Using the Transformational Approach to Build a Safe and Generic Data Synchronizer

Pascal Molli
molli@loria.fr

Gérald Oster
oster@loria.fr

Hala Skaf-Molli
skaf@loria.fr

Abdessamad Imine
imine@loria.fr

ECOO & CASSIS Team
INRIA Lorraine
FRANCE

ABSTRACT

Reconciliating divergent data is an important issue in concurrent engineering, mobile computing and software configuration management. Currently, a lot of synchronizers or merge tools perform reconciliations. However, they do not define what is the correctness of their synchronisation. In this paper, we propose to use a transformational approach as the basic model for reasonning about synchronisation. We propose an algorithm and specific transformation functions that realize a file system synchronisation. Unlike classic synchronizers, our synchronizer ensures properties of convergence, causality and intention preservation and is extensible to new data types.

Categories and Subject Descriptors

D.2 [**Software Engineering**]: Distribution, Maintenance, and Enhancement

General Terms

Algorithms, Reliability

Keywords

Synchronization, Operational transformation

1. INTRODUCTION

Generally, users involved in mobile computing, collaborative computing and concurrent engineering work on replicas of shared data. They can make updates while working disconnected or insulated. This generates divergence on replicas that has to be reconciliated later. Synchronization is a critical application. If safety is not ensured, users can loose data, read inconsistent data and propagate inconsistencies. This can be dramatic in the context of distributed software engineering or mobile computing.

Many systems exist today for reconciliating divergent data: file synchronizers, tools for PDAs, configuration management tools with merge tools, optimistic replication algorithms in databases, groupware algorithms in CSCW and distributed systems algorithms. However, an important issue is still open: what is a *correct* synchronization ? How to write a *safe* synchronizer ? In this paper, we propose to use the transformational approach[6, 25, 21, 24] as a theoretical foundation for a safe and generic data synchronizer. This approach allows to define general correctness criteria for synchronizing any kind of data. To validate this approach, we developed a prototype that allows to synchronize *with the same algorithm* a file system and *file's contents* for text files and XML files. The same correctness properties are ensured at all levels of synchronization. Of course, we can extend this prototype for more data types. In this paper, we present the transformational model and how we can use it for building a safe and generic data synchronizer.

The paper is organized as follows: Section 2 details the problems with current synchronizers. Section 3 gives an overview of the transformational approach. Section 4 presents the generic integration algorithm. Section 5 and 6 detail transformation functions for a file system and text files. Section 8 presents related works. The last section concludes with some points to future works.

2. WHY A SAFE AND GENERIC SYNCHRONIZER ?

Synchronization is a process that takes two divergent copies as inputs and makes them identical. Unfortunately, for two divergent data, there are many states of convergence. A correct synchronizer must force convergence towards a state that enforces predefined properties.

Current synchronizers do not define such properties, they propagate non-conflicting updates to other copies and delegate resolution of conflicting updates to users [1]. So two different existing synchronizers will not produce the same results in the same situation.

Another important issue is the granularity of reconciliation. Suppose two text files edited in parallel by two users, one is appending a new chapter, the other one is checking the grammar. Current file synchronizer will ask to choose between the two versions. In fact, reconciliation is performed at wrong level of granularity. To be more precise, reconciliation must be done at all levels of granularity; at the file system level and at the file content level and maybe at the character level. This leads to another important issue: the genericity of the synchronization. Is the algorithm of reconciliation the same for two integers, blocks of text, XML Trees, Databases tables ? Currently, in software configuration environments, different tools are used to merge objects at different levels of genericity. One reconciles the file system and the other one reconciles file text. Unfortunately, they do not apply the same strategy. For example, with CVS [2], if a conflict occurs at the file system level, users are required to solve the conflict. If a conflict occurs at the content of text file, a merge is done automatically and users can compensate system decisions after synchronisation completion. This raises two new questions : 1) who will resolve conflicts ? the synchronizer, the user or the system administrator. 2) when conflicts have to be resolved ? during the synchronization or after.

In distributed systems like CODA[13] or Bayou[15], when a conflict is detected, the system tries to solve it automatically. But if it fails, the system delegates conflicts resolution to the administrator, data remains frozen until the conflicts are solved. In replicated Database Systems[8], when a conflict is detected, the system tries to perform automatic merge procedures. But these procedures are not defined for all kind of conflicts and so convergence is not ensured in all cases.

In this paper, we propose to use the transformational approach to build a synchronizer that forces convergence in all cases and reconciles divergent data at all possible levels of granularity. The synchronizer resolve *all* conflicts automatically without delegating them to users nor administrator. Users can compensate system decisions after synchronization completion.

3. TRANSFORMATIONAL APPROACH

The model of transformational approach considers n sites. Each site has a copy of the shared objects. When an object is modified on one site, the operation is executed immediately and sent to others sites to be executed again. So every operation is processed in four steps: (a) generation on one site, (b) broadcast to others sites, (c) reception by others sites, (d) execution on other sites.

The execution context of a received operation op_i may be different from its generation context. In this case, the integration of op_i by other sites may lead to inconsistencies between replicas. We illustrate this behavior in figure 1(a). There are two sites $site_1$ and $site_2$ working on a shared data of type $String$. We consider that a $String$ object can be modified with the operation $Ins(p, c)$ for inserting a character c at position p in the string. We suppose the position of the first character in string is 0. $user_1$ and $user_2$ generate two concurrent operations: $op_1 = Ins(2, f)$ and $op_2 = Ins(5, s)$. When op_1 is received and executed on $site_2$, it produces the expected string "effects". But, when op_2 is received on $site_1$, it does not take into account that op_1 has been executed before it. So, we obtain a divergence between $site_1$ and $site_2$.

In the operational transformation approach, received operations are transformed according to local concurrent operations and then executed. This transformation is done by calling transformation functions. A transformation function T takes two concurrent operations op_1 and op_2 defined on the same state s and returns op'_1. op'_1 is equivalent to op_1 but defined on a state where op_2 has been applied. We illustrate the effect of a transformation function in figure 1(b). When op_2 is received on $site_1$, op_2 needs to be transformed according to op_1. The integration algorithm calls the transformation function as follows:

$$T(\overbrace{Ins(5, s)}^{op_2}, \overbrace{Ins(2, f)}^{op_1}) = \overbrace{Ins(6, s)}^{op'_2}$$

The insertion position of op_2 is incremented because op_1 has inserted an f before s in state $efect$. Next, op'_2 is executed on $site_1$. In the same way, when op_1 is received on $site_2$, the transformation algorithm calls:

$$T(\overbrace{Ins(2, f)}^{op_1}, \overbrace{Ins(5, s)}^{op_2}) = \overbrace{Ins(2, f)}^{op'_1}$$

In this case the transformation function returns $op'_1 = op_1$ because, f is inserted before s. Intuitively we can write the transformation function as follows:

```
T(Ins(p1,c1),Ins(p2,c2)):-
  if (p1 < p2) then
    return Ins(p1, c1)
  else
    return Ins(p1+1, c1)
  endif
```

This example makes it clear that the transformational approach defines two main components: the *integration algorithm* and the *transformation functions*. The Integration algorithm is responsible for receiving, broadcasting and executing operations. It is independent of the type of shared data, it calls transformation functions when needed. The transformation functions are responsible for merging two concurrent operations defined on the same state. They are specific to the type of shared data ($String$ in our example).

A more theoretical model is defined in [25, 21, 24, 23]. To be correct, an integration algorithm has to ensure three general properties:

Convergence When the system is idle (no operation in pipes), all copies are identical.

Causality If on one site, an operation op_2 has been executed after op_1, then op_2 must be executed after op_1 in all sites.

Intention preservation If an operation op_i has to be transformed into op'_i, then the effects of op'_i have to be equivalent to op_i.

To ensure these properties, it has been proved [25, 21] that the underlying transformation functions must satisfy two conditions:

1. The condition C_1 defines a *state equivalence*. The state generated by the execution of op_1 followed by $T(op_2, op_1)$ must be the same as the state generated by the execution of op_2 followed by $T(op_1, op_2)$:

$$C_1 : op_1 \circ T(op_2, op_1) \equiv op_2 \circ T(op_1, op_2)$$

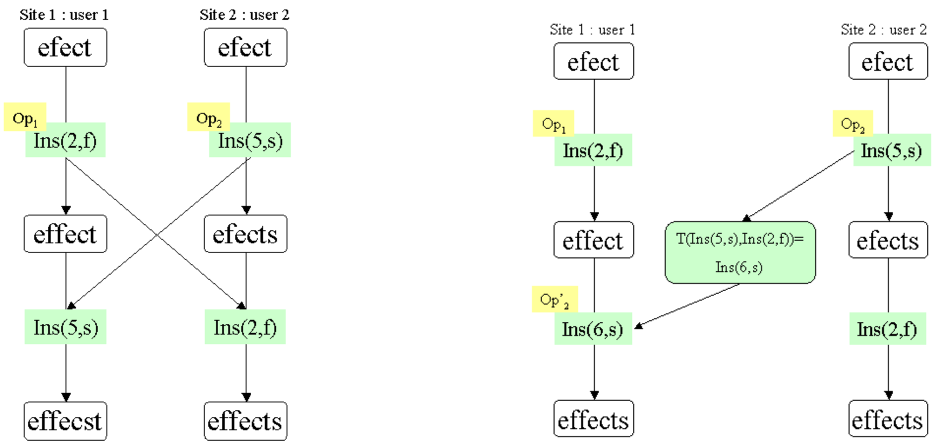
2. The condition C_2 ensures that the transformation of an operation according to a sequence of concurrent operations does not depend on the order in which operations of the sequence are transformed:

$$C_2 : T(op_3, op_1 \circ T(op_2, op_1)) = T(op_3, op_2 \circ T(op_1, op_2))$$

In order to use the transformational model, we must follow these steps:

1. Choose a integration algorithm. Depending of the algorithm, C_2 may be required or not on underlying transformation functions.
2. Define shared data types with their operations
3. Write transformation functions for all combination of operations. For example, on a string object with $Ins(p, c)$, $Del(p)$, we have to define:

```
T(Ins(p1,c1),Ins(p2,c2)):-
```



(a) Incorrect integration

(b) Integration with transformation

Figure 1: Integration of two concurrent operations

```
T(Ins(p1,c1),Del(p2)):-
T(Del(p1),Ins(p2,c2)):-
T(Del(p1),Del(p2)):-
```

4. Prove the required conditions on these transformation functions.

4. AN INTEGRATION ALGORITHM FOR SYNCHRONIZATION

```
Sync(log,Ns) :-
  while ((opr = getOp(Ns+1))!=∅)
    for (i=0;i<log.size();i++)
      opl=log[i];
      log[i]=T(opl,opi)
      op'i=T(opi,opl);
    endfor
    execute(op'i)
    Ns=Ns+1
  endwhile

  for (i=0;i<log.size();i++)
    op'l=log[i];
    if send(op'l,Ns+1) then
      Ns=Ns+1
    else
      error 'need to synchronize'
    endif
  endfor
```

Figure 2: Generic synchronization algorithm

In the transformational approach, the integration algorithm has the responsibility of receiving, integrating, broadcasting and executing operations. Among the existing algorithms, SOCT4[28] with its deferred broadcast is the most suitable algorithm for our synchronization needs. SOCT4 is based on a continuous global order of operations and requires only C_1 to be verified by transformation functions. Each operation is sent with a unique global timestamp. An operation on a site S with a given timestamp cannot be sent if all the preceding operations based on the timestamp order have been received and executed.

Our synchronization algorithm based on SOCT4 is presented in figure 2. Synchronizing a site S takes two parameters: *log* and N_s. *log* is the sequence of operations executed locally since the last synchronization. N_s is an integer which contains the timestamp of the last operation received or sent by site s. We define two functions:

1. getOp(int ticket) →op: retrieves the operation identified by the timestamp *ticket*. If no operation is available, *getOp* return ∅

2. send(Operation op, int ticket) →boolean: sends a local operation with the timestamp *ticket*. If *ticket* already exists, it means that a concurrent synchronization is in progress. In this case, the operation *send* returns false. The current state is not corrupted, it requires just to start again another synchronization.

$site_1$	$site_2$
op_1	op_3
op_2	op_4
$s_1 = synchronize$	
	$s_2 = synchronize$
$s_3 = synchronize$	

Figure 3: Scenario of integrations

Suppose we want to synchronize two sites as illustrated in figure 3. At the beginning, each site has $N_s = 0$. $site_1$ performed two local operations op_1, op_2, and $site_2$ performed two concurrent local operations op_3, op_4.

1. At point s_1, $site_1$ synchronizes. It calls $sync([op_1, op_2], 0)$. There is no concurrent operation available, so we just send op_1, op_2 as it is to $site_2$. Now, $N_s = 2$ on $site_1$.

2. At point s_2, $site_2$ synchronizes by calling $sync([op_3, op_4], 0)$. The following transformation functions are called:

$op'_1 = T(op_1, op_3)$
$op'_3 = T(op_3, op_1)$
$op''_1 = T(op'_1, op_4)$
$op'_4 = T(op_4, op'_1)$
$op'_2 = T(op_2, op'_3)$
$op''_3 = T(op'_3, op_2)$
$op''_2 = T(op'_2, op'_4)$
$op''_4 = T(op'_4, op'_2)$

op_1'', op_2'' are executed on $site_2$. op_1'', op_2'' are sent to others sites. Now $N_s = 4$ on $site_2$.

3. At point S_3, $site_1$ synchronizes again by calling $sync([], 2)$. There is no more local concurrent operations, so remote operations are executed without transformation on $site_2$ and $N_s = 4$.

4. After point S_3, $site_1$ has executed the following sequence:

op_1
op_2
$op_3'' = T(T(op_3, op_1), op_2)$
$op_4'' = T(T(op_4, op_1'), op_2')$

and $site_2$ has executed the following equivalent sequence:

op_3
op_4
$op_1'' = T(T(op_1, op_3), op_4)$
$op_2'' = T(T(op_2, op_3'), op_4')$

This equivalence is ensured if transformation functions verify C_1. It is clear in this example that conflicts detection and conflicts resolution are delegated to transformation functions. However, the problem is now simpler. A transformation function detects and resolves conflicts for *one combination* of *two concurrent operations* defined on the *same state*. If one transformed operation has an effect on the next operation, the cascading effects are handled by the integration algorithm.

This algorithm is a safe generic synchronizer if underlying transformation functions verify condition C_1. It preserves convergence, causality and intention.

5. TRANSFORMATION FUNCTIONS FOR A FILE SYSTEM

We define transformation functions for a file system and for each type of files. We limit our description to text files.

Writing *correct* transformation functions is complex. We have to preserve convergence by verifying condition C_1 and intention by computing equivalent operations. Our general strategy when writing transformation functions is to *converge to a state in which conflicts are represented.* A merge tool does the same thing when it merges two files. For example, rcsmerge [26] handles an update conflict by producing the following output:

```
<<<<<<< testfile.txt
std::string LineReader::readLine()
{
  return std::read_line( cin );
}=======
CString LineReader::ReadLine()
{
  CString line;
  m_archive >> line;
  return line;
}>>>>>>> 1.1.1.1.2.1
```

Users resolve the conflict by just editing the file. We apply the same principle for the file system. We handle conflicts on a file system by renaming files or directories involved in this conflict. For example, if two users create concurrently the same file in the same directory we converge to a state where we have renamed one file. Users can compensate this choice after synchronisation by using the *move* operation and synchronize again.

The safety of the transformational approach relies on correctness of transformation functions. If transformation functions do not verify C_1 then the integration algorithm ensures nothing. Proving condition C_1 is error prone, time consuming and part of an iterative process. It is nearly impossible to do this by hand. We made the proof using SPIKE : an automatic theorem prover [20, 11, 10]. The input of SPIKE is exactly the transformation functions written in this paper.

We consider a file system like a tree where nodes are directories and leaves are files. We define the following operations:

1. mf(int id, int pid, String name): mf stands for mkfile. It creates a file identified with a unique id . pid is the parent identifier. id is referenced with the name $name$ in pid. mf has the following preconditions: id does not exist, pid exists and $name$ is not used by pid.

2. md(int id, int pid, String name): md stands for mkdir. It creates a directory. In order to represent the root of the tree, we consider that a unique identifier 0 exists and represents the root.

The sequence $mf(1, 0, a); md(2, 0, b); mf(3, 2, a); mf(4, 2, b)$ builds the tree illustrated in figure 4(a):

3. mv(int pid1, int id1, String name1,int pid2, String name2). Moves the object identified by id_1 referenced in pid_1 with the name $name_1$ to the node identified by pid_2 with $name_2$. mv has the following preconditions: pid_1, id_1, pid_2 exist. id_1 is referenced with the name $name_1$ in pid_1. $name_2$ is not used in pid_2. If we apply $mv(2, 4, b, 0, c)$ on the tree illustrated in 4(a), we obtain the state described in figure 4(b).

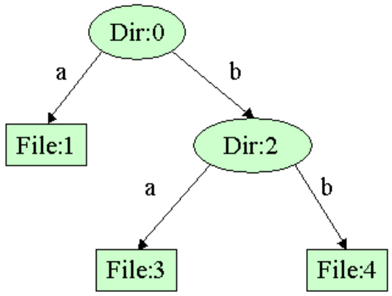
We do not define the remove operation, we consider that removing is equivalent to moving a subtree to a directory representing the garbage.

```
T(mf(id1, idp1, n1, s1), mf(id2, idp2, n2, s2))=
if (idp1 == idp2) then
  if (n1 == n2) then
    if (id1 < id2) then return
    mf(id1, idp1, max(s1) ⊙ id1,
           s1 ∪ {max(s1) ⊙ id1})
    else return
     mv(idp2, id2, n2, idp2, max(s2) ⊙ id2,
          s2 \ {n2} ∪ {max(s2) ⊙ id2})
     ⊎ mf(id1, idp1, n1, s2 ∪ {max(s2) ⊙ id2})
    endif
  else
  return mf(id1, idp1, n1, s2 ∪ {n1})
  endif
else
  return mf(id1, idp1, n1, s1)
endif;
```

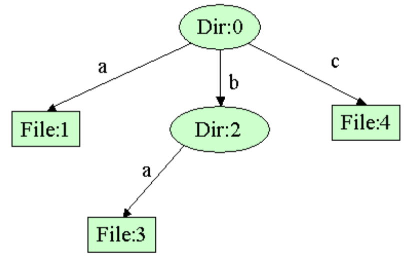
Figure 5: Transformation Function for mkfile-mkfile

Figure 5 represents the transformation function for mkfile-mkfile. Renaming entries in a file system is a little tricky:

1. How to compute a new unique name in a directory ? In order to represent conflicts by renaming files or directories, we need to compute unique names within the transformation function. This must be done using only the state informations where both concurrent operations are defined. Every operation modifying a directory provides the set of names contained in the directory after the execution of the operation. For example, on a directory identified by 1 and containing names $\{a, b, c\}$, the operation $mf(2, 1, d)$ is created with a fourth parameter s containing the set $\{a, b, c, d\}$. On this set, we define an extra operation $max(s)$. It returns the name with highest lexicographical value. If we append id of the renamed object to $max(s)$, we obtain a new unique name $max(s) \odot id$. $\odot$ is the append operator.



(a) Initial tree



(b) tree after $mv(2, 4, b, 0, c)$

Figure 4: File System Representation

```
T( mf( id1 ,idp1 ,n1 ,s1 ) ,
   mv( opid2 ,id2 ,nb ,idp2 ,n2 ,s2 ) ) =
if ( idp1 == idp2 ) then
  if ( n1 == n2 ) then
    if ( id1 < id2 ) then return
      mf( id1 ,idp1 ,max(s1) ⊙ id1 ,
      s2 ∪ {max(s1) ⊙ id1} )
    else return
      mv( idp2 ,id2 ,n2 ,idp2 ,max(s1) ⊙ id2 ,
            s2 \ {n2} ∪ {max(s1) ⊙ id2} )
      ⊎ mf( id1 ,idp1 ,n1 ,s2 ∪ {max(s1) ⊙ id2 )
    endif
  else return
    mf( id1 ,idp1 ,n1 ,s2 ⊙ n1 )
  endif
else return
  mf( id1 ,idp1 ,n1 ,s1 )
endif
```

Figure 6: Transformation Function for mkfile-move

```
T( mv( opid1 ,id1 ,na ,idp1 ,n1 ,s1 ) ,
   mf( id2 ,idp2 ,n2 ,s2 ) ) =
if ( idp1 == idp2 ) then
  if ( n1 == n2 ) then
    if ( id1 < id2 ) then return
      mv( opid1 ,id1 ,na ,idp1 , max(s2) ⊙ id1 ,
            s2 ∪ {max(s2) ⊙ id1} \ {na} )
    else return
      mv( idp2 ,id2 ,n2 ,idp2 ,max(s2) ⊙ id2 ,
            s2 ∪ {max(s2) ⊙ id2} \ {n2} )
      ⊎ mv( opid1 ,id1 ,na ,idp1 ,n1 ,
              s2 ∪ {max(s2) ⊙ id2} \ {na} )
    endif
  else return
    mv( opid1 ,id1 ,na ,idp1 ,n1 ,s2 ∪ {n1} \ {na} )
  endif
else return
  mv( opid1 ,id1 ,na ,idp1 ,n1 ,s1 )
endif ;
```

Figure 7: Transformation Function for move-mkfile

2. Which entry to rename ? In order to converge, we must make the same deterministic choice on all sites. We choose to rename the file with the least id. Thus, if we integrate two concurrent operations creating the same file in the same directory, there are two cases: (a) we are transforming the operation with the least id (cf line 5 in figure 5). In this case, we just rename the file with $max(s_1) \odot id_1$. (b) we are transforming the file with greatest id (cf line 8 in figure 5). In this case, we rename the least, and create the greatest without modifications. By this way, the transformation function returns a sequence of two operations. ⊎ is the sequence constructor.

Figures 6 and 7 describe transformation function for mkfile-move and move-mkfile. We use these functions in section 7.

6. TRANSFORMATION FUNCTIONS FOR TEXT FILES

On a text file, we define the following operations:

1. ab(id_1,s_1,os_1,v_1). Adds a block of text v_1 to the file identified by id_1 at the insert point s_1. os_1 parameter is used to solve some false ambiguous conflicting situation [22]. This parameter remembers the original insertion point. When an operation *ab* is created, $os_1 = s_1$. If the operation is transformed, os_1 remains identical. In order to simplify the transformation functions, we use l_1 to represent the number of lines of block v_1. id_1 and s_1 have to exist.

2. db(id_1,s_1,ov_1). Deletes the block of text ov_1 from the file identified by id_1 at the delete point s_1. l_1 is used to represent the number of lines of block ov_1. id_1 and s_1 have to exist.

Figure 8 presents the transformation function for addblock-addblock. As for the file system, our general strategy for writing transformation function is to converge towards a state where conflicts are represented. In case of conflict, we will produce a block of text containing the effects of both operations like rcsmerge[26]. Conflicts occur when the effects of two concurrent operations are overlapping. For example, a *db* operation can delete lines added concurrently by a *ab* operation. The overlapping between these two concurrent operations can be partial or complete.

For addblock-addblock, a conflict occurs only if both operations insert at the same line two different texts. In this case, we delete the block previously inserted and insert a new block containing both texts. If there is no overlapping, we just manage the insert point. The same strategy is applied for addblock-delblock in figure 9 and for delblock-addblock in figure 10.

For space reasons, we do not present transformation functions for delblock-delblock and move-move. All others transformation

```
T( ab(id1 ,s1 ,os1 ,v1 )  ,
   ab(id2 ,s2 ,os2 ,v2 ))  :-
 if  (id1 !=id2 )  then
  return  ab(id1 ,s1 ,os1 ,v1 )
  else
   if  (s1<s2 )  then
    return  ab(id1 ,s1 ,os1 ,v1 )
   elseif  (s1  >  s2 )  then
    return  ab(id1 ,s1+l2 ,os1 ,v1 )
   else
   if  (os1  <  os2 )  then
    return  ab(id1 ,s1 ,os1 ,v1 )
   elseif  (os1  >  os2 )  then
    return  ab(id1 ,s1 + l2 ,os1 ,v1 )
   else
    if  (v1  ==  v2 )  then
     return  Id(ab(id1 ,s1 ,os1 ,v1 )
    else
     return  db(id2 ,s2 ,l2 ,v2 )
               ⊎  ab(id1 ,s1 ,os1 ,v1 ⊙ v2 )
    endif
   endif
  endif
endif
```

Figure 8: Transformation Function for addblock-addblock

```
T( ab(id1 ,s1 ,os1 ,v1 ) ,  db(id2 ,s2 ,ov2 )) :-
if  (id1  !=  id2 )  then
 return  ab(id1 ,s1 ,os1 ,v1 )
else
 if  (s1 < s2 )  then
  return  ab(id1 ,s1 ,os1 ,v1 )
 elseif  (s1 > s2 + l2 - 1)  then
  return  ab(id1 ,s1 - l2 ,os1 ,v1 )
 else
  return  ab(id1 ,s2 ,s2 ,ov2 ⊙ v1 )
 endif
endif
```

Figure 9: Transformation Function for addblock-delblock

functions $T(op_1, op_2)$ (for example $T(op_1 = move, op_2 = addblock)$) return op_1.

7. EXAMPLE

We suppose three users working concurrently on the same initial state (cf. figure 12(a)): mf(1,0,a,{a}), ab (1,0,0,{" gaspard "," melchior "," balthazar "}).

On this state, users produce concurrent operations described in figure 11. After the synchronization s_5, all users observe the same state (cf. figure 12(b)).

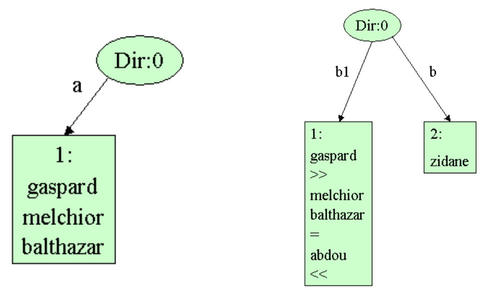
This scenario illustrates how the synchronizer handles conflicts between op_1 and op_3 and between op_2 and op_5. We describe now the effects of each synchronize command.

s_1 At this point, there is no concurrent operation. Merge is straightforward. op_1 and op_2 are just sent to the other sites.

s_2 The synchronizer merges the sequence $op_1; op_2$ to the local log $op_3; op_4$. If we execute the integration algorithm described in figure 2, we obtain the following calls to transformation functions.

```
T( db(id1 ,s1 ,ov1 ) ,  ab(id2 ,s2 ,os2 ,v2 )) :-
if  (id1  !=  id2 )  then
 return  db(id1 ,s1 ,ov1 )
else
 if  (s1  >  s2 )  then
  return  db(id1 ,s1 + l2 ,ov1 )
 elseif  (s1 + l1 - 1 < s2 )  then
  return  db(id1 ,s1 ,ov1 )
 else
  return  db(id2 ,s2 ,v2 )
          ⊎  db(id1 ,s1 ,ov1 )
          ⊎  ab(id1 ,s1 ,s1 ,ov1 ⊙ v2 )
 endif
endif
```

Figure 10: Transformation Function for delblock-addblock



(a) Initial State

(b) State after s_5

Figure 12: Initial and Final State of the Scenario

Operations	*Result*
$op_1^1 = T(op_1, op_3)$	$mv(0, 1, a, 0, b1, \{b, b1\})$
$op_3^1 = T(op_3, op_1)$	$mv(0, 1, b, 0, b1, \{b1\})$ $\uplus mf(2, 0, b, \{b, b1\})$
$op_1^2 = T(op'_1, op_4)$	$mv(0, 1, a, 0, b1, \{b, b1\})$
$op_4^1 = T(op_4, op'_1)$	$ab(2, 0, 0, \{"zidane"\})$
$op_2^1 = T(op_2, op'_3)$	$db(1, 2, \{"melchior", "balthazar"\})$
$op_3^2 = T(op'_3, op_2)$	$mv(0, 1, b, 0, b1, \{b1\})$ $\uplus mf(2, 0, b, \{b, b1\})$
$op_2^2 = T(op'_2, op'_4)$	$db(1, 2, \{"melchior", "balthazar"\})$
$op_4^2 = T(op'_4, op'_2)$	$ab(2, 0, 0, \{"zidane"\})$

For s_3 **to** s_5 we re-execute the integration algorithm as for s_2. After s_5, each site has executed a sequence of operations equivalent to:

```
mf(1,0,a,{a}),
ab(1,0,0,{"gaspard","melchior","balthazar"}),
op1=mv(0,1,a,0,b,{b}),
op2=db(1,2,{"melchior","balthazar"}),
op3^2=mv(0,1,b,0,b1,{b1}) ⊎ mf(2,0,b,{b,b1}),
op4^2=ab(2,0,0,{"zidane"}),
op5^4=ab(1,2,2,{">>","melchior",
              "balthazar","=","abdou","<<"});
```

LibreSource is a platform for hosting virtual teams. It provides a web service of synchronization based on the transformational approach [1]. Users can register and create channels for synchronizing

[1]You can try the LibreSource prototype online at http://woinville.loria.fr/ls. It requires to have the jdk1.4.+ installed on your computer.

u_1	u_2	u_3
$op_1 = mv(0, 1, a, 0, b, \{b\})$	$op_3 = mf(2, 0, b, \{a, b\})$	$op_5 = ab(1, 3, 3, \{"abdou"\})$
$op_2 = db(1, 2, \{"melchior", "balthazar"\})$	$op_4 = ab(2, 0, 0, \{"zidane"\})$	
$s_1 = synchronize$		
	$s_2 = synchronize$	
		$s_3 = synchronize$
	$s_4 = synchronize$	
$s_5 = synchronize$		

Figure 11: Integration Scenario

data. A channel is a queue of timestamped operations. Once a channel is created, users can create replicas on their local disks and start synchronizing.

We use diff algorithms [3, 14] to detect changes since last synchronization. Diff algorithm generates the local logs required by the integration algorithm.

We have used *LibreSource* for several months now, and we observed that the number of operations is growing fast. On some channels we have more than 4000 operations. An algorithm for compressing log of operations using the transformational approach has been developed in [19]. We plan to implement it in order to compress channel queues.

8. RELATED WORKS

Many tools exist in different research areas dealing with synchronizations. We compare our work with file synchronizers, PDAs synchronizers, configuration management tools, synchronization issues in distributed systems and replication in database systems.

File Synchronizer The overall goal of a file synchronizer is to detect conflicting updates and propagate non conflicting ones. To achieve this goal, the semantic of the file system primitives must be well defined as in Unison [1]. However, this approach presents several drawbacks: (a) the approach is restricted to a file system. (b) Synchronization is often limited to two replicas. (c) Reconciliation is coarse grained. It does not attempt to synchronize file contents. (d) A general correctness criterion is not defined. (e) The system interacts with user each time a conflict is detected. If there are 100 conflicts, the system will interact 100 times with the user. If we just make the comparison between S_5 and this kind of synchronizers, S_5 handles n replicas, ensures convergence, causality and intention preservation, synchronizes files contents, resolves conflicts automatically in all cases.

PDA synchronizer ActiveSync, HotSync, I-Sync are now largely used to synchronize data between desktop computers and PDAs. These synchronizers allow to synchronize several kind of data like address books, calendars, tasks, notes , bookmarks, files and so on. However, this approach is an extension of the file synchronizer approach: it detects conflicting updates and propagates non conflicting ones. So we have exactly the same problems: no correctness criteria, problems with conflict resolution ...
The genericity of transformational approach makes it easy to write such synchronizers. We can define a type calendar with three operations: AddRendezVous, RemoveRendezVous and UpdateRendezVous. Then we define all transformation functions and make the proof of the condition C_1. The result is a safe synchronizer, ensuring convergence, causality and intention preservation.

CM and Merge Tools In Configuration Management Environments [2, 27, 4, 7] users can work in parallel, produce data divergence and reconciliate later using the copy-modify-merge paradigm. If we look closer on how things are done, we observe that reconciliation is done by tight cooperation between version manager and merge tools. (a) When a reconciliation is required (i.e. often when a user updates his workspace), version managers provides required version to merge tools [14]. Merge is done locally, in the workspace of the user. (b) Merge tools extract from different versions, concurrent logs of operations using Diff algorithms [3]. Of course, diff algorithms are specific to data types. (c) Finally, concurrent operations are merged using ad-hoc algorithm specific to data types.
The transformational model is more general, more uniform and safer than this model. In this approach, each merge tool has its own merge algorithm. One tool merges two divergent file systems, another tool merges two divergent text files, another one merges two divergent XML files. Maybe, they are not consistent together, they do not apply the same strategy. For example, with CVS, compensation is used by the text file merge tool and not by the file system merge tool.
In the transformational approach, the merge algorithm is shared by all transformation functions. It preserves Convergence, Causality and Intention (CCI) if underlying transformation functions ensure condition C_1. By this way, we can extend the synchronizer by adding new transformation functions without violating CCI properties.

Distributed systems Maintaining consistency of shared data is a big issue in distributed systems. Coda[13], Bayou[15], Ficus[17] allow users to work disconnected and use reconciliation procedures when people reconnect.
Bayou[15] first used an epidemic algorithm to propagate changes between weakly consistent replicas. When a conflict is detected, merge procedures associated with operations are executed. If the merge procedure cannot find a solution, conflict resolution is delegated to users. Bayou use a total update ordering. Other systems [18] use a partial update ordering and then take advantages of update commutativity. Causality is used to determine the partial ordering.
Distributed systems and transformational approach are similar in many points: both approaches detect conflicts, merge procedures and transformation functions looks identical, commutativity and condition C_1 are quite similar and causality are used in both approaches. However, the transformational approach allows to *transform* operations. C_1 is some sort of "transformational commutativity". It allows to compute more complex state of convergence. Unlike merge procedures, transformation functions ensure convergence in all cases.
The IceCube [12] is a generic approach for reconciliating divergent data. IceCube does not define a general correctness criterion for synchronization but uses semantic constraints that the reconciliation algorithm has to preserve. IceCube considers two kind of constraints: (a) Static constraints can be evaluated without using the state of replica. Commutativity of operations can be expressed as a static constraint. (b) Dynamic constraints can refer to the state of replicas.
Basically, IceCube explores all possible combinations of concurrent actions. First, IceCube rejects all combinations violating static constraints. For the others, IceCube simulates integrations on repli-

cas and reject combinations violating dynamic constraints. Resulting combinations are ranked and proposed to user.

This approach is interesting because, IceCube is looking for the combinations of concurrent operations that minimize conflicts of reconciliation. Maybe, on this point, transformational approach will not find the optimal reconciliation. On the other hand, IceCube has some intrinsic drawbacks: (a) Combinatorial explosion can occur during the first stage of reconciliation, even if static constraints restrict the number of possible schedules. (b) Constraints are specific to applications and have to be defined. (c) IceCube is interactive, (d) IceCube does not transform operations. What happens if there are just two concurrent operations *mkfile*("/*a*") and *mkdir*("/*a*"). All possible schedules are bad. In this situation, IceCube will just ask users what it has to do as a classical file synchronizer.

Database Systems Replication and database consistency has been investigated extensively [8, 16]. Replication conflicts can occur in a replication environment that permits concurrent updates to the same data at multiple sites. If two transactions working on two different replicas, update the same row at the same time, a conflict can occur.

Oracle[5] provides built-in resolution methods for resolving update conflicts. The "latest timestamp" value resolves a conflict based most recent update. the Additive method adds the difference of two conflicting "update value" operations to the current value. The "overwrite" method replaces the current value with the new value. Users can define their own conflict resolution methods. If convergence cannot be achieved, then a notification is sent to the administrator. Some built-in resolution methods seem to preserve convergence but not for any kinds of conflicts (uniqueness and delete/update) and not for any configuration of replicas. Transformational approach is more general than replicas management in database systems. We can implement built-in or user defined resolution methods of Oracle as transformation functions and prove formally the convergence.

9. CONCLUSION AND PERSPECTIVES

Transformational Approach can be considered as a theoretical foundation for synchronizing data. We propose a generic and safe synchronizer ensuring convergence, causality and intention preservation. It relies on underlying specific transformation functions verifying condition C_1. We write proved *correct* transformation functions for a file system, text files, XML files [9], String.... We validate our approach with the *LibreSource* prototype.

We have several research directions:

1. We plan to develop transformation functions for handling more shared data types like database primitives, DTDs in XML...

2. We are currently building network of synchronizations. It implies that a single replica can be synchronized with several timestampers. By this way, we can develop the dataflow part of a software process.

3. A lot of work has been done for undoing operations in real-time groupware[23]. It requires that at least transformation functions verifies condition C_2. We have started to improve our transformation functions to handle the undo operation. This approach can be used as an alternative to compensation.

4. We are currently modifying the SPIKE theorem prover in order to build an integrated development environment for transformation functions. Within this environment a user enters functions like in this paper and calls the theorem prover like a compiler. If there are errors, the environment gives counter-examples immediately. We believe that this kind of environment can greatly improve the process of production of transformation functions.

10. REFERENCES

[1] S. Balasubramaniam and B. C. Pierce. What is a file synchronizer? In *Mobile Computing and Networking*, pages 98–108, 1998.

[2] B. Berliner. CVS II : Parallelizing Software Development. In *Proceedings of USENIX*, Washigton D. C., 1990.

[3] S. S. Chawathe and H. Garcia-Molina. Meaningful change detection in structured data. In *Proceedings of the 1997 ACM SIGMOD International Conference on Management of Data*, pages 26–37. ACM Press, 1997.

[4] R. Conradi and B. Westfechtel. Version models for software configuration management. *ACM Computing Surveys (CSUR)*, 30(2):232–282, 1998.

[5] D. Daniels, L. B. Doo, A. Downing, C. Elsbernd, G. Hallmark, S. Jain, B. Jenkins, P. Lim, G. Smith, B. Souder, and J. Stamos. Oracle's symmetric replication technology and implications for application design. In *Proceedings of the 1994 ACM SIGMOD international conference on Management of data*, page 467. ACM Press, 1994.

[6] C. A. Ellis and S. J. Gibbs. Concurrency control in groupware systems. In *SIGMOD Conference*, volume 18, pages 399–407, 1989.

[7] J. Estublier. Software configuration management: a roadmap. In *ICSE - Future of SE Track*, pages 279–289, 2000.

[8] J. Gray, P. Helland, P. O'Neil, and D. Shasha. The dangers of replication and a solution. In *Proceedings of the 1996 ACM SIGMOD International Conference on Management of Data*, pages 173–182. ACM Press, 1996.

[9] A. Imine, P. Molli, G. Oster, and M. Rusinowitch. Development of transformation functions assisted by a theorem prover. In *Fourth International Workshop on Collaborative Editing*, New Orleans, Louisiana, USA, November 2002.

[10] A. Imine, P. Molli, G. Oster, and M. Rusinowitch. Proving correctness of transformation functions in real-time groupware. In *Proceedings of the 8th European Conference on Computer-Supported Cooperative Work*, Helsinki, Finland, September 2003.

[11] A. Imine, P. Molli, G. Oster, and P. Urso. Vote: Group editors analyzing tool. In *International Workshop on First-Order Theorem Proving*, June 2003.

[12] A.-M. Kermarrec, A. Rowstron, M. Shapiro, and P. Druschel. The IceCube approach to the reconciliation of divergent replicas. In *Proceedings of the Twentieth ACM Symposium on Principles of Distributed Computing (PODC)*, Newport RI, USA, August 2001.

[13] P. Kumar and M. Satyanarayanan. Flexible and safe resolution of file conflicts. In *USENIX Winter*, pages 95–106, 1995.

[14] J. P. Munson and P. Dewan. A flexible object merging framework. In *Proceedings of ACM CSCW'94 Conference on Computer-Supported Cooperative Work*, Technologies for Sharing I, pages 231–242, 1994.

[15] K. Petersen, M. J. Spreitzer, D. B. Terry, M. M. Theimer, and A. J. Demers. Flexible update propagation for weakly consistent replication. In *Proceedings of the 16th ACM Symposium on Operating SystemsPrinciples (SOSP-16), Saint Malo, France*, 1997.

[16] M. Rabinovich, N. H. Gehani, and A. Kononov. Scalable update propagation in epidemic replicated databases. In *Extending Database Technology*, pages 207–222, 1996.

[17] P. L. Reiher, J. S. Heidemann, D. Ratner, G. Skinner, and G. J. Popek. Resolving file conflicts in the ficus file system. In *USENIX Summer*, pages 183–195, 1994.

[18] Y. Saito and H. M. Levy. Optimistic replication for internet data services. In *International Symposium on Distributed Computing*, pages 297–314, 2000.

[19] H. Shen and C. Sun. A log compression algorithm for operation-based version control systems. In *Proceedings of IEEE 26th Annual International Computer Software and Application Conference (COMPSAC 2002)*, Oxford, England, august 2002.

[20] S. Stratulat. A general framework to build contextual cover set induction provers. *Journal of Symbolic Computation*, 32(4):403–445, 2001.

[21] M. Suleiman, M. Cart, and J. Ferrié. Concurrent operations in a distributed and mobile collaborative environment. In *Proceedings of the Fourteenth International Conference on Data Engineering (ICDE'98)*, pages 36–45, Orlando, Florida, USA, February 1998. IEEE Computer Society.

[22] M. Suleiman, M. Cart, and J. Ferrié. Serialization of concurrent operations in a distributed collaborative environment. In *Proceedings of the International ACM SIGGROUP Conference on Supporting Group Work : The Integration Challenge (GROUP'97)*, pages 435–445. ACM Press, November 1997.

[23] C. Sun. Undo as concurrent inverse in group editors. *ACM Transactions on Computer-Human Interaction (TOCHI)*, 9(4):309–361, December 2002.

[24] C. Sun and D. Chen. Consistency maintenance in real-time collaborative graphics editing systems. *ACM Transactions on Computer-Human Interaction (TOCHI)*, 9(1):1–41, March 2002.

[25] C. Sun, X. Jia, Y. Zhang, Y. Yang, and D. Chen. Achieving convergence, causality-preservation and intention-preservation in real-time cooperative editing systems. *ACM Transactions on Computer-Human Interaction (TOCHI)*, 5(1):63–108, March 1998.

[26] W. F. Tichy. RCS – A system for version control. *Software–Practice and Experience*, 15(7):637–654, July 1985.

[27] A. van der Hoek. International workshop on software configuration management (scm-10): new practices, new challenges, and new boundaries. *ACM SIGSOFT Software Engineering Notes*, 26(6):57–58, 2001.

[28] N. Vidot, M. Cart, J. Ferrié, and M. Suleiman. Copies convergence in a distributed real-time collaborative environment. In *Proceedings of the ACM Conference on Computer Supported Cooperative Work (CSCW'00)*, Philadelphia, Pennsylvania, USA, December 2000.