**VOTE**: Group Editors Analyzing Tool

Abdessamad Imine\(^a\), Pascal Molli\(^a\), Gérald Oster\(^a\) and Pascal Urso\(^b\)

\(^a\) LORIA, INRIA - Lorraine  
Campus Scientifique, 54506 Vandœuvre-Lès-Nancy Cedex, France  
\{imine, molli, oster\}@loria.fr

\(^b\) Ecole Supérieure en Sciences Informatiques.  
930, Route des Colles, 06903 Sophia Antipolis, France  
urso@essi.fr

**Abstract**
We present an initial version of a tool VOTE\(^1\) for detecting copies inconsistency in group editors. As input, our tool takes an algorithmic-description which consists of the group editor behaviour and the transformation algorithm. VOTE translates this description into rewrite rules. As a verification back-end we use SPIKE, an automated induction-based theorem prover, which is suitable for reasoning about conditional theories. The effectiveness of our tool is illustrated on several case studies.

**1 Motivations**

A group editor is a system that allows for two or more users (sites) to simultaneously edit a document (a text, an image, a graphic, etc.) without the need for physical proximity and enables them to synchronously observe each others changes. In order to achieve good responsiveness, the shared document is replicated at the local memory of each participating user. Every operation is executed locally first and then broadcasted for execution at other sites. So, the operations are applied in different orders at different replicas (or copies) of the document. This potentially leads to inconsistent (or different) replicas – an undesirable situation for group editors. Let us consider the following group text editor scenario (see the figure 1): there are two sites working on a shared document represented by a string of characters. Initially, all the copies hold the string “efect”. The document is modified with the operation $\text{Ins}(p, c)$ for inserting a character $c$ at position $p$. Users 1 and 2 generate two concurrent operations: $op_1 = \text{Ins}(2, \text{“f”})$ and $op_2 = \text{Ins}(6, \text{“s”})$ respectively.

\(^1\) VOTE can be found at [http://www-sop.inria.fr/coprin/urso/logiciels/](http://www-sop.inria.fr/coprin/urso/logiciels/).

©2003 Published by Elsevier Science B. V.
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 an inconsistency between sites 1 and 2.

How to maintain consistency? One proposed solution is the operational transformation approach [2]. It consists of an algorithm \( T \), called transformation algorithm, which takes two concurrent operations \( op_1 \) (remote) and \( op_2 \) (local) 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. In Figure 2 we illustrate the effect of \( T \) on the previous example. When \( op_2 \) is received on site 1, \( op_2 \) needs to be transformed according to \( op_1 \) as follows: \( T((\text{Ins}(6, “s”), \text{Ins}(2, “f”))) = \text{Ins}(7, “s”). \) The insertion position of \( op_2 \) is incremented because \( op_1 \) has inserted a character at position 2, which is before the character inserted by \( op_2 \). Next, \( op'_2 \) is executed on site 1. In the same way, when \( op_1 \) is received on site 2, it is transformed as follows: \( T(\text{Ins}(2, “f”), \text{Ins}(6, “s”)) = \text{Ins}(2, “f”); \) \( op_1 \) remains the same because “f” is inserted before “s”. Intuitively we can write the transformation \( T \) as follows:

\[
T(\text{Ins}(p_1, c_1), \text{Ins}(p_2, c_2)) = \begin{cases} 
\text{return } \text{Ins}(p_1, c_1) & \text{if } (p_1 < p_2) \\
\text{else return } \text{Ins}(p_1 + 1, c_1) 
\end{cases}
\]

However, according to [3,8,7] the transformation algorithm needs to fulfill the following conditions in order to achieve copies consistency (we use the symbol \( \circ \) to represent the sequence of operations):

**Condition \( C_1 \):** Let \( op_1 \) and \( op_2 \) be two concurrent operations defined on the same state. \( T \) satisfies \( C_1 \) iff:
\[
op_1 \circ T(op_2, op_1) \equiv op_2 \circ T(op_1, op_2). \quad (\equiv \text{ denotes a state equivalence}).
\]
**Condition C₂:** For any operations $op_1$, $op_2$, $op_3$, $T$ satisfies $C₂$ iff:

$$T(op_3, op_1 \circ T(op_2, op_1)) = T(op_3, op_2 \circ T(op_1, op_2)).$$

Finding such a transformation algorithm for an group editor application and proving that it satisfies conditions $C₁$ and $C₂$ is not an easy task. This proof is often difficult to produce by hand and unmanageably complicated. Moreover, $C₂$ is particularly difficult to meet even on a simple string object. Consequently, to be able to develop the transformational approach with simple or more complex objects, proving conditions on transformation algorithm must be assisted by an automatic theorem prover.

In this paper, we present an initial version of a tool, **VOTE** (Validation of Operational Transformation Environment), for automatically checking these conditions. The input of our tool consists of a formal specification written in algorithmic style; it specifies the system behaviour in the *situation calculus* [5] – that allows the developer to concisely describe the effects of operations on the state object without representing its inner structure explicitly – and the functional description of the transformation algorithm. The tool builds an algebraic specification described in terms of conditional equations. As a verification back-end we use **SPIKE** [1, 6], an automated induction-based theorem prover, which is suitable for reasoning about conditional theories.

**2 Architecture**

The organization of the tool is depicted in figure 3. The main entry is a “humanly readable” description of a group editor (behaviour and transformation algorithm). The consistency conditions, $C₁$ and $C₂$, are automatically generated with respect to the input description.

![Fig. 3. Tool architecture.](image)

**Input description**

More formally a group editor system is a structure of the form $G = \langle S_t, O, Tr \rangle$ where: (i) $S_t$ is the structure of the shared object (*i.e.*, a string, an XML document, a CAD object), (ii) $O$ is the set of operations applied on the shared object, (iii) $Tr$ is the transformation algorithm.
Since group editors are in essence dynamic systems, the situation calculus is especially well-suited for formalizing them. This formalism allows us to reason about operations concealing the structure of the shared object. In fact, the situations are finite sequences of operations. Starting with an initial situation, operations possible in a current situation are executed to get new situations. We observe the behavior of the group editor through the situations. In other words, we define only the effect of each operation on the characteristics of the shared object. These characteristics are observed by fluents (or observers) which are inductively defined upon the situation by successor state axioms. The state of a situation is defined as being the set of fluents that hold in that situation. Accordingly, two situations $s$ and $s'$ have the same state, and we denote it by $s \approx_{\text{state}} s'$ if the set of fluents that hold is the same.

As a first step, the user describes the group editor system in algorithmic-style. Figure 4 shows an example of the VOTE input. Firstly, the user declares sorts of used data and the signatures of observers and operations. Every operation is preceded by a boolean expression indicating when this operation is enabled. Next, the user defines the transformation rules. This definition is complete, i.e. all cases should be given. Finally, the user gives the observation rules, i.e. successor state axioms, for every observer and operation.

**Algebraic specification**

In the second step, VOTE translates the above description into algebraic specification [9]. Let $\pi$ be a group editor system. Two sorts are used: $\text{sit}$ and $\text{opn}$ for situations and operations respectively. Let $S$, $S_{bs} = \{\text{sit}, \text{opn}\}$ and $S_{is} = S \setminus S_{bs}$ be the set of all sorts, the set of basic sorts and the set of individuals sorts, respectively. We use $\sharp(\omega, s)$ for denoting the number of occurrences of the sort $s$ in the sequence $\omega$. Then, $\pi$ is modeled by an algebraic specification $SP^\pi = (\Sigma^\pi, A^\pi x)$ where:

- $\Sigma^\pi = (F, X)$ is a signature. $F$ is defined as $C \cup D$, where $C$ and $D$ are constructor and non-constructor (or defined) functions, such that: (i) $C_{e, \text{sit}} = \{S_0\}$, $C_{\text{opn, sit, sit}} = \{\bullet\}$ and $C_{\omega, s} = \emptyset$ if $s$ is $\text{sit}$ or $\omega$ contains an element of $S_{bs}$. (ii) $D_{\text{opn, opn, opn}} = \{T\}$, $D_{\text{opn, sit, bool}} = \{\text{poss}\}$ and $D_{\omega, s} = \emptyset$ if either $s$ is $\text{sit}$, $\omega$ contains $\text{opn}$, or $\sharp(\omega, \text{sit}) > 1$. (iii) $X$ is $S$-indexed family of variable sets.

- $A^\pi x = D_{S_0} \cup D_P \cup D_{SS} \cup D_T$ is the set of axioms (written as conditional equations) such that: (i) $D_{S_0}$ is the set of axioms describing the initial situation, $S_0$; (ii) $D_P$ is the set of operation precondition axioms, i.e. $\text{poss}$; (iii) $D_{SS}$ is the set of successor state axioms for every fluent; (iv) $D_T$ contains axioms corresponding to the transformation function $T$.

The sort $\text{sit}$ has two constructor functions: the constant constructor $S_0$ and the constructor symbol $\bullet$. The set $C_{\omega, s}$ ($\omega \in S_{is}^*$) contains all constructor operations which represent the operation types of $\omega$. All the necessary conditions for the execution of an operation are given by $D_P$. The set $D_{\omega, \text{sit, s}}$
contains all fluent symbols, where $\omega \in S^*_t$ and $s$ is an element of $S_{is}$; these ones are used to define the observations related to the characteristics of the shared object. Precisely when $\pi$ evolving, the change of these characteristics is described by the set of successor state axioms, $D_{SS}$. Finally, the transformation algorithm used by $\pi$ is given as a set of axioms $D_T$. 

Fig. 4. Specification of Group Editor in VOTE tool.
Proving consistency conditions

As a verification back-end we use SPIKE, first-order implicit induction prover. SPIKE was chosen for the following reasons: (i) its high automation degree, (ii) its ability on case analysis (to deal with multiple operations and many case of transformations), (iii) its refutational completeness (to find counter-examples), (iv) its incorporation of decision procedures (to automatically eliminate arithmetic tautologies produced during the proof attempt). In the sequel, we use the following notations: (i) \[ [b_1, \ldots, b_n] \bullet s = b_n \bullet ([b_1, \ldots, b_{n-1}] \bullet s), \]
and, (ii) \[ \text{Legal}([b_1, \ldots, b_n], s) = \text{poss}(b_1, s) \land \ldots \land \text{poss}(b_n, [b_1, \ldots, b_{n-1}] \bullet s), \]
where \( b_1, \ldots, b_n \) are terms of sort \( \text{opm} \) and \( s \) is of sort \( \text{sit} \). We use also \( \models_{\text{ind}} \) for denoting the inductive consequence.

The consistency conditions are formulated as theorems to be proved. Let \( SP_\pi = (\Sigma^\pi, A^\pi_x) \) be an algebraic specification modeling an group editor system \( \pi \). The first condition \( C_1 \) expresses a semantic equivalence between two operation sequences. Given two operations \( \text{op}_1 \) and \( \text{op}_2 \), the sequences \( [\text{op}_1, T(\text{op}_2, \text{op}_1)] \) and \( [\text{op}_2, T(\text{op}_1, \text{op}_2)] \) must produce the same state.

**Theorem 2.1 (Condition \( C_1 \)).**

If for all operations \( \text{op}_1 \) and \( \text{op}_2 \), and for all \( n + 1 \)-ary fluent \( f \):

\[
A^\pi x \models_{\text{ind}} (\text{Legal}([\text{op}_1, T(\text{op}_2, \text{op}_1)], s_1) = \text{true} \\
\quad \land \text{Legal}([\text{op}_2, T(\text{op}_1, \text{op}_2)], s_2) = \text{true} \\
\quad \land f(x_1, \ldots, x_n, s_1) = f(x_1, \ldots, x_n, s_2) \\
\implies f(x_1, \ldots, x_n, [\text{op}_1, T(\text{op}_2, \text{op}_1)] \bullet s_1) = \\
f(x_1, \ldots, x_n, [\text{op}_2, T(\text{op}_1, \text{op}_2)] \bullet s_2)
\]

holds then,

\[
(\text{Legal}([\text{op}_1, T(\text{op}_2, \text{op}_1)], s_1) = \text{true} \\
\land \text{Legal}([\text{op}_2, T(\text{op}_1, \text{op}_2)], s_2) = \text{true} \\
\land s_1 \approx_{\text{state}} s_2) \\
\implies [\text{op}_1, T(\text{op}_2, \text{op}_1)] \bullet s_1 \approx_{\text{state}} [\text{op}_2, T(\text{op}_1, \text{op}_2)] \bullet s_2
\]

also holds.

The second condition \( C_2 \) stipulates a syntactic equivalence between two operation sequences. Given three operations \( \text{op}_1 \), \( \text{op}_2 \) and \( \text{op}_3 \), transforming \( \text{op}_3 \) with respect two sequences \( [\text{op}_1, T(\text{op}_2, \text{op}_1)] \) and \( [\text{op}_2, T(\text{op}_1, \text{op}_2)] \) must give the same operation.

**Theorem 2.2 (Condition \( C_2 \)).**

For all operations \( \text{op}_1 \), \( \text{op}_2 \) and \( \text{op}_3 \):

\[
A^\pi x \models_{\text{ind}} T(\text{op}_3, [\text{op}_1, T(\text{op}_2, \text{op}_1)]) = T(\text{op}_3, [\text{op}_1, T(\text{op}_1, \text{op}_2)]).
\]

\( \text{like } x + z > y = \text{false } \land z + x < y = \text{false } \implies x + z = y \)
All axioms of $\mathcal{A}^r x$ are automatically oriented into rewrite rules by SPIKE. For proving theorem 2.1 (resp. 2.2), SPIKE replaces first the variables $op_1$ and $op_2$ (resp. $op_1$, $op_2$ and $op_3$) with the elements of the test set describing the sort $opm$. This replacement generates many instances of the theorem to be verified, enabling to cover all possible cases. Next, SPIKE simplifies these instances by rewriting. The proof of $C_1$ and $C_2$ is either successful and transformation algorithm is verified, or failed and the SPIKE’s proof-trace is used by VOTE to extract the problematic cases to the user. In the later case, there are two possibilities. The first one concerns valid conjectures where appear undefined auxiliary functions or arithmetic symbols which SPIKE’s decision procedure cannot manage; in this case, the user can introduce lemmas. The second one concerns cases violating condition $C_1$ or $C_2$. VOTE gives the scenario (operation and conditions) of each cases to help user to rectify its transformations.

3 Experiments

We have detected a lot of bugs in well-known group editors such that GROVE [2], Joint Emacs [3], REDUCE [8] and SAMS [4], which are based on transformational approach for maintaining consistency of shared data. The results of our experiments are reported in Table 1. GROVE, Joint Emacs and REDUCE are group text editors whereas SAMS is XML document-based group editor. $S^5$ is a file synchronizer which uses a transformation algorithm for synchronizing many file system replicas.

<table>
<thead>
<tr>
<th>Group editors</th>
<th>$C_1$</th>
<th>$C_2$</th>
</tr>
</thead>
<tbody>
<tr>
<td>GROVE</td>
<td>violated</td>
<td>violated</td>
</tr>
<tr>
<td>Joint Emacs</td>
<td>violated</td>
<td>violated</td>
</tr>
<tr>
<td>REDUCE</td>
<td>correct</td>
<td>violated</td>
</tr>
<tr>
<td>SAMS</td>
<td>correct</td>
<td>violated</td>
</tr>
<tr>
<td>$S^5$</td>
<td>correct</td>
<td>violated</td>
</tr>
</tbody>
</table>

Table 1
Case studies.

Let consider the group text editor GROVE designed by Ellis and Gibbs – the pioneers of the operational transformation. The text is modified by two operations: (i) $Ins(p, c, pr)$ to insert a character $c$ at position $p$. (ii) $Del(p, pr)$ to delete the character located at position $p$. The $pr$ parameter represents the priority (site identifier where the operation is generated). Let us consider the

---

4 http://woinville.loria.fr/sams
5 http://woinville.loria.fr/S5
following transformations (the complete description of this editor is given in Figure 4):

\[
T(\text{Ins}(p_1, c_1, pr_1), \text{Del}(p_2, pr_2)) =
\begin{align*}
\text{if } (p_1 < p_2) & \text{ then return } \text{Ins}(p_1, c_1, pr_1) \\
\text{else return } & \text{Ins}(p_1 - 1, c_1, pr_1) \\
\end{align*}
\]

\[
T(\text{Del}(p_1, pr_1), \text{Ins}(p_2, c_2, pr_2)) =
\begin{align*}
\text{if } (p_1 < p_2) & \text{ then return } \text{Del}(p_1, pr_1) \\
\text{else return } & \text{Del}(p_1 + 1, pr_1) \\
\end{align*}
\]

After submitting this system to VOTE, it has detected that condition $C_1$ is violated by giving the counter-example depicted in figure 4. The counter-example is simple: (i) $user_1$ inserts $x$ in position 2 ($op_1$) while $user_2$ concurrently deletes the character at the same position ($op_2$). (ii) When $op_2$ is received by site 1, $op_2$ must be transformed according to $op_1$. So $T(\text{Del}(2), \text{Ins}(2, x))$ is called and $\text{Del}(3)$ is returned. (iii) In the same way, $op_1$ is received on site 2 and must be transformed according to $op_2$. $T(\text{Ins}(2, x), \text{Del}(2))$ is called and return $\text{Ins}(1, x)$. Condition $C_1$ is violated. Accordingly, the final results on both sites are different.

Fig. 5. Counter-example violating condition $C_1$.

The error comes from the definition of $T(\text{Ins}(p_1, c_1, pr_1), \text{Del}(p_2, pr_2))$. The condition $p_1 < p_2$ should be rewritten $p_1 \leq p_2$. 

4 Conclusion

This tool is a first step towards to assist the development of correct transformation algorithms in order to ensure copies consistency in group editors. We have detected bugs in many well-known systems. So, we think that our approach is very valuable because: (i) it can help significantly to increase confidence in a transformation algorithm; (ii) having the theorem prover ensures that all cases are considered and quickly produces counter-example scenarios;

Many features are planned to deal effective and large systems. We plan to ensure the correct composition of many transformation algorithms for handling composed objects. Finally, we intend to improve strategy proofs underlying to SPIKE for increasing more the degree of automation.

References


