Development of Transformation Functions Assisted by a Theorem Prover

Imine Abdessamad, Pascal Molli, Gérald Oster, Michaël Rusinowitch

Development of Transformation Functions Assisted by a Theorem Prover

Abdessamad Imine[2], Pascal Molli[1], Gérald Oster[1] and Michaël Rusinowitch[2]
ECOO[1] & CASSIS[2]
LORIA, INRIA Lorraine, BP 239, 54506 Vandœuvre-lès-Nancy, France
{imine,molli,oster,rusi}@loria.fr

ABSTRACT
Transformational approach requires to write transformation functions that ensure properties $C1$ and $C2$. Proving these conditions on complex typed objects is a serious bottleneck for the application of this approach. We propose to use a theorem prover to assist the development of safe transformation functions. In this paper, we present how we have designed in that way a set of safe transformation functions for an XML typed object.

Keywords
Theorem Prover, Transformational Approach, Transformation Functions, XML.

1. INTRODUCTION

The Operational Transformation approach [3, 8] allows one to build real-time groupware. Algorithms such as aDOPTed [6], GOTO [9], SOCT 2,3,4 [7, 10] suppose that the transformation functions T respect conditions $C1$ and $C2$:

- $C1 : op_1 \circ T(op_2, op_1) \equiv op_2 \circ T(op_1, op_2)$,
- $C2 : T(op_3, op_1 \circ T(op_2, op_1)) = T(op_3, op_2 \circ T(op_1, op_2))$,

where $T(op_2, op_1)$ means that op_2 is transformed according to op_1. It is quite difficult to prove these conditions even on simple typed objects. If we have more operations on more complex typed objects, the proof gets nearly impossible to achieve without a computer. This is a serious bottleneck for building more complex real time groupware software.

We propose to assist the development of transformation functions with SPIKE theorem prover [2].This new approach requires specifying the transformation functions in first order logic. Then, SPIKE automatically determines if the transformation functions respect $C1$ and $C2$. If not, SPIKE returns sequences of operations that violate $C1$ or $C2$.

As proofs are automatic, we can handle more operations and more complex ones. In this article, we present how we have developed a set of transformation functions for XML that respects $C1$ and $C2$.

In section 2, we present operations on an XML typed object. In section 3, we present the results of SPIKE analysis on our initial set of transformation functions. Section 4 briefly overviews the SPIKE characteristics. In section 5, we describe the formal specification of transformation functions in SPIKE . The complete set of safe XML transformation functions and their specification appear respectively in appendices A and B.

2. XML OPERATIONS

An XML document is an unordered tree whose nodes can be decorated with attributes. We consider that any XML tree can be built with the following set of operations:

1. $CN(int\, n, String\, tn)$: $int\, nn$ (C)reate (N)ode nn, child of n with tag name tn where nn is a unique new identifier.
2. $DN(int\, n) : void$ (D)elete (N)ode n where n exists.
3. $CA(int\, n, String\, a) : void$ (C)reate (A)ttribute a on node n where n exists and a does not exist (it is a new attribute for n).
4. $DA(int\, n, String\, a) : void$ (D)elete (A)ttribute a of node n where n and a exist.
5. $CHA(int\, n, String\, a, any\, v)$: $void$ (CH)ange the (A)ttribute a of node n with the value v, where n and a exists and v is an arbitrary value. We can assume that v can be serialized in a `String`.

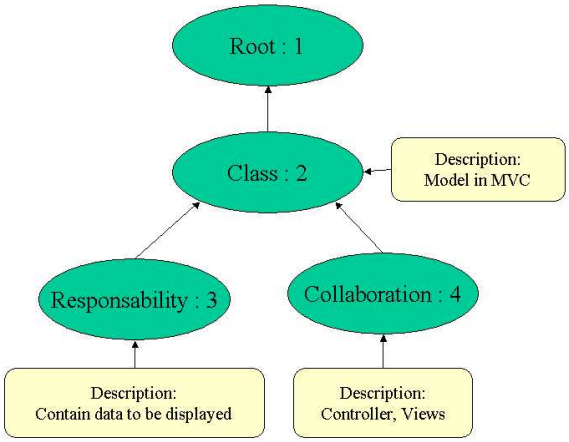

Figure 1: Example of XML tree

The following script illustrates how to use XML operations to build an XML tree. Figure 1 shows the resulting tree.

```
CN(1,"Class")−>2
CA(2,"Description")
CHA(2,"Description","Model in MVC")

CN(2,"Responsibility")−>3
CA(3,"Description")
CHA(3,"Description","Contain data to be displayed")

CN(2,"Collaborations")−>4
CA(4,"Description")
CHA(3,"Description","Controller, Views")
```

3. DEVELOPPING TRANSFORMATION FUNCTIONS WITH SPIKE

We have developped the following transformation function without SPIKE . We have proved by hand that condition $C1$ is verified. And we have attempted to verify condition $C2$ with SPIKE . We denote the identity operation by $NOOP$.

```
T(CHA(n1,a1,v1),CHA(n2,a2,v2)):−
   if n1=n2 ^ a1=a2 ^ v1=v2 return NOOP
   if n1=n2 ^ a1=a2 ^ v1<>v2 return CHA(n1,a1,max(v1,
        v2))
   else return CHA(n1,a1,v1)
```

SPIKE proves that $C2$ is violated in the following scenario with $max(v_1, v_2) = max(v_2, v_3)$:

s_1	s_2	s_3
$CHA(n, a, v_1)$	$CHA(n, a, v_2)$	$CHA(n, a, v_3)$

We illustrate on Figure 2 the broadcast of an instance of this sequence. The problem comes from the integration of op_3 on site 2.

1. First transformation of op_3 with op_2 (denoted by $op_3^{op_2}$) gives $T(CHA(1, a, 2), CHA(1, a, 5)) = CHA(1, a, 5)$.
2. Then, the transformation of $op_3^{op_2}$ with $op_1^{op_2}$ gives $T(CHA(1, a, 5), CHA(1, a, 5)) = NOOP$.
3. Moreover, at Site 1, transformation of op_3 with op_1 gives $T(CHA(1, a, 2), CHA(1, a, 1)) = CHA(1, a, 2)$.
4. Then the transformation of $op_3^{op_1}$ with $op_2^{op_1}$ gives $T(CHA(1, a, 2), CHA(1, a, 5)) = CHA(1, a, 5)$. Consequently, $C2$ is not verified.

To ensure $C2$, we have redefined the transformation function as follows:

```
T(CHA(n1,a1,v1),CHA(n2,a2,v2)):−
        if n1=n2 ^ a1=a2 return CHA(n1,a1,max(v1,v2))
        else return CHA(n1,a1,v1)
```

If we resubmit the new specification to SPIKE , then the whole specification respects $C1$ and $C2$. Furthermore, SPIKE found another problem in our initial specification. It concerns a possible interleaving between operations on attributes.

```
T(CA(n1,a1),DA(n2,a2)) :−
        return CA(n1,a1)

T(CA(n1,a1),CHA(n2,a2,v2)) :−
        if n1=n2 ^ a1=a2 return NOOP
        else return CA(n1,a1)

T(DA(n1,a1),CA(n2,a2)) :−
        if n1=n2 ^ a1=a2 return NOOP
        else return DA(n1,a1)

T(DA(n1,a1),CHA(n2,a2,v2)) :−
        return DA(n1,a1)

T(CHA(n1,a1,v1),CA(n2,a2)):−
        return CHA(n1,a1,v1)

T(CHA(n1,a1,v1),DA(n2,a2)):−
        if n1=n2 ^ a1=a2 return NOOP
        else return CHA(n1,a1,v1)
```

SPIKE proves also that $C2$ is violated in the following scenario:

s_1	s_2	s_3
$CA(n, a)$	$CHA(n, a, v)$	$DA(n, a)$

Figure 3 shows the broadcast of an instance of this sequence. We have deliberately applied tranformation functions on operations which are not defined on same state, in order to illustrate a possible instance of this sequence. The problem comes from the integration of op_3 on Site 2.

1. First the transformation of op_3 with op_2 (denoted by $op_3^{op_2}$) gives $T(DA(1, a, 2), CHA(1, a, 5)) = DA(1, a)$.
2. Then, the transformation of $op_3^{op_2}$ with $op_1^{op_2}$ gives $T(DA(1, a), NOOP) = DA(1, a)$.
3. On the other hand, at Site 1, the transformation of op_3 with op_1 gives $T(DA(1, a), CA(1, a)) = NOOP$.
4. Then the transformation of $op_3^{op_1}$ with $op_2^{op_1}$ gives $T(NOOP, CHA(1, a, 5)) = NOOP$. So $C1$ and $C2$ are not verified.

We can fix this problem by correcting the specification as follows:

```
T(CA(n1,a1),DA(n2,a2)) :
        if n1=n2 ^ a1=a2 return NOOP // new
        else
        return CA(n1,a1)

T(DA(n1,a1),CA(n2,a2)) :−
// deleted
//          if n1=n2 ^ a1=a2 return NOOP
//          else
            return DA(n1,a1)
```

However, although SPIKE can prove that this scenario violates $C2$, it cannot prove that this scenario is reachable. In this scenario, it is not possible to have op_2 without a preceding operation $CA(1, a)$ that is concurrent to op_1. And then, the pre-condition of op_1 is violated. Anyway, we prefer to fix the specification even if this scenario is not reachable.

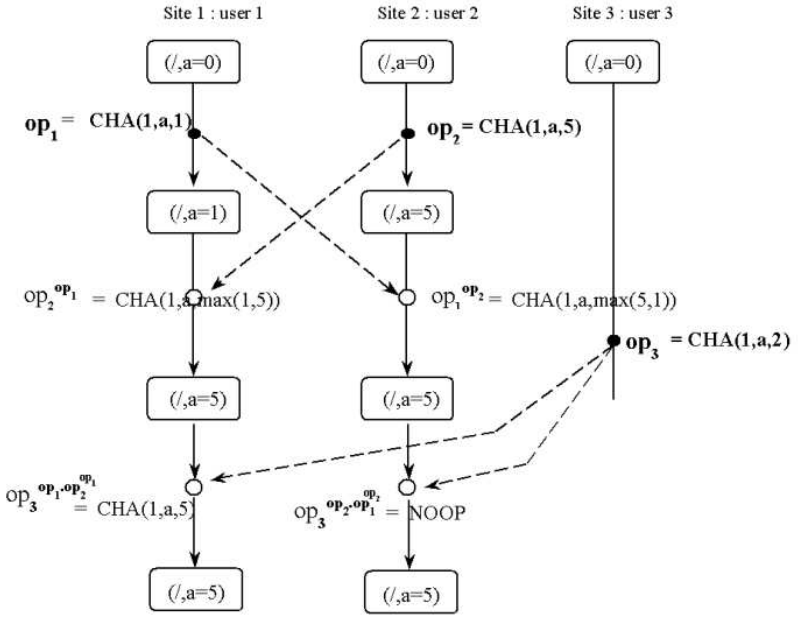

Figure 2: Counter example scenario that violates condition $C2$

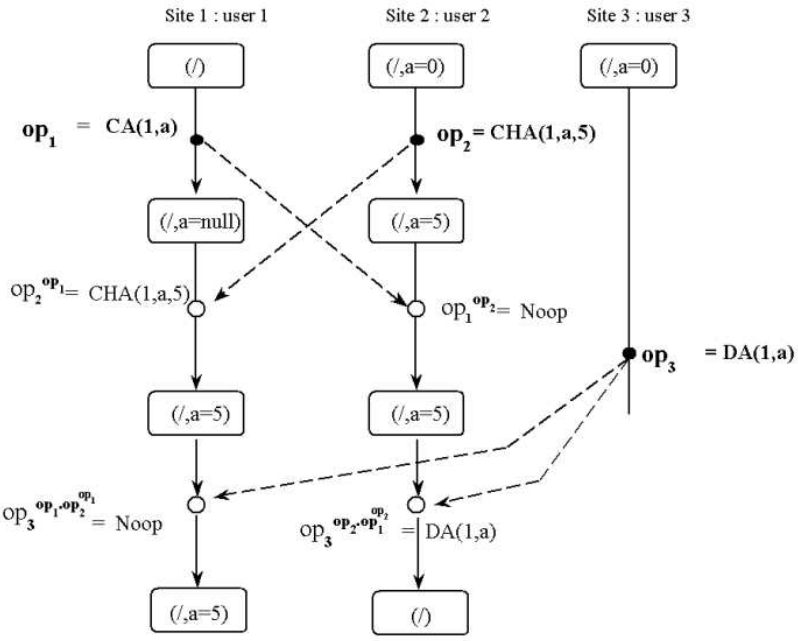

Figure 3: Counter example scenario that violates conditions $C1$ and $C2$

These examples illustrate how a theorem prover can find quickly unobvious mistakes in the sequence of transformations. The gain is quite important:

- During the writing of the transformation functions, developers have a quick feedback about problems in the transformation functions,
- At the end of the development, we are sure to have a safe set of transformation functions.

Of course, these advantages have a cost: specifying the transformation functions in the theorem prover formal language. In the next section we explain the basic principles of SPIKE and how a developer can specify transformation functions in SPIKE .

4. 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.

The 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.

The nice feature of SPIKE is that not only it can prove theorems but it can also disprove wrong conjectures by providing counter-examples.

5. FORMAL SPECIFICATION

The goal of the specification is to prove conditions $C1$ and $C2$.

- $C1$ is a state equivalence property. It requires representing the states and how to construct them. For our description formalism we choose the constructor-based algebraic approach (see e.g. [5]).
- $C2$ is a syntactic equality on operation. It does not require representing states; we only need to specify transformation functions and profiles of operations on typed objects.
- It is interesting to note that, for the theorem prover, it is more complex to prove $C1$ than $C2$.

5.1 Specification of Conditions C1 and C2

We now express the convergence conditions as theorems to be proved in our algebraic setting. For this purpose, we use two sets: Opn is the set of operations and Tree is the set of states.

The first condition, $C1$, expresses a *semantic equivalence* between two sequences where everyone 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 tree as the execution of the sequence of op_2 followed by $T(op_1, op_2)$.

THEOREM 5.1. **(Condition C1).**

$$\forall op_i, op_j \in \text{Opn } \textit{and } \forall t \in \text{Tree}: \\ t \circ op_i \circ T(op_j, op_i) = t \circ op_j \circ T(op_i, op_j)$$

The second condition, $C2$, stipulates a *syntactic equivalence* between two sequences where everyone is composed of three operations. Given three operations op_1, op_2 and op_3, the transformation of op_3 with regard to the sequence formed by op_2 followed by $T(op_1, op_2)$ must gives the same operation as the transformation of op_3 with regard to the sequence formed by op_1 followed by $T(op_2, op_1)$.

THEOREM 5.2. **(Condition C2).**

$$\forall op_i, op_j, op_k \in \text{Opn}: \\ T(op_k, op_i \circ T(op_j, op_i)) = T(op_k, op_j \circ T(op_i, op_j))$$

5.2 Specification of Transformation Functions

Writing the specification of transformation functions in first order logic is straightforward:

T(**CN**(n1,tn1),**CN**(n2,tn2)):−
  return CN(n1,tn1)

in SPIKE :
$T(CN(n1, tn1), CN(n2, tn2)) = CN(n1, tn1)$

T(**CN**(n1,tn1),**DN**(n2)):−
  if (n1 **childof** n2) **return NOOP**
  else return CN(n1,tn1)

in SPIKE :
$Childof(n1, n2) = true \Rightarrow T(CN(n1, tn1), DN(n2)) = Nop$
$Childof(n1, n2) = false \Rightarrow T(CN(n1, tn1), DN(n2)) = CN(n1, tn1)$

Then SPIKE needs to know the profile of the managed operations:

---

specification : convergence
use : Trees ;

sorts : Opn Tag Node Att SetAtt Val Tree;

constructors :

CN__: Node Tag −> Opn;
DN_: Node −> Opn;
CA__: Node Att −> Opn;
DA__: Node Att −> Opn;
CHA___: Node Att Val −> Opn;
Nop : −> Opn;

defined functions :

T__: Opn Opn −> Opn;

If a user only wants to prove $C2$, the specifications of typed objects are very simple. For XML, it has required one day work. Next, to prove $C1$, we need to represent the state of the XML tree. This specification is more involved.

5.3 Specification of the XML Tree

The XML object is a tree-like hierarchical structure built from *nodes* with a particular one called the *root*. The components of this object are as follows: (i) every node consists of an identifier and a set (possibly empty) of *attributes*; (ii) each attribute has an identifier and a value.

To verify $C1$ we have to write the algebraic specification of a tree in first order logic. This is quite standard.

The complete specification is in Section B.

6. IMPLEMENTATION

We have used this set of proved transformation functions for our environment SAMS [4]. This environment implements an original concept described in [1]. In this environment, a team member can use a working style according to his needs and the environment still ensures the consistency. Multi-synchronous mode is suitable for production phases when the user wants to work in insulation and synchronous mode is suitable for discussion phases where user needs to work with others in order to converge towards a state that satisfy all people.

A SAMS environment is independent of shared objects types. We have developed in this environment two editors: a CRC cards editor and an HTML editor (cf figure 4). We could easily add an SVG editor, UML, CAD editor ... As this environment is flexible, we can develop a SAMS environment for text editors, drawings, diaries ...

The XML SAMS environment can be tested online at the following url: `http://woinville.loria.fr/simu/`

7. CONCLUSION AND PERSPECTIVES

We have presented in this paper how to write transformation functions with the assistance of an automatic theorem prover. This 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 produces quickly counter-examples.

We are convinced that this approach allows the transformational approach to be applied on more complex typed objects.

As expected, the proving has a cost. We have shown that the cost of proving $C2$ is very low. The developer has only to translate the transformation functions in first order logic and to give the profiles of the primitives operations. This is an important result. Indeed, it is nearly impossible to prove $C2$ by hand for complex objects and in all cases it is error-prone.

The cost of proving $C1$ is more important because it requires the specification of states. However there exist a large number of algebraic specifications for many data structures in the literature. We think that it is important to check also $C1$ with a theorem prover. Even if it is possible to prove $C1$ by hand, this process is error prone and can be very damaging at exploitation stage.

It took 1 week to specify the XML operations in SPIKE and to converge to a proved set of transformation functions. The specification and the proof have been performed by CASSIS Team without any knowledge about transformational approach. They have given feedback about bogus scenario after a few days.

To give an idea about computation time needed by the prover, SPIKE computes the complete proof of $C1$ and $C2$ in less than 1 hour on a Pentium 4 computer. We think that this approach can scale if the number of operation increase.

We are working in several directions now:

- As we can prove $C1$ and $C2$ on large number of operations, we are currently composing XML operations with string operations.
- CASSIS Team is currently improving the SPIKE theorem prover in order to build an integrated development environment dedicated to write transformation functions.

REFERENCES

[1] Abdelmajid Bouazza and Pascal Molli. Unifying coupled and uncoupled collaborative work in virtual teams. In *ACM CSCW'2000 workshop on collaborative editing systems, Philadelphia, Pennsylvania, USA*, December 2000.

[2] Adel Bouhoula and Michael Rusinowitch. Implicit Induction in Conditional Theories. *Journal of Automated Reasoning*, 14(2):189–235, 1995.

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

[4] Pascal Molli, Hala Skaf-Molli, Gérald Oster, and Sébastien Jourdain. Sams: Synchronous, asynchronous, multi-synchronous environments. In *The Seventh International Conference on CSCW in Design*, Rio de Janeiro, Brazil, September 2002.

[5] Peter Padawitz. Swinging types = relations + functions + transition systems. *TCS*, 243(1-2):93–165, 2000.

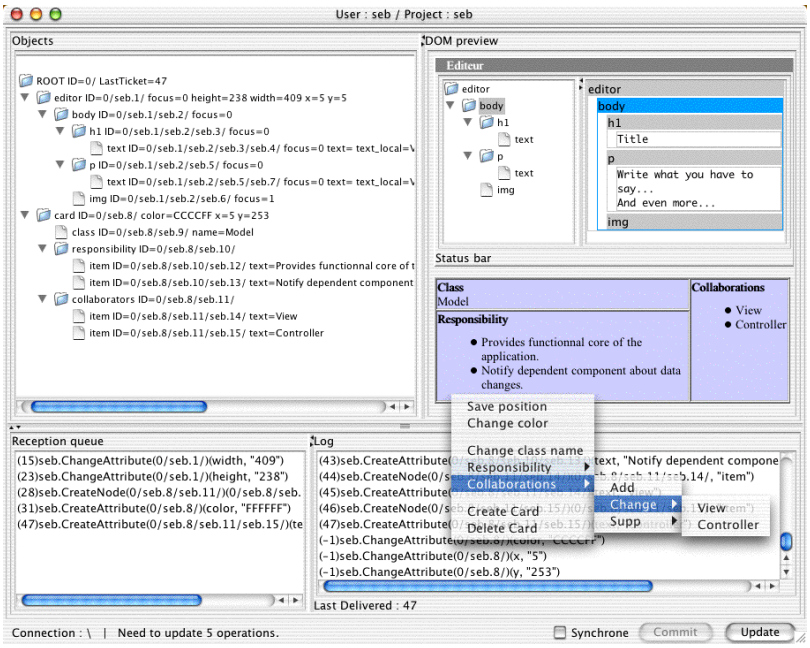

Figure 4: SAMS Environment based on XML typed object

[6] Matthias Ressel, Doris Nitsche-Ruhland, and Rul Gunzenhauser. An integrating, transformation-oriented approach to concurrency control and undo in group editors. In *Computer Supported Cooperative Work*, pages 288–297, 1996.

[7] Maher Suleiman, Michèle Cart, and Jean Ferrié. Concurrent operations in a distributed and mobile collaborative environment. In *Proceedings of the Fourteenth International Conference on Data Engineering, February 23-27, 1998, Orlando, Florida, USA*, pages 36–45. IEEE Computer Society, 1998.

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

[9] C. Sun, X. Jia, Y. Zhang, Y. Yang, and D. Chen. Achieving convergence, causality-preservation and intention-preservation in real-time cooperative editing systems. In *ACM Transactions on Computer-Human Interactions*, volume 5, pages 63–108, 1998.

[10] 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. ACM Press.

APPENDIX

A. XML TRANSFORMATION FUNCTIONS

```
// ----------
// childof:
// n1 childof n2 return true if n1=n2 or n1 is a child of n2

//----------
// NOOP

// foreach x in {CN,DN,CA,DA,CHA}
T(NOOP, x):-
    return NOOP

T(x, NOOP):-
    return x

//----------
// Create node

T(CN(n1,tn1),CN(n2,tn2)):-
        return CN(n1,tn1)

T(CN(n1,tn1),DN(n2)):-
        if( n1 childof n2) return NOOP
        else return CN(n1,tn1)

T(CN(n1,tn1),CA(n2,a2)):-
        return CN(n1,tn1)

T(CN(n1,tn1),DA(n2,a2)):-
        return CN(n1,tn1)

T(CN(n1,tn1),CHA(n2,a2,v2)):-
        return CN(n1,tn1)

//----------
// Delete node

T(DN(n1),DN(n2)):-
        if (n1 childof n2) return NOOP
        else return DN(n1)

T(DN(n1),CN(n2,tn2)):-
```

```
        return DN(n1)

T(DN(n1),CA(n2,a2)):−
        return DN(n1)

T(DN(n1),DA(n2,a2)):−
        return DN(n1)

T(DN(n1),CHA(n2,a2,v2)):−
        return DN(n1)

//−−−−−−−−−−−−−
// Create Attribute
// CA(n1,a1) creates an attribute with a empty value (called
      null)

T(CA(n1,a1),CA(n2,a2)) :−
        if n1=n2 ^ a1=a2 return NOOP
        else return CA(n1,a1)

T(CA(n1,a1),CN(n2,tn2)) :−
        return CA(n1,a1)

T(CA(n1,a1),DN(n2)) :−
        if n1 childof n2 return NOOP
        else return CA(n1,a1)

T(CA(n1,a1),DA(n2,a2)) :−
        if n1=n2 ^ a1=a2 return NOOP
        else
        return CA(n1,a1)

T(CA(n1,a1),CHA(n2,a2,v2)) :−
        if n1=n2 ^ a1=a2 return NOOP
        else return CA(n1,a1)

//−−−−−−−−−−−−−−−−
// Delete Attribute

T(DA(n1,a1),DA(n2,a2)) :−
        if n1=n2 ^ a1=a2 return NOOP
        else return DA(n1,a1)

T(DA(n1,a1),CN(n2,tn2)) :−
        return DA(n1,a1)

T(DA(n1,a1),DN(n2)) :−
        if n1 childof n2 return NOOP
        else return DA(n1,a1)

T(DA(n1,a1),CA(n2,a2)) :−
        return DA(n1,a1)

T(DA(n1,a1),CHA(n2,a2,v2)) :−
        return DA(n1,a1)

//−−−−−−−−−−−−−−−−
// Change Attribute

T(CHA(n1,a1,v1),CN(n2,tn2)):−
        return CHA(n1,a1,v1)

T(CHA(n1,a1,v1),DN(n2)):−
        if n1 childof n2 return NOOP
        else return CHA(n1,a1,v1)

T(CHA(n1,a1,v1),CA(n2,a2)):−
        return CHA(n1,a1,v1)

T(CHA(n1,a1,v1),DA(n2,a2)):−
        if n1=n2 ^ a1=a2 return NOOP
        else return CHA(n1,a1,v1)

T(CHA(n1,a1,v1),CHA(n2,a2,v2)):−
        if n1=n2 ^ a1=a2 return CHA(n1,a1,max(v1,v2))
        else return CHA(n1,a1,v1)
```

B. SPECIFICATION OF XML TYPED OBJECT IN SPIKE LOGIC

```
specification : convergence
use : Trees ;

sorts : Opn Tag Node Att SetAtt Val Tree;

constructors :
CN__: Node Tag −> Opn;
DN_: Node −> Opn;
CA__: Node Att −> Opn;
DA__: Node Att −> Opn;
CHA___: Node Att Val −> Opn;
Nop : −> Opn;

defined functions :

_._: Tree Opn −> Tree;
T__: Opn Opn −> Opn;
Add___: Tree Node Node −> Tree;
Create__: Tree Node −> Tree;
Creatt___: Tree Node Att −> Tree;
Del__: Tree Node −> Tree;
Delatt___: Tree Node Att −> Tree;
Chat____: Tree Node Att Val −> Tree;
Childof__: Node Node −> Bool;
Null_: Tag −> Bool;
Eqn__: Node Node −> Bool;
Eqa__: Att Att −> Bool;
Eqv__: Val Val −> Bool;
Exist_: Node −> Bool;
Exista_: Att −> Bool;
List_: Node −> SetAtt;
In__: Att SetAtt −> Bool;
Max__: Val Val −> Val;
New__: Node Tree −> Node;

axioms:

%Childof properties
Exist(n1)=false =>
     Childof(n1,n2)=false;
Exist(n2)=false =>
     Childof(n1,n2)=false;

%Properties of Equality nodes
Exist(n1)=false =>
     Eqn(n1,n2)=false;
Exist(n2)=false =>
     Eqn(n1,n2)=false;

%Create Node
Exist(n1)=true, Null(tn1)=false, Exist(New(n1,St))=false =>
     St.CN(n1,tn1)=Add(St,n1,New(n1,St));
Exist(n1)=false =>
     St.CN(n1,tn1) = St;
Null(tn1)=true =>
     St.CN(n1,tn1) = St;

%Delete Node
Exist(n1)=true =>
     St.DN(n1)=Del(St,n1);
Exist(n1)=false =>
     St.DN(n1)=St;

%Create an attribute with a empty value
Exist(n1)=true, In(a1,List(n1))=false  =>
     St.CA(n1,a1)=Creatt(St,n1,a1);
Exist(n1)=false =>
```

St.**CA**(n1,a1)=St;
In(a1,List(n1))=true =>
St.**CA**(n1,a1)=St;

%Delete attribute
Exist(n1)=true, In(a1,List(n1))=true =>
St.**DA**(n1,a1)=Delatt(St,n1,a1);
Exist(n1)=false =>
St.**DA**(n1,a1)=St;
In(a1,List(n1))=false =>
St.**DA**(n1,a1)=St;

%Change attribute
Exist(n1)=true, In(a1,List(n1))=true =>
St.**CHA**(n1,a1,v1)=Chat(St,n1,a1,v1);
Exist(n1)=false =>
St.**CHA**(n1,a1,v1)=St;
In(a1,List(n1))=false =>
St.**CHA**(n1,a1,v1)=St;

%Nop
St.Nop=St;

%Transpose T(remote operation, local operation (executed))
% T(**CN**(n1,tn1),i)
T(**CN**(n1,tn1),**CN**(n2,tn2)) = **CN**(n1,tn1);
Childof(n1,n2)=true =>
T(**CN**(n1,tn1),**DN**(n2)) = Nop;
Childof(n1,n2)=false =>
T(**CN**(n1,tn1),**DN**(n2)) = **CN**(n1,tn1);
T(**CN**(n1,tn1),**CA**(n2,a2))=**CN**(n1,tn1);
T(**CN**(n1,tn1),**CHA**(n2,a2,v2))=**CN**(n1,tn1);
T(**CN**(n1,tn1),**DA**(n2,a2))=**CN**(n1,tn1);
T(**CN**(n1,tn1),Nop)=**CN**(n1,tn1);

%T(**DN**(n1),i)
Childof(n1,n2)=true =>
T(**DN**(n1),**DN**(n2)) = Nop;
Childof(n1,n2)=false =>
T(**DN**(n1),**DN**(n2)) = **DN**(n1);
T(**DN**(n1),**CN**(n2,tn2)) = **DN**(n1);
T(**DN**(n1),**CA**(n2,a2))=**DN**(n1);
T(**DN**(n1),**DA**(n2,a2))=**DN**(n1);
T(**DN**(n1),**CHA**(n2,a2,v2))=**DN**(n1);
T(**DN**(n1),Nop)=**DN**(n1);

%T(**CA**(n1,a1),i)
Eqn(n1,n2)=true, Eqa(a1,a2)=true =>
T(**CA**(n1,a1),**CA**(n2,a2))=Nop;
Eqn(n1,n2)=false =>
T(**CA**(n1,a1),**CA**(n2,a2))=**CA**(n1,a1);
Eqa(a1,a2)=false =>
T(**CA**(n1,a1),**CA**(n2,a2))=**CA**(n1,a1);
T(**CA**(n1,a1),**CN**(n2,tn2))=**CA**(n1,a1);
Childof(n1,n2)=true =>
T(**CA**(n1,a1),**DN**(n2))=Nop;
Childof(n1,n2)=false =>
T(**CA**(n1,a1),**DN**(n2))=**CA**(n1,a1);
Eqn(n1,n2)=true, Eqa(a1,a2)=true =>
T(**CA**(n1,a1),**DA**(n2,a2))=Nop;
Eqn(n1,n2)=false =>
T(**CA**(n1,a1),**DA**(n2,a2))=**CA**(n1,a1);
Eqa(a1,a2)=false =>
T(**CA**(n1,a1),**DA**(n2,a2))=**CA**(n1,a1);
Eqn(n1,n2)=true, Eqa(a1,a2)=true =>
T(**CA**(n1,a1),**CHA**(n2,a2,v2))=Nop;
Eqn(n1,n2)=false =>
T(**CA**(n1,a1),**CHA**(n2,a2,v2))=**CA**(n1,a1);
Eqa(a1,a2)=false =>
T(**CA**(n1,a1),**CHA**(n2,a2,v2))=**CA**(n1,a1);
T(**CA**(n1,a1),Nop)=**CA**(n1,a1);

%T(**DA**(n1,a1),i)
Eqn(n1,n2)=true, Eqa(a1,a2)=true =>
T(**DA**(n1,a1),**DA**(n2,a2))=Nop;
Eqn(n1,n2)=false =>
T(**DA**(n1,a1),**DA**(n2,a2))= **DA**(n1,a1);
Eqa(a1,a2)=false =>
T(**DA**(n1,a1),**DA**(n2,a2))=**DA**(n1,a1);
T(**DA**(n1,a1),**CN**(n2,tn2))=**DA**(n1,a1);
Childof(n1,n2)=true =>
T(**DA**(n1,a1),**DN**(n2))=Nop;
Childof(n1,n2)=false =>
T(**DA**(n1,a1),**DN**(n2))=**DA**(n1,a1);
T(**DA**(n1,a1),**CA**(n2,a2))=**DA**(n1,a1);
T(**DA**(n1,a1),**CHA**(n2,a2,v2))=**DA**(n1,a1);
T(**DA**(n1,a1),Nop)=**DA**(n1,a1);

%T(**CHA**(n1,a1,v1),i)
Eqn(n1,n2)=true, Eqa(a1,a2)=true =>
T(**CHA**(n1,a1,v1),**CHA**(n2,a2,v2))=**CHA**(n1,a1,Max(v1,
v2));
Eqn(n1,n2)=false =>
T(**CHA**(n1,a1,v1),**CHA**(n2,a2,v2))=**CHA**(n1,a1,v1);
Eqa(a1,a2)=false =>
T(**CHA**(n1,a1,v1),**CHA**(n2,a2,v2))=**CHA**(n1,a1,v1);
T(**CHA**(n1,a1,v1),**CN**(n2,tn2))=**CHA**(n1,a1,v1);
Childof(n1,n2)=true =>
T(**CHA**(n1,a1,v1),**DN**(n2))=Nop;
Childof(n1,n2)=false =>
T(**CHA**(n1,a1,v1),**DN**(n2))=**CHA**(n1,a1,v1);
T(**CHA**(n1,a1,v1),**CA**(n2,a2))=**CHA**(n1,a1,v1);
Eqn(n1,n2)=true, Eqa(a1,a2)=true =>
T(**CHA**(n1,a1,v1),**DA**(n2,a2))=Nop;
Eqn(n1,n2)=false =>
T(**CHA**(n1,a1,v1),**DA**(n2,a2))=**CHA**(n1,a1,v1);
Eqa(a1,a2)=false =>
T(**CHA**(n1,a1,v1),**DA**(n2,a2))=**CHA**(n1,a1,v1);
T(**CHA**(n1,a1,v1),Nop)=**CHA**(n1,a1,v1);

%T(Nop,i)
T(Nop,i)=Nop;

conjectures:

%C1
St.i.T(j,i) = St.j.T(i,j);

%C2
T(T(k,i),T(j,i)) = T(T(k,j),T(i,j))