EFFICIENT PARTITIONING OF GRAPHS IN COLLABORATIVE WORKFLOW EDITOR SYSTEMS

Gergely Sipos, Péter Kacsuk
MTA SZTAKI, Hungarian Academy of Sciences
Budapest, Hungary
{sipos, kacsuk}@sztaki.hu

ABSTRACT

Collaborative editing systems allow a group of users to view and edit a shared item from geographically dispersed sites. Consistency maintenance in the face of concurrent accesses to shared entities is one of the core issues in the design of these systems. Workflow modelling is a popular technique to describe business processes, scientific experiments, distributed applications. A workflow is directed graph which specifies tasks and data/control dependencies. The paper introduces protocols by which workflow developer environments can enable the concurrent editing of graphs by multiple users. The proposed graph partitioning and pessimistic locking algorithms assure that collaborators cannot break the consistency criteria of workflows by introducing cycles or invalid edges to them. We prove that the solution results correct graphs even when collaborative parties know separate parts of the workflow and do not share their own sub-graphs with each other in real time. A method to compare the efficiency of different graph partitioning algorithms is also provided.

KEYWORDS

workflow, collaboration, DAG, groupware, locking, evaluation

1. INTRODUCTION

Workflow techniques are widely used methods for parallel and distributed processing in business [1] and scientific environments [2]. Due to the widespread adoption of workflow technologies, the term “workflow” is heavily overloaded and has various different definitions. We consider a workflow as a process that consists of several steps (tasks) and defined in the form of a directed graph, in which vertices represent tasks to be performed by human or computer services and edges represent dependencies among these tasks. (Data or control dependency). A workflow has a development phase, during which the graph is defined in a graphical environment, and an execution phase, when an enactment engine instantiates the process of the workflow.

With the advance of workflow technologies, workflow applications become more complex they integrate more human knowledge. The development of complex and/or large workflows requires support for collaborative work. Recent works from the field of collaborative development of workflows studied concurrent access to workflows [3][4][5], but did not deal with inconsistency, a common issue in computer supported collaborative working (CSCW) environments. In the current paper we propose a solution that guarantees the consistency properties of workflow applications and enables not only effective, but also *correct interaction among users*.

Maintaining consistency of workflows is highly important in any CSCW application because faults not discovered at early stages can be debugged later only at high costs [6]. This paper focuses on the maintenance of three graph consistency criteria: (1) workflows must remain acyclic, (2) edges in the workflow must point to existing vertexes and (3) maximum one input edge can be connected to an input *channel* of a workflow vertex (a vertex that represents e.g. a Web service can have more than one input channels). These consistency rules are critical in those workflow languages, environments and managers that do not support cycles and/or cannot handle broken connections between vertexes. Protecting against multiple incoming edges assures that input data of a task is not overwritten from multiple sources. Several workflow tools require these criteria, e.g. GriPhyn (Pegasus), Taverna (SCUFL), Grid Service Flow Language, Condor

DAGMan, Workflow Enactment Engine, just to name a few. Moreover, the approach is also applicable in other domains where collaboratively edited entities are represented as acyclic graphs, e.g. in mind mapping tools or automating product design processes [7]. The contribution of the paper to the collaborative computing and to the workflow fields is a protocol that enables the real-time concurrent editing of graphs by multiple parties and the proof that the protocol guarantees consistency of the shared entities.

The next section provides an overview of related works. In Section 3 the concept of lock based collaborative editing of workflows is described. The problems related to workflow consistency are explained in Section 4, where our consistency aware locking protocol is introduced and a proof of correctness given. Section 5 provides discussion on additional properties of the solution such as deadlock and outlines a performance analysis method to compare the new protocol with two of our earlier algorithms. Our summary and conclusions are given in Section 6.

2. RELATED WORK

Real-time collaboration through shared, editable entities has recently become an intensively researched topic within computer supported collaborative work (CSCW). Most of the related efforts focused on enabling concurrent editing of items that are either unstructured (such as a drawing) or have hierarchical structure (such as a document consisting of sections, subsections). Unstructured items do not have consistency requirements and do not require consistency maintenance frameworks. Although consistency maintenance of hierarchical structures is well understood, *directed acyclic graphs cannot be mapped to hierarchical graphs* (tree graphs) and that is why they require slightly different consistency maintenance solutions.

Usage of CSCW approaches for the collaborative editing of workflow applications is a less researched field. Although some solutions support the collaboration of workflow developers, these systems are often do not provide services for real-time collaboration. For example, MyExperiment.org is a Web 2 style community site where scientist can publish, discover and download workflows [8], but the concurrent editing of these entities is not possible.

Some recent efforts in workflow research have focused on the knowledge sharing aspects of collaborative workflows in multi-organizational environments. While these works provide detailed analysis on the users' awareness and propose system architectural components, they rarely discuss concurrency control mechanisms and implementations [9][10]. There are a few exceptions though. Lock based concurrency control frameworks are given in [3][5]. Unfortunately none of these solutions protect the consistency of workflow graphs so some concurrent editing scenarios can result inconsistent applications that can be later executed only after corrections.

The HOBBES environment also provides lock based approach for concurrent editing of grid workflows [4], however as we discussed in [11] it provides very poor collaborative performance, i.e. it allows only very few users to work on the same graph concurrently. In our previous work [11] two versions of a lock-based workflow partitioning algorithm have been introduced. Although both partitioning algorithms protect the consistency of directed acyclic grid workflows, compared to the new solutions that is described in this paper they either lock too large sub-graphs for each user – allowing only a few people to concurrently edit a workflow – or use two types of locks, resulting a complicated system architecture that is difficult to add to existing workflow editor environments. The current solution applies only one lock to distinguish the different users' graph components. Version Control Systems (VCS) such as CVS, SVN, Git, Mercurial are the typical tools to support the concurrent engineering of software [6]. However these environments cannot help in the collaborative development of workflows because (i) workflows are typically stored in a single file (e.g. workflow description file) meanwhile VCS supports collaboration on file sets, not allowing the distribution of a single file among several users; and (ii) VCS work at the level of text and cannot interpret and resolve conflicts that happen at a higher abstraction level (in our case at the level of graph vertexes or edges).

3. COLLABORATIVE EDITING OF ACYCLIC GRAPHS

Collaborative development of workflows involves concurrent access to a single graph by multiple users. Collaborative development tools must assure that the users' contributions are integrated into a single, coherent application without resulting loss of data or invalid state. Workflows are developed manually, i.e. changes made on a workflow graph are results of human actions. In a collaborative environment if user *A*'s changes on a workflow are lost, or dropped due to a user *B*'s concurrent changes, then this results wasted, sometimes irreproducible effort for user *A*. *An important requirement for collaborative development of workflows is that no user's development session should be aborted because of another concurrent user's work.*

Turn-taking, serialization (often extended with operation transformation) and locking are the most commonly used concurrency-control techniques in groupware environments [12]. In our previous papers we argued that lock based concurrency control matches most with the needs of workflow developers [3][11]. We suggested locks at graph component level, i.e. vertices and edges are the lockable units. In our collaborative workflow editor architecture the lock manager appears as a central component. The role of the lock manager is to accept lock requests from the users, evaluate these requests and grant or deny the locks based on compatibility rules and protocols. In our previous paper [11] we introduced two different algorithms that can be integrated into a lock manager to grant and deny locks. In Section 4 a brief overview of these two algorithms will be given, and a third algorithm will be described. In the remaining part of the paper we call these algorithms as "*lock evaluator algorithms*". A lock evaluator algorithm can be modelled with a function that receives a workflow graph and a lock request as inputs, and gives a partitioning of the graph as output. The function partitions the graph to locked and unlocked parts. A locked sub-graph contains those elements of the workflow that the lock requestors' editing work will affect and must be locked for him/her to avoid conflicts. If a lock cannot be granted, then the user must be informed about the denial so he/she can modify the request or submit it at a later time.

When multiple users work on the same graph concurrently, then a series of locking requests reach the lock manager. The lock evaluator algorithm evaluates each request separately, and allocates different parts of the graph for the users:

$$G, R^S \rightarrow \text{Lock evaluator algorithm} \rightarrow G = G_1+G_2+ \ldots +G_n+G^U$$

where G marks the workflow graph on which n users work concurrently. $R^S = R_1,R_2,\ldots,R_n$ marks the locking requests of the users. R_i includes those graph components (vertices, edges) that user i wants to modify on the graph. Such a request can be generated by the user e.g. through the graphical editor where he/she selects the workflow components that he/she wishes to modify. G_1, G_2, ...G_n mark n sub-graphs of G that become locked for the users. G_i gets locked owner of the R_i lock request. G^U is the sub-graph of G that remains unlocked even after all the n users begin concurrently editing the graph. Note that each of the sub-graphs contains some vertices and some edges from G, but a G_i sub-graph is not necessarily connected. If e.g. a user intends to modify the first and the last nodes of a pipe-line graph, then his/her locked sub-graph may contain only these two nodes so the sub-graph is disconnected. Because of using exclusive locks $G_i \cap G_j = \emptyset, \forall i, j, 1 \leq i, j \leq n, i \neq j$ and $G_i \cap G^U = \emptyset, \forall 1 \leq i \leq n$.

4. CONSISTENCY OF WORKFLOW GRAPHS

Collaborative workflow editors must assure that editing sessions bring graphs from a consistent state to another consistent state. We identified the following three consistency criteria for workflow graphs [11]:

1. There is no "dangling" edge in the graph, i.e. an edge that refers to non-existing (already deleted) node as its source and/or sink.
2. Maximum one edge can provide input data for an input channel of a node. If multiple edges are connected to the same input channel, then the execution of the workflow would be unpredictable as the data items that arrive through the edges overwrite each other. (This condition still allows a vertex to have more than one input channel.)
3. The graph is acyclic.

In a collaborative editing scenario no party has the complete view of the workflow, thus maintaining consistency is not self-evident. In sections 4.1, 4.2 and 4.3 protocols are introduced to maintain the above consistency criteria in lock based environments.

4.1 Preventing Dangling Edges and Multiple Incoming Edges

A dangling edge appears in the graph when a transaction deletes a node without the removal of its edges. Multiple incoming edges come into existence when concurrent users connect input edges to the same input channel of the same node. In [11] we proved that any lock evaluator algorithm can protect against dangling edges and against multiple incoming edges if it implements the following protocol (See also Fig. 1):

- An edge can be locked for a transaction only if both of its end nodes can be locked for the same user. This assures that a locked G_i sub-graph is *always* enclosed by nodes.
- Within each locked sub-graph the users' editors can check for dangling edges. (In the same way as a single-user editor does.)
- Only the owner of a locked vertex can connect an edge to that vertex. *This rule implicitly declares that no vertex can be shared between multiple users, so locks cannot be used at a finer granularity, e.g. at the level of parameters of vertexes.*
- When an editing session is finished (the user clicks "Save workflow" in the editor) then the updated components of his/her locked sub-graph must be merged into the complete graph residing on the server. If the user deleted a vertex that was connected to other sub-graphs, then these edges must be deleted by the server. *Because these edges are not locked for any user, their removals do not make any transaction abort, do not waste any users' work.*

This protocol guarantees that workflow developers can perform any modification within their own sub-graphs and when these changes are merged into the complete graph they do not result dangling edges, multiple incoming edges and aborted transactions.

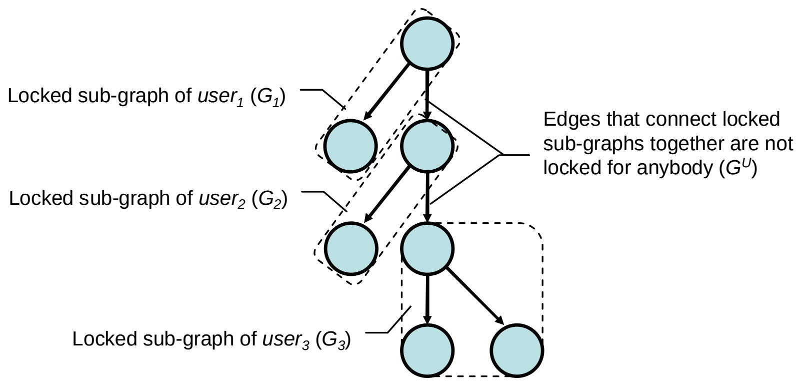

Locked sub-graph of $user_1$ (G_1)
Locked sub-graph of $user_2$ (G_2)
Locked sub-graph of $user_3$ (G_3)
Edges that connect locked sub-graphs together are not locked for anybody (G^U)

Fig 1. Graph partitioning protocol that prevents dangling edges and multiple incoming edges.

4.2 Preventing Cycles

Collaborative workflow editors can recognize and eliminate cycles within their own sub-graphs, however, none of the editors or even the central lock manager knows the current state of the whole workflow during an editing session. *In extreme cases it can happen that nonetheless no cycle exists in any of the locked sub-graphs, a cycle exists in the whole workflow [11].* Although a central component that validates every edge-creation operation could allow only valid modifications to the graph, such a component would surely slow down client side editors and would make offline editing impossible. Our proposed solution for this problem is a lock partitioning protocol that creates sub-graphs in such a way that no edge creation operation within any of the locked sub-graphs can result cycle in the whole graph. In our previous paper we introduced two graph partitioning algorithms that implement the protocol from Section 4.1 and that also prevent cycles [11].

The first algorithm of that paper (to be called v1 algorithm now) locks complete branches of a graph, i.e. if a V vertex needs to be locked for user A, then every child vertex of V and every child edge of V also become locked for A. If either V, or any of its child components are already locked for a user B, then A's locking request is denied, the system does not try to find a smaller, but lockable sub-graph.

The second algorithm from our previous paper (to be called v2 algorithm now) is an improved version of v1, because it uses two types of locks. Those components that user A requested to lock should be locked with USER lock (U lock). The child vertices and child edges of these components are locked with SYSTEM locked (S lock). While U locks are visible for users and mark the editable components, S locks are invisible to users and seen only by the lock evaluator algorithm to decide about lock compatibility. U locks are incompatible with each other, i.e. no more than one user can have U lock on a component. On the other hand, we proved that multiple S locks can exists on the same component, moreover, in some cases both an S lock and an U lock is allowed on the same component. It has been proven that a lock request must be denied *only*, if it puts an U lock on a component that already has an S lock and at the same time it puts an S lock on a component that already has an U lock.

Our new algorithm (to be called v3 algorithm now) uses only one lock. (Like the v1 algorithm, or like to U lock in the v2 algorithm.). It uses a new concept, called "contiguity graph" to decide about granting/denying locks. Before describing the v3 algorithm we provide a definition of the "contiguity graph".

Definition: A *contiguity graph* is a graph that represents the connections among locked sub-graphs, and among locked sub-graphs and unlocked vertices of a workflow. The contiguity graph $C=C(V^C, E^C)$ of a $G=G(V, E)=G_1+G_2+..+G_n+G^U$ workflow can be generated with the below described Algorithm (1). In the contiguity graph every locked sub-graph of the G graph appears as a single node (Step 2); every edge that connects two locked sub-graphs together appears as an edge (Step 3), and every unlocked component (vertex or edge) appears as it is (a vertex or an edge) (Step 4 and 5).

Algorithm (1): Generating contiguity graph:

```
1) $V_C = \{\}$, $E_C = \{\}$
2) For $\forall$ $G_i$
      Add a new vertex denoted as $V_i^C$ to $V^C$
3) For $\forall$ $G_i, G_j \in V$ $(i \neq j)$
     For $\forall$ $e=e(v_k, v_l)$, $e \in E$ | $v_k \in G_i$ , $v_l \in G_j$
        Add $e^C=c(v_i^C, v_j^C)$ to $E^C$
4) For $\forall$ $v \in G^U$
      Add a new vertex denoted as $V_i^C$ to $V^C$
5) For $\forall$ $e=e(v_i, v_j) \in G^U$
      Add a new edge denoted as $e^C=e^C(v_i^C, v_j^C)$ to $V^C$
```

The v3 lock evaluator algorithm uses the contiguity graph to decide about lock requests and it is defined with Algorithm (2) below. V3 first allocates all those components that the submitter of R_i wants to write (Step 2). It then allocates the end nodes of all the requested edges (Step 3). This is required to protect against dangling edges and multiple incoming edges as it defined in Section 4.1. In Step 4 the manager checks whether any of the components that should be locked is already locked. It denies the lock if this is the case. If still all the components are available for locking then it generates the contiguity graph (Step 5) and checks whether there is a cycle in it (Step 6). If a cycle is found, then it denies the lock. If there is no cycle in the contiguity graph then it locks the components and notifies the user (Step 7, 8). The editor can allow the user edit these components and when the workflow is saved it propagates the modified components back to the server where they are used to update the complete workflow.

Algorithm (2): Consistency-aware lock evaluator algorithm (v3 algorithm):

```
Input: $R_i$ - The locking request to evaluate. $R_i$ includes those components of the
graph that its owner wants to modify or delete
Output: $G_i$ - The sub-graph that must be locked to serve the request. If $G_i$ is
empty, then the request is denied and no lock is granted.

1) $G_i = \{\}$
2) Add every component from $R_i$ to $G_i$
3) Add the end nodes of every edge from $R_i$ to $G_i$
4) If (any component of $G_i$ is already locked) then $G_i = \{\}$
5) Generate the contiguity graph of G $\rightarrow$ $G^C$
6) If (cycle exist in $G^C$) then $G_i = \{\}$
7) Lock elements of $G_i$
8) Return $G_i$ to the owner of $R_i$
```

The v3 lock evaluator algorithm assures that there is no need to abort or modify any users' editing session, his/her changes can be integrated into the complete workflow without breaking its consistency. An important step in algorithm (2) is Step 6, as it claims that *a lock request must be denied if a cycle exists in the contiguity graph.* We prove now that checking for cycle in the contiguity graph is enough to protect against cycles in workflows.

Statement: If an editing session is able to create a cycle in a workflow, then a cycle must have existed in the contiguity graph of the workflow when the editing session started.

Proof: Assume that $user_A$, $user_B$, … $user_M$ concurrently edit a G graph. Assume that the modifications made by $user_1$, $user_2$, … $user_i$ $(i<=M)$ result a cycle in G. Denote with e_1, e_2, ..., e_i those edges that were added by $user_1$, $user_2$, … $user_i$ to the cycle respectively. Observations:

- Because these user could add new edges to their sub-graphs (G_1,G_2,,...G_i) there must be at least 2-2 nodes in every sub-graph. (If a transaction holds lock on an edge then it must hold locks on its source and sink nodes too – see Step 3 of the algorithm.)
- Because adding the e_1, e_2, ..., e_i edges to the G_1, G_2,, ...G_i sub-graphs results a cycle in G, there must be directed routes among the G_1, G_2,, ...G_i sub-graphs before the editing sessions started. If this would not be true, and the route between some G_a and G_b sub-graphs was created by some $user_x$, then G_x sub-graph should hold common nodes with G_a and G_b sub-graphs. This is impossible because the above algorithm denies the start of editing sessions with overlapping components, so it would deny the parallel work of $user_x$. $user_a$ and $user_b$.

So when the R_i was evaluated by the algorithm then already at least 2-2 nodes exist from the cycle in G_1,G_2,,...G_i and there were already directed routes between these sub-graphs. In the continuity graph every of these sub-graphs is represented as a single node, so there must be a cycle in the contiguity graph. □

5. DISCUSSION AND PERFORMANCE ANALYSIS

All the three lock evaluator algorithms (v1, v2 and v3) provide such a partitioning of acyclic graphs, that the consistency rules cannot be broken by any editing user. However, none of the solutions guarantee deadlock free editing scenarios. It can happen that user A locks sub-graph G_a, user B locks sub-graph G_b and they both wait for each others sub-graphs to become unlocked in order to extend their own sub-graphs with those components. Prohibiting users to hold locks on more than one workflow component could be a solution as it is applied in the GroupGraph system [5]. However, in this way operations that affect multiple components (e.g. a definition of a new edge) cannot be performed. That is why we suggest the inclusion of awareness tools in the graph editors, so users can see the topology of locks on the whole graph, can contact each other and can manually delay locks in favour of others. Groupware widgets, such as the MAUI Toolkit [14] can be used to increase the developers' group awareness.

The three algorithms use different policies to evaluate lock requests, and consequently can result different partitioning of a graph. In the same situation one algorithm can deny the lock request, while another algorithm can grant the locks. *From the users' point of view the algorithm that allows the most developers to work on the same graph concurrently, provides the best collaborative performance.* The more users can access to a workflow the sooner they can finish the definition process and can proceed to workflow execution. Groupware systems are typically evaluated through prototype implementations with laboratory studies or through production implementations with real-life scenarios [13]. Because the decision made by a lock evaluator algorithm in a given locking situation can be predicted, we are in better situation and can compare the collaborative performance of the algorithms in a purely theoretical way.

The decision made by a lock evaluator algorithm depends only on the current state of the workflow graph, the topology of the existing locks, and the topology of the requested locks. The decision is independent from several other parameters, such as the unsaved parts of the graph (These are visible only in the client side editors.) When an algorithm evaluates the $R^S = R_1,R_2,...,R_n$ sequence of locking requests on a G graph, then besides partitioning G to locked sub-graphs and an unlocked sub-graph, the algorithm implicitly returns a 0 or a 1 value for each request. 0 means that the lock request is denied, 1 means that the request can be served. *The more 1 digits are given for the evaluation of an R^S lock request on a G graph, the more users are* *allowed to concurrently edit the workflow. Consequently, by comparing the binary vectors of the algorithms for a given editing situation, the collaborative performance of these algorithms can be compared.* It is easy to see that the v2 algorithm provides a better collaborative performance than the v1 algorithm. In situations when a user's lock request does not require S lock on the graph the v2 algorithm results the same lock topology than the v1 algorithm, so it allows exactly the same persons to work with the graph. However, when S locks must be used by v2 and only S locks cause collision, then v2 allows more people to edit the workflow. *As a consequence the v2 algorithm gives better user satisfaction than the v1 algorithm in any collaborative workflow editor environment.*

The comparison of the v2 and v3 algorithms does not result such a clear answer. We found, that in some editing situations the v2 algorithm allows more users to work on the graph, while in other cases the v3 algorithm provides better collaborative performance. Fig. 2 gives one example for each of these situations.

The upper half of Fig 2. shows three users requesting locks on a G graph. While v3 algorithm allows all the three persons to lock and work, v2 would deny the third request. The bottom half of Fig. 2 shows that v2 allows two users to work on an F graph, while v3 allows locks only to the first person. Our current research focuses on the categorisation of editing cases and then on the definition of an intelligent lock evaluator algorithm, that uses both S locks and contiguity graphs to maximise the success of locks, to provide better collaborative performance than v2 and v3.

V3 provides better performance than v2

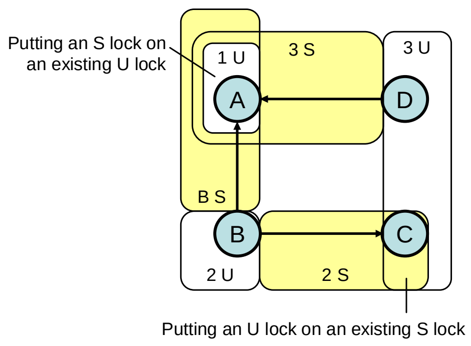
Users' lock requests:

R_1={A},
R_2={B},
R_3={C, D}

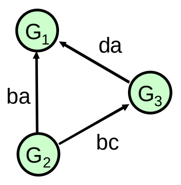
Evaluation outcome:

- **v2 denies** lock for R_3 because it puts S lock on U lock and U lock on S lock.
- **V3 allows** lock for R_3 because there is no cycle in the contiguity graph

G graph with locks issued by v2:


Putting an S lock on an existing U lock
1 U
3 S
3 U
A
D
B S
B
C
2 U
2 S
Putting an U lock on an existing S lock

Contiguity graph of G:



v2 provides better performance than v3

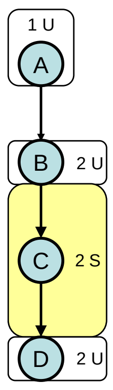
Users' lock requests:

R_1={A}
R_2={B, D}

Evaluation outcome:

- **v2 allows** lock for R_2 because it causes no lock collision
- **v3 denies** lock for R_3 because ther is cycle in the contiguity graph

F graph with locks issued by v2:



Contiguity graph of F:

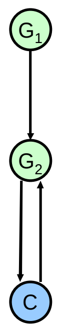


Fig 2. Example scenarios that demonstrate the difference in the collaborative performance of v2 and v3 algorithms. (Notations: 1 U: component locked with U lock for R_1; 1 S: component locked with S lock for R_1, …)

6. SUMMARY AND CONCLUSIONS

The paper presented a method for the collaborative editing of workflows with protecting consistency of graphs. The solution is based on a pessimistic locking approach which ensures that graphs are partitioned for users in such a way that their contributions cannot result cycles, dangling or overlapping edges in the graph. *The solution assures that no user's editing transaction is aborted and no user's development effort is wasted*. The introduced algorithm uses a different method to decide about locks than the two other algorithms that we discussed in our previous work [11]. A method to compare the collaborative performance of graph locking algorithms was also presented. While the second algorithm can perform better than the third, there cannot be clear decision made between the second and third algorithms. Currently, we are performing a detailed analysis of the collaborative graph editing scenarios to be able to define the most powerful algorithm, which would allow the highest number of users to share a graph.

We have a prototype implementation of a collaborative workflow editor environment [3]. It is called Collaborative P-GRADE Grid Portal and it demonstrates that an originally single-user workflow tool can be turned into a groupware application with lock based concurrency control techniques. Once the most efficient graph partitioning algorithm is found we intend to further develop the Collaborative P-GRADE Grid Portal and make it use this solution for lock request evaluation.

REFERENCES

1. Diimitrios Georgakopoulos, Mark Hornick, Amit Sheth: An overview of workflow management: From process modeling to workflow automation infrastructure, Journal of Distributed and Parallel Databases, Volume 3, Number 2. 1995, pp. 119-153.
2. Ian Foster, Carl Kesselman (eds.), The Grid: Blueprint for a New Computing Infrastructure, Morgan Kaufmann, 1998.
3. Sipos, G. et al, 2005, Workflow-oriented Collaborative Grid Portals, Advances in Grid Computing, In. Proc. of European Grid Conference, EGC 2005, LNCS 3470, Amsterdam, The Netherlands, pp. 434-443.
4. Held, M., Blochinger, W., 2009, Structured Collaborative Workflow Design, In. *Future Generation Computer Systems*, Vol. 25, Issue 6, Pages 638-65.
5. H. A. S. Lima Filho, Celso M. Hirata, GroupGraph: A Collaborative Hierarchical Graph Editor Based on the Internet, Proceedings of the 35th Annual Simulatin Symposium, 2002.
6. Dewan, P., Riedl, J., 1993, Toward computer-supported concurrent software engineering, In IEEE Computer, vol. 26, no. 1, pp. 17-27.
7. Huang, C.J. Liao, L.M., 2007, An intelligent agent-based collaborative workflow for inter-enterprise PCB product design, In Proc of IEEE International Conference on Industrial Engineering and Engineering Management, Singapore, pp. 189-193
8. Goble C.A., De Roure D.C., 2007, MyExperiment: Social Networking for Workflow-Using E-scientists, In *Proceedings of the 2nd workshop on Workflows in support of large-scale science*, Monterey, California, USA. ACM, New York, USA
9. Schuster, H. et al., 2000, The collaboration management infrastructure, In *Proc. of ICDE Conference*, San Diego, California., USA, pp 677–678.
10. Friese, T., et al, 2006, Collaborative Grid Process Creation Support in an Engineering Domain, In proc of High Performance Computing - HiPC 2006, pp. 263-276.
11. Sipos, G., Kacsuk, P., 2009, Maintaining Consistency Properties of Grid Workflows in Collaborative Editing Systems, *Proc. of Grid and Collaborative Computing Conference (GCC09)*, IEEE-publishing, Lanzhou, China, pp.168-175.
12. Chengzheng Sun , Xiaohua Jia , Yanchun Zhang , Yun Yang , David Chen, Achieving convergence, causality preservation, and intention preservation in real-time cooperative editing systems, ACM Transactions on Computer-Human Interaction (TOCHI), vol.5 no.1, pp.63-108, 1998.
13. Pinelle, D., and Gutwin, C., 2000, A Review of Groupware Evaluations, Proceedings of Ninth IEEE WETICE 2000 Workshops on Enabling Technologies, Gaithersburg, Maryland, pp. 86-91, 2000.
14. Hill, J., Gutwin, C., 2004, The MAUI Toolkit: Groupware Widgets for Group Awareness, In Journal of Computer Supported Cooperative Work (CSCW), Vol. 13 No. 5-6, pp. 539-571.