ULTRAflow – A Lightweight Workflow Management System

Alfred Fent and Burkhard Freitag

University of Passau, Department of Computer Science, 94030 Passau, Germany
Tel. (+49) 851 509 3131 Fax (+49) 851 509 3182
{fent,freitag}@fmi.uni-passau.de

1 Introduction

Workflow management systems usually use a graphical notation to define workflows, often based on Petri nets or specialized workflow graphs. [3, 4] lists the most important patterns for the specification of workflows and evaluates various commercial workflow management systems w.r.t. their support of these patterns. It turns out that most systems have to struggle with some fundamental problems: The exact semantics of graphical constructs varies from system to system, thus complicating the exchange of workflow descriptions even on the very abstract specification level; specification of workflows often resembles assembler programming more than high-level specifications, e.g. for iterations; and, most surprisingly for people that are used to rule-based specifications, all those systems have severe problems with subworkflows, i.e. workflows that are defined once and reused as a part of several other activities.

[1] already proposed to use a rule based formalism for workflow specification. In this paper, we describe how the building blocks of workflows can be expressed in the rule based update language ULTRA [2, 5]. It turns out that these building blocks have counterparts in the rule language that are easy and natural to specify. The ideas presented here are implemented in the prototype system ULTRAflow.

The work described in this paper has been funded by the German Research Agency (DFG) under contract number Fr 1021/3-2.

2 The ULTRA Language and Semantics

ULTRA [2, 5] is a framework for rule-based update languages, i.e. languages that can not only retrieve data based on a given rule set, but can as well be used to manipulate the underlying data. The framework semantics is defined for a transition system that can be instantiated to accomodate a given application domain. All the usual semantical properties of logic programming languages also hold for ULTRA, especially the existence of a unique minimal model and a fixpoint characterization of this minimal model.

As opposed to the classical Datalog and Prolog languages, ULTRA features a concurrent conjunction "|", which specifies the *parallel* execution of the goals to the left and to the right, as well as a connective ":" to specify *sequential* execution: The goals on the right hand side of the ":" are executed in the state that results from the execution of the goals on the left of ":".

ULTRA distinguishes between the usual *EDB* and *IDB predicates* on the one hand, and so-called *update predicates*, which contain actions changing the current state, on the other hand. The semantics of the former is set-oriented, i.e. all valid solutions are returned. For update predicates, a *deferred* semantics is used, where all possible action sequences are derived, but only *one* of them is "materialized", i.e. actually executed.

The language also features a bulk quantifier "$\#X\,[p(\boldsymbol{X}) \rightarrow u(\boldsymbol{Y})]$" which can be read as "for all X such that $p(\boldsymbol{X})$ holds, do $u(\boldsymbol{Y})$". This quantifier in some sense bridges the gap between the *all solutions* semantics for EDB and IDB, and the *one solution* semantics for update predicates. See [2, 5] for a complete definition of the ULTRA syntax, the formal definition of its semantics, and an extensive comparison to related work. Note that "," is used instead of "|" for the concurrent conjunction in these papers.

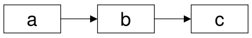
3 Workflow Patterns as ULTRA Rules

In this section we introduce how the basic workflow patterns can be expressed as ULTRA rules. The extension of the given correspondences to more complex situations, e.g. involving more than two parallel actions, is straightforward. One central feature that most commercial workflow management systems are missing – the modular composition and reuse of existing workflows – is inherited from the rule language: Workflow reuse is just a call of a predicate that constitutes a (sub)workflow of its own.

In the rules below, we denote by $\boldsymbol{X}$ an arbitrary list of variables; we require that the variables used as input arguments for update predicates are bound, e.g. by using an adequate sideways information passing strategy; the same holds for negated subgoals, i.e. we assume that no floundering occurs. In the workflow context, it can be assumed that these restrictions are satisfied. Note that besides the logical variables there is an external state, i.e. files, databases, forms, etc. that are manipulated by the workflows.

Sequence: This is the most basic workflow pattern. It is required when there is a dependency between two or more tasks so that one task cannot be started before another task is finished. In the diagram, the three tasks a, b and c have to be executed one after the other. This pattern can be implemented easily with the sequential composition of ULTRA.

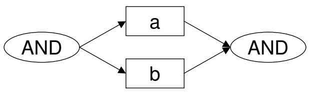
Graphical notation: ULTRA formulation:



$$sequence(\boldsymbol{X}) \leftarrow a(\boldsymbol{X}_a) : b(\boldsymbol{X}_b) : c(\boldsymbol{X}_c)\ .$$

AND-Split and AND-Join: Also known as "parallel split" and "synchronization", this pattern specifies that several tasks, a and b in the diagram, are executed in parallel. The workflow can only continue if all of the parallel tasks have finished. Parallel execution and synchronization is handled by the ULTRA concurrent composition:

Graphical notation: ULTRA formulation:

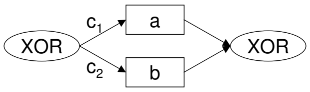


$$parallel(\boldsymbol{X}) \leftarrow a(\boldsymbol{X}_a) \mid b(\boldsymbol{X}_b).$$

XOR-Split and XOR-Join: This pattern implements an exclusive choice (in contrast to the multiple choice below) and a simple merge where it is guaranteed that only one predecessor of the join node will be active. Consequently no synchronization is needed. The conditions c_1 and c_2 in the diagram below denote the conditions that must hold for the corresponding branches to be activated, so they act as a kind of guard.
The specification with ULTRA rules can make use of the logical disjunction. No matter whether the guards c_i are mutually exclusive or not, only *one* successful branch is mate-

rialized, i.e. executed, even if more than one such branch exists: As described in Sect. 2, the semantics ensures that only one of several possible action sequences is executed.

Graphical notation: ULTRA formulation:

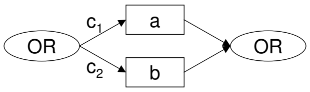


$exclusive\_choice(\boldsymbol{X}) \leftarrow c_1(\boldsymbol{X}_1) : a(\boldsymbol{X}_a).$
$exclusive\_choice(\boldsymbol{X}) \leftarrow c_2(\boldsymbol{X}_2) : b(\boldsymbol{X}_b).$

The rule based formalism allows the guards to be arbitrary logical predicates, i.e. complex deductions can be used, not just simple arithmetic or string comparisons as in many commercial systems. Note, however, that if the guards are not only used to "control access" to their branches, but also provide bindings to variables for the following actions, this may cause an implicit XOR-Split: If e.g. in a rule body *c(X) : action(X)* the variable X is bound to several values by predicate c, then the semantics chooses exactly one binding for *action*, so there is an implicit XOR over the various values for X.

OR-Split and OR-Join: An OR-Split implements a multiple choice, an OR-Join a synchronizing merge. One or several of the alternatives after the split may be activated in parallel during the execution; the join waits for the corresponding number of tasks to finish. Again, the conditions c_i act as guards for their branches.

Graphical notation: ULTRA formulation:



$$
\begin{array}{ll}
multiple\_choice(\boldsymbol{X}) & \leftarrow a'(\boldsymbol{X}_a) \mid b'(\boldsymbol{X}_b). \\
a'(\boldsymbol{X}) & \leftarrow c_1(\boldsymbol{X}_1) : a(\boldsymbol{X}_a). \\
a'(\boldsymbol{X}) & \leftarrow \neg c_1(\boldsymbol{X}_1). \\
b'(\boldsymbol{X}) & \leftarrow c_2(\boldsymbol{X}_2) : b(\boldsymbol{X}_b). \\
b'(\boldsymbol{X}) & \leftarrow \neg c_2(\boldsymbol{X}_2).
\end{array}
$$

To guarantee the logical success of the concurrent rule *multiple_choice*, both of the alternatives *a'* and *b'* must succeed, even if the guard is false. Therefore we also provide the rules with negated guards but without action. This implies that, if none of the guards is true, the multiple choice succeeds without performing any operation. Though the definition of the OR-split pattern [3, 4] is not specific about this case, we think that this behavior is acceptable in the workflow context. Moreover, careful design of the guards can ensure that always at least one alternative is executed.

Note that the situation of the AND-Split and -Join from above is just a special case of an OR-Split/Join with all the guards evaluating to *true*. Similarly, a XOR-Split/Join may arise as a special case of the OR-Split when the guards are mutually exclusive.

Multiple instances: Instantiating a given workflow several times poses a challenge to most commercial workflow management systems. If the number of instances is known in advance, the instances of the (sub)workflow can be generated by an AND-Split/Join with the known number of branches. Yet, if the number of instances is known only at runtime, awkward constructions are often necessary, and some systems even require all the instances to be executed sequentially. The rule-based language of ULTRAflow does not pose such restrictions. Through the use of recursion an arbitrary number of instances of a workflow can be generated, and the ULTRA bulk quantifier allows the parallel execution of multiple instances of a subworkflow whenever a certain condition is satisfied. For example, the ULTRA rule

$$notify(Topic) \leftarrow \#X\ [interested(X,Topic) \rightarrow write\_letter(X)]$$

activates the (sub)workflow *write_letter* for every X that satisfies the condition *interested(X,Topic)*. This, again, may be a database table which records every person who is interested in a certain topic, and so the whole rule spawns several concurrent instances of the *write_letter* workflow – one for every person that is interested in *Topic*. As described in Sect. 2, the bulk quantifier bridges the gap between the all-solutions semantics for retrieval and the one-solution semantics for updates.

4 The ULTRAflow Prototype

Although in the previous section we have shown how the patterns of workflows can be expressed using ULTRA rules, the prototype implementation of ULTRAflow does not work directly on the rules. Instead, the rules are compiled into an internal graph based representation that in some aspects resembles the well-known rule-goal graph. During this compilation, the whole workflow graph is generated from the rules, which poses some additional restrictions on the class of allowed programs: First, obviously the bulk quantifier is not supported in our prototype, as it cannot be decided at compile time how many instances of the given subworkflow have to be generated. Then, the treatment of recursive calls allows only tail-recursive programs in our prototype. Finally, implicit XOR-Splits arising from retrieval predicates or guards (see XOR-Split/Join above) are forbidden, i.e. only deterministic programs are allowed. Yet, even with these restrictions the prototype proved to be a valuable tool in real-life workflows of our research group.

5 Summary and Outlook

We have shown that the building blocks of workflow management systems can be expressed in the rule based update language ULTRA in an elegant and concise manner. The commonplace features of the rule language, like recursion and logical reasoning, give additional power for the specification of workflows, which usually is missing in a purely graph-based approach.

We are currently working on the complete formal definition of an instance of the ULTRA framework that is tailored to the specification of workflows. The prototype system is currently re-implemented as an Enterprise Java Bean. The new version will no longer demand a compilation of rules into graphs in advance, but will do so on the fly; this will also remove the restrictions mentioned above and allow bulk quantification, arbitrary recursion, and implicit XORs.

References

[1] A. J. Bonner. Workflow, transactions, and datalog. In *Proc. 18th ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, Philadelphia*, pages 294–305, 1999.

[2] A. Fent, C.-A. Wichert, and B. Freitag. Logical update queries as open nested transactions. In *Transactions and Database Dynamics*, volume 1773 of *LNCS*. Springer, 2000.

[3] W. M. P. van der Aalst, A. P. Barros, A. H. M. ter Hofstede, and B. Kiepuszeski. Advanced workflow patterns. In *Proc. 7th Int. Conf. on Cooperative Information Systems, Eilat, Israel*, volume 1901 of *Lecture Notes in Computer Science*, pages 18–29. Springer, 2000.

[4] W. M. P. van der Aalst, A. P. Barros, A. H. M. ter Hofstede, and B. Kiepuszeski. Workflow patterns homepage. Available at http://tmitwww.tm.tue.nl/research/patterns, 2001.

[5] C.-A. Wichert, A. Fent, and B. Freitag. A logical framework for the specification of transactions (extended version). Technical Report MIP-0102, University of Passau (FMI), 2001. Available at http://daisy.fmi.uni-passau.de/papers/.