workflow Prolog features *persistence*: when a query terminates, normally with suspended goals, the query's state is saved in a file for later use. The state includes (at least):

- all suspended goals (because of the language restriction, these must be ground);
- the workflow variables on which each goal is suspended;
- the goals that are suspended on each workflow variable;
- the message time (for all workflow variables);
- the reply content (for workflow variables that have been replied to);
- the reply time (for workflow variables that have been replied to).

When a participant replies on a workflow variable `WV`, meaning that an activity has completed:

- The state is read from the state file.
- The reply content and reply time of `WV` are stored in the state.
- The goals suspended on `WV` are removed from `WV` and all other workflow variables they are suspended on, and converted to a Prolog conjunction by appending them together in any order (because they are not dependent).
- The woken conjunction is executed in the same way as the initial query. We call this a *reply query*.
- The updated state is saved in the state file.

All of these steps are done mutually exclusively: if two participants reply about the same time, one will have to wait until the other has updated the state.

This way, a Workflow Prolog query is solved in *phases*: following the initial query, there is a reply query phase whenever a participant responds. The phases may be separated by long time intervals. Eventually the query may succeed or fail, whereupon the state file may be deleted.

Naturally, there may be any number of queries active at any time; these are independent of each other and each have their own state file.

2.6 Deep and nested guards

There is no restriction on the goals that can appear in a guard: it can be an arbitrary Prolog conjunction, excluding `rpc` goals and other goals with side-effects. In this sense, guards can be "deep", as in the earliest concurrent logic programming languages. (Later languages used "flat" guards, containing only specific built-in predicates, for implementation reasons [26].)

However, guards should still be as small as possible, because they are executed again from the beginning whenever the parent goal is woken after being suspended.

A particularly useful kind of deep guard is one that calls a user-defined suspendable goal, itself defined by clauses with guards. Nested guards can be used for what Conlon [8] called "peeling or-parallelism". Even without or-parallelism, nested guards are useful for expressing a disjunction between an arbitrary number of alternatives. The suspension criteria for a goal are determined dynamically, which cannot be done with flat guards or the `freeze` or `when` predicates provided in some versions of Prolog.

For example, the code in Program 1-3 works only for two markers, because of the way in which the `decide` predicate is defined: the first clause handles an '`ok`' reply from both markers while the second and third clauses each deal with comments from one of the two markers. To change the program to allow *N* markers, `decide` would need to be rewritten to use *N*+1 clauses. A better solution is to generalize it to allow an arbitrary number of markers by exploiting nested guards, as shown in Program 4. The first clause of `decide` handles an '`ok`' reply from *all* markers while the second clause handles comments from *any* marker.

```
decide(Rs, _, Proposal, Student) :- all_ok(Rs) :
  rpc(database, accepted(Student,Proposal), D),
  replycontent(D, ok).
decide(Rs, Markers, Proposal, Student) :- one_comment(Rs, C) :
  proposal(Student, revise(Proposal,C), Markers).

all_ok([]).
all_ok([R|Rs]) :-
  replycontent(R, ok),
  all_ok(Rs).

one_comment([R|_], C) :- replycontent(R, c(C)) : true.
one_comment([_|Rs], C) :- one_comment(Rs, C) : true.
```

Program 4: Nested guards

2.7 Time constraints

Workflow Prolog handles time in a more declarative way than assuming the existence of a "timer" activity. As mentioned above, each workflow variable has a message time and reply time. Each of these can be obtained, respectively, by:

```
messagetime(WV, MT)
replytime(WV, RT)
```

`messagetime(WV,MT)` is not suspendable; it unifies `MT` with the message time of `WV`, assuming that `WV` is a workflow variable on which a message has been sent. `replytime(WV,RT)` is suspendable: it suspends on workflow variable `WV` if it has no reply yet, and otherwise unifies `RT` with the reply time of `WV`.

The final type of time predicate is the *precedence constraint*

```
T1 << T2
```

which allows testing of a workflow variable's reply time even *before* a reply has been received. `T1<<T2` is a suspendable goal. `T1` and `T2` must each be an absolute time or of the form `replytime(WV)` where `WV` is a workflow variable. The four forms of this constraint, where $\mathtt{AT}_i$ is an absolute time and $\mathtt{WV}_i$ a workflow variable, are:

AT1 << AT2	Succeeds if AT1 precedes AT2, fails otherwise.
replytime(WV1) << replytime(WV2)	If both WV1 and WV2 have been replied to, succeeds or fails, depending on their actual reply times. If neither has been replied to, suspends on both WV1 and WV2. If WV1 has been replied to but WV2 has not, succeeds. If WV2 has been replied to but WV1 has not, fails.
replytime(WV) << AT	If WV has been replied to, succeeds if WV's actual reply time is earlier than AT, otherwise fails. If not, and the current time is later than AT, fails. Otherwise, suspends on both WV and time AT.
AT << replytime(WV)	If WV has been replied to, succeeds if WV's actual reply time is later than AT, otherwise fails. If not, and the current time is later than AT, succeeds. Otherwise, suspends on both WV and time AT.

In general, a suspended goal may now be suspended on workflow variables and/or time events. Goals suspended on a *time event* are woken up any time soon after the corresponding time is reached, in the same way as goals suspended on workflow variables are woken up. We call this a *time query*.

The precedence constraint can be used to implement a timeout. In Program 4, suppose that each marker is required to respond within 24 hours (86400 seconds); otherwise, the activity should be cancelled and the proposal sent to another marker for approval. Program 5 shows a variant of the decide predicate that does this. The first two clauses handle replies by markers, as before. The new third clause handles the case where one marker has not replied before the deadline: it invokes a new approve_proposal activity and replaces the old activity's workflow variable by the new one.

```
decide(Rs, _, Proposal, Student) :- all_ok(Rs) :
  rpc(database, accepted(Student,Proposal), D),
  replycontent(D, ok).
decide(Rs, Markers, Proposal, Student) :- one_comment(Rs, C) :
  proposal(Student, revise(Proposal,C), Markers).
decide(Rs, Markers, Proposal, Student) :- one_late(Rs, Rs1) :
  choose(Markers, Marker),
  rpc(Marker, approve_proposal(Proposal), R),
  decide([R|Rs1], Markers, Proposal, Student).

one_late([R|Rs], Rs) :- messagetime(R, T),
  Deadline is T+86400, Deadline << replytime(R) : true.
one_late([R|Rs], [R|Rs1]) :- one_late(Rs, Rs1) : true.
```

Program 5: Implementing a timeout using the precedence constraint

Restrictions. A replytime goal can only occur in a guard (unless RT is ground, which is unlikely), and must appear at the Prolog top level. T1<<T2 must be ground and must appear at the Prolog top level.

2.8 Data flow

Although we have concentrated on control flow issues, the use of message passing via workflow variables also allows data to be transferred to and from activities (as the message and reply content). For real applications, this would not be a ground Prolog term, but might need to be a file: the workflow system would send to the participant the *contents* of the file, not just a filename, which may be meaningless to the receiver. Similarly, for a file appearing in the reply content, data would be received from the participant and stored in a local temporary file at the time of reply. This implies that Workflow Prolog's persistent state (Section 2.5) may need to be extended to include a set of temporary files.

2.9 Cancellation and garbage collection

In workflow systems it is sometimes necessary to cancel activities after they have started. For example, an `approve_proposal` activity is invoked in two or more markers at the same time: if one marker replies with comments, the activity in each of the other markers should be cancelled. These markers are informed that they need not approve the proposal sent earlier because it is now obsolete.

It is easy to identify redundant activities: those whose workflow variable no longer appears in the suspended goals in the state. These are cancelled automatically by the garbage collector, which can also delete temporary files from the state (Section 2.8).

The garbage collector identifies each workflow variable `WV` that appears in the state but is not referenced by the suspended goals, and:

- If `WV` has not been replied to, a cancellation message is sent to the corresponding participant.
- If `WV` has been replied to, and the reply includes temporary files, these files are deleted.

2.10 Shared workflow variables

In the above examples, each concurrent workflow "thread" comprised only a single activity, but it is quite possible for a thread to contain a sequence of activities or to be even more complex. In these cases, each thread can be represented by a suspendable goal which monitors its progress. These threads may be independent, but it may be necessary for them to synchronize or communicate with each other. This can be done by having some workflow variables shared between the corresponding suspendable goals. Only one goal can send a message (by `rpc`) on a workflow variable, but all goals can wait for its reply and access its value.

To allow a workflow variable `WV` to be shared by several goals it must first be created, by a call to the `wvar` built-in predicate:

```
wvar(WV)
```

Restrictions. Like `rpc`, `wvar` must appear at the Prolog top level (i.e., there must be no choicepoints) and it must not appear in a guard.

3. Implementation considerations

3.1 Concurrency vs. coroutining

Workflow Prolog is a coroutining language, not a concurrent one. This means that the implementation need not maintain a queue of runnable processes. A query is a single process (a Prolog conjunction), and so is a reply query (Section 2.5). While executing a query, it may spawn processes (each process being a suspendable goal) but these are always suspended, not runnable; if a suspendable goal is not actually suspended, it is run in place. Because there is no runnable queue, there can be no preemptive scheduling. Therefore, the programmer is responsible for avoiding nonterminating goals, but this is no more onerous than in Prolog.

Assuming there are no nonterminating goals, fairness is assured. This is because (see Section 2.5) whenever a goal suspends, it will not be woken up again until *all* goals in the query have suspended or succeeded.

3.2 Suspendable goals and workflow variables

Since Workflow Prolog is based on Prolog, a query is executed in essentially the same way as Prolog. Indeed, if no suspendable goals are used, there is almost no difference from Prolog, and no overhead relative to it. Even when a suspendable goal is called, the only difference is the search for a candidate clause, by executing guards until the goal commits or suspends. This search for a clause could be implemented by an extension to the Prolog implementation or by a simple interpreter. The latter approach (used in the prototype implementation) is more expensive, but not significantly, because the interpreter is called only for suspendable goals and is very small: the guards themselves are not interpreted but are executed directly by Prolog, as are clause bodies.

No special unification or matching is needed in a Workflow Prolog implementation, because workflow variables are distinct from regular variables and are used only by suspendable predicates. For example, a workflow variable cannot be unified with another workflow variable or any term except an unbound variable.

3.3 Suspending goals

One way to suspend a suspendable goal on a set of workflow variables is to exploit coroutining features provided in some Prolog systems, such as “attributed variables” [17, 15]. Each workflow variable would contain a pointer to the goals suspended on it, and the other variables that each goal is suspended on. This kind of structure allows suspended goals to be directly accessed from the variable in order to wake them up. However, this is not an advantage in Workflow Prolog because, between the times of suspending goals and waking them up, the whole structure needs to be saved to and restored from a file.

Another method (used in the prototype implementation) is to store workflow variables and suspended goals in the Prolog “database”, using `record` or `assert`. This approach is possible only because suspended goals are ground and workflow variables are used only by suspendable predicates (e.g., they cannot be unified). If, for example, suspendable goals contained variables, these variables would be lost (renamed) when the goal is copied to the database.

3.4 Sending messages and receiving replies

The `rpc` predicate sends a message to a participant. These messages could either be sent immediately or delayed until the end of the current query phase. Delaying them cannot cause deadlock because the reply to a message is never received during the same query phase.

The exact form of a message is not prescribed, except that it must contain a unique identifier to enable the corresponding reply to be matched with the same workflow variable. Possible ways to implement `rpc(Dest,MC,WV)` include:

- For a human participant, the message and reply could be sent by email. `Dest` would be the participant's email address. Email is sent to `Dest` with a unique identifier in the "Subject" field and `MC` in the email body. A mail client process is run, which monitors a mail (e.g., IMAP) server for incoming mail. When a message is received that contains a recognized identifier, the mail client assigns the email body to the reply content of workflow variable `WV` and invokes a reply query.
- Also for a human participant, a web interface could be used. Again, `Dest` would be the participant's email address. A web page is created containing `MC` and a form for the reply, and its URL mailed to `Dest`. The participant replies by accessing the URL and completing the form, which "posts" the form contents to the web server. This assigns the form contents to the reply content of the corresponding workflow variable and invokes a reply query. One advantage of this method is that the reply content can be structured.
- If the participant is a program (on the same machine), `Dest` could be the pathname of the program. A process is created in the host operating system to run the named program with `MC` as its standard input and, when the program terminates, assign the standard output to the reply content of the corresponding workflow variable and invoke a reply query.

3.5 Time constraints

Time constraints can be implemented by, at regular intervals, restoring the state from the state file, finding time events that are earlier than the current time, executing the goals that are suspended on them as a time query, and saving the updated state. This could be done by a thread or process that runs continuously, or by making use of the operating system's scheduler, such as the cron utility of Unix. The question is how frequently to check the state: if too frequent, it will be less efficient, but more effective at responding to deadlines.

A better method would be to check the state file only at the time of each time event stored in it. If this is done by a process that runs continuously, it would need to be informed by a message each time a new time event is added to the state. The process would sleep until the time of the earliest time event or until such a message arrives.

3.6 Garbage collection

Garbage collection (Section 2.9) could be done at the end of each phase of a query. Alternatively, it could be performed by a separate process that runs from time to time. This could be a more efficient method since garbage collection could then be done less frequently when contention is higher.

Figure 1 shows the structure of a Workflow Prolog implementation: the state file is created by the initial query and then updated by reply queries, time queries, and garbage collection. All of these require exclusive access to the state file while they update it.

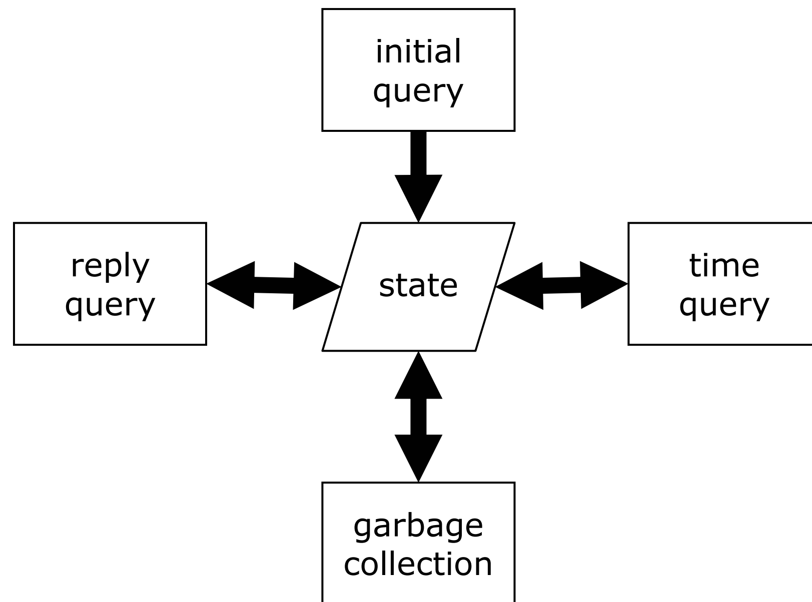

Figure 1: Workflow Prolog implementation

4. Workflow patterns in Workflow Prolog

In [1], the authors describe 20 workflow patterns which represent, in an abstract form, all of the control flow features found in the wide variety of workflow management systems that they investigated. They believe that this set of patterns represent comprehensive workflow functionality. The set of workflow patterns could therefore be regarded as a specification of what an ideal workflow language should provide. In this section we describe each of the workflow patterns, together with an example, and show how it can be implemented as a program in Workflow Prolog, in an attempt to demonstrate its power as a workflow language.

4.1 Pattern 1: sequence

An activity is executed after the completion of another activity. For example, a "review proposal" activity is executed after a "write proposal" activity in a process to write, review, and submit proposals (Figure 2).

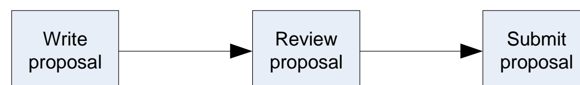

Figure 2: Pattern 1 (sequence)

To sequence two activities in Workflow Prolog, the first activity is invoked by an `rpc` goal, a suspendable goal waits for the reply on the first activity's workflow variable (by a

clause's guard) and the same clause's body invokes the second activity by another `rpc` goal. See Program 6.

```
:- rpc(write_proposal, 0, P), review_proposal(P).

review_proposal(P) :- replycontent(P, Proposal) :
  rpc(review_proposal, Proposal, RP),
  submit_proposal(RP).

submit_proposal(RP) :- replycontent(RP, ReviewedProposal) :
  rpc(submit_proposal, ReviewedProposal, A),
  replycontent(A, accepted).
```

Program 6: Pattern 1 (sequence)

4.2 Pattern 2: parallel split

A thread splits into two or more threads that are executed concurrently. For example, an "arrange meeting" activity enables the concurrent execution of the following activities "person 1 check diary", "person 2 check diary", and "person 3 check diary", in a process to arrange a meeting (Figure 3).

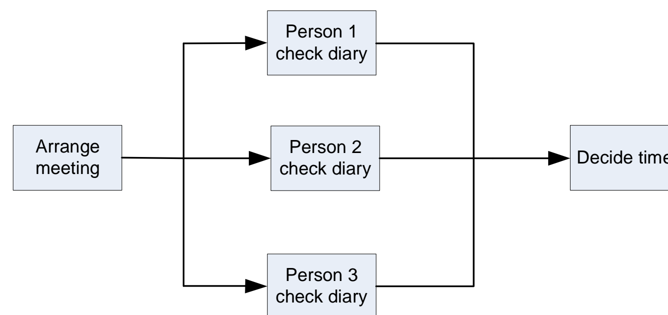

Figure 3: Pattern 2 (parallel split) and pattern 3 (synchronization)

In Workflow Prolog, concurrent activities are each invoked by an `rpc` goal, like the three `check_diary` activities in Program 7.

```
:- rpc(arrange_meeting, 0, D), check_times(D).

check_times(D) :- replycontent(D, Date) :
  rpc(person1, check_diary(Date), T1),
  rpc(person2, check_diary(Date), T2),
  rpc(person3, check_diary(Date), T3),
  decide_time(T1, T2, T3).

decide_time(T1, T2, T3) :-
  replycontent(T1, Time1),
  replycontent(T2, Time2),
  replycontent(T3, Time3) :
  rpc(decide_time, [Time1,Time2,Time3], D),
  replycontent(D, done).
```

Program 7: Pattern 2 (parallel split) and pattern 3 (synchronization)

In general, multiple concurrent *threads* (*subprocesses*) can be invoked in a similar way: as well as an `rpc` goal to invoke each activity, there would be a suspendable goal suspended on each activity's workflow variable, which would invoke subsequent activities in the thread; an example of this is Program 18.

4.3 Pattern 3: synchronization

A number of concurrent threads join together into a single thread. The following activity waits until all of the concurrent threads have completed (synchronization). For example, the "decide time" activity is enabled after the completion of the "person 1 check diary", "person 2 check diary", and "person 3 check diary" activities, in a process to arrange a meeting (Figure 3).

In Workflow Prolog, a suspendable goal suspends on all of the workflow variables that indicate the end of each activity or thread. In Program 7, the `decide_time` goal has this role: it waits for the three `check_diary` activities to complete.

4.4 Pattern 4: exclusive choice

One alternative branch is chosen to be executed. For example, after the execution of an "assessment type" activity, one of the following activities is enabled: either "submit assignment" or "take exam", in a process for student assessment (Figure 4).

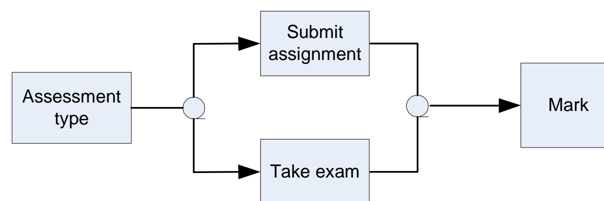

Figure 4: Pattern 4 (exclusive choice) and pattern 5 (simple merge)

In Workflow Prolog this is simply implemented using guards. If the choice between branches is to be made on the basis of the result of a workflow activity, the guard can both wait for the activity to complete *and* test its result. In Program 8, the `assessment` goal uses the result of the `assessment_type` activity to choose between the `submit_assignment` and `take_exam` activities.

```
:- rpc(course, assessment_type, A), assessment(A).

assessment(A) :- replycontent(A, coursework) :
  rpc(student, submit_assignment, C),
  mark(C).
assessment(A) :- replycontent(A, exam) :
  rpc(student, take_exam, E),
  mark(E).

mark(A) :- replycontent(A, Submission) :
  rpc(marker, Submission, D),
  replycontent(D, done).
```

Program 8: Pattern 4 (exclusive choice) and pattern 5 (simple merge)

4.5 Pattern 5: simple merge

Two or more alternative branches join together without synchronization (because only one branch has been executed). For example, after a student submits an assignment or takes an exam, the work (assignment or exam) is marked by the "mark" activity (Figure 4).

The simplest way to implement this is to include the continuation code at the end of every branch. In Program 8, a `mark` goal appears in both of the clauses for `assessment`. This duplication is not a problem because, of course, the `mark` predicate is defined only once.

4.6 Pattern 6: multi-choice

A number of branches exist such that one or more of them can be selected for execution. For example, in a process to assess dissertations, after the execution of a "submit dissertation" activity, the activity "assess dissertation by marker 1" or the activity "assess dissertation by marker 2" is executed, or both. At least one of the branches is executed, but sometimes both of them can be executed as concurrent threads (Figure 5).

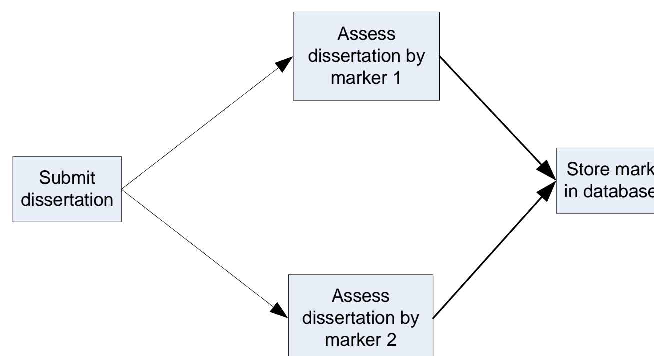

Figure 5: Pattern 6 (multi-choice) and pattern 7 (synchronizing merge)

Because of Workflow Prolog's committed choice, this cannot be implemented with one clause for each alternative, like exclusive choice. It requires an extra clause for each combination of alternatives. For example, in Program 9, `mark1` has a clause to invoke the `assess` activity for each marker and a third clause to invoke that activity for *both* markers.

4.7 Pattern 7: synchronizing merge

Two or more branches join together into a single thread. If more than one branch was active concurrently, these branches are synchronized. Alternatively, if only one branch was chosen, no synchronization is needed. For example, after the execution of "assess dissertation by marker 1" or "assess dissertation by marker 2" or both of them, the activity "store mark in database" is enabled (Figure 5).

In Workflow Prolog, this can be handled quite simply: a suspendable goal suspends on a variable number of workflow variables, each being the reply to one of the concurrent activities or threads invoked in a multi-choice construct. In Program 9, the `decide` goal is suspended on the list of workflow variables produced by the `mark1` goal that originally invoked the activities.

```
:- submit_mark(zz1234, N, marker1, marker2).

submit_mark(Student, N, Marker1, Marker2) :-
  rpc(Student, submit, D),
  mark(D, Student, N, Marker1, Marker2).

mark(D, Student, N, Marker1, Marker2) :-
  replycontent(D, Diss) :
  mark1(Diss, N, Marker1, Marker2, Rs),
  decide(Rs, Student).

mark1(Diss, 1, Marker1, _, [R1]) :- !,
  rpc(Marker1, assess(Diss), R1).
mark1(Diss, 2, _, Marker2, [R2]) :- !,
  rpc(Marker2, assess(Diss), R2).
mark1(Diss, both, Marker1, Marker2, [R1,R2]) :-
  rpc(Marker1, assess(Diss), R1),
  rpc(Marker2, assess(Diss), R2).

decide(Rs, Student) :- received_marks(Rs, Marks) :
  rpc(database, marks(Student,Marks), D),
  replycontent(D, done).

received_marks([R], [M]) :- !,
  replycontent(R, M).
received_marks([R1,R2], [M1,M2]) :-
  replycontent(R1, M1),
  replycontent(R2, M2).
```

Program 9: Pattern 6 (multi-choice) and pattern 7 (synchronizing merge)

4.8 Pattern 8: multi-merge

Two or more branches join together into a single one without synchronization, such that if more than one branch is executed (as concurrent threads), the activity that follows the merge is executed for every branch. For example, the process of applying for a course involves two concurrent activities, "verify application" and "accept or reject application", both of which lead to the execution of the same activity "inform student" (Figure 6).

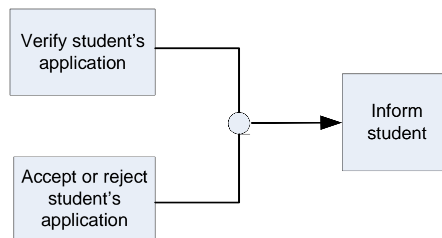

Figure 6: Pattern 8 (multi-merge)

Similarly to pattern 5 (simple merge), multi-merge can be implemented by including the required continuation code at the end of each concurrent activity or thread. In Program 10, an `inform` goal monitors the result of both the `verify_application` and `accept_reject` activities, so when each of them is complete, the `inform_student` activity is invoked.

```
:- rpc(receive_application, 0, A), process_application(A).

process_application(A) :- replycontent(A, Application) :
  rpc(verify_application, Application, V),
  rpc(accept_reject, Application, AR),
  inform(V),
  inform(AR).

inform(R) :- replycontent(R, Result) :
  rpc(inform_student, Result, D),
  replycontent(D, done).
```

Program 10: Pattern 8 (multi-merge)

4.9 Pattern 9: discriminator

Two or more branches join together into a single one, such that when the first one is completed the following activity is enabled. The discriminator waits for the remaining branches (if any) to complete but then ignores them. For example, in a process to assess dissertations, suppose that two markers should assess a student's dissertation but only one mark is stored. When the first marker responds, the following activity "store mark in database" is enabled. The second marker's response is completed but is ignored (Figure 7).

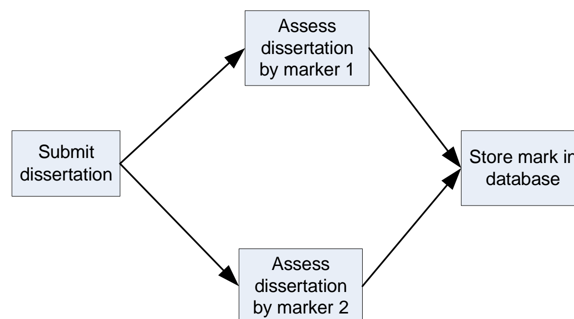

Figure 7: Pattern 9 (discriminator)

A discriminator can be implemented in the same way as pattern 2 (parallel split) but with additional suspendable goals to wait for, and ignore, all but the first "branch" that completes. In Program 11, the `mark` clause invokes two concurrent activities and the `decide` goal suspends on both of them. The first `decide` clause waits for the first activity to complete and then ignores the second; the second clause waits for the second activity to complete and then ignores the first.

4.10 Pattern 10: arbitrary cycles

One or more activities can be executed repeatedly. Workflow languages vary in whether cycles can be arbitrary or must be structured (like the while and for loops of structured programming languages). For example, in a process to write and submit a dissertation, the "write draft" activity is executed after the "background research" activity is complete. Then the "revise draft" activity must be executed 0 or more times before the "submit" activity is executed (Figure 8).

```
:- submit_mark(zz1234, marker1, marker2).

submit_mark(Student, Marker1, Marker2) :-
  rpc(Student, submit, D),
  mark(D, Student, Marker1, Marker2).

mark(D, Student, Marker1, Marker2) :- replycontent(D, Diss) :
  rpc(Marker1, assess(Diss), R1),
  rpc(Marker2, assess(Diss), R2),
  decide(R1, R2, Student).

decide(R1, R2, Student) :- replycontent(R1, M1) :
  ignore_result(R2),
  rpc(database, marks(Student,M1), D),
  replycontent(D, done).
decide(R1, R2, Student) :- replycontent(R2, M2) :
  ignore_result(R1),
  rpc(database, marks(Student,M2), D),
  replycontent(D, done).

ignore_result(R) :- replycontent(R, _) : true.
```

Program 11: Pattern 9 (discriminator)

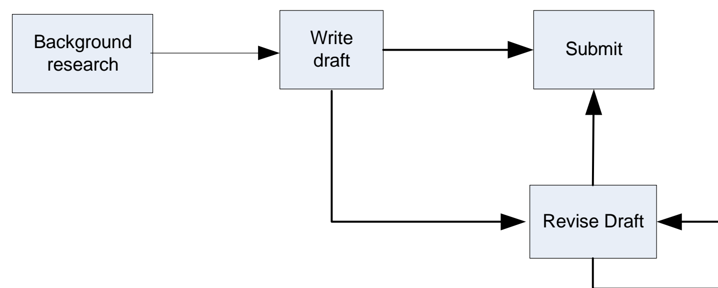

Figure 8: Pattern 10 (cycles)

Cycles (loops) can be implemented in the same way as in Prolog: by tail recursion. The result of an activity is tested in a clause guard, and the recursive clause has a body that invokes a new activity and tests its result in a tail-recursive goal, as in Program 12.

```
:- rpc(background_research, 0, R), write_draft(R).

write_draft(R) :- replycontent(R, Research) :
  rpc(write_draft, Research, D),
  revise_draft(D).

revise_draft(D) :- replycontent(D, Draft), good(Draft) :
  rpc(submit, Draft, D1),
  replycontent(D1, done).
revise_draft(D) :- replycontent(D, Draft), \+ good(Draft) :
  rpc(revise_draft, Draft, D1),
  revise_draft(D1).
```

Program 12: Pattern 10 (cycles)

4.11 Pattern 11: implicit termination

If a subprocess comes to an end (none of the activities are executed and none of them can be enabled) then the subprocess is terminated. Workflow languages vary in whether termination is implicit or by an explicit instruction. For example, in all the above examples, the processes should terminate when all activities shown have been completed.

Workflow Prolog programs terminate implicitly, when there are no activities that are active or can be activated. As in other logic programming languages, there is no explicit "halt" instruction.

4.12 Pattern 12: multiple instances without synchronization

Multiple instances of an activity are generated that are independent, so that there is no need to synchronize them. For example, consider a mailshot process with a list of addresses. A concurrent "mail" activity instance is created for each address to send a letter to that address (Figure 9).

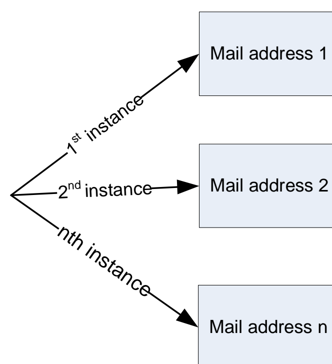

Figure 9: Pattern 12 (multiple instances without synchronization)

It is simple to create multiple instances of activities or threads in Workflow Prolog. Each `rpc` goal invokes a new activity and workflow variable, and there can be a suspendable goal suspended on each workflow variable. For example, each `mailing` goal in Program 13 creates a new `mail` activity and a goal that simply waits for it to complete.

```
:- mailshot([cust1,cust2,cust3]).

mailshot([]).
mailshot([Addr|Addrs]) :-
  mailing(Addr),
  mailshot(Addrs).

mailing(Addr) :-
  rpc(mail, Addr, Mailed),
  replycontent(Mailed, done).
```

Program 13: Pattern 12 (multiple instances without synchronization)

4.13 Pattern 13: multiple instances with *a priori* design time knowledge

Multiple instances of an activity are generated which need to synchronize, such that another activity may start when they all complete. In this pattern the number of instances is known at design time. For example, in a process to assess dissertations, the number of markers may be fixed. Figure 10 shows this situation where the number of markers is 2.

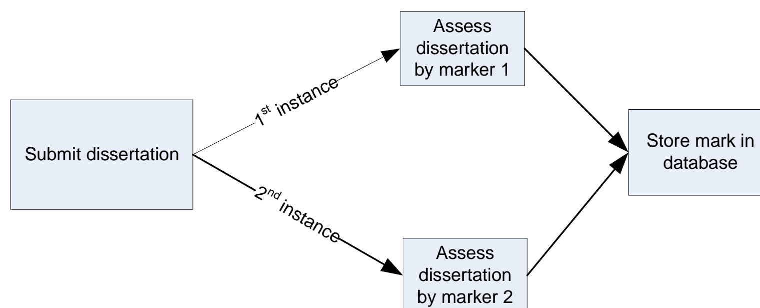

Figure 10: Pattern 13 (multiple instances with *a priori* design time knowledge)

This is similar to patterns 2 and 3 (parallel split and synchronization), in which all threads are of the same type: i.e., multiple instances. Since we know the number of instances in advance we can include in the program *N* copies of an `rpc` goal, to invoke *N* activities, and *N* calls to `replycontent` to wait for them all to complete, as in Program 14.

```
  :- submit_mark(zz1234, marker1, marker2).

submit_mark(Student, Marker1, Marker2) :-
  rpc(Student, submit, D),
  mark(D, Student, Marker1, Marker2).

mark(D, Student, Marker1, Marker2) :- replycontent(D, Diss) :
  rpc(Marker1, assess(Diss), R1),
  rpc(Marker2, assess(Diss), R2),
  decide(R1, R2, Student).

decide(R1, R2, Student) :-
  replycontent(R1, M1),
  replycontent(R2, M2) :
  rpc(database, marks(Student,M1,M2), D),
  replycontent(D, done).
```

Program 14: Pattern 13 (multiple instances with *a priori* design time knowledge)

4.14 Pattern 14: multiple instances with *a priori* runtime knowledge

Multiple instances of an activity are generated which need to synchronize, such that another activity may start when they all complete. In this pattern the number of instances is not known at design time but is known before the instances are created. For example, in a process to assess dissertations, suppose that the number of markers is a number *n*, which may be different for each dissertation. Figure 11 shows this situation.

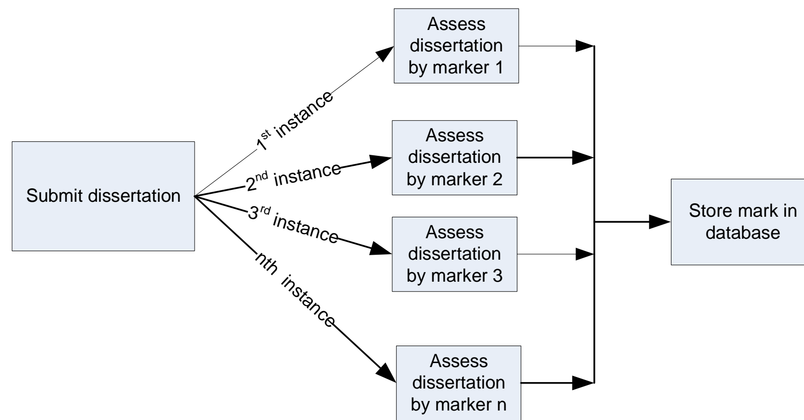

Figure 11: Pattern 14 (multiple instances with *a priori* runtime knowledge)

This is implemented as a conjunction of `rpc` goals and a conjunction of `replycontent` goals, the numbers of which are determined at run time. In Program 15, the `submit_mark` goal takes an arbitrarily long list of markers instead of two markers as in Program 14.

```
:- submit_mark(zz1234, [marker1,marker2]).

submit_mark(Student, Markers) :-
  rpc(Student, submit, D),
  mark(D, Student, Markers).

mark(D, Student, Markers) :- replycontent(D, Diss) :
  rpcs(Markers, assess(Diss), Rs),
  decide(Rs, Student).

decide(Rs, Student) :- replycontents(Rs, Ms) :
  rpc(database, marks(Student,Ms), D),
  replycontent(D, done).

rpcs([], _, []).
rpcs([Dest|Dests], Message, [WV|WVs]) :-
  rpc(Dest, Message, WV),
  rpcs(Dests, Message, WVs).

replycontents([], []).
replycontents([R|Rs], [M|Ms]) :-
  replycontent(R, M),
  replycontents(Rs, Ms).
```

Program 15: Pattern 14 (multiple instances with *a priori* runtime knowledge)

4.15 Pattern 15: multiple instances without *a priori* runtime knowledge

Multiple instances of an activity are generated which need to synchronize, such that another activity may start when they all complete. In this pattern the number of instances is not known before the instances are created. This means that new instances can be created even if some of the instances are still being executed or have already ended. For example, in a process to assess dissertations, suppose that the number of markers is normally 2, but it is possible for the two markers to disagree on the mark. In this case the workflow process finds a third marker, creating a third instance of the "assess dissertation" activity. The third

instance will be created after the other instances have been created (and completed). Figure 12 shows this situation.

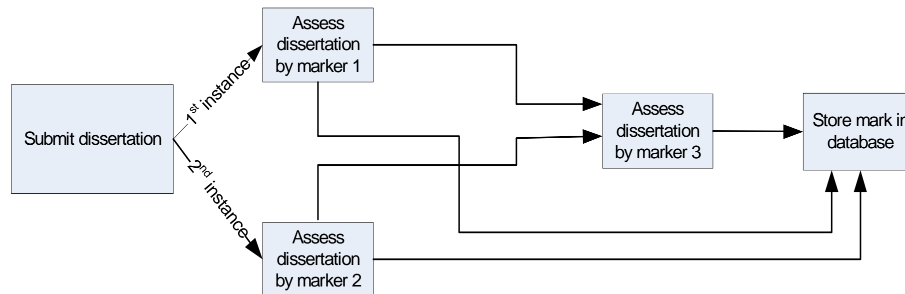

Figure 12: Pattern 15 (multiple instances without *a priori* runtime knowledge)

In Workflow Prolog, it is easy to create and synchronize new activities or threads at run time. In Program 16, if the original markers disagree, a new `assess` activity is invoked, and synchronized by adding its workflow variable to the list of workflow variables from the existing activities.

```
:- submit_mark(zz1234, [marker1,marker2]).

submit_mark(Student, Markers) :-
  rpc(Student, submit, D),
  mark(D, Student, Markers).

mark(D, Student, Markers) :- replycontent(D, Diss) :
  rpcs(Markers, assess(Diss), Rs),
  decide(Rs, Student, Diss).

decide(Rs, Student, Diss) :-
  replycontents(Rs, Ms), disagree(Ms) :
  rpc(marker3, assess(Diss), R),
  decide([R|Rs], Student, Diss).
decide(Rs, Student, _) :-
  replycontents(Rs, Ms), \+disagree(Ms) :
  rpc(database, marks(Student,Ms), D),
  replycontent(D, done).

disagree([M1,M2]) :- M1 < M2-10.
disagree([M1,M2]) :- M2 < M1-10.
```

Program 16: Pattern 15 (multiple instances without *a priori* runtime knowledge)

4.16 Pattern 16: deferred choice

One of several alternative branches is chosen to be executed; however, the choice is not made in advance, but depends on the environment. For example, in an exam process, a participant has to register for an exam and then take the exam. Suppose there are two alternative exams he can take, and the choice is determined by which registration attempt succeeds first. The "register exam 1" and "register exam 2" activities are tried concurrently; if/when one succeeds, the corresponding "take exam" activity is started and the other "register exam" activity is cancelled (Figure 13).

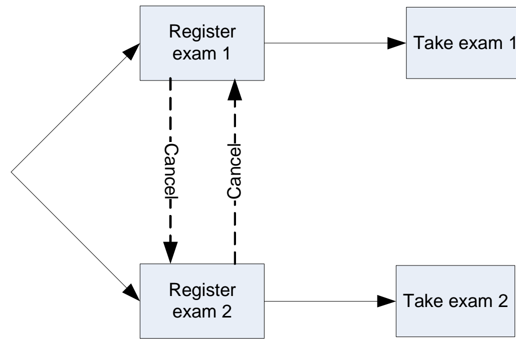

Figure 13: Pattern 16 (deferred choice)

A deferred choice is implemented in Workflow Prolog by first invoking all of the activities, waiting for one of them to complete, and then cancelling all others. The cancellation of unwanted activities is handled automatically by Workflow Prolog's special garbage collection. In Program 17, a candidate tries to register for two exams by invoking the `register1` and `register2` activities; if one (say `register2`) completes, the `register1` activity is cancelled and the candidate then performs the `exam2` activity.

```
:- rpc(register1, Name, Reg1), rpc(register2, Name, Reg2),
   cand(Name, Reg1, Reg2).

cand(Name, Reg1, _) :- replycontent(Reg1, done) :
  rpc(exam1, Name, Exam),
  replycontent(Exam, done).
cand(Name, _, Reg2) :- replycontent(Reg2, done) :
  rpc(exam2, Name,Exam),
  replycontent(Exam, done).
```

Program 17: Pattern 16 (deferred choice)

Pattern 16 is described as a *state-based* pattern in [1] because it involves testing conditions that become true temporarily; e.g., the `register1` activity should be enabled only until `register2` succeeds. However, as van der Aalst *et al*. point out, the same effect can be achieved in a message passing framework by using "messages to cancel previous messages"; this is what Workflow Prolog does, automatically by garbage collection.

4.17 Pattern 17: interleaved parallel routing

A number of activities are executed in an arbitrary order, which is decided at run time. This is similar to parallel split but with the restriction that two activities cannot be executed at the same time. For example, in an exam process, suppose that there are several participants each trying to register for an exam and then take it, but only one participant can take the exam at a time. Hence, the "take exam" activities are mutually exclusive (Figure 14).

Although this is described as a state-based pattern in [1], it can be implemented in Workflow Prolog without the need for cancellation messages; see Program 18. To implement the required mutual exclusion, we use workflow variables that are shared between suspendable goals (Section 2.10), and test the reply times of workflow variables.

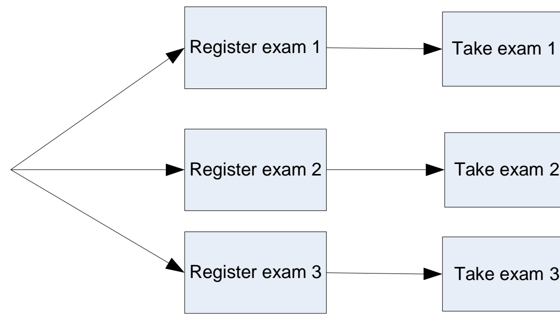

Figure 14: Pattern 17 (interleaved parallel routing)

```
:- wvar(AR), wvar(BR), wvar(AE), wvar(BE),
   cand(a,AR,BR,AE,BE), cand(b,BR,AR,BE,AE).

cand(Name,R,OR,E,OE) :-
  rpc(register,Name,R),
  cand1(Name,R,OR,E,OE).

cand1(Name,R,_,E,OE) :-
  replytime(R) << replytime(E), replytime(OE) << replytime(E) :
  rpc(exam,Name,E), replycontent(E,done).
cand1(Name,R,OR,E,_) :-
  replytime(R) << replytime(OR) :
  rpc(exam,Name,E), replycontent(E,done).
```

Program 18: Pattern 17 (interleaved parallel routing)

Each `cand` goal (thread) can access the workflow variables of the others. After a candidate (`cand` goal) successfully registers for the exam, it waits until all those that registered earlier complete the exam before taking the exam itself. Program 18 shows the case of two candidates, but can easily be generalized. The first `cand1` clause requires this candidate to register and the other candidate to complete the exam before taking the exam; the second clause requires this candidate to register before the other does.

4.18 Pattern 18: milestone

The enabling of an activity depends on the state in which a workflow process is. This means that an activity is enabled *temporarily* only if some thread has reached a particular point in its execution. For example, in an exam process, there could be a milestone between the "register exam" and "take exam" activities. An "eat" activity can be executed (zero or more times) between these two activities (Figure 15).

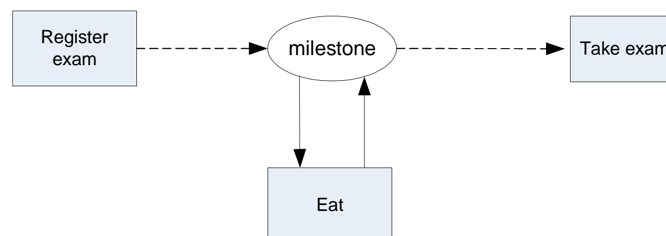

Figure 15: Pattern 18 (milestone)

Like deferred choice, this can be implemented in Workflow Prolog using cancellation messages (issued automatically by garbage collection). In Program 19, a candidate first completes the `register` activity, and then tries to invoke both the `exam` and `eat` activities. If the `eat` activity succeeds, the `exam` activity is cancelled and retried later. If the `exam` activity succeeds, the process terminates.

```
:- cand(a).

cand(Name) :-
  rpc(register, Name, Register),
  cand1(Name, Register).

cand1(Name, Register) :- replycontent(Register, done) :
  cand2(Name).

cand2(Name) :-
  rpc(eat, Name, Eat),
  rpc(exam, Name, Exam),
  cand3(Name, Eat, Exam).

cand3(Name, Eat, _) :- replycontent(Eat, done) :
  cand2(Name).
cand3(_, _, Exam) :- replycontent(Exam, done) :
  true.
```

Program 19: Pattern 18 (milestone)

4.19 Pattern 19: cancel activity

An enabled activity is disabled. For example, in Figure 13, one "register exam" activity is cancelled when the other completes.

As mentioned in Section 2.9, activities are cancelled automatically by garbage collection. If there is a thread (suspendable goal) waiting for the result of an activity, the activity will not be cancelled because it is still referenced by the program. Any such suspended goals must be cancelled explicitly, by causing them to wake up and terminate; the activities will then be cancelled.

4.20 Pattern 20: cancel case

An instance of a process is entirely stopped. For example, in Figure 3, if the meeting is cancelled, the "check diary" activities performed by all of the participants can be stopped.

This is similar to cancelling activities. Any suspendable goals represent threads must be terminated explicitly, but all associated activities will be cancelled automatically.

5. Conclusions and related work

5.1 Summary

We have presented a new extension to Prolog that we believe is sufficiently expressive for workflow applications but simple enough to implement efficiently and to use.

The `rpc` predicate is simple but neatly captures a workflow participant's behaviour as a relation on a workflow variable. Garbage collection of workflow variables is fed back to the

workflow participant by cancelling activities. In these ways, Workflow Prolog is intended to *live in* the real world in the sense suggested by Morozov [20]: "To reason about the outer world and to live in the outer world are not the same things. Actually, [a] logic language can operate in [the] outer world without reasoning about it."

Workflow Prolog's coroutining using guards, especially nested guards, is more powerful than the `freeze` or `when` predicates. Persistence is also essential, and time constraints provide a powerful and declarative way to express time. Implementation is simplified by restricting suspended goals to be ground and because goals are never woken up during a query.

A Workflow Prolog process cannot interact with itself or with another Workflow Prolog process, because it has no facility to act as a server (to receive unsolicited requests), but this is because the language is intended to coordinate workflow activities, not implement them.

A prototype implementation has been developed in SWI-Prolog. It includes all features described in this paper, with messages to participants displayed on screen and replies read from the keyboard. Garbage collection is not yet implemented except on success or failure of the query. The Workflow Prolog implementation is available from `http://www.cs.bris.ac.uk/~steve/wp/`.

5.2 Future work

In the near future we plan to enhance the Workflow Prolog implementation. First, garbage collection should be fully implemented. Second, the system should be linked with some practical methods of sending and receiving messages, like those listed in Section 3.4. Finally, practical forms of data flow (files, etc.) should be implemented.

There are a few ways in which the Workflow Prolog language could be extended. One is to allow suspendable goals to be non-ground at the time of call. This could allow data to be transferred from them, when they terminate, to subsequent goals. The problems are (a) how to delay a goal following a suspendable goal (only) if it shares a variable with it; (b) how to preserve variables in suspended goals.

Another language extension is to allow precedence constraints (Section 2.7) to constrain a workflow variable's message time. For example, `T<<messagetime(WV)` would delay the sending of a message until a specified time, `T`. This would have to be executed before the `rpc` goal that sends the message. This would complicate the language slightly because both `rpc` and `messagetime(WV,T)` would then become suspendable goals.

More ambitiously, we should allow Workflow Prolog programs to be modified while queries are still active, because workflow specifications typically need to be updated at intervals shorter than the duration of some "business processes".

5.3 Related work

The idea of extending an existing language, instead of developing a new one, to handle workflow applications was also proposed by Forst *et al*. [14]. Their approach uses a "coordination toolkit" that can be added to computation languages, including Prolog.

The workflow patterns of van der Aalst *et al*. have received a lot of attention since they were published. Recently they were used to evaluate the capabilities of the orchestration language Orc [9]; unlike Workflow Prolog, which is a practical programming language, Orc is a new process algebra. Coordination languages have also been applied to workflow recently by Omicini *et al*. [21].

References

1. W.M.P. van der Aalst, A.H.M. ter Hofstede, B. Kiepuszewski, and A.P. Barros. Workflow Patterns. In *Distributed and Parallel Databases 14*, 1 (July), 5-51. 2003.
2. W.M.P. van der Aalst and A.H.M. ter Hofstede. YAWL: yet another workflow language. In *Information Systems 30*, 4 (June), 245-275. 2005.
3. H.H. Bi and J.L. Zhao. Applying propositional logic to workflow verification. In *Information Technology and Management 5*, 3-4 (July), 293-318. 2004.
4. D. Chan and K. Leung. Valmont: a language for workflow programming. In *Proceedings of the 31st Annual Hawaii International Conference on System Sciences*, 744-753. IEEE. 1998.
5. K.L. Clark and S. Gregory. A relational language for parallel programming. In *Proceedings of the 1981 Conference on Functional Programming Languages and Computer Architecture* (Portsmouth, NH), 171-178. ACM. 1981.
6. K.L. Clark and S. Gregory. Parallel programming in logic. In *ACM Transactions on Programming Languages and Systems 8*, 1 (January), 1-49. ACM. 1986.
7. K.L. Clark, F.G. McCabe, and S. Gregory. IC-PROLOG language features. In *Logic Programming*, ed. Clark and Tärnlund, 253-266. Academic Press. 1982.
8. T. Conlon. *Programming in Parlog*. Addison-Wesley. 1989.
9. W. Cook, S. Patwardhan, J. Misra. Workflow patterns in Orc. In *Proceedings of the 8th International Conference on Coordination Models and Languages* (LNCS 4038), 82-96. Springer. 2006.
10. H. Davulcu, M. Kifer, C.R. Ramakrishnan, and I.V. Ramakrishnan. Logic based modeling and analysis of workflows. In *Symposium on Principles of Database Systems* (Seattle), 25-33. 1998.
11. E.W. Dijkstra. Guarded commands, nondeterminacy and formal derivation of programs. In *Communications of the ACM 18*, 8 (August), 453-457. ACM. 1975.
12. Y. Duan and H. Ma. Modeling flexible workflow based on temporal logic. In *Proceedings of the 9th International Conference on Computer Supported Cooperative Work in Design vol 1*, 508-513. IEEE. 2005.
13. M.H. van Emden and G.J. de Lucena Filho. Predicate logic as a language for parallel programming. In *Logic Programming*, ed. Clark and Tärnlund, 189-198. Academic Press. 1982.
14. A. Forst, E. Kühn, and O. Bukhres. General purpose work flow languages. In *Distributed and Parallel Databases 3*, 2 (April), 187-218. 1995.
15. M. Hermenegildo, D. Cabeza, and M. Carro. Using attributed variables in the implementation of concurrent and parallel logic programming systems. In *Proceedings of the 1995 International Conference on Logic Programming*, 631-645. MIT Press. 1995.
16. C.A.R. Hoare. Communicating sequential processes. In *Communications of the ACM 21*, 8 (August), 666-677. ACM. 1978.
17. C. Holzbaur. Metastructures vs. attributed variables in the context of extensible unification. In *Proceedings of International Symposium on Programming Language*

Implementation and Logic Programming (LNCS 631), 260-268. Springer Verlag. 1992.

18. D. Liu, J. Wang, S. Chan, J. Sun, L. Zhang. Modeling workflow processes with colored Petri nets. In *Computers in Industry 49*, 3 (December), 267-281. Elsevier. 2002.
19. H. Ma. A workflow model based on temporal logic. In *Proceedings of the 8th International Conference on Computer Supported Cooperative Work in Design vol 2*, 327-332. IEEE. 2004.
20. A. Morozov. On the next logic programming language. In *ALP Newsletter* (August). http://www.cs.kuleuven.ac.be/~dtai/projects/ALP/newsletter/aug04/. 2004.
21. A. Omicini, A. Ricci, N. Zaghini. Distributed workflow upon linkable coordination artifacts. In *Proceedings of the 8th International Conference on Coordination Models and Languages* (LNCS 4038), 228-246. Springer. 2006.
22. L.R. Pokorny and C.R. Ramakrishnan. Modeling and Verification of Distributed Autonomous Agents Using Logic Programming. In *Declarative Agent Languages and Technologies II* (LNCS 3476), 148-165. Springer. 2005.
23. P. Senkul, M. Kifer, and I. Toroslu. A logic framework for scheduling workflows under resource allocation constraints. In *Proceedings of the 28th VLDB Conference* (Hong Kong), 694-702. 2002.
24. E.Y. Shapiro. The family of concurrent logic programming languages. In *ACM Computing Surveys 21*, 3, 413-510. ACM. 1989.
25. C. Stefansen. SMAWL: a small workflow language based on CCS. In *Proceedings of the CAISE05 Forum* (Porto). 2005.
26. A. Takeuchi and K. Furukawa. Parallel logic programming languages. In *Proceedings of the 3rd International Conference on Logic Programming* (LNCS 225), 242-254. Springer. 1986.
27. Y. Wang and Y. Fan. Using temporal logics for modeling and analysis of workflows. In *IEEE International Conference on E-Commerce Technology for Dynamic E-Business*, 169-174. IEEE. 2004.
28. Workflow Management Coalition. *WFMC specification: the workflow reference model*. Document no. WFMC-TC-1003. 1995.
29. Workflow Management Coalition. *WFMC specification: terminology and glossary*. Document no. WFMC-TC-1011. 1999.
30. Workflow Patterns Home Page. `http://www.workflowpatterns.com/`