WebWorkFlow: An Object-Oriented Workflow Modeling Language for Web Applications

Zef Hemel, Ruben Verhaaf, and Eelco Visser

Software Engineering Research Group, Delft University of Technology, The Netherlands
`Z.Hemel@tudelft.nl, rverhaaf@gmail.com, visser@acm.org`

Abstract. Workflow languages are designed for the high-level description of processes and are typically not suitable for the generation of complete applications. In this paper, we present WebWorkFlow, an object-oriented workflow modeling language for the high-level description of workflows in web applications. Workflow descriptions define procedures operating on domain objects. Procedures are composed using sequential and concurrent process combinators. WebWorkFlow is an embedded language, extending WebDSL, a domain-specific language for web application development, with workflow abstractions. The extension is implemented by means of model-to-model transformations. Rather than providing an exclusive workflow language, WebWorkFlow supports interaction with the underlying WebDSL language. WebWorkFlow supports most of the basic workflow control patterns.

1 Introduction

Workflow is concernced with the coordination of *activities* performed by *participants* involving *artifacts* [9,19]. Workflow and business process modeling languages such as UML activity diagrams [5], BPEL [4], and YAWL [17], are designed for the high-level description of a wide variety of workflows or *business processes* ranging from the documentation of the operating procedures for a factory, the administrative processes involving (paper) documents of a business, or the procedures carried out by medical staff with patients in a hospital. Thus, participants in a workflow may be people, machines, or machines operated by people, and artifacts may be electronic data or physical artifacts. A worklow description may be just the documentation of a procedure to be carried out by humans, or it may be the specification of an interactive automated process. If automated, a workflow may be coordinated by a central machine (e.g. a web server), or it may consist of a network of collaborating (web) services. To cover this wide range of applications, workflow languages are restricted to modeling processes and not complete applications. That is, using a workflow engine for the execution of a process definition requires external applications or code to implement individual activities.

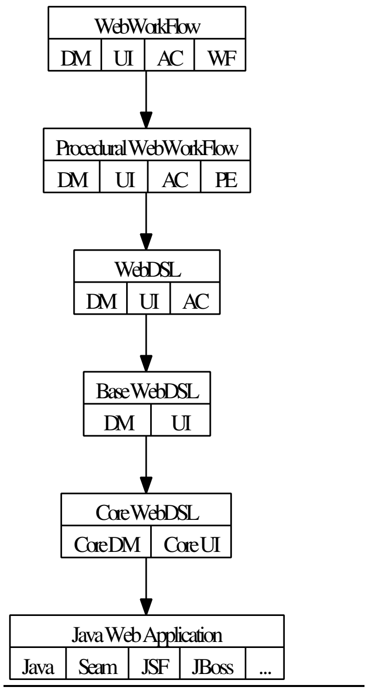
Web applications are concerned with presenting information to, and obtaining information from users interactively through a web browser. There are many types of web applications that contain workflow elements, i.e. the coordination of activities performed by participants. Consider for instance the following three examples. (1) An issue tracker coordinating the activities of the members of a project through registration, assignment and monitoring progress on issues. (2) A conference management system coordinating the activities of authors, program chairs, program committee members, external reviewers, meeting planners, and attendees to produce, review, select, and present a collection of scientific publications. (3) A user registration component, creating an account for a new user by subsequently registering, checking of credentials and confirming by email, involving a user and administrator. Thus, workflow concepts can be used as organizing principle for the engineering of many web applications, supporting the high-level implementation of the administration and monitoring of a process.

Rather than deriving an incomplete skeleton or boilerplate application from a process definition, a customized application with workflow requires *integration* of a workflow description language with a web engineering language.

In this paper, we present WebWorkFlow, an object-oriented workflow modeling language for the high-level description of workflows in web applications. WebWorkFlow is an *embedded language* [3] extending WebDSL [18], a domain-specific language for web application development, with workflow abstractions (Figure 1). From the definition of procedures operating on objects, and a control flow description to connect these procedures, complete custom web applications can be generated.

The WebWorkFlow generator is designed and built using a number of best practices for *domain-specific language engineering* [18]. Rather than providing an exclusive workflow language, WebWorkFlow supports interaction with the underlying WebDSL language. This approach enables the use of workflow abstractions where possible, and the use of the regular web modeling facilities where needed. This practice is called *language integration and separation of concerns [7]*.

The target language (WebDSL) is a subset of the source language. The high-level language is more expressive (more concise models), but may not have the same coverage. For example, process expressions support only



DM = data model, UI = user interface, AC = access control, PE = procedure events, WF = workflow

Fig. 1. WebWorkFlow is implemented as extension of WebDSL, which is itself implemented by means of model-to-model transformations to a core subset, which is transformed to code for the Java/Seam web platform

structured control-flow, while the underlying procedure event model supports unstructured control-flow. Thus, this approach makes it possible to use high-level abstractions where possible, but allows escaping to the next lower level where needed, thus increasing coverage of the language. This approach is called *compilation by normalization [10]* and is sketched in Figure 1. At the top level is WebWorkFlow, a rich DSL with sub-languages for data, user interface, access control and workflow modeling. WebWorkFlow is translated to lower-level procedural WebWorkFlow where workflow process descriptions have been translated to procedure events. Procedures are translated to a combination of data models, user interface elements and access control rules. This chain of transformations continues until all that is left is core WebDSL, a relatively low level model that can be easily mapped to the target platform, in this case Java/SEAM.

The rest of this paper is structured as follows. In the next section we introduce WebWorkFlow procedures by means of an example. In Section 3 we describe our implementation approach, explaining the procedure event model underlying the implementation of high-level process descriptions, the transformation of process expressions to procedure events, and the transformation of procedures to WebDSL, building on its high-level data model, user interface, and access control abstractions. In Section 4 we evaluate the coverage of WebWorkflow by examing the encoding of the control-flow patterns of Van der Aalst et al. [16]. In Section 5 we discuss the relation of WebWorkFlow to other process modeling approaches.

2 WebWorkFlow by Example

Workflows in WebWorkFlow are defined by means of *workflow procedures* that operate on *workflow objects*. In this section we introduce the high-level language constructs for defining objects and procedures, using as running example a simple workflow for organizing 'progress meetings' between managers and their employees. More precisely, rather than organizing the meeting itself, the workflow organizes the organization surrounding the meeting. Prior to the meeting, the manager and employee provide their own view on the progress of the employee. After the meeting the manager writes a `report` about the meeting. The employee may approve the report or may provide `comment`s on the report, which may cause the manager to revise the report. When the report is approved, the manager finalizes it. The complete WebWorkFlow implementation of this `ProgressMeeting` workflow is defined in Figures 2 and 3.

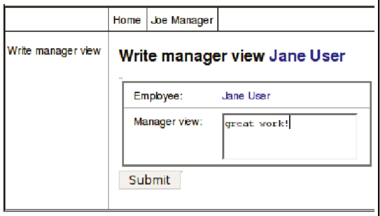
```
entity User {
  username  :: String
  password  :: Secret
  name      :: String
  manager   -> User
  employees -> Set<User>
}
entity ProgressMeeting {
  employee     -> User
  employeeView :: Text
  managerView  :: Text
  report       :: Text
  approved     :: Bool
  comment      :: Text
}
```

Fig. 2. WebDSL data model for progress meeting workflow

Workflow Objects. WebWorkFlow is an *object-oriented* workflow language. Central to the definition of a workflow is a workflow object that accumulates the data produced in the process and documents its progress. Typically, a workflow object is a domain object in the domain model of the application. For example, in a conference management system natural workflow objects are `Paper` and `Review`. If the only purpose of a workflow is to schedule a number of steps without a natural domain object, a special entity can be created to represent the instances of the workflow.

Workflow objects are instances of entities described using WebDSL data models [18]. Figure 2 describes the data model for the progress meeting workflow. A data model consists of *entity declarations* such as `User` and `ProgressMeeting`.

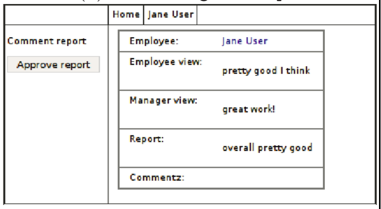
```
procedure meeting(p : ProgressMeeting) {
  process {
    (writeEmployeeView(p)
                    |AND| writeManagerView(p));
    repeat {
      writeReport(p);
      (approveReport(p) |XOR| commentReport(p))
    } until finalizeReport(p)
  }
}
procedure writeEmployeeView(p : ProgressMeeting) {
  who  { principal = p.employee }
  view {
    derive procedurePage from p
       for (view(employee), employeeView)
  }
}
procedure writeManagerView(p : ProgressMeeting) {
  who  { principal = p.employee.manager }
  view {
    derive procedurePage from p
       for (view(employee), managerView)
  }
}
procedure writeReport(p : ProgressMeeting) {
  who  { principal = p.employee.manager }
  view {
    derive procedurePage from p
       for (view(employee), view(employeeView),
            view(managerView), report)
  }
}
procedure approveReport(p : ProgressMeeting) {
  who  { principal = p.employee }
  do   { p.approved := true; }
}
procedure commentReport(p : ProgressMeeting) {
  who  { principal = p.employee }
  view {
    derive procedurePage from p
       for (view(employee), view(report),
            commments)
  }
  do   { email(commentNotification(p));
  }
}
procedure finalizeReport(p : ProgressMeeting) {
  who  { principal = p.employee.manager }
  when { p.report != "" && p.approved }
}
```



Write manager view Jane User

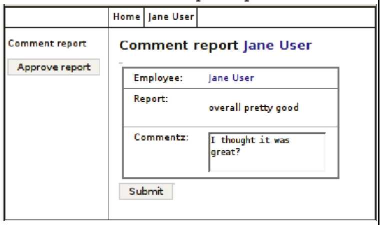
Employee:	Jane User
Manager view:	great work!

(a) `writeManagerView(p)`



Employee:	Jane User
Employee view:	pretty good I think
Manager view:	great work!
Report:	overall pretty good
Commentz:	

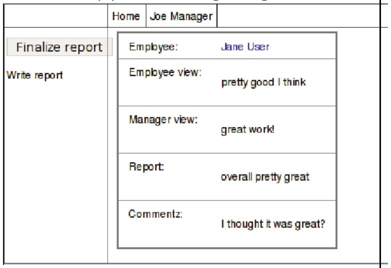
(b) `approveReport(p) |XOR| commentReport(p)`



Comment report Jane User

Employee:	Jane User
Report:	overall pretty good
Commentz:	I thought it was great?

(c) `commentReport(p)`



Employee:	Jane User
Employee view:	pretty good I think
Manager view:	great work!
Report:	overall pretty great
Commentz:	I thought it was great?

(d) `repeat{ writeReport(p) ... } until finalizeReport(p)`

Fig. 3. Progress meeting workflow procedure with screenshots

An entity declaration has *properties*, which associate data with entity instances. A property has a *name* and a *type*, which may be either a *value type* indicated by `::` (e.g. `String`, `Text`, `Secret`) or a reference type, indicated by `->`, referring to other entities or collections of entities (e.g. `Set<User>`).

Workflow Procedures. A workflow in WebWorkFlow is formalized by means of a *procedure*, which describes activities to be performed by one or more participants in a particular order. A procedure may consist of a single step, or may be a composition of procedures. A procedure may be automatic or may require a user to provide input, which may require a simple button click or filling in a complete form. Figure 3 defines the procedures for the progress meeting workflow. The `meeting` procedure defines the overall process of the workflow by composing the other procedures, which each define a single step. The screenshots on the right of Figure 3 are snapshots from a workflow conducted by 'Joe Manager' and 'Jane User'. The name in the menubar indicates the logged in user.

Figure 4 defines the syntax of high-level procedure definitions. Thus, a procedure definition has a *name* (`f`), exactly one typed parameter (`x:A`) indicating the workflow object to which the procedure applies, and a number of optional clauses, `who`, `when`, `view`, `do`, and `process`, which are discussed below. In the next section, the list of procedure clauses is extended to cater for the definition of *procedure events*.

```
procedure f(x : A) {
  who     { who }
  when    { when }
  view    { elem* }
  do      { stat* }
  process { pexp }
}
```

Fig. 4. Procedures

Actor. The `who` clause determines which participants can apply the procedure by means of an access control predicate, based on the declarative access control model of WebDSL [7]. The expression is a constraint on the current session and the workflow object, and any objects reachable from those via properties. The session includes a pointer to the *principal*, i.e. the logged in user associated with the session. For example, the `writeReport` procedure requires that the `principal` corresponds to `p.employee.manager`, that is, the manager of the employee for which the meeting is organized.

Activation. The `when` clause provides additional constraints on the applicability of a procedure. This is used for enforcing the ordering of procedures, as we will see in the next sections. However, in the high-level language, ordering of procedures is achieved by means of process expressions. Thus, further utility of the `when` clause is to test for preconditions on the workflow object. For example, the `finalizeReport` procedure tests that the `report` has actually been written by requiring that it is not the empty string, and that the report has been approved. Another application of the `when` clause is to test timing constraints, e.g. the deadline for a `submitPaper` procedure. A procedure is only applicable when the actor and activation constraints are satisfied. Thus, the page for applying a procedure and the links to that page are not accessible if these constraints are not satisfied.

User interface. The `view` clause defines the user interface for applying a procedure. This may be an arbitrary WebDSL page definition, allowing a completely customizable user interface [18]. The page definition can display any relevant information accessible through the workflow object and session, and will typically display a form for user input required by the procedure. It is often convenient and appropriate to derive a page definition from the data model. The WebDSL `derive` construct

```
derive style from e for (p1,...,pn)
```

provides a flexible mechanism for deriving a page from an entity declaration [8]. The `style` argument declares the style of the page, the expression `e` indicates the object and thus the type for which to generate the page, and the `pi` properties indicate which properties of the object should be displayed (`view`) and which should be edited. The `commentReport` procedure in Fig. 3 demonstrates how a `procedurePage` can be generated for two view properties (`employee` and `report`) and one edit property (`comments`).

Action. The `do` clause describes the action taken when the procedure is applied. Actions can be described using a simple imperative language. For a standard `procedurePage` the default action is to save the changes for the edit fields in the form, and no further action is needed. Additional actions may be taken to implement business logic, to send a notification email as in the `commentReport` procedure, or to create sub-workflows as described below.

Process. The `process` clause contains a *process expression* defining the composition of procedures to apply after invoking the containing procedure. For example, the `meeting` procedure in Figure 3 defines the composition of the individual steps in the `ProgressMeeting` workflow. Process expressions are composed from procedure invocations with several combinators. The sequential composition `e1 ; e2` of two expressions first applies `e1` and then `e2`. The parallel composition `e1 |AND| e2` applies `e1` and `e2` in parallel waiting for both to terminate. The iteration `repeat{e} until f(o)` applies `e` until procedure `f` is applied. The parallel composition `e1 |XOR| e2` enables the application of `e1` and `e2`, but cancels one if the other has terminated.

```
extend entity ProgressMeeting {
  meetings -> Set<ProgressMeeting>
}
procedure meeting(p : ProgressMeeting) {
  process {
    employeeMeetings(p);
    (employeeView(p) |AND| managerView(p));
    ... as before ...
  }
}
procedure employeeMeetings(p : ProgressMeeting) {
  who { principal = p.employee }
  do {
    for(u : User in p.employee.employeesList) {
      p.meetings
        .add(ProgressMeeting{employee := u})
    }
  }
  process {
    AND(q : ProgressMeeting in p.meetingsList) {
      meeting(q)
    }
  }
}
```

Fig. 5. Recursive workflow procedure

Recursive Procedures. The meeting example defines a workflow on a single workflow object `ProgressMeeting`. However, there can be multiple instances of this workflow in different stages of execution in parallel. For each user there can be a `ProgressMeeting` instance, or even several, say if an employee has more than one manager, or one for each year of employment. Thus, a workflow instance corresponds to an instance of the corresponding workflow object. Procedures can instantiate new sub-workflows by invoking procedures on linked objects (through properties). Such sub-workflows can also be *recursive* in the sense that a procedure may call itself on another workflow object. The example in Figure 5 illustrates recursion by extending the progress meeting example. In this workflow users do not only have managers, they can also *be* managers. Before evaluating the progress of a manager, all his or her own employees should be evaluated first. Thus, the `meeting` workflow procedure is adapted to invoke `employeeMeetings(p)`. In the `do` clause, a `ProgressMeeting` object is created and added to the set of `meetings` for each employee of the user. Then, in the `process` clause, the `meeting` workflow is initiated for each employee meeting in parallel so that all employees can start working on their `employeeView` simultaneously. The `employeeMeetings` procedure needs to be finished (all reports approved and finalized) before the managers meeting can proceed.

3 Transforming Procedures

In this section we explain how we implement the compilation by normalization approach to realize WebWorkFlow. We describe how WebWorkFlow procedures are implemented by means of model-to-model transformations to the underlying WebDSL language. The conceptual design of WebWorkFlow may suggest that it is an object-oriented language that can be directly translated to a regular object-oriented language such as Java. However, due to the statelessness of the HTTP protocol, state has to be kept in between requests. Traditionally, sessions have been used for this purpose, however sessions typically only last a few hours whereas a workflow can last years. Furthermore sessions are bound to one particular user, whereas many users can participate in a workflow. So rather than using sessions, the workflow state is persisted in the database through extension of the application's data model. Furthermore, page definitions are used to implement the user interface for applying a procedure, and access control rules to regulate the applicability of a procedure. The transformations rely on the data, user interface and access control modeling languages of WebDSL.

Procedures to Pages. The basic idea for the implementation of a procedure is illustrated in Figure 6 with the transformation of `procedure` *`f`*`(`*`a`*`:`*`A`*`){...}`. To record the state of a procedure, the workflow entity *`A`* is extended with a property with the name of the procedure referring to a `ProcedureStatus` object. The basic implementation of `ProcedureStatus` provides an `enabled` property, which indicates whether the procedure may be applied, and an `enable()` function, which can be used to set this property to true. The user interface for the procedure is realized by means of a page definition with the name of the procedure and the workflow object as argument. The *`view`* from the procedure is used as specification of the presentation and the `do` action is performed on submit and disables the procedure by resetting the `enabled` flag. Finally, an access control rule uses the *`who`* and *`when`* expressions to regulate access to the procedure. The `enabled` property of the status object is used as an additional requirement for applicability of the procedure.

Task Lists and Navigation. In addition to the basic page for applying a procedure, further elements for the user interface of an application can be derived from procedure definitions. In particular, a definition of a task list with links to pages for applicable procedures for a particular workflow object, or a list with all available procedures for a particular user. The access control mechanism of WebDSL ensures that links in such work lists are only displayed if the pages they point to are accessible. Thus, the user interface is dynamically adapted to the state of the application. This is illustrated in Figure 3, where links to applicable procedures can be seen in the sidebar.

```
procedure f(a : A) {
  who  { who }
  when { when }
  view { elem* }
  do   { stat* }
}
```

⇓

```
extend entity A {
  f -> ProcedureStatus
}
entity ProcedureStatus {
  enabled :: Bool
  function enable() {
    enabled := true;
  }
}
define page f(a : A) {
  elem*
  action do() {
    a.f.enabled := false;
    stat*
  }
}
access control rules {
  rule page f(a : A) {
    who && when && a.f.enabled
  }
}
```

Fig. 6. Transformation of procedure into access control rule, page definition, and entity declaration

Procedure Event Model. A procedure has a life cycle that starts with the creation of the workflow object it is associated with and ends with its destruction. WebWorkFlow provides an event model for observing the changes in the life cycle of a procedure. Observation of events is realized using the following event handling clauses in a procedure definition (Figure 7):

– `enabled` is triggered after a call of `enable()`
– `disabled` is triggered after a call of `disable()`
– `done` is triggered after the execution of the `do` clause
– `processed` is triggered after the procedure's process has terminated; in case a procedure has no `process` clause the `processed` event follows directly after the `done` event.

```
procedure f(a : A) {
  enabled   { stat* }
  who       { who }
  when      { when }
  view      { elem* }
  do        { stat* }
  done      { stat* }
  process   { pexp }
  processed { stat* }
  disabled  { stat* }
}
```

Fig. 7. Procedure events

To support modular subscription to the events of a procedure, the `extend procedure` mechanism can be used to add additional statements to an event handler. For example, the definition

```
extend procedure f(a : A) { processed { stat* } }
```

extends the `processed` event handler with the *`stat*`* statements.

Encoding Procedure Dependencies. The procedure event model provides a general mechanism for encoding a wide variety of policies for ordering procedures. For example, the definition

```
extend procedure writeReport(p : ProgressMeeting){
  processed { p.approveReport.enable(); p.commentReport.enable(); } }
```

enables the procedures `approveReport` and `commentReport` after the `write Report` procedure has terminated, which corresponds to a parallel split.

Process Expressions. While the procedure event handlers provide a flexible mechanism for composing procedures, it is also a rather tedious mechanism. A large number of procedure composition patterns can be captured using concise process expressions from which the correct event handling code can be generated automatically. For example, the sequential composition of two procedures can be encoded as in the `writeReport` definition above. That is, the sequential composition `f(x); g(x)` is encoded by calling `x.g.enable()` in the `processed` clause of `f`. This direct enabling of the successor of a procedure works fine provided that procedures are only called from one call site. Since workflow procedures are intended for human consumption, it is generally not a good idea to require the same activity in many different contexts. However, this constraint is typically violated in the case of recursion, which requires an initial call and the actual recursive call.

```
entity ProcedureStatus {
  caller      -> ProcedureStatus
  returnstate :: Int
  function enable(c : ProcedureStatus,
                  r : Int) {
    this.enabled := true;
    this.caller := c;
    this.returnstate := r;
    this.enabled(); }
  function disable() {
    this.enabled := false;
    this.disabled(); }
}
```

Fig. 8. Re-definition of `ProcedureStatus` for recording return address of procedure call

```
procedure f(a:A){ process{ g(a); h(a) } }
```

$$\Downarrow$$

```
entity FStatus : ProcedureStatus {
  a -> A
  function next(state : Int) {
    if(state = 0) { a.g.enable(this, 1); }
    if(state = 1) { a.h.enable(this, 2); }
    if(state = 2) { this.processed(); }
  }
}
extend procedure f(a : A) {
  enabled { this.next(0); }
  processed {
    this.caller.next(this.returnstate);
  }
}
```

Fig. 9. Sequential composition with state machine

Dealing with multiple call sites requires a more dynamic approach to sequencing of calls. In order to return control to the proper callee, it is necessary to record the 'return address'. The return address of a workflow procedure call can be represented by the identity of the caller, i.e. its `ProcedureStatus` object, and its state. Figure 8 redefines `ProcedureStatus` with an `enable` function taking the caller identity and its state as arguments.

To determine the next step to take after a procedure returns, a process expression is transformed to a finite state machine encoded by the `next` function of the `FStatus` entity, which specializes `ProcedureStatus` for a procedure `f`. To compute the state machine, all positions in the process expressions are assigned a unique number. For each combinator there are special rules for computing the transitions. We illustrate the computation with two examples, sequential composition (Figure 9) and parallel split (Figure 10). Procedure `f` in Figure 9 applies the sequential composition of `g` and `h`. It is transformed to the `next` function in the `FStatus` entity declaration and an extension of the procedure event handlers. When `f` is enabled, the transition from the start state (0) is taken, which will lead to `g` being enabled with state 1. When the `next` transition is taken from `g`, `h` is enabled with state 2. On return from `h` the process is completed and the `processed` event handler is called, which itself returns to the caller of `f`.

Figure 10 defines the implementation of the `e1 |AND| e2` combinator, which applies processess `e1` and `e2` in parallel and waits for both to complete before proceeding. The transformation assumes a normalized process expression in which the expression is preceded and succeeded by a simple procedure call. This assumption is also made for the branches of the split. Expressions that do not match this pattern (e.g. `(e1 |AND| e2); (e3 |AND| e4)` can be transformed to this form by insertion of automatic identity procedures. Note that the `[i]` expressions are state labels. When the split is reached (state 1), the heads of the two branches are enabled. When the first of the branches returns, the counter `count` is incremented. When the next branch returns the counter is 1 and the continuation `q` is enabled.

```
procedure f(a : A) {
  process {
    p(a); [1]
    ((g1(a); e1*; h1(a) [3])
     |AND| (g2(a); e2*; h2(a) [5]));
    q(a) } }
```

⇓

```
extend entity A { count :: Int }
extend entity FStatus {
  extend function next(state : Int) {
    if(state = 1) {
      a.g1.enable(this, 0);
      a.g2.enable(this, 0);
    }
    if(state = 3 || state = 5) {
      if(a.count = 1) {
        a.count := 0;
        a.q.enable(this, 0);
      } else {
        a.count := a.count + 1;
      } } } }
```

Fig. 10. Implementation of the split and join transitions of the `|AND|` parallel combinator

4 Encoding Patterns

A lot of research has been conducted on the assessment of workflow languages. Van der Aalst et al. [16] describe an extensive set of workflow patterns from the process perspective ranging from simple patterns such as sequential execution to complicated patterns such as loops and cancellation patterns. Recently, a revised version of these patterns was published [13]. Patterns have also been devised for the resource perspective [15] and the data perspective [14]. In this paper, we focus on the use of control patterns for evaluating workflow languages from the control perspective. Fig. 11 shows the coverage of these patterns for WebWorkFlow. In this table `+` means this pattern directly supported by WebWorkFlow, `+/-` means it is possible to implement through a workaround, and `-` means the pattern is not supported by WebWorkFlow. A number of patterns that are particularly noteworthy for WebWorkFlow are patterns 10, 12-15 and 22.

Although *arbitrary cycles (10)* are not often needed, they can be implemented using the lower-level procedure events `processed` or `done`, in which an arbitrary procedure can be enabled. This is an example of why it useful to have a high-level process description language, while at the same time still having access to a lower level of abstraction where not directly supported process features can be implemented.

Multiple instances (12-15) of procedures in WebWorkFlow are supported through multiple instances of the objects they operate on. One instance of a process can run on each object that the process is defined on. By creating new object instances, any number of instances of a process can be created. Multiple instances without synchronization (pattern 12) can be implemented by simply calling `enable()` on them, which instantiates them in a non-blocking manner. Synchronizing on a number of instances (pattern 13 and 14) can be realized by instantiating them using the `AND(a : A in o.aList){proc}` construct. Pattern 15 (Multiple instances without a priori runtime knowledge) can be implemented as follows:

Pattern	Support
1. Sequence	+
2. Parallel split	+
3. Synchronization	+
4. Exclusive Choice	+
5. Simple merge	+
6. Multi choice	+
7. Synchronizing merge	+
8. Multi-merge	+/-
9. Discriminator	+
10. Arbitrary cycles	+/-
11. Implicit termination	-
12-15. Multiple instance patterns	+
16. Deferred choice	+
17. Interleaved parallel routing	+/-
18. Milestone	+/-
19. Cancel activity	+
20. Cancel case	+/-
21. Structured Loop	+
22. Recursion	+
23. Transient Trigger	+
24. Persistent Trigger	+

Fig. 11. Control flow pattern coverage of WebWorkFlow

```
multiproc(o).enable(); stopmultiproc(o)
```

In the `done` clause of `multiproc`, the procedure instantly re-enables itself using `enable()`, creates a new object, and starts a process with `enable()`. Finally, `stopmultiproc` disables `multiproc` through a call to `multiproc.disable()`. To synchronize all processes created, a `when` clause can be added to the `stopmultiproc` procedure, to require for all instances that their processes are finished.

For *recursion (22)* we can distinguish three cases. First, tail recursion can be implemented using `repeat` and `while`. Second, recursion on properties of the object is supported by simply calling a procedure recursively on the property (e.g. Figure 5). Recursive self calls on an object `a` from a process defined on `a` are not supported. We have not yet found a use-case for non tail-recursive self recursion, so it does not seem much of an issue. (But examples are most welcome!)

Data Patterns. Rusell et al. in [14] discuss a number of workflow data patterns. WebWorkFlow defines its processes directly on top of data entities. This gives the procedures in the process direct access to the data entity and all the data that is linked from it. Through this mechanism many of the data patterns are naturally supported. WebWorkFlow is mostly lacking in data hiding, for which it provides no explicit support. Lack of space prevents us from providing a thorough evaluation of the data patterns that are supported by WebWorkFlow.

5 Discussion

During the design of WebWorkFlow a number of design decisions were made that distinguish it from other workflow systems. We compare our approach to other workflow approaches and discuss opportunities for future work.

Evaluation. Most workflow systems (YAWL, JBPM, BPEL) dynamically load a workflow description and *interpret* it. The advantage of this approach is the ability to adapt the workflow while the system is running. The workflow description can then simply be reloaded. This ability is often limited, however, to process descriptions, which means that no new tasks or procedures can be defined at runtime. In WebWorkFlow on the other hand, workflows are *compiled* to WebDSL (which in in its turn compiles to Java/JSF). As a consequence, run-time adaptations of the workflow process are not supported. However, the compiled application is much more light-weight than an interpreted worfklow management system.

The main concern of most workflow management systems is *controlling* the process. User interfaces and the rest of the application are handled in separate systems and are thus outside the scope of the workflow system. WebWorkFlow is an embedded DSL in WebDSL and therefore integrates well with the rest of WebDSL. This integration enables the developer to more easily develop and *generate* complete user interfaces, automated tasks and control flow. At the same time, it gives a maximum of flexibility as one can always resort to a lower level of abstraction in case the ultimate abstraction layer does not support a certain construction.

Following the tendency to design languages that are understandable by both business analysts and technical developers, and the shift from workflow to business process modeling, most workflow approaches use a *graphical language* for specifying processes. WebWorkFlow is specifically aimed at web developers and uses a *textual language*. The first reason to use a textual language is because it is an embedded DSL within WebDSL, which is a textual language. The second reason is that it turns out to be a very efficient and expressive way of expressing workflows.

Related Work. The approach that is taken by Brambilla et al. in [1], is that of adding workflow support to any domain model by performing a model transformation, by which the original model is extended with the necessary domain model elements to support workflow. WebWorkFlow applies these ideas to WebDSL. WebWorkFlow is more expressive than the approach discussed by Brambilla, because it does not cover nested sub-processes, which precludes recursion. Brambilla et al. themselves applied the ideas from [1] to WebML [2], where they describe how processes described in BPMN can be used to enact workflow processes in WebML applications. The authors envision that their explicit design styles in the future could be automated to generate skeleton applications based on process descriptions. WebWorkFlow realizes this vision; it does not only generate a skeleton application, but a complete application that can be *customized at the model level* to better suit application-specific needs.

YAWL is a graphical worklow language and system designed by Van der Aalst and Ter Hofstede, authors of the work on workflow patterns [13,15,14]. YAWL is designed to support almost every workflow pattern. Its formalization is based on high-level Petri nets [17]. Although YAWL is a graphical language, a lot of configuration needs to be done by setting parameters in property boxes. The process diagram does not always shows a complete picture because so much information is hidden inside properties. While YAWL is very expressive, it does not have the layered implementation that WebWorkFlow has, which allows to use lower level constructs to implement workflows not directly supported by the workflow language.

Panta Rhei [6] is a web-based workflow system that interprets workflows specified in a textual language. Workflow data in Panta Rhei is persisted in a database, which, like in WebWorkFlow, makes it possible to alter the execution of workflows at run-time by changing this data. A web browser is used as interface to the user. Communication with other systems is possible using an internal form representation, which remote systems must be able to interpret. Panta Rhei also has support for timing and transactional features, both of which are future work for WebWorkFlow.

WebWork is a web-only implementation of the workflow management system Meteor-2 [11]. It uses a graphical designer to specify a workflow, from which HTML and CGI scripts are generated. Automated tasks are performed using a socket connection with a web server and invoking CGI programs. WebWork is not as expressive as BPEL or BPMN, as only somewhat more than half of the first 20 workflow pattern are supported [16]. Also, the use of CGI adds flexibility, but a custom task is not easily constructed, as opposed to WebWorkFlow, where all code can be specified in one language.

BPMN (Business Process Modeling Notation) is a business process modeling notation language designed by the Business Process Modeling Initiative [20] and now maintained by the Object Management Group [12]. Its goal is to provide a notation for describing business processes understandable for both business analysts and technical developers. WebWorkFlow on the other hand is much more technical and mainly aimed at developers. In contrast to WebWorkFlow, BPMN is not directly executable. It is possible however, to derive executable workflow specifications from workflows specified using a subset of BPMN.

Future Work. An interesting area of research for WebWorkFlow is making procedures accessible through a web service interface (as opposed to the current HTML interface) and accessing procedures of remote servers. WebWorkFlow could then be used for web service orchestration, similar to BPEL.

Because of the user-initiated nature of WebDSL (and WebWorkFlow by extension), using timed events is not yet possible. Conceivable applications of timing features are scheduling tasks and using deadlines to automatically influence the execution of procedures.

As WebWorkFlow is used to generate a fully functional workflow system instead of specifying a workflow that is interpreted by a WFMS, regenerating and deploying the workflow system could potentially break ongoing workflows. This is a data model evolution problem and is future work. The automatically generated navigation based on procedure definitions is useful, but can still be enhanced. Process descriptions can be more fully utilized to generate navigation. Although it is possible to disable (parts of) procedures in the current version of the language, it is not straightforward to roll back the state of an application when errors occur. A transaction system similar to Panta Rhei might be helpful.

6 Conclusion

In this paper we introduced WebWorkFlow, an embedded DSL extending WebDSL with object-oriented workflow abstractions. Based on the definition of workflow procedures a full fledged executable application can be generated, including navigation and work lists. Following the 'compilation by normalization' [10] approach, WebWorkFlow achieves great flexibility and customizability by making the lower abstraction levels of the procedure event model and WebDSL web application modeling accessible next to the high-level workflow abstractions. WebWorkFlow covers most of the workflow control patterns. The patterns that are not directly expressible through the process expression language can often be implemented on the procedure event level, demonstrating the advantages of building a (workflow) language as an abstraction on a lower-level language.

Acknowledgments. This research was supported by NWO/JACQUARD project 638.001.610, *MoDSE: Model-Driven Software Evolution*.

References

1. Brambilla, M., Cabot, J., Comai, S.: Automatic generation of workflow-extended domain models. In: Engels, G., Opdyke, B., Schmidt, D.C., Weil, F. (eds.) MODELS 2007. LNCS, vol. 4735, pp. 375–389. Springer, Heidelberg (2007)
2. Brambilla, M., Ceri, S., Fraternali, P., Manolescu, I.: Process modeling in web applications. ACM Trans. Softw. Eng. Methodol. 15(4), 360–409 (2006)
3. Bravenboer, M., Visser, E.: Concrete syntax for objects. Domain-specific language embedding and assimilation without restrictions. In: Schmidt, D.C. (ed.) Proceedings of the 19th ACM SIGPLAN Conference on Object-Oriented Programing, Systems, Languages, and Applications (OOPSLA 2004), Vancouver, Canada, October 2004, pp. 365–383. ACM Press, New York (2004)
4. Curbera, F., Goland, Y., Klein, J., Leymann, F.: Business process execution language for web services, version 1.1. Technical report, IBM (2003)

5. Dumas, M., ter Hofstede, A.H.M.: Uml activity diagrams as a workflow specification language. In: Gogolla, M., Kobryn, C. (eds.) UML 2001. LNCS, vol. 2185, pp. 76–90. Springer, Heidelberg (2001)
6. Eder, J., Groiss, H., Liebhart, W.: The Workflow Management System Panta Rhei. In: Advances in Workflow Management Systems and Interoperability, Istanbul, Turkey, August, pp. 129–144. Springer, Heidelberg (1997)
7. Groenewegen, D., Visser, E.: Declarative access control for WebDSL: Combining language integration and separation of concerns. In: Schwabe, D., Curbera, F. (eds.) International Conference on Web Engineering (ICWE 2008), July 2008. IEEE CS Press, Los Alamitos (2008)
8. Hemel, Z., Kats, L.C.L., Visser, E.: Code generation by model transformation. In: Gray, J., Pierantonio, A., Vallecillo, A. (eds.) ICMT 2008. LNCS, vol. 5063, pp. 183–198. Springer, Heidelberg (2008)
9. Hollingsworth, D.: The Workflow Reference Model. Workflow Management Coalition, Document Number TC00-1003 - Issue 1.1 edn. (1995)
10. Kats, L.C.L., Bravenboer, M., Visser, E.: Mixing source and bytecode. A case for compilation by normalization. In: Kiczales, G. (ed.) Proceedings of the 23rd ACM SIGPLAN Conference on Object-Oriented Programing, Systems, Languages, and Applications (OOPSLA 2008), Nashville, Tenessee, USA, October 2008. ACM Press, New York (2008)
11. Miller, J.A., Palaniswami, D., Sheth, A.P., Kochut, K.J., Singh, H.: Webwork: Meteor$_2$'s web-based workflow management system. J. Intell. Inf. Syst. 10(2), 185–215 (1998)
12. Recker, J., Strategy, M.: Process Modeling in the 21 stCentury. In: BPTrends, pp. 1–8 (May 2006)
13. Russell, N., ter Hofstede, A.H.M., van der Aalst, W.M.P., Mulyar, N.: Workflow control-flow patterns: A revised view. Technical report, BPMcenter.org. (2006)
14. Russell, N., ter Hofstede, A., Edmond, D., van der Aalst, W.: Workflow Data Patterns. In: Delcambre, L.M.L., Kop, C., Mayr, H.C., Mylopoulos, J., Pastor, Ó. (eds.) ER 2005. LNCS, vol. 3716, pp. 353–368. Springer, Heidelberg (2005)
15. ter Hofstede, A., Edmond, D., van der Aalst, W.: Workflow resource patterns. BETA Working Paper Series, pp. 216–232 (January 2004)
16. van der Aalst, W.M.P., Hofstede, A.H.M.T., Kiepuszewski, B., Barros, A.P.: Workflow patterns. Distrib. Parallel Databases 14(1), 5–51 (2003)
17. van der Aalst, W.M.P., ter Hofstede, A.H.M.: YAWL: yet another workflow language. Information Systems 30(4), 245–275 (2005)
18. Visser, E.: WebDSL: A case study in domain-specific language engineering. In: Lammel, R., Saraiva, J., Visser, J. (eds.) GTTSE 2007. LNCS, vol. 5235. Springer, Heidelberg (2008)
19. WfMC. Terminology and glossary, 3rd edn. Document Number WFMC-TC-1011, Workflow Management Coalition (1999)
20. White, S.: Introduction to BPMN. In: IBM Cooperation (2004)