International Journal of Cooperative Information Systems
Vol. 12, No. 1 (2003) 1–36


World Scientific
www.worldscientific.com

# A REUSE-ORIENTED WORKFLOW DEFINITION LANGUAGE

MARIE JOSÉ BLIN
*LAMSADE - Université Paris IX, 75775 Paris CEDEX France*
*blin@lamsade.dauphine.fr*

JACQUES WAINER and CLAUDIA BAUZER MEDEIROS
*IC - UNICAMP - CP 6176, 13081-970 Campinas SP Brazil*
*wainer-cmbm@ic.unicamp.br*

This paper presents a new formalism for workflow process definition, which combines research in programming languages and in database systems. This formalism is based on creating a library of workflow building blocks, which can be progressively combined and nested to construct complex workflows. Workflows are specified declaratively, using a simple high level language, which allows the dynamic definition of exception handling and events, as well as dynamically overriding workflow definition. This ensures a high degree of flexibility in data and control flow specification, as well as in reuse of workflow specifications to construct other workflows. The resulting workflow execution environment is well suited to supporting cooperative work.

*Keywords*: Workflow reuse; workflow evolution; and flexible workflow definition.

## 1. Introduction and Motivation

Workflows are gaining increasing acceptance in the business world as a means of documenting and organizing procedures, as well as helping the coordination of groups. Workflows were conceived to describe procedures which can be repeated over and over again, for business environments and enterprise modeling.

The term *workflow* denotes "automation of a business process, in whole or part, during which documents, information or tasks are passed from one participant to another for action, according to a set of procedural rules".[18] Formally, a *workflow definition* describes the part of a business process that comprises automated processing (i.e. does not involve specification of manual activities),[30] whereas *process definition* also includes manual activities. Workflow process definition comprises a number of discrete steps, including human and computer tasks as well as rules that define how the process should progress.[30] In this paper, however, we follow the trend of using the generic term *workflow definition* to include both automated and manual tasks.

Several workflow definition formalisms have been proposed in the last five years, ranging from very high level abstraction constructs to low level programming specifications. Each formalism aims to provide the persons who specify workflows — here called *workflow designers* — with better means to describe processes. Workflow

definitions are expressed through set of tasks, along with their interdependencies, inputs and outputs,[26] as well as the actors responsible for executing these tasks. Formalisms differ among each other in terms of how tasks are specified, the kinds of control and data flow considered, and in actor/role designation. Nevertheless, there are still several open problems in the domain of workflow definition. Examples include the constraints imposed on a process, exceptions raised during its execution, and flexibility in workflow definition. Furthermore, according to Sheth and Kochut,[27] new solutions are needed to support scalability and adaptable and dynamic workflows.

This paper is a step towards filling some of these gaps. Our proposal is based on two main concepts:

- The construction of an open and extensible "library" of *workflow building blocks*, which encapsulate rules and variables. These blocks can be progressively composed and nested to build more complex blocks and workflows. Furthermore, they can be dynamically redefined at execution time. These two features ensure *specification reuse* and flexibility. The main advantages of reusability are: increase in flexibility, adaptability to change, and quality control of workflow specifications, which evolve over time. As mentioned in Ref. 19, workflow evolution has several causes, including progressive improvements and corrections, customization to the needs of a case, or adjustments to real world situations. This is discussed at more length at Sec. 5.
- A well defined *workflow definition language*, which constructs workflows by combining blocks in the library. Unlike other languages for workflow specification, our language allows defining not only the "static" workflow constructs — i.e. the sequencing and nesting of the blocks and tasks — but also their dynamic behavior, by associating event handling rules, and control/parameter passing among workflow components.

We combine two research domains in order to support our approach: databases and programming languages. The building block library as well as the documents managed by workflows are handled within a database management system (DBMS) environment. Database systems are already used in workflow management systems as a basis for sharing documents and ensuring their security. We extend their use to that of storing the workflow building blocks, similar to what is already done in other contexts (e.g. of scientific workflows[1,33,34]).

Workflows are defined and instantiated using a declarative programming language treatment. The design of the building blocks in the library uses an object-oriented approach, separating their interface (the parameters) from their contents — encapsulated tasks, rules and other blocks. A workflow definition is instantiated (i.e. executed) through dynamic activation of its blocks according to the "flow" specified in the definition. The use of a building block, in this sense, can be compared to that of procedure invocation in a program — at runtime, procedures are executed according to parameter values. By the same token, during workflow

instantiation, blocks are activated according to the values of the parameters (events and state variables) they accept in their interface. Furthermore, block parameters have a naming scope within a workflow (e.g. events are only recognized within a given workflow section), and can be renamed, just like variables in a programming language.

As will be seen, our approach supports reuse and scalability, not only in workflow specification but also at runtime. First, from a specification point of view, any workflow can be progressively augmented by composing library blocks and creating new blocks using sequencing, nesting and parallelizing of blocks. Second, at runtime we postulate a flexible instantiation of blocks using a language approach. This allows communication among blocks by parameter passing via their interfaces.

Furthermore, dynamic and distributed workflow specification is also ensured, thanks to the dynamic overriding of block definitions and the event-handling features provided by our language, similar to those found in active databases. This allows generic specification of workflow behavior, which can then be instantiated and executed differently for each case. As remarked in Ref. 28, particular computational coordination mechanisms are designed to support cooperating actors in specific aspects of their work, and thus workflows need to be customized and adapted to contextual situations.

These properties — reuse and dynamic runtime modification — are the two main advantages of our proposal. These are features considered necessary for establishing a cooperative work environment. Actors can jointly participate in workflow construction by storing and reusing/exchanging library blocks. At runtime, communication is supported via event and parameter passing among blocks. Our language is compliant with the standards proposed by the Workflow Management Coalition[30] and subsumes other language proposals. Furthermore, its redefinition facilities are an additional feature not found in the literature.

The rest of this paper is organized as follows. Section 2 presents our workflow language. Section 3 shows an example of how this language can be used to specify a complex workflow application. Section 4 presents an overview of an architecture to implement our proposal. Section 5 gives an overview of the literature, stressing related work on workflow specification formalisms. Finally, Sec. 6 presents conclusions and future work.

## 2. The Workflow Specification Language

Our formalism consists in a declarative structured language based on the concept of *block* — our main reuse unit, which extends the usual notion of activity block of Workflow Management Coalition.[32] A block is a workflow that encapsulates tasks, events, state variables and event handling rules. State variables and rules are associated to the execution of the block, and are used to control specific application needs. Events correspond to situations in which task interruptions may be signalled, in which case some actions may optionally be taken.

The *state* or *activation context* of a block corresponds to the instantiation of a block's (declarative) specification with appropriate values substituted for the block's state variables, enabled rules and events that may be signalled within it. A block activation corresponds to the execution of the tasks and blocks which it encapsulates, within the proper context. A block may be activated several times within the same workflow, possibly simultaneously and using different values for its parameters, therefore configuring a distinct activation context. From now on, whenever we talk about "execution of block $b$" we also mean "execution of the task(s) and block(s) encapsulated by $b$, within the context of $b$'s activation". The execution of a block (and thus of any workflow) is controlled by rules and variable values.

A block $b$ may be constructed out of other blocks $b_1,\ldots,b_n$, in which case $b$ is called *composite* and $b_i$ are the *components* of $b$. A component block may again be *atomic* or *composite*. Blocks can be connected using the standard *iteration*, *sequence* and *parallelization* constructors[30] for expressing activity coordination. Atomic blocks are the smallest execution unit of a workflow. Tasks are also executable units, but are not defined within the workflow.

Figure 1 gives a schematic view of block and task. From a functional point of view, both are executable units within a workflow. The main difference between these concepts is that we are not concerned with task specification (and therefore the task's body is not shown). From a workflow definition point of view, a block just needs to know a task's interface. Our language assumes that tasks are executable units which cannot be modified (e.g. have been provided by others) whereas blocks can be coded and, as we shall see, even have their code dynamically redefined at execution time.

The figure shows the separation of a block's "code" (encapsulated) from its interface. The interface is composed of the block *name* (optional) and its input/output parameters: *state variables* and *events*. A block may *export* events to other blocks (i.e. equivalent to event signalling). It may also make its state variables visible outside its context by making them part of its interface. Furthermore, all blocks accept four specific commands — *stop, resume, exit* and *start* — which will change their state.

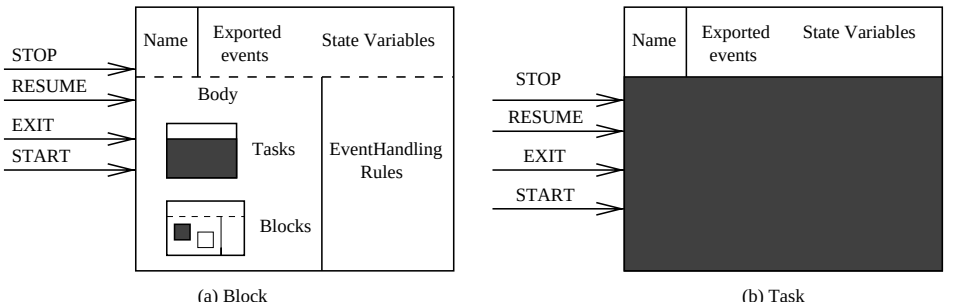


Fig. 1. Block and task schematic diagrams.

### 2.1. *Blocks and tasks*

Tasks are declared by the `deftask` construction, which defines the task name, its interface (the events that it exports and the state variables that it imports or exports) and the path for a directory or file where the task's contents is stored (e.g. a programming library):

```
(DEFTASK task name interface definition path definition)
```

Blocks are declared by the `defblock` construction, which defines an optional block name, the block's interface (the events that the block exports and the state variables that it imports or exports) and body. The `external-events` section of the block definition lists all external events (i.e. events generated outside the workflow) for which this block has an event handling rule.

```
(DEFBLOCK [block name] interface definition
          [(EXTERNAL-EVENTS external-events section )]
          (BODY body section )
          [ (EH-RULES eh-rules section) ]
)
```

The `body` section defines the internal components (blocks and tasks) of a block and is intended to describe the standard execution of these components. The `eh-rules` section lists all event handling rules.

These means of definition are natural for a workflow application and, what is more important, facilitate reuse. In terms of naturalness, the block allows expressing two important concepts in workflow applications: the normal or standard flow of activities (which in our language is expressed in the *body* section of the block definition); and event handling rules. Thus, blocks embody aspects needed in workflow definition, such as task ordering, as well as information to be used at run-time (such as task enabling conditions).

Most blocks have a name, to allow reuse and help workflow definition. Anonymous blocks are special blocks which have no name. This allows the definition of sub-workflows which serve special purposes — e.g. environment setting — but which cannot be reused later.

Blocks and tasks within a block's body may be organized according to `seq`, `par`, `loop`, and `parloop` constructs. `(SEQ` $A$ $B$ `)` states that $B$ is executed after $A$ terminates. It corresponds to the standard sequencing of activities in workflow languages. `(PAR` $A$ $B$ `)` states that $A$ and $B$ start at the same time, and that both must be terminated for the execution to continue. It corresponds to an *and-split* followed by an *and-join* in workflow languages.

`(LOOP` $A$) will execute $A$ in an infinite loop, whose termination is forced when $A$ raises some specific event. `(PARLOOP` $n$ $A$) is a non-standard control operator in workflow languages, which launches $n$ parallel instances of block $A$. It is useful when

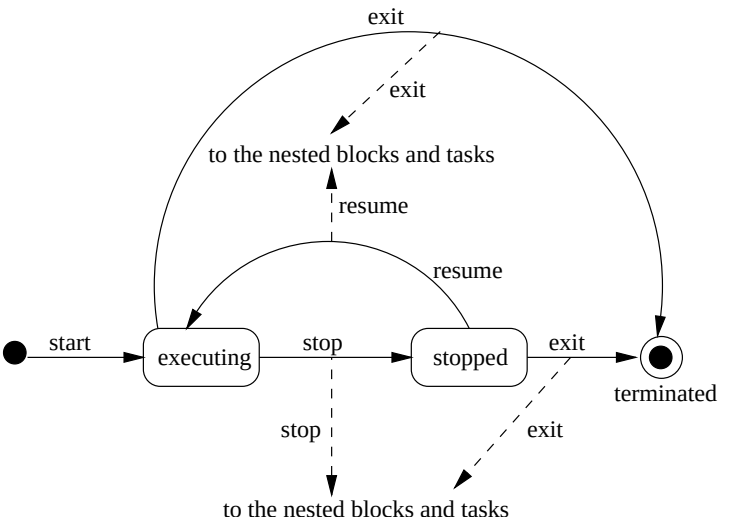


Fig. 2. Block state-transition diagram.

one must start a number of parallel branches of the same block, but at modeling time one does not know how many: $n$ is dynamically determined at execution time.

Blocks can be in three states, as described in Fig. 2. A block can be executing, which means that some of its internal blocks or tasks are executing. A block can be in the stopped state, which means that all its internal blocks and tasks that were executing are also stopped. And finally a block can be in the terminated state (meaning its execution cannot be resumed). In the figure, full lines correspond to block's state transitions. Dashed lines represent the fact that a given state change in a block requires the same state changes to be forced on the blocks and tasks it encapsulates.

If an instantiation of a block receives a `start` signal, and it has not yet started, it starts executing — i.e. becomes active for a given activation context. Thus, its variables are instantiated and its event handling rules are added to the set of rules that can be fired at any time. If the block is already in the executing state, or the stopped state, it ignores the signal.

If the block is in the executing state, and receives the `stop` signal, this signal is propagated to all its executing sub-blocks and tasks; the block's state is saved and it goes into the stopped state. If the block is in any other state, it ignores the `stop` signal.

If the block is in the stopped state, and receives a `resume` signal, its internal state is restored and the signal is propagated to its stopped sub-blocks and tasks. For all other states, the resume signal is ignored.

Finally the `exit` signal terminates the block execution: if received in the executing or stopped states, it will be propagated to all its executing or stopped sub-blocks and tasks, and the block goes to the terminated state, disabling all of its

Table 1. Options for rule definition.

| | |
|---|---|
| Event | Role = Mandatory<br>Source ⊂ {Behavior invocation, Block, Clock, External}<br>Type ⊂ {Primitive, Composite}<br>Operators ⊂ {or, and, seq, times} |
| Condition | Role = Optional<br>Context = $B_C$ |
| Action | Options ⊂ {Behavior invocation, Event, Inform, External}<br>Context ⊂ {$B_S$, $B_E$, $B_C$} |

rules. Furthermore, a block's termination provokes the abort of all eh-rules which were part of its activation state, and which had been triggered.

### 2.2. *Events and event handling rules*

Events are captured by event handling rules, declared in the `eh-rules` section. Event handling rules are similar to *<E, C, A>* rules in active databases. E stands for the Event upon which the rule is triggered, C the Condition (predicate) to be tested, and A the Action to be performed if the condition is met. From now on, the term *rule* in this paper will denote this triple.

In order to describe our rules in detail, we use the same approach as in Ref. 23, separating rule specification from the execution semantics. Table 1, adapted from Ref. 23, shows the basic rule components, divided in *event*, *condition* and *action*. EH-rules ensure flexibility during workflow execution.

**Basic notions.** An event is something that happens at a point in time, and all rules must have an event specified (*role* = mandatory). Events can have different types of *sources* — see the Table. *Behavior invocation* events are those raised by activation of tasks and nested blocks. Examples of behavior invocation events are "BOB" and "EOB", denoting respectively begin and end of a block. *Block* events are those specifically raised and exported by blocks, being part of a block's interface. *Clock* events are those that are signalled by the passing of time (e.g. "12-noon"). Finally, *external* events are those which happens outside the workflow execution environment (e.g. arrival of documents in an office). A block must declare in its `external-events` section, which external events it treats.

External events can be seen and processed by all the blocks composing a workflow. Their names are global to the entire wokflow.

Events can be primitive or composite. Composite events are those created by recursively combining less complex events using the set operators of Table 1 — e.g. "$e_1$ or $e_2$".

Conditions are predicates on the state of specific blocks or predicates on existence or non-existence of events. The *condition* field of a rule is not compulsory (and thus a condition's *role* is optional). A condition's *context* defines for which block state condition is to be evaluated. In our case, the context is always $B_C$ — at the

point where the condition is tested. Examples of conditions are `(active?block)`, `(terminated?block)` and `(stopped?block)` verifying if a block is in the corresponding states.

*Actions* may have distinct execution options. *Behavior invocation* actions activate blocks or tasks. *Inform* actions notify given users/addresses — (`notify adress`), whereas *external* actions invoke external procedures (`call procedure`). *Event* actions are those that raise events (`raise`), that can send a start, stop, resume, or exit signal to a block or a task (e.g. `start` *block* | *task*, `stop` *block* | *task*). The `stop` and `exit` actions without any argument refer to the block that contains the `eh-rule` itself. An action's *context* defines when the action is to be performed: at the start of the execution of the current block — $B_S$; at the point where the event was raised — $B_E$, or at the time the condition is actually checked — $B_C$.

**Rule execution environment.** Once the syntax for defining a rule is specified, we must consider the execution environment. In terms of rule execution, there are five stages which must be considered again[23]: the *signalling* phase in which an event occurs; the *triggering* phase in which the relevant rules are triggered; the *evaluation phase* in which conditions are evaluated, generating a "conflict set" containing all rules where conditions are satisfied and thus where actions are prone to be activated; the *scheduling* phase determines how to process rules within the conflict set; and the *execution* phase which corresponds to action execution.

These phases are executed according to two coupling modes: Event-condition and Condition-action. The usual coupling modes found in rule execution environments include *immediate* — e.g. the condition is evaluated immediately after an event is raised — and *deferred* — e.g. condition evaluation is deferred to some point in the execution (for instance, until block end). In our case, all coupling modes are *immediate* and therefore we will not concern ourselves with discussing other alternatives.

The *consumption mode*, which defines how an event should be treated by the event manager in case several events of the same type arrive is *chronicle*, i.e. events are managed in a queue by the event manager, which must take into consideration which block raised the event. When a rule triggered by an event is activated, the event is consumed. Since blocks are reusable units, the same event may be raised by several instanciations of a given block, indicating that the corresponding rule must be fired distinct times, for each activation context.

Scheduling of actions may follow distinct priority policies. Again, in our case, we assume strictly FIFO, meaning actions are executed in the same order the rules were triggered. This means, among other things, that a workflow designer cannot associate priorities to rules which would override this execution policy.

Some events are generated internally to a block and processed by it (we call these internal events). An internal event of a block `b` may be produced by the blocks or tasks inside `b` or may refer to the begining or end of the component blocks. For example, if the block `a` is used in the body section of `b`, then during the execution

of `b`, the event `(eob a)` refers to the event "end of block `a`". `(eob a +1h)` is a composite event that has two sources (behavior — eob — and clock — 1h) and will be signalled one hour after block `a` ends. `(bob +2h)` is an event that will be raised 2 hours after block `b` itself started.

A block's exported events are declared in its interface. A block `b'` can make available an internal event to its immediate ancestor `b`, by declaring the event as *exported* in its interface. Then, the event becomes an internal event of `b`, i.e. `b` can process the event by an event handling rule or it can again export it. To allow for reuse, exported events are named similarly to parameters and arguments of procedure definitions and procedure calls in programming languages. When a block is activated, there is a positional matching between the exported event internal and external names.

The execution of `eh-rules` is asynchronous with the execution of the blocks where they are declared: if while a block is executing an event is raised, the relevant `eh-rules` will execute. The event handling rules are only active when the block in which they are defined is executing or stopped. If the block is terminated the rules cannot be activated.

For example, consider the following block specification, where `a` is invoked by `b`.

```
(DEFBLOCK a (ev1) ()
...
)
(DEFBLOCK b (x) ()
    (BODY
        (a (x) () ) )
    (EH-RULES
        (x, , actionx) )
)
```

`ev1` is an exported event of `a`. When `a` is activated within `b`, `ev1` is known as `x`, which is also exported by `b`. If, during the execution of `a`, the event `ev1` (and thus `x` for `b`) is raised, the action *actionx* will be executed.

### 2.3. *State variables*

State variables are the third central concept of the language. The basic idea of this concept is to be able to represent resources that will be used by and transferred among the tasks and the blocks at execution time. A calling block gives effective resources to a callee block in transferring data to it and callee block tranfers results in the same manner. In the interface, resources granted to a block (input) are preceded by symbol "+", whereas resources it transfers to other blocks (output) are preceded by symbol "∧". A symbol "?" before a variable name in a call declares that the variable is shared between the calling and the callee blocks. At execution

time, variable declarations are matched from calling to callee blocks according to their order, just as in a programming language. The transfer of variables is usually by copy except when the variable is preceded by "?".

### 2.4. *Features to enable reuse*

A primordial objective of our language is to allow flexibility in workflow creation. This is attained by supporting the reuse of the definitions of existing workflow, by allowing their integration as building blocks (sub-workflows) into new workflows, and by providing the possibility of dynamically redefining workflows. We recall that flexibility in workflow specification is one of the basis for a cooperative workflow construction environment.

Several specific features of the language were developed to achieve the reuse goal: the first one, explained in the previous section, is the dynamic matching between calling/callee interfaces. Other features that ensure this are: (i) the `PARLOOP` instruction; (ii) mechanisms for overriding rules, interface definitions, and task and block definitions; and (iii) the possibility to permanently modify tasks and blocks.

As explained in Sec. 3.1, the `PARLOOP` instruction allows to launch simultaneous parallel executions of the same workflow a number $x$ of times, where the value of $x$ is defined dynamically at execution time. The remaining reuse-oriented features are explained below.

#### 2.4.1. *Rule overriding*

To facilitate reuse, an event handling rule defined in a block `b` may be redefined by a block `a` which uses `b`. Thus, when there is block nesting, a rule may be redefined several times. When there is rule redefinition, in the case of event signalling, it is the outermost declaration of the rule that takes precedence. Let us consider the following example, where `b` invokes `a` and is invoked by `c`:

```
(DEFBLOCK a (ev1) ()
...
)
(DEFBLOCK b (x) ()
    (BODY
        (a (x) () ) )
    (EH-RULES
        (x, , actionx) )
)
(DEFBLOCK c () ()
    (BODY
        (b (y) ()) )
    (EH-RULES
        (y true actiony) )
)
```

Suppose `ev1` is raised in block `a` (while executing within `b`, which has been activated within `c`). This event is known as `x` within `b` and as `y` in `c`. The outermost rule for this event is `(y true` ***actiony)***, declared in block `c`, and thus *actiony* will be executed. If, on the other hand, the definition for block `c` were:

```
(DEFBLOCK c () ()
    (BODY
        (b (y) ()) )
)
```

then even though `c` is the outermost block, there is no rule to deal with event `y` (or `ev1` or `x`, for that matter). So the rule in `b` would be activated.

This rule overriding mechanism is well-adapted to the reuse of blocks. Let us examine, for example, the block `larger`, specified below.

```
(DEFBLOCK larger (late) ()
    (BODY
        ...
        (process-order (late) ())
        ... )
    (EH-RULES
        ...
        (late, , (start notice () ()) )
)
```

Let us assume that the block `process-order` generates an exported event called `late` as soon as the processing of a customer's order has taken more than a certain time. Block `process-order` is used within a larger block, with other activities and blocks and which implements the standard policy for late orders. The standard policy for late orders is just to start the block `notice` that, among other things, notifies the customer that the order is late.

Now, let us suppose that a new customer policy is decided on. This new policy starts a block `apologize` that involves running tasks which are different from the ones performed in `notice`. The new policy is implemented by invoking block `larger` and overriding the rule, capturing the `late` event.

```
(DEFBLOCK even-larger (late) ()
    (BODY
        ...
        (larger (late) ()) )
        ...
    (EH-RULES
        ...
        (late, , (start apologize () ()) )
)
```

2.4.2. *Interface definition overriding*

The interface of a block `b` may be redefined by a block `a` which invokes `b`, by adding exported events or/and exported variables. Let us examine the example below:

```
(DEFTASK a1 (ev1) () tpath)
(DEFTASK a2 (ev2) () tpath)
(DEFBLOCK b (z) ()
    (BODY
        .......
        (a1 (x) () ) )
        (a2 (y) () ) )
        .......
    (EH-RULES
        (x, , RAISE z) )
        (y, , RAISE z) )
)
(DEFBLOCK c (w) ()
    (BODY
        (b (w) ()) )
    (EH-RULES
        (w true actionx) )
)
```

Event `z` is exported by `b` and processed by `c`, where it is named `w`. The two rules within `b` raise event `z` upon detection of event `x` (from task `a1`) or `y` (from task `a2`). Since both tasks `a1` and `a2` raise `z`, in either case the rule within `c` will be activated. Suppose now that we want to specifically process the event exported by task `a1`. The definition of block `c` would be:

```
(DEFBLOCK c (w1) ()
(REDEFINESINTERFACE b (z, x) () )
    (BODY
        (b (w1,w2) ()) )
    (EH-RULES
        (w2 true actiony) )
)
```

2.4.3. *Block and task definition overriding*

Besides having its interface redefined dynamically, an entire block may be redefined by a block which invokes it. It is indeed possible to have a cascading of redefinitions of the same block. In this case, only the outermost definition is considered. In addition, a block redefinition overrides an interface redefinition of the same block. Let us suppose we add the following definition to the previous example:

```
(DEFBLOCK d () ()
;---------------------------
(REDEFINESBLOCK b (x) ()
    (BODY
        .......
        (a1 (x) () ) )
        .......
); end of redefinition
;---------------------------
    (BODY
        (c (α) () ) )
    (EH-RULES
        (α true actionα) )
)
```

Block `d` calls block `c` which in turn calls block `b`. The interface redefinition of block `b` in `c` is overriden by the redefinition of block `b` in `d`. Thus, for the execution of `c` within `d`, the redefined block `b` no longer has any EH-rule. The net effect of this redefinition cascading is that it is the outermost definition of `b` which will actually be executed within `d`. The actual execution occurs as if we had the definitions below:

```
(DEFTASK a1 (ev1) () tpath)
(DEFTASK a2 (ev2) () tpath)
(DEFBLOCK b (x) ()
    (BODY
        .......
        (a1 (x) () ) )
        .......
)
(DEFBLOCK c (w1) ()
    (BODY
        (b (w1,w2) ()) )
    (EH-RULES
        (w2 true actiony) )
)
(DEFBLOCK d () ()
    (BODY
        (c (α) () ) )
    (EH-RULES
        (α true actionα) )
)
```

Block redefinitions are dynamic and take place only during the execution of the workflow where they are redefined.

A workflow may also override the task which is invoked within a block without recoding the block. This is achieved by the `redefinestask` command, which serves to indicate that a different task is to be invoked within a reused block. Just as block definition overriding, this allows customizing workflows without affecting the block library or existing workflows. Section 3.3 contains an example of overriding task invocation.

#### 2.4.4. *Permanent modifications of blocks*

Blocks may also be definitively modified, in which case their new definition is stored in the block library. Some modifications have no consequence on existing workflows. However, just as in any programming environment, it is up to the people who define the blocks (or manage the library) to make sure there is no side effect on existing workflows. Furthermore, again as in a programming environment, tasks may also be redefined.

In order to provide a minimum of consistency to existing workflows when a block they invoke is updated in the library, we impose restrictions on their interface definition. Basically, if a block `b` is modified in the library, the only allowed modifications in its interface are those that add parameters (exported events or state variables) — i.e. the updated definition of `b` cannot have less parameters in its interface. In this case, pre-existing workflow definitions which use `b` will still execute and perform positional parameter passing, but will ignore the additional parameters. Naturally, this does not eliminate the need for ensuring the maintenance of the semantics of these pre-existing workflows. Section 3.3 contains an example of this feature. The same proviso applies to task modification.

## 3. Application

We will now discuss an application of the workflow language definition. We will use a variation of the standard paper reviewing procedure that, we believe, is familiar to the reader. The basic idea is to describe the flow of the paper reviewing procedure in a conference. The program committee chair sends papers to reviewers according to some pre-arranged protocol, and then gathers up the reviews for each paper. The examples concern the reviewer assignment/paper review/ final decision protocols. We begin by presenting a basic example of this protocol. Then, in order to demonstrate the reuse features of the language, we show how to reuse the blocks defined in the first example to describe two other protocol examples with some different procedures and policies.

### 3.1. *Paper evaluation — Basic example*

The main aspects of this basic example is that there is a list of possible reviewers. Reviewers in the list are invited to review a paper and they may refuse to do it.

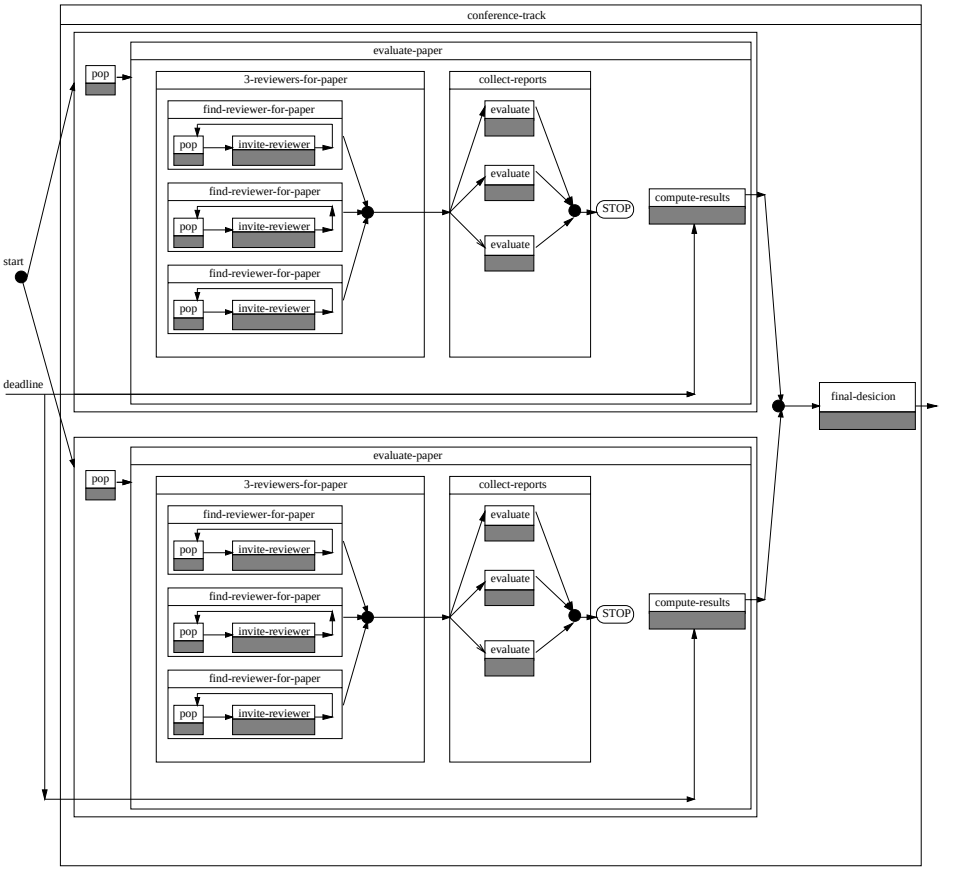


Fig. 3. A simplified graphical representation of paper evaluation example.

Each paper should have at least two and at most three reviews. If it is not the case that at least two reviewers accepted to review the paper, the paper is automatically rejected, based on the assumption it is outside the conference's scope.

The workflow is defined by a main (outermost) block named `conference-track` which embeds four blocks `find-reviewer-for-paper`, `3-reviewers-for-paper`, `collect-reports` and `evaluate-paper`. The workflow blocks invoke six tasks, `pop`, `push`, `invite-reviewer`, `compute-results`, `evaluate` and `final-decision`. The general organization of the workflow is shown in Fig. 3, which represents the different invoked blocks and tasks. The figure shows a situation where there are only two papers to be reviewed.

The figure portrays the general idea of the process, which consists of launching several `evaluate-paper` blocks in parallel and computing the final results after all papers have their evaluations returned. Each paper to be evaluated is popped from a paper list, and passed on to the `evaluate-paper` block. The latter consists of the execution of `3-reviewers-for-paper` followed by `collect-reports`. Block

`3-reviewers-for-paper`, in turn, consists on finding at least 2 and at most 3 people who accept to evaluate the paper; block `collect-reports` sends these people letters to confirm the request to evaluate, and waits for their reports. The results of the evaluations are collected at some predefined deadline and finally the final decision is taken on which papers should be accepted and rejected. The details and formal definition of the workflow are described below.

The outermost block, `conference-track`, receives a list a papers, a list of potential reviewers, and returns the final decision, with a list of accepted and rejected papers.

```
(DEFBLOCK conference-track () (+size-of-list-of-papers
                               +list-of-reviewers
                               ^final-accept ^final-reject)

  (BODY
    (SEQ
       (PARLOOP size-of-list-of-papers
          (BLOCK ( ) ( )
             (BODY
               (SEQ
                (pop (empty) (paper list-of-papers))
                (evaluate-paper (accept reject wait)
                 (paper list-of-reviewers))
               ))
             (EH-RULES
              (empty true EXIT)
              (accept true (push () (paper  ?list-of-accepted)))
              (reject true (push () (paper  ?list-of-rejected)))
              (wait true (push () (paper  ?list-of-pending)))
             )
           ) ; end of anonymous block
        ) ; end of PARLOOP

       (final-decision () (list-of-accepted list-of-rejected
                           list-of-pending final-accept final-reject))
    ))
)
```

Tasks `push` and `pop` respectively push and pop one element at a time from a list. The `pop` task also raises event `empty` if the list is empty.

```
(DEFTASK push () (+element +list) t_path)

(DEFTASK pop (empty) (^element +list) t_path)
```

Notice the `PARLOOP` structure, which allows defining that a block will be launched for the number of times provided in the variable `size-of-list-of-papers`. Since this variable is known only at execution time, this block can be used for any kind of conference that employs the work procedures embedded in block `evaluate-paper`.

The `final-decision` task has the following definition:

```
(DEFTASK final-decision () (+list-of-accepted +list-of-rejected
                            +list-of-pending ^final-accept
                            ^final-reject) t_path)
```

From the list of accepted, rejected and pending papers, the task produces the final list of accepted and the final list of rejected papers.

The main evaluation policy is implemented by block `evaluate-paper`, whose code is

```
(DEFBLOCK evaluate-paper (accept reject wait)
(+paper +list-of-reviewers)

   (EXTERNAL-EVENTS (deadline))

   (BODY
      (SEQ
        (3-reviewers-for-paper (x3 x2 nok) (paper list-of-reviewers
        r1 r2 r3))
        STOP
        (collect-reports () (paper r1 r2 r3 report1 report2 report3))
        STOP )
   (EH-RULES
      (nok true (SEQ (RAISE reject) EXIT))
      (OR(x2 x3) true RESUME)
      (deadline true (SEQ (compute-results (accept reject wait)
                          (paper x3 x2 report1 report2 report3))
                           EXIT))
    )
)
```

This block initially invokes block `3-reviewers-for-paper`, and goes to the stopped state, waiting for either a failure in finding an acceptable number of reviewers or a success. In the first case, block `3-reviewers-for-paper` signals event `nok`, which will raise the first rule.

Events `x2` and `x3` respectively indicate that 2 or 3 reviewers were found, in which case the block can resume its execution (second rule).

Event `deadline` is a time event which will raise the third rule of the block; this rule invokes the task `compute-results`.

Notice that here, as in several other parts of this section, the `stop` event is signalled between block invocations, forcing the block to enter a stopped state. This is needed to ensure synchronization between block execution and rule execution. Since rules are executed asynchronously, this is needed in some cases to make sure of the correct block execution semantics.

The task `compute-results` receives the number of reviewers that were assigned to the paper, and their reports, and raises `accept`, `reject`, or `wait` if the paper should be accepted, rejected, or is pending further discussion. The task has the following definition:

```
(DEFTASK compute-results (accept reject wait) (+paper
                          +three-reviewers +two-reviewers +report1
                          +report2 +report3) t_path)
```

Block `3-reviewers-for-paper` is in charge of finding reviewers for one paper and is coded as

```
(DEFBLOCK 3-reviewers-for-paper (three-rev two-rev not-ok)
                                (+paper +list-of-reviewers ^rev1
                                 ^rev2 ^rev3)

   (BODY
     (SEQ
        (PAR
         (find-reviewer-for-paper (ok1 nok1) (paper rev1
         ?list-of-reviewers))
         (find-reviewer-for-paper (ok2 nok2) (paper rev2
         ?list-of-reviewers))
         (find-reviewer-for-paper (ok3 nok3) (paper rev3
         ?list-of-reviewers))
)
     RAISE f
     STOP )
)
   (EH-RULES
      (f (and ok1 ok2 ok2) (SEQ (RAISE three-rev) EXIT))
      (f (or (and ok1 ok2) (and ok1 ok3) (and ok2 ok3))
         (SEQ (RAISE two-rev) EXIT))
      (f (or (and nok1 nok2) (and nok1 nok3) (and nok2 nok3))
         (SEQ (RAISE not-ok) EXIT))
   )
)
```

This block raises `three-rev` if three reviewers were found, `two-rev` if only two reviewers were found, and `not-ok` if less than two reviewers were found.

Block `find-reviewer-for-paper` will attempt to find a reviewer that accepts reviewing the paper. Reviewers are taken from the list `list-of-reviewers`. It will raise `fail-assign` if no one from the `list-of-reviewers` accepts to review the paper.

```
(DEFBLOCK  find-reviewer-for-paper (accept fail-assign)
                                   (+paper ^reviewer
                                    +list-of-reviewers)

  (BODY
    (LOOP
      (SEQ
         (pop (empty-list) (reviewer ?list-of-reviewers))
         (invite-reviewer (accept refuse) (paper reviewer))
   STOP
      )
    )
  )

  (EH-RULES
     (empty-list true (SEQ (RAISE fail-assign) EXIT))
     (accept true EXIT)
     (refuse true RESUME))
)
```

The actual formal invitation is performed by task `invite-reviewer`. This task will cause the paper and a letter of invitation to review to be sent to the `reviewer` who may `accept` or `refuse` to review that paper:

```
(DEFTASK invite-reviewer (accept refuse) (+paper +reviewer) t_path)
```

Finally, block `collect-reports` returns the reports of each reviewer of a paper.

```
(DEFBLOCK collect-reports () (+paper +reviewer1 +reviewer2 +reviewer3
                              ^report1 ^report2 ^report3)

   (BODY
      (SEQ
        (extract-paper-id () (paper, paper-id))
        (PAR
          (evaluate () (paper-id reviewer1 report1))
          (evaluate () (paper-id reviewer2 report2))
          (evaluate () (paper-id reviewer3 report3))
         )))
)
```

The task `evaluate` called in the block is defined as

```
(DEFTASK evaluate () (+paper +reviewer ^report) t_path)
```

It confirms to the `reviewer` that this paper has actually been assigned and the reviewer eventually responds in sending the report. If the variable reviewer is null, the task does nothing.

Task `extract-paper-id` defined as

```
(DEFTASK extract-paper-id () (+paper, ^paper-id))
```

returns the identifier for a paper, that is used in further communication with the reviewers. The paper identifier is supposed to be in the paper header.

### 3.2. *Paper evaluation — A different conference*

Consider now a variation of the previous example. In this variation, if no reviewer accepts to review the paper, it is assigned to two PC members. A new issue arises, in comparison with the previous example: there is the problem of passing along the list of PC members, from the `conference track` block down to the `3-reviewers-for-paper` block, where it is used.

The outermost block, here `conference-track2`, is defined as:

```
(DEFBLOCK conference-track2 () (+size-of-list-papers +list-of-papers
                                +list-of-reviewers +list-of-PC
                                ^final-accept ^final-reject)
  (BODY
    (SEQ
       (PARLOOP size-of-list-of-papers
          (BLOCK
             (BODY
               (SEQ
                (pop (empty) (paper list-of-papers))
                (evaluate-paper2 (accept reject wait)
                                 (paper list-of-reviewers list-of-PC))
                ))
             (EH-RULES
              (empty true EXIT)
              (accept true (push () (paper list-of-accepted)))
              (reject true (push () (paper list-of-rejected)))
              (wait true (push () (paper list-of-pending)))
             )
          ) ; end of anonymous block
       ) ; end of PARLOOP
```

```
       (final-decision () (list-of-accepted list-of-rejected
                           list-of-pending final-accept final-reject))
    ))
)
```

The block `evaluate-paper2` redefines the interface of the block `evaluate-paper` to be able to override the rule for processing the event `nok`. It is defined as

```
(DEFBLOCK evaluate-paper2 (accept reject wait)(+paper
+list-of-reviewers +list-PC)
(REDEFINESINTERFACE evaluate-paper (accept reject wait nok) (+paper
                                    +list-of-reviewers))

   (BODY
      (evaluate-paper (accept reject wait nok)
                           (paper list-of-reviewers))
   )
   (EH-RULES
      (nok true (SEQ
                  (pop (x) (rev1 ?list-PC))
                  (notify-PC () (paper rev1 report1))
                  (pop (y) (rev2 ?list-PC))
                  (notify-PC () (paper rev2 report2))
                  (collect-reports() (paper rev1 rev2 null report1
                   report2 null))
                  (compute-results (accept reject wait) (paper null
                   x2 report1 report2 null)
                )))
)
```

The task `notify-PC` is defined as

```
(DEFTASK notify-PC () (+paper +PC-member ^report) t_path)
```

and it will cause the paper to be reviewed to be sent to the `PC-member` who responds in sending his report.

### 3.3. *Yet another different conference*

As a third example, let us suppose that in the first example, we modify the reviewer invitation process. Instead of inviting and expecting an answer, we send the paper, with an invitation letter to the reviewer. If the reviewer refuses to evaluate the paper, he will indicate this, otherwise it is expected that the report will be sent before the deadline.

This modification is embedded in a new task `invite-reviewer`. The new task is automatically invoked by the block because there is a `redefinestask` declaration.

This new task is stored in a different path and its interface is distinct from the other `invite-reviewer` task used until now.

```
(DEFBLOCK conference-track3 () (+size-of-list-of-papers
                                +list-of-papers +list-of-reviewers
                                ^final-accept ^final-reject)

;-------------------------------------------------------------
(REDEFINESTASK invite-reviewer (ack no) (+doc +name ^report) tpath2)
;-------------------------------------------------------------
(REDEFINESBLOCK evaluate-paper (accept reject wait)(+paper
 +list-of-reviewers)
    (EXTERNAL-EVENTS (deadline))
    (BODY
        (SEQ
            (3-reviewers-for-paper (x3 x2 nok)
                        (paper list-of-reviewers r1 r2 r3 rp1
                         rp2 rp3))
            STOP)
    )
    (EH-RULES
        (nok true (SEQ (RAISE reject) EXIT))
        (deadline (true) (SEQ
                            (compute-results (accept reject wait)
                                             (paper x3 x2 report1
                                              report2 report3))
                             EXIT))
    )
 ) ; end of redefined block
 ;-------------------------------------------------------------

    (BODY
        (conference-track () (list-of-papers list-of-reviewers
                              final-accept final-reject))
    )
)
```

The blocks `find-reviewer-for-paper` and `3-reviewers-for-paper` are permanently modified as:

```
(DEFBLOCK 3-reviewers-for-paper (three-rev two-rev not-ok)
                  (+paper +list-of-reviewers ^rev1 ^rev2 ^rev3
                   ^rp1 ^rp2 ^rp3)
```

```
   (BODY
     (SEQ
        (find-reviewer-for-paper (ok1 nok1) (paper rev1
         ?list-of-reviewers rp1))
        (find-reviewer-for-paper (ok2 nok2) (paper rev2
         ?list-of-reviewers rp2))
        (find-reviewer-for-paper (ok3 nok3) (paper rev3
         ?list-of-reviewers rp3))
     )
   )

   (EH-RULES
      (EOB (and ok1 ok2 ok2) (RAISE three-rev))
      (EOB (or (and ok1 ok2) (and ok1 ok3) (and ok2 ok3))
           (RAISE two-rev))
      (EOB (or (and nok1 nok2) (and nok1 nok3) (and nok2 nok3))
           (RAISE not-ok))
   )
)
```

The interface of the block `3-reviewers-for-paper` is modified to output the `report` variable and the invocations of the blocks `find-reviewer-for-paper` are updated.

This new version of the block `3-reviewers-for-paper` is stored in the library, but will not affect the execution of pre-existing workflows (e.g. the one described in Sec. 3.1). That original workflow assumes the old interface of `3-reviewers-for-paper`, which does not return the reports. If the first workflow is now used, the variables `rp1`, `rp2` and `rp3` will not match with any calling block parameters.

```
(DEFBLOCK  find-reviewer-for-paper (ok fail-assign)
                                   (+paper ^reviewer +list-of-reviewers
                                    ^report)
(BODY
    (LOOP
      (SEQ
         (pop (empty-list) (reviewer ?list-of-reviewers))
         (invite-reviewer (accept refuse) (paper reviewer report))
       )))
(EH-RULES
     (empty-list true (SEQ (RAISE fail-assing) EXIT))
     (accept true EXIT)))
)
```

Notice that the interface of the block `find-reviewer-for-paper` is modified to output the `report` variable, and the call to the task `invite-reviewer` is also changed to invoke the redefined task `invite-reviewer`.

Like before, this new version of the block is stored in the library, but will not affect the execution of pre-existing workflows. The original workflow will continue using the ancient definition of the task `invite-reviewer`, whose interface moreover does not return the report.

## 4. Architecture

A workflow management system (WFMS) must offer the designer several tools for specifying workflows and the data they manipulate. Several architectures rely on the so-called *workflow repository*, which stores workflow definitions. WFMS architectures dispose of two main functional modules: *repository services*, where workflows are stored and managed, and *enactment service*, to coordinate workflow execution. The latter enforces inter-task dependencies, schedules activities, manages data and is concerned with the overall reliability of the execution environment. Additional modules provide security, authentification, user group monitoring and reporting (e.g. workflow logs).

Our architecture, schematically shown in Fig. 4, follows the basic model of the Workflow Management Coalition WfMC,[30] comprising three main component types:

- Software components for providing support for functions within the workflow system.
- Control data and systems.
- Applications and application databases, which are accessed by a workflow instance during run-time.

To ensure exception handling and workflow interoperability, we add two components to the architecture proposed by WfMC: the *event manager* and the *repository*. The repository contains workflow definitions and shared data to be used by all workflows and their component sub-workflows.

During workflow/process execution, the workflow engine coordinates the sequencing of activities and the invocation of application tools. We recall that we consider blocks to be sub-workflows. Thus, rather than having one single workflow engine, our architecture ensures that the activation of a block will instantiate a temporary workflow engine, which will be responsible for the block's execution. Each engine manages the block's internal control data, as well as its state variables. This optimizes execution of the nested blocks, and block reuse. As blocks may be simultaneously executed, several workflow engines may be active at the same time.

Figure 4 shows the architecture of our system. At the beginning of the execution of a workflow, only one workflow engine exists and executes the highest level (outermost) block. This block is activated with the parameters provided by the

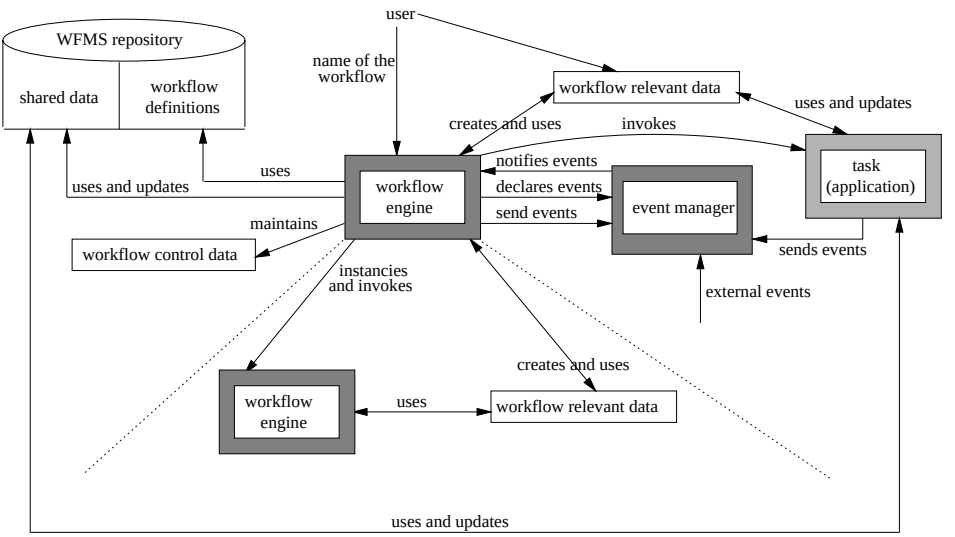


Fig. 4. Architecture of the WFMS.

user (workflow relevant data). When the engine encounters a task definition, it invokes the associated task module (which is stored in the workflow repository). If it encounters a block definition, it instantiates the corresponding (sub-)workflow and triggers its execution by means of a temporary engine.

Workflow instantiation is as follows. If a block's body is not redefined, its definition is extracted from the workflow repository. Interface redefinitions, when found, overide the ones stored in the repository. Next, a workflow/process is defined, together with the relevant data containing the effective parameters of the block, including block redefinition, and the identification of exported events handled by event-handling rules. When a workflow execution is terminated (corresponding to the terminated state of the block) its engine sends an exit event together with the block's identification to the event manager.

The event manager processes all the events generated during the execution of a workflow. Each workflow engine declares to the manager the external events it processes and sends to it the events it exports together with its identification. The event manager notifies each workflow engine of the arrival of the external events it recognizes, of the events exported by its sub-workflows and of their `exit` events. When a workflow engine receives an `exit` event from one of its sub-workflows, it first extracts the sub-workflow's relevant data and then deletes its temporary engine.

Figure 5 shows an instantiation of this architecture for the execution of block `3-Reviewers-for-paper`. To keep the figure readable, we did not show the execution of the block `find-reviewer-for-paper`. The figure represents the normal case, i.e. the case when block `find-reviewer-for-paper` finds a reviewer who accepts the paper.

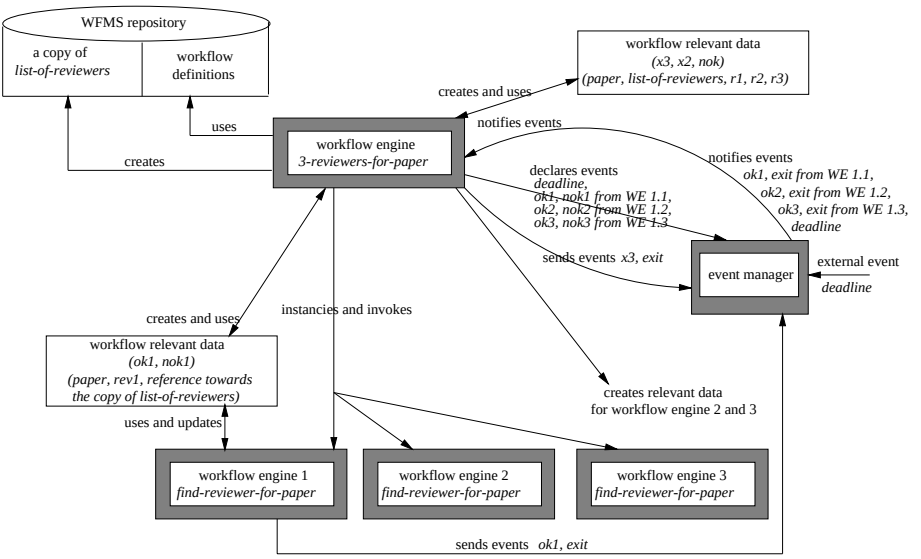


Fig. 5. A WFMS execution example.

First, the workflow engine for block `3-reviewers-for-paper` starts. It receives relevant data containing the effective parameters (in particular, the reference towards a copy of `list-of-reviewers` which has been created by the outermost block `conference-track`). During its execution, it instantiates three workflow engines, one for each instance of the block `find-reviewer-for-paper` and launches them with their associated relevant data. These engines, in turn, dynamically construct the associated workflow definitions (selecting the block definition in the database) and execute them. They send an `ok` event to the event manager followed by an `exit` event. The event manager notifies the workflow engine of the block `3-reviewers-for-paper` of the arrival of each of the events `ok1, ok2 and ok3`, together with the identification of their source. When an `exit` event sent by a sub-workflow engine is notified to the engine `3-reviewers-for-paper`, the latter picks up the variables `reviewer1, reviewer2` or `reviewer3` from the data maintained by the corresponding sub-workflow engine and deletes that sub-workflow engine. When the three sub-workflow engines have finished their execution, event `f` is raised by the workflow engine `3-reviewers-for-paper` which sends its identification plus two events — `three-rev` followed by `exit` — to the event manager.

## 5. Related Work

Research on workflows covers a wide gamut of aspects, ranging from the work environment where they are used (e.g. office procedures, communication paradigms[7]) to their specification and execution. A recent trend concerns the use of workflows across the Web, to support cooperative work organization (e.g. the special issue

on internet-based workflows[24], or the work on coordinating communication among workflows[5]). Within this same context, a first draft of XPDL, an XML standard for workflow definition has been published.[31]

Our paper concentrates on workflow specification and instantiation mechanisms. Related work is usually linked to that on WFMS and covers several aspects which vary from architectural issues to prototype implementation. Surveys[2,21] discuss some open research issues in the area. As stressed in these papers, workflow management systems originated from office automation or CSCW applications, and thus have been geared towards routing, task identification, sharing and cooperation. Limitations mentioned range therefore from lack in flexibility in their specification to operational problems (such as scalability, fault tolerance, exception handling, concurrent execution and long transactions). Several of the languages and systems described but not all are WfMC compliant.

### 5.1. *Main research trends*

Roughly speaking, research on workflow definition and workflow management systems is centered on three main axes: (i) architectures and prototypes; (ii) extensions to support non-business-oriented applications; and (iii) specification formalisms and environments.

Research on *architectures and prototypes* is either based on extending DBMS mechanisms (to ensure appropriate transaction management, concurrency control and persistency — e.g. in Refs. 11, 15 and 20) or on artificial intelligence agent-based proposals (to optimize parallelism, execution dynamics and task distribution — e.g. Ref. 8). Support for collaborative facilities and reuse is also a common goal (see, for instance in Ref. 27). Reuse and parallelism are frequently mentioned in the context of enterprise-wide management, where several sites and/or enterprises need to share and exchange documents.

Research on *extensions* to support other types of applications requires construction of new WFMS. An example of this type of work is that geared towards the so-called scientific workflows, proposed to describe activities typical of experimental (scientific) environments — e.g., Ref. 1, 33 and 34.

Finally, research on *specification formalisms* — which is the main focus of this paper — aims to provide workflow designers with more expressive power in order to better describe their applications. In the rest of this section, we concentrate on this research aspect. This kind of work is either formulated in terms of state transition constructs (e.g. based on graphs, Petri nets, or statecharts) or on declarative languages (e.g. standard structured languages, or object-oriented and reflexive languages, eventually extended with ECA active database rules). In some cases, specification tools combine declarative programming with a graphical interface.

Specification formalisms have evolved from simple state transition graphs with little semantics to progressively more complex structures, separating data from control flow, to provide more expressive power and to allow flexibility in specification.

However, the richer the constructs, the more complex becomes the specification. Indeed, some formalisms are restricted to low level details such as variable values, and are yet unable to allow designers to abstract data and control flows, complicating the task of workflow definition.

**State-centered proposals.** Workflow formalisms which adopt state-centered specifications are usually based on graphical notations. Most of these proposals rely on digraph representations, where nodes correspond to activities and directed edges correspond to data and/or control flow. An example is the work of Gal and Montesi,[14] where the emphasis is on modeling the communication between different enterprises, represented by their business workflows, at a very high abstraction level. Here, the workflow is expressed by means of Petri nets, which allows defining synchronization events. An exception to the digraph specification paradigm is the work of Wodtke *et al.*[35] — the Mentor project — which uses *Statecharts* to specify workflows. This type of formalism presents a marked evolution in comparison to other state-based proposals, since it allows expressing events and triggering of activities in the diagram, with as many abstraction levels as desired. Nevertheless, their events are still synchronous, and can only be raised at the end of an activity. We, on the other hand, consider asynchronous events, which can furthermore be raised during the execution of an activity.

Yet another state-based formalism relies on UML Activity Diagrams to specify workflows, and design patterns.[9] UML (the Unified Modeling Language[25]) is a software engineering theoretical framework which is being increasingly used to model and document software. Based on several kinds of complementary diagrams, it allows designing static and dynamic software components, associated databases, and actors involved in software execution. Activity Diagrams correspond to the formalism within UML which is responsible for modeling computational and organizational processes. They are transition diagrams akin to state machines. Because of their characteristics, there have been some attempts to use these diagrams to specify workflows. As pointed out by Dumas and ter Hofstede,[10] however, though they allow specification of several workflow characteristics, they do not support synchronization of blocks or activities, and some of their constructs lack precise syntax and semantics.

**Language-based proposals.** Most language-based proposals for workflow specification concern the definition of one language. An exception is the Ariadne[29] notation system, which provides tools for specifying a language which a designer can then use to specify a workflow. Ariadne defines families of symbols for specifying workflows, and rules for this specification. It comprises three levels for workflow specification, in which the most abstract level determines possible grammars to be applied and the two others construct workflows by applying rules of the chosen grammar. Though all tasks considered are primitive, its main goal is the coordination of cooperative actions.

All other language-based proposals concentrate on specification of one language and its constructs. Representative examples, described below, are the work of E. Gokkoca *et al.*[12] and R. Hull *et al.*,[16] and that of F. Casati *et al.*[6] and Z. Luo *et al.*[22] in the case of active rules.

E. Gokkoa *et al.*[12] propose a block-based declarative structured language to specify workflows. Blocks are used to encapsulate tasks and therefore provide more abstraction to workflow specification, but have no specific semantics associated. Thus, their blocks cannot be defined within other blocks (a usual feature workflow graphical representations).

The goal of the Vortex environment[16,17] is to specify heuristic workflow descriptions, as well as to allow tuning of their execution. The specification language combines graphical properties (called DecisionFlows) with textual specification — the language is based on the OQL database language standard. Vortex takes advantage of active database facilities to define attribute values: it is ECA rules that determine how and when to query a database to obtain attribute values.

Several other language-based proposals use active databases to serve as the platform for WFMS, and as a consequence workflows are defined in terms of ECA rules. Workflows and rules are stored in a database; the rules may either handle data states or workflow execution (e.g. coordinating state transitions). Active database approaches are usually low-level (rules concern variables or database states, but not complex workflows) and do not allow nesting of workflows, due to limitations of active database systems. The work of F. Casati[6] exemplifies this type of system. They employ a textual declarative means for defining a workflow, which combines database specification (to define schemata) and rule definition. The reactive approach is also taken by Z. Luo *et al.*[22] who enhance ECA rules by defining the context in which they are examined, becoming J-ECA rules (Justified ECA). In this proposal, reactive capabilities allow a workflow to reason about its current state and act accordingly to handle exceptions.

Finally, the WIDE project[4] combines the use of patterns and event handling within database ECA rules. The goal — like that of our paper — is to allow execution flexibility. Their event handling concerns are similar to ours; however, these concerns are dealt with using a database perspective, and workflow specification is based on patterns and their specialization using inheritance mechanisms. In particular, inheritance is one of the means of achieving reuse.

**Combining graphic and textual notation.** Finally, some formalisms require combining declarative languages to state-based diagrams. An example is the Atreus[13] environment. It associates textual metadata to a workflow graphical specification, in order to document activities and specify execution conditions. Besides the usual concepts of top-down task refinement and sequence and parallel constructs, it also raises the question of deterministic and non-deterministic parallel tasks. Many other authors also use metadata attached to workflows to help documentation of activities (e.g. in Ref. 36).

### 5.2. *Comparison with our work*

Our work can be considered similar to that of Atreus[13] as far as events are concerned. However, it extends[13] by allowing an arbitrary complexity of block nesting, as well as event propagation within and across blocks. We can also draw a parallel of our language to the Mentor statechart specification. However, they do not allow reuse of blocks, and their use of data and variables is mostly for documentation purposes in the activity chart. Ours can control the flow.

There is very little research on the expressiveness of languages to specify workflows. One such effort is the work of Bonner,[3] who is intent on showing how Transaction Datalog can be used to specify workflows in a database framework. Transaction Datalog is a powerful formal tool used in studying database programming languages. It is a (database) programming language that supports concurrent process specification and advanced transaction modeling. Since it is a database language, but also supports concurrency and embedded processes, it allows definition of both behavior and state features. Bonner's work is concerned with issues such as preservation of states, and computational complexity of workflow specification using this language, under specific conditions. While his study is centered on formal issues, the work reported in Ref. 10 studies expressivenes from a practical perspective. It analyzed the expressiveness of UML activity diagrams against a set of 21 design patterns that specify desirable workflow constructs, some of which not supported by commercial WFMS.[9] These two papers[3,10] — represent the two extremes that are found in the literature when studying existing workflow specification proposal — from practical specification concerns to theoretical complexity issues.

The main goal of our research was to support reusability in workflow specification. Reusability is a major concern in software engineering, but has so far attracted little attention in terms of workflow definition. Among the exceptions can be found in Dumas and ter Hofstede,[9] who use patterns or in Ref. 19, that consider using versions to accomodate evolution. Neither proposal can be considered a language, and thus needs further refinement for construction and execution of workflows. Furthermore, they do not consider dynamic (temporary run-time) modifications or propose language- specific features.

Workflow reuse is a practical issue, notably in the context of evolution, maintenance and adaptability. To start with, organizations have to deal with many processes that are similar to each other, but with slight changes — e.g. in a government agency, various departments may deal with the same problem in distinct ways. The differences may be very small — such as a process having an optional review, or a distinct termination activity. In such situations, the workflows used by each department are almost identical. Reuse in this case consists in adopting one single specification, parts of which are redefined or overriden. This a typical situation in which we recommend applying our *Redefines* features. If a basic definition is reused and just small updates are performed, not only is time saved, but several specification errors may be avoided. In the same vein, many activities are repeated

in distinct processes, and again one single reusable specification saves time. Second, reuse is one of the basic requirements to attain maintainability and evolution which occur naturally due to process reengineering activities and the natural dynamics of the real world. For instance, enterprises have to change policies and procedures to adapt to global and local market constraints, political context and technological progress. Nevertheless, most changes are not radical, and many work methods can be adapted — and again reuse is of interest.

Our main goal was proposing a language to foster reuse, rather than to provide constructs that would add expressive power as compared to other proposals. By introducing reuse features, however, we also gained on expressiveness. First, we are WfMC-compliant. Though our language does not explicitly contain certain operators (e.g. OR), they can be implemented by adequate use of events and event-handling rules. Second, like several other proposals, we support different kinds of workflow — and block — states, in particular wait-for and cancel, which are needed in real life situations. Third, our language allows expressing all of the concepts specified by the workflow patterns in Ref. 9.

Our language provides some features which are not common in workflow specification proposals, notably *PARLOOP*, asynchronous events, event propagation, dynamic event management via event handling rules, and the different kinds of parameter passing among workflows and their components. These features allow, for instance, expressing the complex Multiple Instance patterns of Ref. 10 which include dynamic (run-time) definition of loops and parallel branches. This same set of properties can be defined using distinct specifications of *PARLOOP* with appropriate parameter and/or event definitions.

The unique language feature offered by us is the *Redefines* clause, which does not appear in other languages and which fosters dynamic modifications and reuse. It is true that a clause of the same name appears in the XML proposal of WfMC, XPDL.[31] The XPDL redefines, however, does not have the same scope. It only allows redefining the specification of header (documentation) items such as workflow author or version. Thus, even though XPDL contemplates parameter passing, there is no support to this kind of reusability.

## 6. Conclusions and Future Work

This paper presented a new formalism for specifying extensible workflows, based on combining paradigms of programming languages and database systems. This formalism is based on creating a library of workflow building blocks, which can be progressively combined and nested to construct complex workflows. Control flow along and across blocks is achieved by different kinds of events and exceptions, which are handled by active rules. Workflows are specified declaratively, using a simple high level language, which allows dynamically defining exception and event scopes. This enables flexible workflow specification and orthogonality between static and dynamic aspects.

The proposed framework allows reuse of previously stored workflow definitions. This is particularly appropriate for distributed cooperative work environments, which are becoming more widespread with the advent of the Web.

Future work includes two kinds of activity — extending the language to add new constructs, especially for different kinds of event handling and exception consideration, and implementation of the architecture. We are also working on using our language to specify workflows for several application domains, to check their expressive power and limitations.

As compared to other similar efforts, the main advantage is that, besides features offered by formalisms such as those discussed in Sec. 5, we support asynchronicity of events, event propagation and explicit declaration of events and rules, and their dynamic redefinition. This provides more flexibility as well as supports specification of more complex situations. Furthermore, we allow specification of variables and data which are part of a workflow state but are also used to help control the execution. At present, we are mapping our language to the the first draft of the XML Workflow Process Definition Language of WfMC (XPDL).[31] So far, most of our constructs permit a direct mapping, though some of these mappings are not immediate. The main differences concern the PARLOOP construct and redefinition facilities.

One limitation to our work is the fact that the language is not user-friendly. In fact, even workflows with few states may become ungainly and hard to understand and specify, as seen from the simple examples presented in the paper. Thus, one extension would be to provide a programming environment which would hide this complexity from the designer. Another limitation is that one cannot redefine just part of a block; redefinition demands rewriting the whole block, even if just some parts are changed. So far, we have not found a situation where this is critical, but further experimentation may be needed. Finally, if the designer wants to redefine a block which is deeply embedded in a block composition hierarchy, this may lead in some cases to the redefinition of all the calling blocks within this hierarchy. The solution in this case is to create new block specifications (in the paper called permanent modifications). This does not affect pre-existing workflows. This kind of approach is akin to creating a library of block *versions*, whose management is also to be considered.

## Acknowledgments

The work reported in this paper was partially financed by a binational cooperation program CNPq-Brazil and CNRS-France. Financing was also complemented by grants from CNPq and FAPESP, and by the PRONEX/MCT project on Advanced Information Systems (SAI).

## Appendix — Language Syntax

The language syntax follows closely a LISP style, with the exception of the definition and use of a block, which would correspond to a function definition and function call in LISP. The difference is that in our language there are two classes of parameters for blocks, exported events and state variables, and we decided to group them separately, for clarity's sake.

Below is the grammar for our workflow definition language. The symbol ( is part of the language, but not to overload the rules we wrote them as ( instead of ‘‘(’’.

*task* ::= (DEFTASK *task-name* interface path)

*block*::= *named-block* | *anonymous-block*

*named-block* ::= (DEFBLOCK *block-name block-definition*
*block-definition* ::= *interface*
[(REDEFINESINTERFACE *block-name interface*
[(REDEFINESTASK *task-name* interface tpath)]
[(REDEFINESBLOCK *block-name block-definition* )]
[(EXTERNAL-EVENTS {*event-name*}* ) ]
(BODY *body-form* )
[(EH-RULES {*eh-rule*}*) ] )

*interface* ::= ({*event-name*}* ) ( {*state-var-name1*}* )

*state-var-name1* ::= {+ | ∧} {*state-var-name*}

*body-form* ::= ({*task-name* | *block-name*} interface)
| *anonymous-block*
| (SEQ {*body-form*}+)
| (PAR {*body-form*}+)
| (LOOP *body-form*)
| (PARLOOP *number-of-instances body-form*)
| (RAISE *event-name*)
| (CALL {*application-function+args*})
| (EXIT)
| (STOP)

*state-var-name2* ::= [?] {*state-var-name*}

```
anonymous-block ::= (BLOCK block-definition)


eh-rule ::= (event-moment condition action)

event-moment ::= event-name | event-expression

event-expression ::= (EOB [block-name][+time-expression) |
(BOB [block-name][+time-expression)


condition ::= event-name
          |   (ACTIVE? block-name)
          |   (STOPPED? block-name)
          |   (TERMINATED? block-name)
          |   (AND {condition}+)
          |   (OR {condition}+)
          |   (NOT condition)

action ::= ({task-name | block-name} interface)
       |   (STOP [block-name] )
       |   (EXIT [block-name] )
       |   (RESUME block-name)
       |   (RAISE event-name)
       |   (SEQ {action}+)
       |   (CALL {application-function+args})
```

## References


1. A. Ailamaki, Y. Ioannidis and M. Livny, Scientific workflow management by database management, *Proc. 10th IEEE Int. Conf. Sci. Stat. Database Management* 190–201, 1998.
2. G. Alonso and H. Schek, Research issues in large workflow management systems, *Proc. NSF Workshop Workflow Process Automat. Inf. Syst.*, 1996.
3. A. Bonner, Workflow, transactions and datalog, *Proc 18th ACM SIGMOD-SIGACT-SIGART Symp. Prin. Database Syst.* (1999) 294–305.
4. F. Casati, S. Castano, M. Fugini, I. Mirbel and B. Pernici, Using patterns to design rules in workflows, *IEEE Trans. Software Eng.* **26**, 8 (2000) 760–785.
5. F. Casati and A. Discenza, Supporting workflow cooperation within and across organizations, *Proc. ACM Symp. Appl. Comput.* **1**, (2000) 196–202.
6. F. Casati, S. Ceri, B. Pernici and G. Pozzi, Deriving active rules for workflow enactment, *Proc. DEXA Conf.* (1996) 94–110.
7. S. Carlsen and R. Gjersvik, Organization metaphors as lenses for analyzing workflow technology, *Proc. ACM SIGGROUP Conf. Supporting Group Work: Integration Challenge* (1997) 261–270.

8. Q. Chen, M. Hsu, U. Dayal and M. Griss, Multi-agent cooperation, dynamic workflow and XML for e-commerce automation, *Proc. Int. Conf. Autonomous Agt.* (2000) 255–256.
9. M. Dumas and A. ter Hofstede, Workflow patterns, Technical Report WP 47, Technische Universiteit Eindhoven, 2000, tmitwww.tm.tue.nl/research/patterns/, accessed January 2002.
10. M. Dumas and A. ter Hofstede, UML activity diagrams as a workflow specification language, *Proc. Int. Conf. UML*, October 2001.
11. C. Dengi and S. Neftci, Dflow workflow management system, *Proc. 8th DEXA Conf.*, 1997.
12. E. Gokkoca, M. Altinel, I. Cingil, E. Tatbul, P. Koksal and A. Dogac, Design and implementation of a distributed workflow enactment service, *Proc. IEEE Int. Conf. Coop. Inf. Syst. COOPIS* (1997) 89–98.
13. P. Grifoni, D. Luzi, P. Merialdo and F. Ricci, ATREUS — A model for the conceptual representation of a workflow, *Proc. DEXA Conf.* (1997) 400–405.
14. A. Gal and D. Montesi, Inter-enterprise workflow management systems, *Proc. 10th Int. Conf. Workshop Database Expert Syst. Appl. (DEXA'99)* (1999) 623–627.
15. P. Grefen, J. Vonk and P. Apers, Global transaction support for workflow management systems: From formal specification to practical implementation, *VLDB J.* **10** (2001) 316–333.
16. R. Hull, F. Llirbat, E. Simon, J. Su, G. Dong, B. Kumar and G. Zhou, Declarative workflows that support easy modification and dynamic browsing, *Proc. Int. Joint Conf. Work Activities Coordination Collaboration* (1999) 69–78.
17. R. Hull, F. Llirbat, J. Su, G. Dong, B. Kumar and G. Zhou, Efficient support for decision flows in e-commerce applications, *Proc. Conf Telecomm. Electron. Comm. ICTEC*, 1999.
18. D. Hollingsworth, Workflow management coalition — The workflow reference model, Workflow Management Coalition, 1995, Doc Number WFMC-TC00-1003.
19. G. Joeris and O. Herzog, Managing evolving workflow specifications, *Proc. 3rd Int. Conf. Coop. Inf. Syst.*, 1998.
20. P. Karagoz, S. Arpinar, P. Koksal, N. Tatbul, E. Gokkoca and A. Dogac, Task handling in workflow management systems, *Proc. Int. Workshop Issues Appl. Database Technol.*, Berlin, 1998.
21. K. Kim and S. Paik, Practical experiences and requirements on workflows, eds. W. Conen and G. Neumann, *Coordination Technology for Collaborative Applications — Organizations, Processes and Agents*, Lecture Notes in Computer Science 1364 (Spring Verlag, 1998) 145–160.
22. Z. Luo, A. Sheth, J. Miller and K. Kochut, Defeasible workflow, its computation and exception handling, *ACM Workshop Towards Adaptive Workflow Syst.* 1998, Electronic Document, within CSCW'98 Conference.
23. N. Paxton and O. Diaz, Active database systems, *ACM Comput. Sur.* **31**, 1 (1999) 63–103.
24. C. Petrie and S. Sarin (eds.), Internet-based workflows, Special issue, *IEEE Internet Comput.*, May–June, 2000.
25. J. Rumbaugh, I. Jacobson and G. Booch, *The Unified Modeling Language Reference Manual* (Addison Wesley, 1999).
26. M. Rusinkiewicz and A. Sheth, Specification and execution of transactional workflows, ed. W. Kim, *Modern Database Systems. The Object Model, Interoperability and Beyond* (ACM Press, 1995) 592–620.
27. A. Sheth and K. Kochut, Workflow applications to research agenda: Scalable and dynamic work coordination and collaboration systems, *Proc. NATO Workshop Workflow Management Syst. Interoperability*, 1997.

28. K. Schmidt and C. Simone, Coordination mechanisms: Towards a conceptual foundation of CSCW systems design, *J. CSCW*, **5**, 2–3 (1996).
29. C. Simone and K. Schmidt, Taking the distributed nature of cooperative work seriously, *Proc. 6th Euromicro Workshop Par. Distr. Processing* (1998) 295–301.
30. WfMC, Workflow Management Coalition, http://www.wfmc.org/, as of January 2002.
31. WfMC, Workflow Management Coalition, workflow process definition interface process ML definition language, www.wfmc.org/standards/docs/xpdl_010522.pdf (as of January 2002), Draft Version dated May 2001.
32. Workflow Management Coalition, *Workflow Management Coalition Terminology and Glossary*, 1999 Doc Number WFMC-TC-1011.
33. M. Weske, G. Vossen, C. B. Medeiros and F. Pires, Workflow management in geoprocessing applications, *Proc. 6th ACM Int. Symp. Geog. Inf. Syst. — ACMGIS'98*, (1998) 88–93.
34. J. Wainer, M. Weske, G. Vossen and C. B. Medeiros, Scientific workflow systems, *Proc. NSF Workshop Workflow Process Automat. Inf. Syst.*, 1996.
35. D. Wodtke, J. Weissenfels, G. Weikum and A. Dittrich, The mentor project: Steps towards enterprise-wide workflow management, *Proc. IEEE Data Eng.* 1996.
36. D. Ziegler, D. Markopoulos, L. Steffens and T. Chou, Dynamic workflow changes: A metadata approach *Comput. Eng.* **35**, 1–2 (1998) 125–126.