Proving Correctness of Transformation Functions in Real-Time Groupware

Abdessamad Imine, Pascal Molli, Gérald Oster
and Michaël Rusinowitch
ECOO and CASSIS Teams - LORIA France
{*imine,molli,oster,rusi*}*@loria.fr*

Abstract. Operational transformation is an approach which allows to build real-time groupware tools. This approach requires correct transformation functions. Proving the correction of these transformation functions is very complex and error prone. In this paper, we show how a theorem prover can address this serious bottleneck. To validate our approach, we have verified the correctness of state-of-art transformation functions defined on Strings with surprising results. Counter-examples provided by the theorem prover have helped us to define new correct transformation functions for Strings.

Introduction

Real-time groupware systems allow a group of users to manipulate the same object (*i.e.* a text, an image, a graphic, etc.) at the same time from physically dispersed sites that are interconnected by a supposed reliable network. In order to achieve good responsiveness and friendly collaboration, the shared objects are *replicated* at the local memory of each participating user. One of the most significant issues in building real-time groupware systems with replicated architecture is *consistency maintenance* of shared objects (Sun, Jia, Zhang, Yang & Chen 1998).

Operational transformation is an approach (Ellis & Gibbs 1989)(Sun & Chen 2002) which allows to build real-time groupware like shared editors. Algorithms like aDOPTed (Ressel, Nitsche-Ruhland & Gunzenhauser 1996), GOTO (Sun et al. 1998), SOCT 2,3,4 (Suleiman, Cart & Ferrié 1998)(Vidot, Cart, Ferrié &

Suleiman 2000) are used to maintain the consistency of shared data. However these algorithms rely on the definition of transformation functions. If these functions are not correct then these algorithms cannot ensure the consistency of shared data.

Proving the correctness of transformation functions even on a simple typed object like a String is a complex task. If we have more operations on more complex typed objects, the proof is almost impossible without a computer. This is a serious bottleneck for building more complex real-time groupware software.

We propose to assist development of transformation functions with SPIKE , an automated theorem prover which is suitable for reasoning about functions defined by conditional rewrite rules (Stratulat 2001)(Imine, Molli, Oster & Rusinowitch 2002). This approach requires specifying the transformation functions in first order logic. Then, SPIKE automatically determines the correctness of transformation functions. If correctness is violated, SPIKE returns counter-examples. Since the proofs are automatic, we can handle more (even complex) operations and develop quickly correct transformation functions.

This paper is organized as follows. The second section briefly presents the transformational approach. In the third section, we give the surprising results we have obtained when verifying the correctness of existing transformation functions about Strings. Thanks to counter-examples provided by SPIKE we define new *correct* transformation functions for Strings. The fourth section briefly overviews the features of SPIKE and describes how to specify transformation functions in this prover. Finally, we conclude with some remarks and with some perspectives for future works.

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 the other sites to be executed again. So every operation is processed in four steps:

(1) generation on one site,

(2) broadcast to other sites,

(3) reception by other sites,

(4) execution on other sites.

The execution context of a received operation op_i may be different from the generation context of op_i. In this case, the integration of op_i by other sites may lead to inconsistencies between replicates. We illustrate this behavior in Figure 1. There are two sites 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 the string is 1 (and not 0). The users 1 and 2 generate two concurrent operations: $op_1 = Ins(2, f)$ and $op_2 = Ins(6, s)$ respectively. 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 sites 1 and 2.

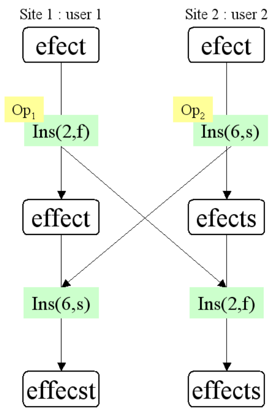

Figure 1. Incorrect integration.

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 and returns op'_1 which 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 2. 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:

$$T((\overbrace{Ins(6,s)}^{op_2}, \overbrace{Ins(2,f)}^{op_1}) = \overbrace{Ins(7,s)}^{op'_2}$$

The insertion position of op_2 is incremented because op_1 has inserted the character "`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:

$$T(\overbrace{Ins(2,f)}^{op_1}, \overbrace{Ins(6,s)}^{op_2}) = \overbrace{Ins(2,f)}^{op'_1}$$

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($p_1$,$c_1$),Ins($p_2$,$c_2$)) :−
  **if** $p_1 < p_2$ **return** Ins($p_1$, $c_1$)
  **else** **return** Ins($p_1 + 1$, $c_1$)
```

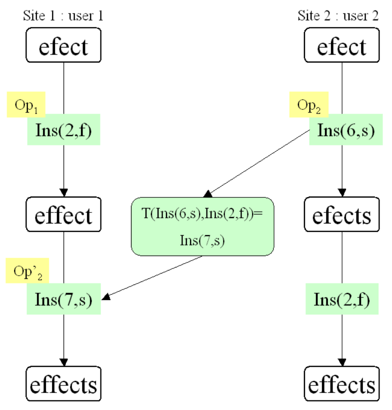

Figure 2. Integration with transformation.

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

A lot of work has been devoted to defining a theoretical model (Sun et al. 1998)(Suleiman et al. 1998)(Sun & Chen 2002, Sun 2002). Basically, transformational approach defines a new consistency criteria for replicates. To be correct, an 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 (Sun et al. 1998)(Suleiman et al. 1998) that the underlying transformation functions must satisfy two conditions:

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

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

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

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

It is important to note that although many algorithms have been designed, only few sets of transformation functions have been delivered to the community (Palmer & Cormack 1998)(Davis, Sun & Lu 2002)(Molli, Skaf-Molli, Oster & Jourdain 2002). Proving C_1 and C_2 on transformation functions is very hard and error prone even on a simple string object. For example, there are 123 different cases to explore when trying to prove C_2 on a String object. Each time the specification of transformation functions is changed, it is necessary to redo the proof.

Without a correct set of transformation functions, the integration algorithm cannot ensure consistency and the resulting groupware tools would not be reliable. Consequently, to be able to develop the transformational approach with simple or more complex objects, proving conditions on transformation functions must be automatic.

Verifying Transformation Functions

In this section, we return to existing transformation functions defined on String objects, where a String is considered as an array of characters starting at range 1 (and not 0). We have formalized them using SPIKE and checked their correctness. We show in the fourth section how to specify these functions in SPIKE . Two operations are defined on String:

- $Ins(p, c)$: Inserts a character c at position p.
- $Del(p)$: Deletes the character located at position p.

Ellis's Transformation Functions

Ellis and Gibbs (Ellis & Gibbs 1989) are the pioneers of the operational transformation. They have defined the transformation functions shown below. Operations Ins and Del are extended with a new parameter pr representing the priority. Priorities are based on the site identifier where operations have been generated [1]. $Id()$ is the Identity operation, which does not affect state.

```
$T_{ii}$(Ins($p_1$,$c_1$,$pr_1$), Ins($p_2$,$c_2$,$pr_2$)) :−
  **if** $p_1 < p_2$ **return** Ins($p_1$,$c_1$,$pr_1$)
  **else if** $p_1 > p_2$ **return** Ins($p_1 + 1$, $c_1$,$pr_1$)
          **else if** $c_1 == c_2$ **return** **Id**()
                  **else if** $pr_1 > pr_2$ **return** Ins($p_1 + 1$,$c_1$,$pr_1$)
                          **else return** Ins($p_1$,$c_1$,$pr_1$)
```

[1] This priority becomes even more complex since it is also used like a list.

```
$T_{id}$(Ins($p_1$,$c_1$,$pr_1$), Del($p_2$,$pr_2$)) :−
  **if** $p_1 < p_2$ **return** Ins($p_1$,$c_1$,$pr_1$)
  **else** **return** Ins($p_1 - 1$,$c_1$,$pr_1$)

$T_{di}$(Del($p_1$,$pr_1$),Ins($p_2$,$c_2$,$pr_2$)) :−
  **if** $p_1 < p_2$ **return** Del($p_1$,$pr_1$)
  **else** **return** Del($p_1 + 1$,$pr_1$)

$T_{dd}$(Del($p_1$,$pr_1$),Del($p_2$,$pr_2$)) :−
  **if** $p_1 < p_2$ **return** Del($p_1$,$pr_1$)
  **else** **if** $p_1 > p_2$ **return** Del($p_1 - 1$,$pr_1$)
         **else** **return** **Id**()
```

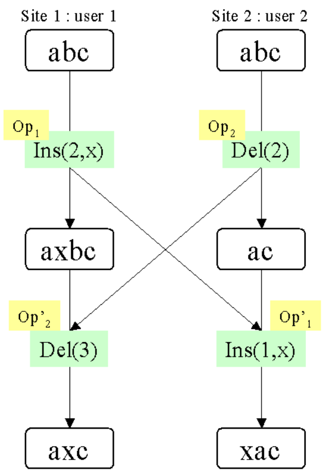

Figure 3. Counter-example violating condition C_1.

It is well known that these transformation functions are not correct (Sun et al. 1998)(Suleiman et al. 1998)(Ressel et al. 1996). Nevertheless, they were submitted to SPIKE in order to verify if the problem can be automatically detected. SPIKE found the counter-example depicted in figure 3 in a few seconds. SPIKE detected that condition C_1 is violated.

The counter-example is simple:

(1) $user_1$ inserts x in position 2 (op_1) while $user_2$ concurrently deletes the character at the same position (op_2).

(2) When op_2 is received by site 1, op_2 must be transformed according to op_1. So $T_{di}(Del(2), Ins(2,x))$ is called and $Del(3)$ is returned.

(3) In the same way, op_1 is received on site 2 and must be transformed according to op_2. $T(Ins(2,x), Del(2))$ is called and return $Ins(1,x)$. Condition C_1 is violated. Accordingly, the final results on both sites are different.

The error comes from the definition of T_{id}. The condition $p_1 < p_2$ should be rewritten $p_1 \leq p_2$. But if we re-submit this version to the theorem prover, it is still not correct with the counter-example detailed in the next section.

This gives a typical example of working with SPIKE . In some way, we use this prover like a compiler. We express transformation functions using the SPIKE syntax and SPIKE checks conditions in few seconds or few minutes depending of the number of different cases induced by the specification.

Ressel's Transformation Functions

Matthias Ressel (Ressel et al. 1996) have modified Ellis's transformation functions in order to satisfy C_1 and C_2. Priorities are replaced by the parameter $u_i \in 1, 2, ..., n$. This parameter represents the user who generates the operation. M. Ressel wrote that T_{id} and T_{di} are exactly the same as those of Ellis. In this case, the set of transformation functions does not satisfy C_1 as in the counter-example of Figure 3. We assume M. Ressel refers to a corrected version of Ellis where T_{id} is redefined with $p_1 \leq p_2$. On the other hand, Ressel modified the definition of T_{ii} as follows: when two insert operations have the same position p, the character produced by the site with the lower range is inserted at p.

```
$T_{ii}$(Ins($p_1$,$c_1$,$u_1$),Ins($p_2$,$c_2$,$u_2$)) :−
  if $p_1$<$p_2$ or ($p_1$=$p_2$ and $u_1$<$u_2$) return Ins($p_1$,$c_1$,$u_1$)
  else return Ins($p_1$+1,$c_1$,$u_1$)

$T_{dd}$(Del($p_1$,$u_1$),Del($p_2$,$u_2$)) :−
  if $p_1$<$p_2$ return Del($p_1$,$u_1$)
  else if $p_1$>$p_2$ return Del($p_1 - 1$,$u_1$)
       else return Id()

$T_{id}$(Ins($p_1$,$c_1$,$u_1$), Del($p_2$,$u_2$)) :−
  if $p_1 \leq p_2$ return Ins($p_1$,$c_1$,$u_1$)
  else return Ins($p_1 - 1$,$c_1$,$u_1$)

$T_{di}$(Del($p_1$,$u_1$),Ins($p_2$,$c_2$,$u_2$)) :−
  if $p_1 < p_2$ return Del($p_1$,$u_1$)
  else return Del($p_1 + 1$,$u_1$)
```

This strategy seems to work but SPIKE found the counter-example given in figure 4. This counter-example requires three users where operations $op_1 = Ins(2, x)$, $op_2 = Del(2)$ and $op_3 = Ins(3, y)$ are concurrent:

(1) First of all, op_2 is integrated on $user_3$'s site. So, we apply $T(Del(2), Ins(3, y))$ which returns $op'_2 = Del(2)$.

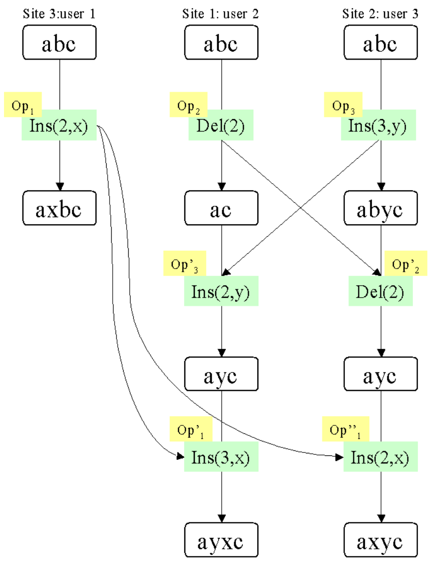

Figure 4. Counter example violating condition C_2.

(2) When integrating op_3 on site 2, we apply $T(Ins(3, y), Del(2))$ which returns $op'_3 = Ins(2, y)$.

(3) Next, op_1 is integrated on site 2 as follows: op_1 must be transformed according to op_2 and the result of this transformation must be transformed according to op'_3. $T(op_1 = Ins(2, x), op_2 = Del(2))$ which returns a new operation $Ins(2, x)$. This operation must be transformed again according to op'_3:

$$T(Ins(2, x), \overbrace{Ins(2, y)}^{op'_3}) = \overbrace{Ins(3, x)}^{op'_1}$$

(4) op_1 is integrating on site 3 in the same way. So we calculate the result of:

$$\overbrace{Ins(2, x)}^{op''_1} = T(T(Ins(2, x), \overbrace{Ins(3, y)}^{op_3}), \overbrace{Del(2)}^{op'_2})$$

Copies on site 2 and 3 do not converge. Consequently, transformation functions of Ressel do not verify C_2.

Sun's Transformation Functions

Chengzheng Sun (Sun et al. 1998) has derived the set of transformation functions below. The signature of operations Ins and Del are slightly different. Indeed, Ins may be used to insert either a character or a string at position p.

```
T(Ins($p_1$,$s_1$,$l_1$),Ins($p_2$,$s_2$,$l_2$)) :−
  **if** $p_1 < p_2$ **return** Ins($p_1$, $s_1$, $l_1$)
  **else return** Ins$(p_1 + l_2, s_1, l_1)$

T(Ins$(p_1$,$s_1$,$l_1)$,Del$(p_2$,$l_2$)) :−
  **if** $p_1 \leq p_2$ **return** Ins($p_1$,$s_1$,$l_1$)
  **else if** $p_1 > (p_2 + l_2)$ **return** Ins($p_1 - l_2$,$s_1$,$l_1$)
        **else return** Ins$(p_2$,$s_1$,$l_1)$

T(Del($p_1$,$l_1$),Ins($p_2$,$s_2$,$l_2$)) :−
  **if** $p_2 \geq p_1$ **return** Del($p_1$,$l_1$)
  **else if** $p_1 \geq p_2$ **return** Del($p_1 + l_2$,$l_1$)
        **else return** [Del($p_1$,$p_2 - p_1$);Del($p_2 + l_2$,$l_1 - (p_2 - p_1)$)]

T(Del($p_1$,$l_1$),Del($p_2$,$l_2$)) :−
  **if** $p_2 \geq p_1 + l_1$ **return** Del($p_1$,$l_1$)
  **else if** $p_1 \geq p_2 + l_2$ **return** Del($p_1 - l_2$,$l_1$)
  **else if** $p_2 \leq p_1$ and $p_1 + l_1 \leq p_2 + l_2$ **return** Del($p_1$,0)
  **else if** $p_2 \leq p_1$ and $p_1 + l_1 > p_2 + l_2$ **return** Del($p_2$, $(p_1 + l_1) - (p_2 + l_2)$)
  **else if** $p_2 > p_1$ and $p_2 + l_2 \geq p_1 + l_1$ **return** Del($p_1$, $p_2 - p_1$)
  **else return** Del($p_1$, $l_1 - l_2$)
```

For a better comparison with others set of transformation functions, we have rewritten Sun's transformations functions for characters. The result is given below.

```
T(Ins$(p_1$,$c_1)$,Ins($p_2$,$c_2$)) :−
   **if** $p_1 < p_2$ **return** Ins($p_1$, $c_1$)
   **else return** Ins$(p_1 + 1, c_1)$

T(Ins$(p_1$,$c_1)$,Del($p_2$)) :−
  **if** $p_1 \leq p_2$ **return** Ins($p_1$,$c_1$)
  **else return** Ins$(p_1 - 1$,$c_1)$

T(Del($p_1$),Ins($p_2$,$c_2$)) :−
  **if** $p_1 < p_2$ **return** Del($p_1$)
  **else return** Del($p_1 + 1$)

T(Del($p_1$),Del($p_2$)) :−
  **if** $p_1 < p_2$ **return** Del($p_1$)
  **else if** $p_1 > p_2$ **return** Del($p_1 - 1$)
        **else return Id**()
```

SPIKE has found that this set of transformation functions violates C_2 with the counter-example presented in Figure 5. Let us consider the following three concurrent operations $op_1 = Ins(2, y)$, $op_2 = Del(2)$ and $op_3 = Ins(3, y)$.

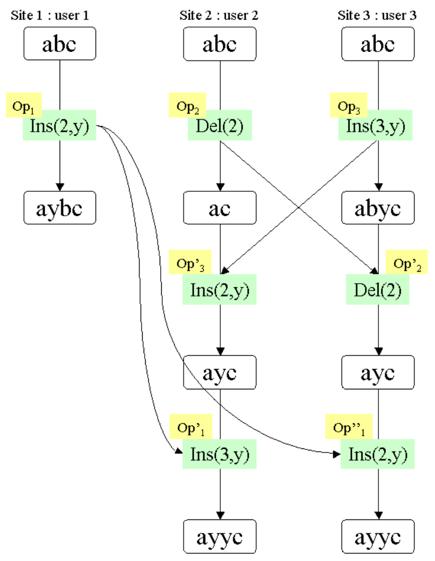

Figure 5. Counter example scenario which violates condition C_2.

(1) Site 3 integrates op_2:

$$\overbrace{Del(2)}^{op'_2} = T(\overbrace{Del(2)}^{op_2}, \overbrace{Ins(3,y)}^{op_3})$$

(2) Then, Site 2 integrates op_3:

$$\overbrace{Ins(2,y)}^{op'_3} = T(\overbrace{Ins(3,y)}^{op_3}, \overbrace{Del(2)}^{op_2})$$

(3) Next, Site 2 integrates op_1:

$$\overbrace{Ins(3,y)}^{op'_1} = T(T(\overbrace{Ins(2,y)}^{op_1}, \overbrace{Del(2)}^{op_2}), \overbrace{Ins(2,y)}^{op'_3})$$

(4) Finally, Site 3 integrates op_1:

$$\overbrace{Ins(2,y)}^{op''_1} = T(T(\overbrace{Ins(2,y)}^{op_1}, \overbrace{Ins(3,y)}^{op_3}), \overbrace{Del(2)}^{op'_2})$$

The final result is the same as in site 2 and 3, but C_2 is not satisfied. In fact:

$$\overbrace{Ins(3,y)}^{op'_1} = T(op_1, op_2 \circ T(op_3, op_2)) \neq \overbrace{Ins(2,y)}^{op''_1} = T(op_1, op_3 \circ T(op_2, op_3))$$

Note that if we use $op_1 = Ins(2,x)$ instead of $op_1 = Ins(2,y)$, then the result will diverge on site 2 and 3 as in the counter-example of Figure 4.

Suleiman’s Transformation Functions

Suleiman (Suleiman, Cart & Ferrié 1997) proposes a different set of transformation functions. He adds two new parameters to function Ins which is defined as follows: $Ins(p_i, c_i, b_i, a_i)$ where b_i (a_i respectively) is the set of concurrent operations to this insertion operation and that have deleted a character before (after respectively) the position p_i. Hence, for two concurrent operations $Ins(p_1, c_1, b_1, a_1)$ and $Ins(p_2, c_2, b_2, a_2)$ defined on the same state, the following cases are given:

- if $(b_1 \cap a_2) \neq \emptyset$ then c_2 was inserted before c_1,
- if $(a_1 \cap b_2) \neq \emptyset$ then c_2 was inserted after c_1,
- if $(b_1 \cap a_2) = (a_1 \cap b_2) = \emptyset$ then c_1 and c_2 were inserted at same position. Hence, we can use the $code$ of character $code(c_i)$ to determine which character we have to insert at this position.

```
T(Ins($p_1$,$c_1$,$b_1$,$a_1$),Ins($p_2$,$c_2$,$b_2$,$a_2$)) :−
  if $p_1 < p_2$ return Ins($p_1$,$c_1$,$b_1$,$a_1$)
  else if $p_1 > p_2$ return Ins($p_1 + 1$,$c_1$,$b_1$,$a_1$)
       else // $p_1 == p_2$
       if ($b_1 \cap a_2$) $\neq \emptyset$ return Ins($p_1 + 1$,$c_1$,$b_1$,$a_1$)
       else if ($a_1 \cap b_2$) $\neq \emptyset$ return Ins($p_1$,$c_1$,$b_1$,$a_1$)
            else if code($c_1$)>code($c_2$) return Ins($p_1$,$c_1$,$b_1$,$a_1$)
                 else if code($c_1$)<code($c_2$) return Ins($p_1 + 1$,$c_1$,$b_1$,$a_1$)
                      else return Id()

T(Ins($p_1$,$c_1$,$b_1$,$a_1$),Del($p_2$)) :−
   if $p_1 > p_2$ return Ins($p_1 - 1$,$c_1$,$b_1$+Del($p_2$),$a_1$)
   else   return Ins($p_1$,$c_1$,$b_1$,$a_1$+Del($p_2$))

T(Del($p_1$),Del($p_2$)) :−
  if $p_1 < p_2$ return Del($p_1$)
  else if $p_1 > p_2$ return Del($p_1 - 1$)
       else return Id()

T(Del($p_1$,$pr_1$),Ins($x_2$,$p_2$,$b_2$,$a_2$)) :−
  if $p_1 < p_2$ return Del($p_1$)
  else return Del($p_1 + 1$)
```

SPIKE has proved that this set of transformation functions is correct. The only problem is the management of the sets a_i and b_i associated with each Ins operation. The implementation is more difficult and transferring the Ins operation is not efficient.

Proposed Transformation Functions

We propose a new set of correct transformation functions which is simpler than the ones by Suleiman. In fact, Suleiman's transformation functions are over-specified. Managing the set of operations before and after each Ins operation is not necessary. By studying the counter-examples given by SPIKE, we have noted that the problem came from the conflict between the insertion operations which have the same position. To solve this problem we need to know the exact insertion positions at the generation of each of the two operations. Then two cases are possible: either the two users had effectively inserted the characters in the same position or they had inserted them into different positions and other users concurrently erased the characters located between these two positions.

Accordingly, we propose to add a new parameter ip_i to every Ins operation. This parameter represents the *initial position* of character c_i. Suppose the user insert a character x at position 3, then an operation $Ins(3, 3, x)$ is generated. If this operation is transformed, only the position will change. The initial position parameter is not affected.

```
T(Ins($p_1$,$ip_1$,$c_1$),Ins($p_2$,$ip_2$,$c_2$)) :−
  if ($p_1$<$p_2$) return Ins($p_1$,$ip_1$,$c_1$)
  else if ($p_1$>$p_2$) Ins($p_1$+1,$ip_1$,$c_1$)
       else // $p_1$==$p_2$
       if ($ip_1$<$ip_2$) return Ins($p_1$,$ip_1$,$c_1$)
       else if ($ip_1$>$ip_2$) return Ins($p_1$+1,$ip_1$,$c_1$)
            else // $ip_1$==$ip_2$
            if (code($c_1$) < code($c_2$)) return Ins($p_1$,$ip_1$,$c_1$)
            else if (code($c_1$) > code($c_2$)) return Ins($p_1$+1,$ip_1$,$c_1$)
                 else // $c_1$==$c_2$
                      return Id()

T(Ins($p_1$,$ip_1$,$c_1$),Del($p_2$)) :−
  if ($p_1$>$p_2$) return Ins($p_1 - 1$,$ip_1$,$c_1$)
  else return Ins($p_1$,$ip_1$,$c_1$)

T(Del($p_1$),Del($p_2$)) :−
  if ($p_1$<$p_2$) return Del($p_1$)
  else if ($p_1$>$p_2$) return Del($p_1 - 1$)
       else return Id()

T(Del($p_1$,$pr_1$),Ins($p_2$,$ip_2$,$c_2$)) :−
  if ($p_1$<$p_2$) return Del($p_1$)
  else return Del($p_1 + 1$)
```

This set of transformation functions is correct, *w.r.t.* C_1 and C_2. This kind of result shows an important aspect of our approach. By studying counter-examples of Ellis and Ressel, we were sure that the priority systems are unsafe. After proving that Suleiman's functions were safe, we tried to simplify them. With the theorem prover, it was easy for us to try different kind of simplifications and finally converge towards these transformation functions. One serious bottleneck for verifying transformation functions is the number of possible cases to be considered. With our approach, we delegate this task to the theorem prover. Hence we can try a lot of different solutions in a short time. By this way, we have a process to develop quickly correct transformation functions.

Formalization of Transformation Functions

In this section, we describe the principles of SPIKE prover and then we explain how to specify transformation functions and convergence conditions (C_1 and C_2).

The Theorem Prover: SPIKE

Theorem provers have been applied to the formal development of software. They are based on logic-based specification languages and they provide support to the proof of correctness properties, expressed as logical formulas. Theorem provers can be roughly classified in two categories: (i) the *proof assistants* need many interactions even sometimes for simple proof steps; (ii) the *automatic provers* are working in a push-button mode. Tools from the second category are especially useful for handling problems with numerous but relatively simple proof obligations.

For the analysis of collaborative editing systems we have employed the SPIKE induction prover, which belongs to the second category and seems particularly adapted to the task.

SPIKE induction prover has been designed to verify quantifier-free formulas in theories built with first-order conditional rules. SPIKE proof method is based on the so-called *cover set induction*: Given a theory SPIKE computes in a first step induction variables where to apply induction and induction terms which basically represent all possible values that can be taken by the induction variables. Typically for a nonnegative integer variable, the induction terms are 0 and $x + 1$, where x is a variable.

Given a conjecture to be checked, the prover selects induction variables according to the previous computation step, and substitutes them in all possible way by induction terms. This operation generates several instances of the conjecture that are then *simplified* by rules, lemmas, and induction hypotheses.

Note that if the conjecture is false, then it is guaranteed that the prover will exhibit a counter-example. This is very important for our approach.

Formal Specification

For modelling the structure and the manipulation of data in programs, *abstract data types* (ADTs) are frequently used (Wirsing 1990). Indeed, the *structure* of data is reflected by so called *constructors* (*e.g.*, zero 0 and successor $s(x)$, meaning $x+1$, may construct the ADT nat of natural numbers). Moreover, all (potential) data are covered by the set of *constructors terms*, exclusively built by constructors. An ADT may have different *sorts*, each characterized by a separate set of constructors. Furthermore, the *manipulation* of data is reflected by *function symbols* (*e.g.*, $plus$ and $minus$ on nat). The value computed by such functions are specified by *axioms*, usually written in equational logic. An *algebraic specification* is a description of one or more such abstract data types (Wirsing 1990).

Specification of Functions

More formally a real-time groupware system can be considered as a structure of the form $G =< S, O, Tr >$ where:

- S is the structure of the shared object (*i.e.*, a string, an XML document, a CAD object),
- O is the set of operations applied to the shared object,
- Tr is the transformation function.

In our approach, we construct an algebraic specification from a real-time groupware system . Indeed, the shared object structure is transformed in ADT specification $State$. We define a sort Opn for the operation set O, where each operation serves as a constructor of this sort. For instance, a collaborative editing text has a character string as shared object structure, and $O = \{O_1, O_2\}$ where:

- $O_1 = Ins(p, c)$ inserts character c at position p,
- $O_2 = Del(p)$ deletes the character at position p.

For the character string we may specify it with the list ADT; its constructors are $\langle\rangle$ and $l \bullet x$ (*i.e.*, an empty list and a list composed by an element x added to the end of the list l respectively). Because all operations are applied to the object structure in order to modify it, we give the following function:

$$\odot : State \times Opn \rightarrow State$$

All appropriate axioms of the function $\odot$ describe the transition between the object states when applying an operation. For example, the operation $Del(p)$ changes the character string as follows:

$$l \odot Del(p) = \begin{cases} \langle\rangle & \text{if } l = \langle\rangle \\ l & \text{if } l = l' \bullet c \text{ and } p \geq |l| \\ l' & \text{if } l = l' \bullet c \text{ and } p = |l| - 1 \\ (l' \odot Del(p)) \bullet c & \text{if } l = l' \bullet c \text{ and } p < |l| - 1 \end{cases}$$

where $|l|$ returns the length of the list l.

To overcome the user-intention violation problem, a transformation function is used in order to adjust the parameters of one operation according to the effects of other operations executed independently. Writing the specification of a transformation function in first-order logic is straightforward. For this we define the following function:

$$T : Opn \times Opn \rightarrow Opn$$

which takes two arguments, namely remote and local operations and produces another operation. The axioms concerning this function show how the considered real-time groupware transforms its operations when they are broadcasted. For example, the following transformation:

```
T(Del($p_1$),Ins($p_2$,$c_2$) :–
    if  $p_1 > p_2$ return Del($p_1 + 1$)
    else  return  Del($p_1$)
```

is defined by two conditional equations:

$p_1 > p_2 \Longrightarrow$ T(Del(p_1),Ins(p_2,c_2))=Del($p_1 + 1$)
$p_1 \not> p_2 \Longrightarrow$ T(Del(p_1),Ins(p_2,c_2))=Del(p_1)

This example illustrates how it is easy to translate transformation function into the formalism of SPIKE . This task is straightforward and can be done automatically. The cost of formalisation is not expensive.

Specification of Conditions C_1 and C_2

We now express the convergence conditions as theorems to be proved in our algebraic setting. For this purpose, we use a predicate $Enabled : Opn \times State \rightarrow Bool$ expressing the condition under which an operation can be executed on a given state. Adding this predicate allows to avoid the generation of unreachable executions which violate conditions C_1 and C_2 (Imine et al. 2002).

The first condition, C_1, expresses a *semantic equivalence* between two sequences. Each sequence consists of two operations. Given two operations op_1 and op_2, the execution of the sequence of op_1 followed by $T(op_2, op_1)$ must produce the same state as the execution of the sequence of op_2 followed by $T(op_1, op_2)$.

Theorem 1 (Condition C_1).

$$\begin{aligned}&\forall op_i, op_j \in Opn \textit{ and } \forall st \in State :\\ &\quad Enabled(op_i, st) \wedge Enabled(op_j, st) \Longrightarrow\\ &(st \odot op_i) \odot T(op_j, op_i) = (st \odot op_j) \odot T(op_i, op_j)\end{aligned}$$

The second condition C_2 stipulates a *syntactic equivalence* between two sequences, where every sequence is composed of three operations. Given three operations op_1, op_2 and op_3, the transformation of op_3 with regards to the sequence formed by op_2 followed by $T(op_1, op_2)$ must give the same operation as the transformation of op_3 with regards to the sequence formed by op_1 followed by $T(op_2, op_1)$.

Theorem 2 (Condition C_2).

$$\begin{aligned}&\forall op_i, op_j, op_k \in Opn \textit{ and } \forall st \in State :\\&\quad Enabled(op_i, st) \wedge Enabled(op_j, st) \wedge Enabled(op_k, st) \Longrightarrow\\&T(T(op_k, op_i), T(op_j, op_i)) = T(T(op_k, op_j), T(op_i, op_j))\end{aligned}$$

Conclusion and Perspectives

We have demonstrated in this paper the difficulty of building correct transformation functions. Even on a simple String object, all existing transformation functions are incorrect or over-specified. The difficulty stems from the complexity of correctness proof for transformations functions. On a simple String object, each time a function definition changes, you have to explore 123 different cases carefully. We are convinced that this task cannot be done properly without assistance of a computer. Our approach is very valuable:

- The result is a set of safe transformation functions.
- During the development, the guidance of the theorem prover gives a high value feedback. Indeed, the theorem prover quickly produces counter-examples.
- Formalization is easy.

We are convinced that this approach allows the transformational approach to be applied on more complex typed objects (Imine et al. 2002). We are working in several directions now:

- As we can prove C_1 and C_2 on large number of operations, we are currently developing correct transformation functions for a file system, XML files, blocks of text, etc. We are working not only on new sets of safe transformation functions but also on correctness of composition of these sets.
- We are currently modifying the SPIKE theorem prover in order to build an integrated development environment for transformation functions. Within this environment a user will enter functions like in this paper and will call the theorem prover on them like a compiler. If errors are reported then the environment gives counter-examples immediately. We believe that this kind of environment can greatly improve the process of deriving transformation functions.

Acknowledgments

Many thanks for Fethi A. Rabhi, Senior lecturer at University of New South Wales of Sydney and Hala Molli, Associate Professor at University of Nancy, for their reviewing of this paper.

References

Davis, A. H., Sun, C. & Lu, J. (2002), Generalizing operational transformation to the standard general markup language, *in* 'Proceedings of the 2002 ACM conference on Computer supported cooperative work', ACM Press, pp. 58–67.

Ellis, C. A. & Gibbs, S. J. (1989), Concurrency control in groupware systems, *in* 'SIGMOD Conference', Vol. 18, pp. 399–407.

Imine, A., Molli, P., Oster, G. & Rusinowitch, M. (2002), Development of transformation functions assisted by a theorem prover, *in* 'Fourth International Workshop on Collaborative Editing', New Orleans, Louisiana, USA.

Molli, P., Skaf-Molli, H., Oster, G. & Jourdain, S. (2002), Sams: Synchronous, asynchronous, multi-synchronous environments, *in* 'The Seventh International Conference on CSCW in Design', Rio de Janeiro, Brazil.

Palmer, C. R. & Cormack, G. V. (1998), Operation transforms for a distributed shared spreadsheet, *in* 'Proceedings of the 1998 ACM conference on Computer supported cooperative work', ACM Press, pp. 69–78.

Ressel, M., Nitsche-Ruhland, D. & Gunzenhauser, R. (1996), An integrating, transformation-oriented approach to concurrency control and undo in group editors, *in* 'Proceedings of the ACM Conference on Computer Supported Cooperative Work (CSCW'96)', Boston, Massachusetts, USA, pp. 288–297.

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

Suleiman, M., Cart, M. & Ferrié, J. (1997), 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)', ACM Press, pp. 435–445.

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

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

Sun, C. & Chen, D. (2002), 'Consistency maintenance in real-time collaborative graphics editing systems', *ACM Transactions on Computer-Human Interaction (TOCHI)* **9**(1), 1–41.

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

Vidot, N., Cart, M., Ferrié, J. & Suleiman, M. (2000), 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.

Wirsing, M. (1990), 'Algebraic specification', *Handbook of theoretical computer science (vol. B): formal models and semantics* pp. 675–788.