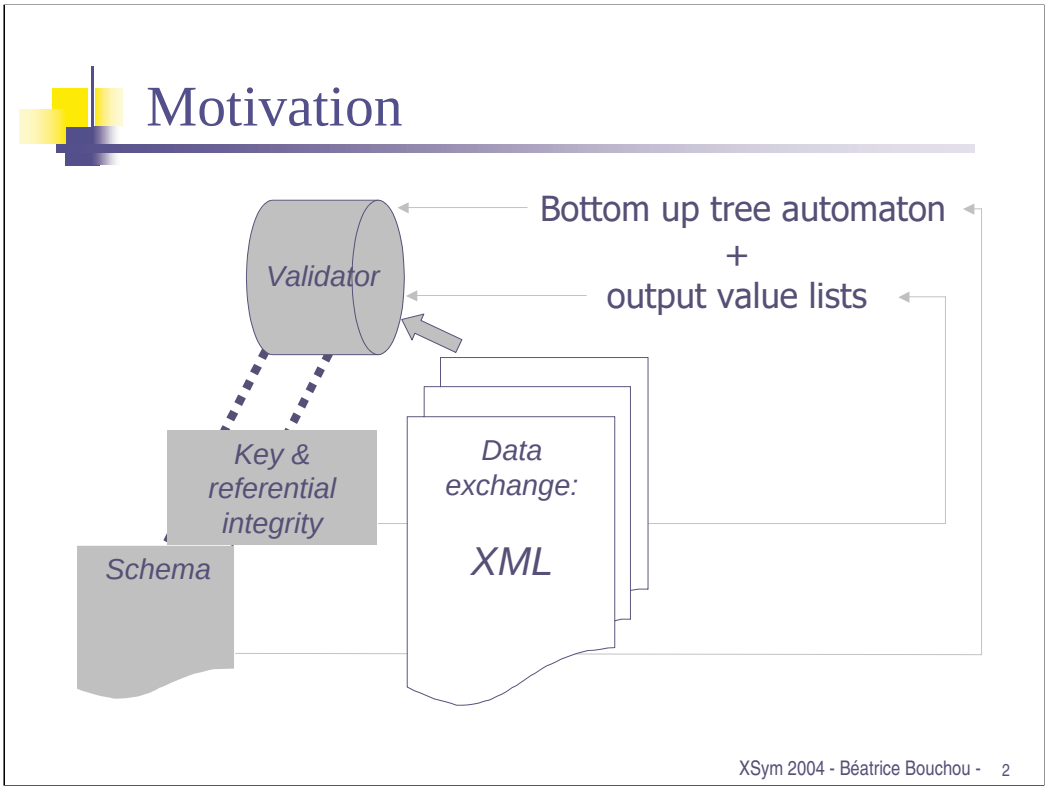
Incremental Constraint Checking for XML Documents

Maria Adriana ABRÃO[1], Béatrice BOUCHOU[1],
Mírian Halfeld FERRARI[1], Dominique LAURENT[2],
Martin MUSICANTE[3]

[1]Université de Blois-Tours-Chinon – LI, France
[2]Université de Cergy-Pontoise – LICP, France
[3]Universidade Federal do Paraná – DI, Brazil

I will present a work developped at Blois in France.

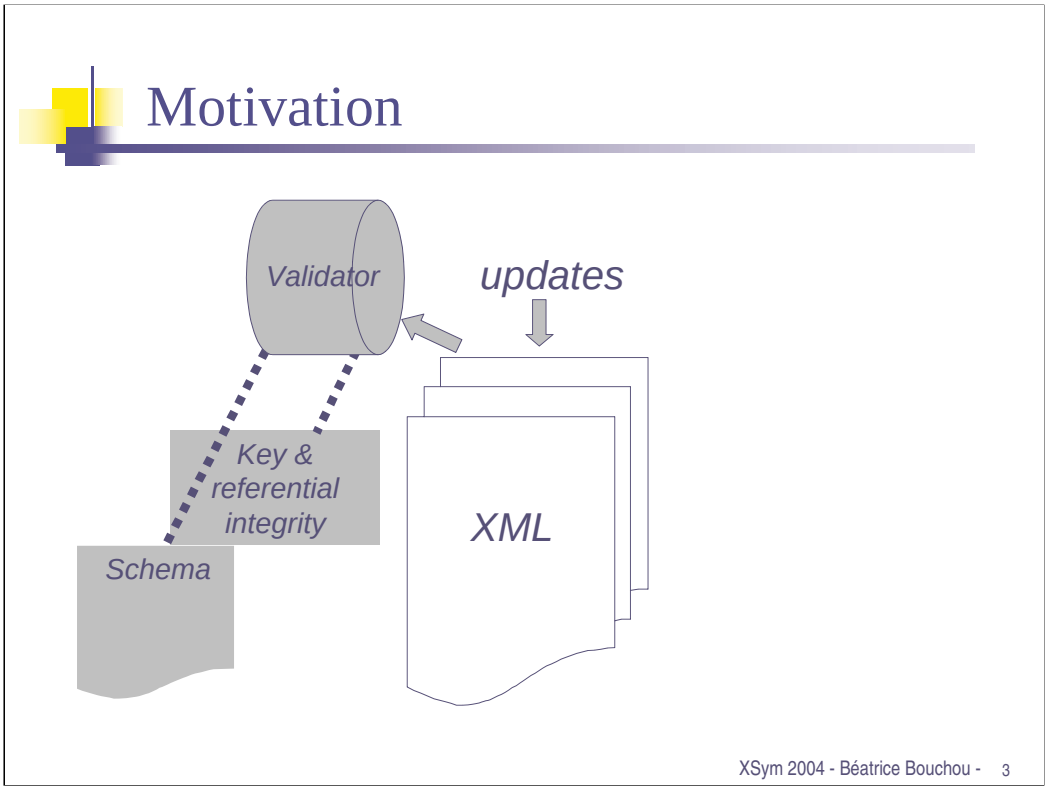
We are interested in incremental update of XML documents, in presence of schema, key and foreign key constraints.



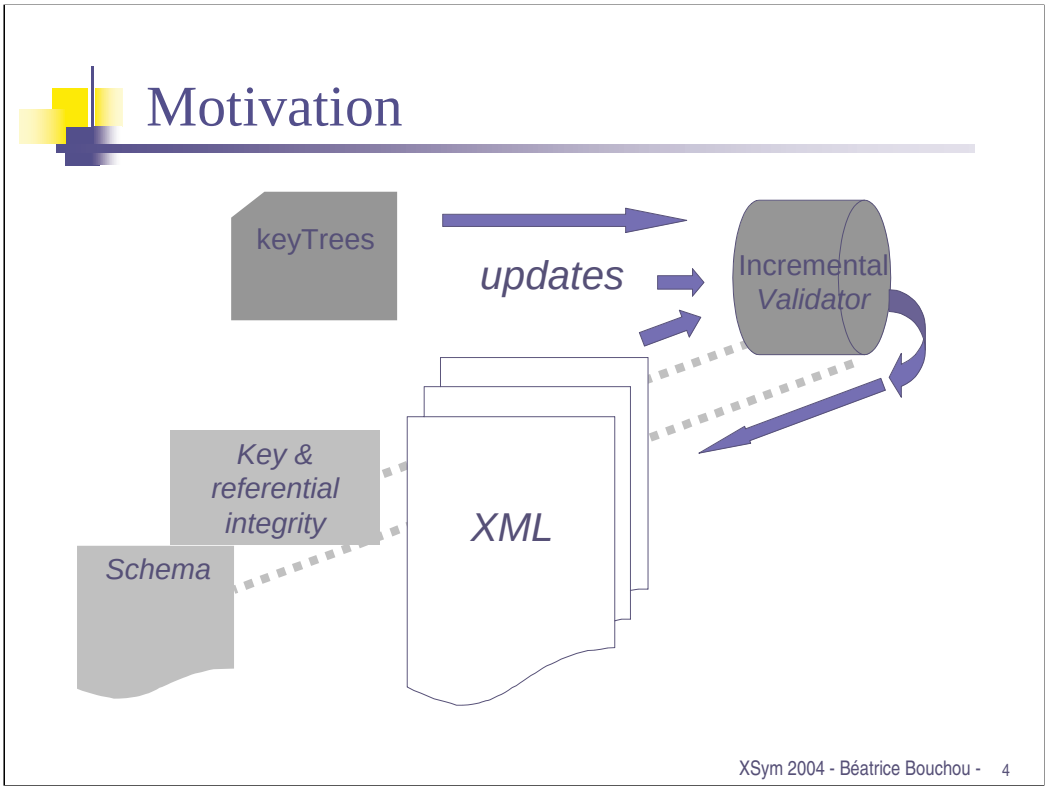
We have first designed a schema validator from a given DTD. In a very classical way, it is a bottom up tree automaton, which is implemented as usual with SAX.

Latter we have considered integrity constraints, as a complementary specification for XML documents. For this purpose, we have integrated key and referential integrity verification in the schema validation process.

Then we verify integrity constraints in a bottom up framework, carrying up output lists of values up to the nodes where verification can be performed.

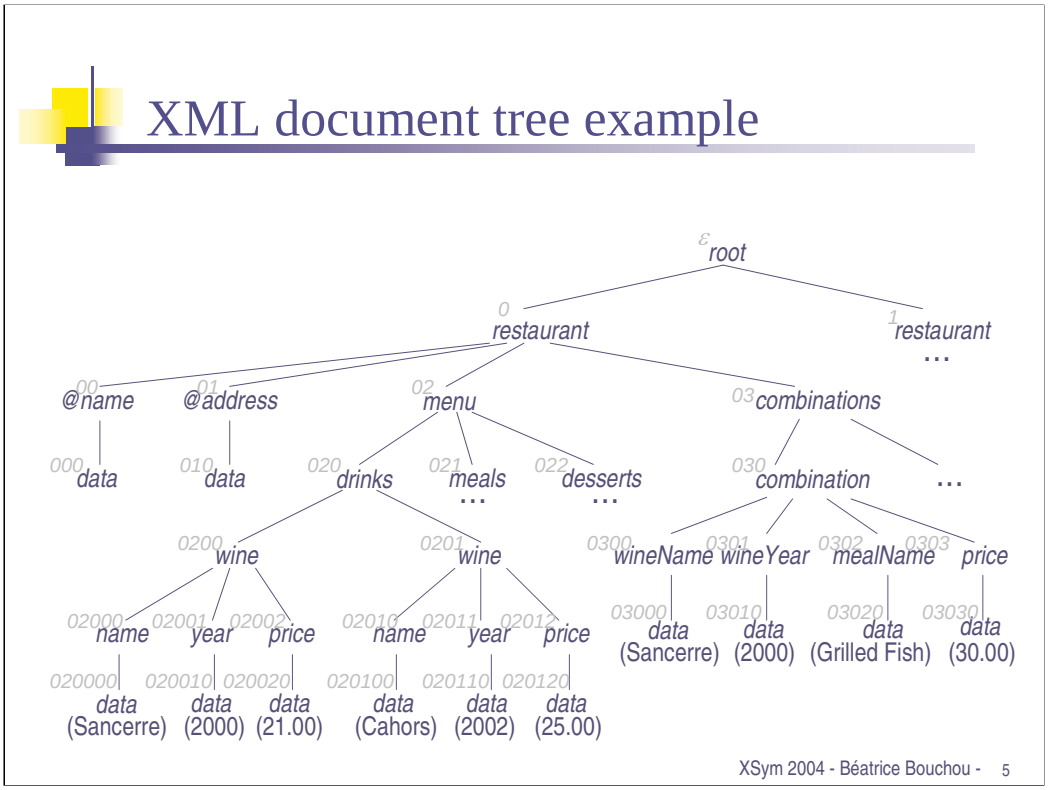


Valid documents, consistent with some keys and some foreign keys, may be updated. In that case one can have to verify if they still be consistant with all constraints.



Moreover, when one have to maintain valid documents, it becomes important to incrementally verify constraints, at each update.

In order to check referencial integrity as well as keys when documents are updated, we use auxiliary data structures that we call keyTrees.



Here is the example I will use all along this talk:

It is a document which desribes menus and combinations in some French restaurants. They contain « a la carte » choices, or combinations, that are restricted groups of dishes and drinks.

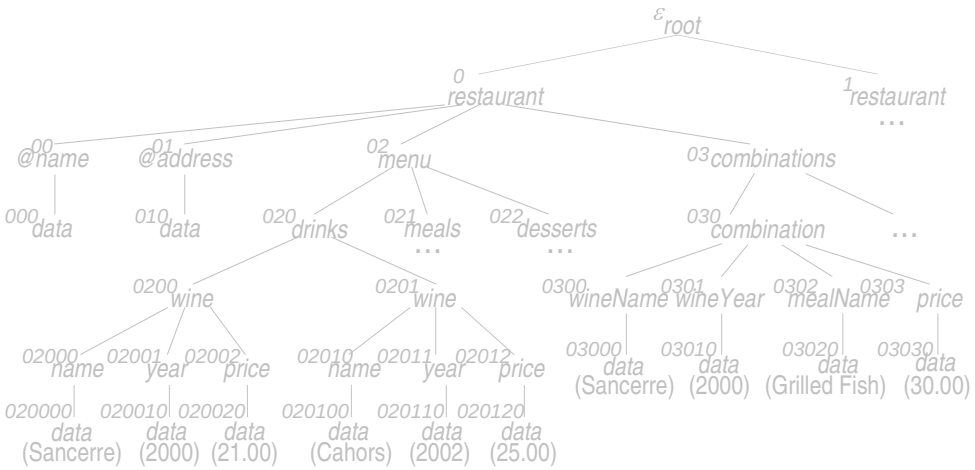
The document is a tree, each node having a position, a label, a type and sometimes a value.

Notice that we treat attributes, but I will disregard their specificity here.

Key (and Foreign Key) Constraints

[WWW 2001 – P. Buneman, S. Davidson, W. Fan, C. Hara, W. Tan]

- K_1= (/restaurant, (./menu/drinks/wine, {./name, ./year}))



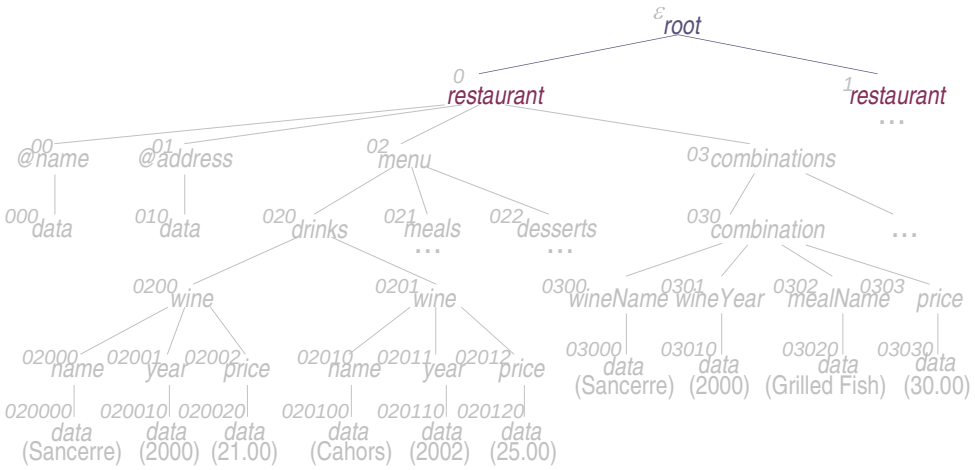
Using the syntax of Buneman et al., keys are written with paths in the document:

Here, to assign a key to a wine element in a menu of a restaurant we write these four paths.

Such a definition is read as follows:

In the context,

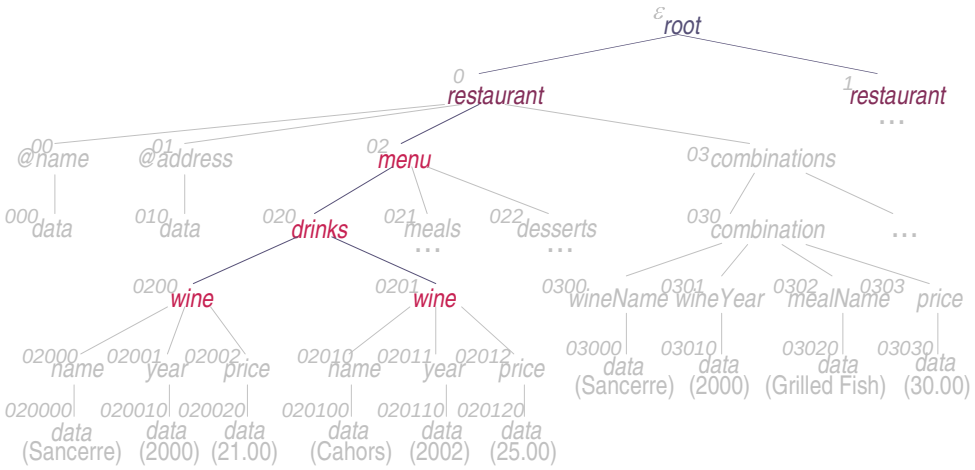
- K_1= (/restaurant, (, { }))



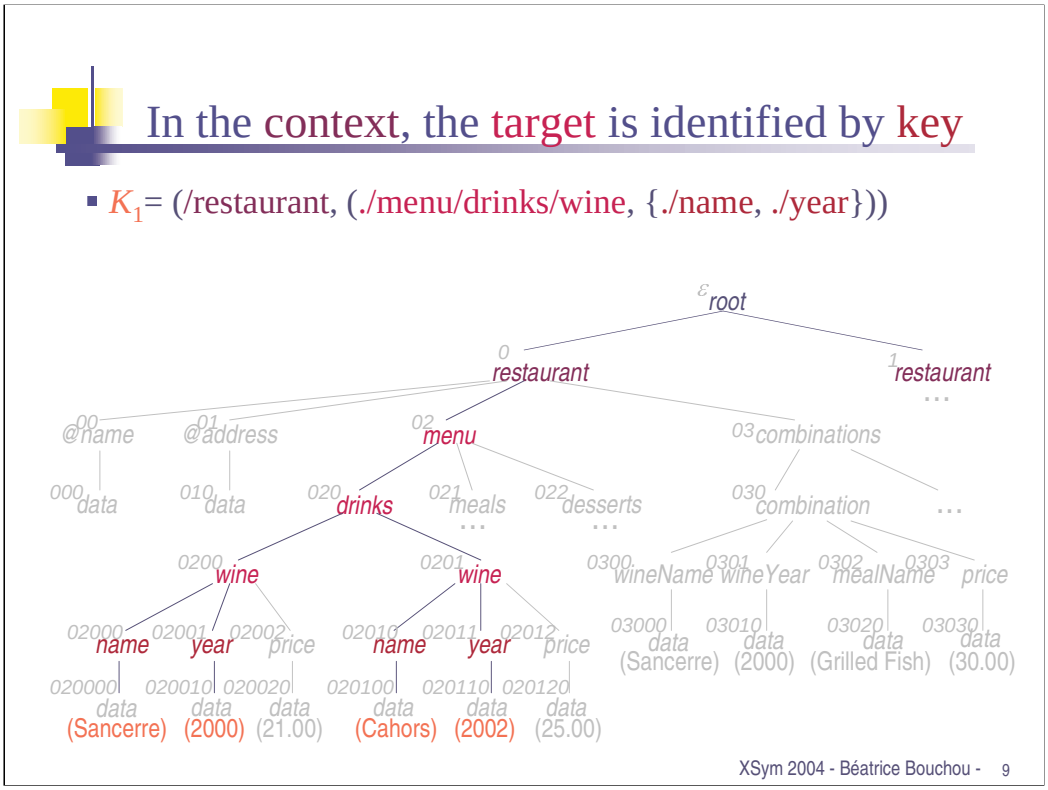
The first path is called the context path: it defines in which context the key is verified.

In the context, the target

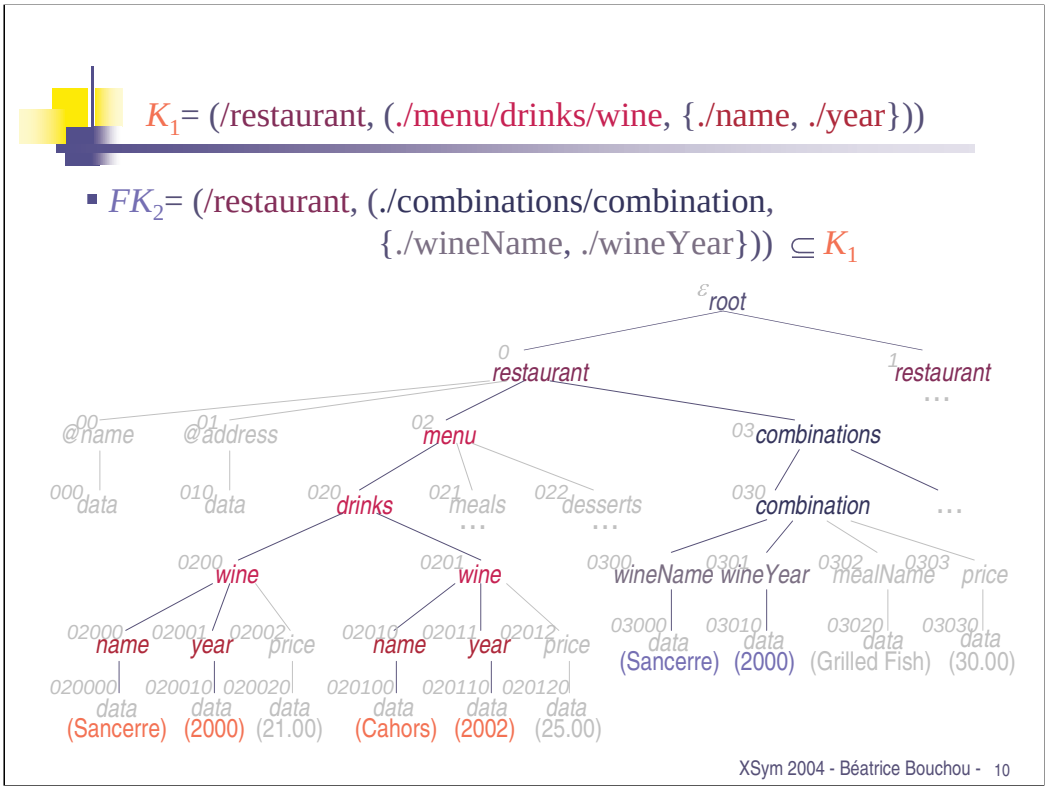
- K_1= (/restaurant, (./menu/drinks/wine, { }))



The next path is called the target path and defines target of the constraint, i.e. which element is represented by the key.



The last component is a set of paths, each one ending at a node associated with a value. It defines the key tuple, here the name and the year of a wine.

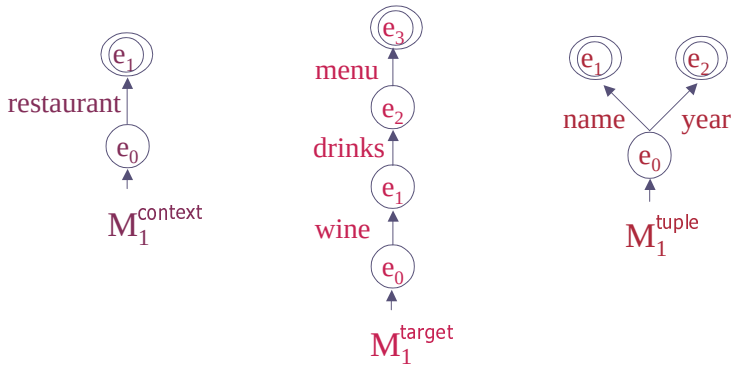


Foreign keys are defined in a same way: here our foreign key denotes that, in a restaurant, the wine proposed in any combination should appear in the menu of the same restaurant.

A foreign key has to reference one key in the same context: this key is specified with this set inclusion notation.

FSA for keys

K_1= (/restaurant, (./menu/drinks/wine, {./name, ./year}))



As I was saying, we verify integrity constraints in a bottom up way, together with schema validation. For that purpose we build 3 finite state automata for each constraint:

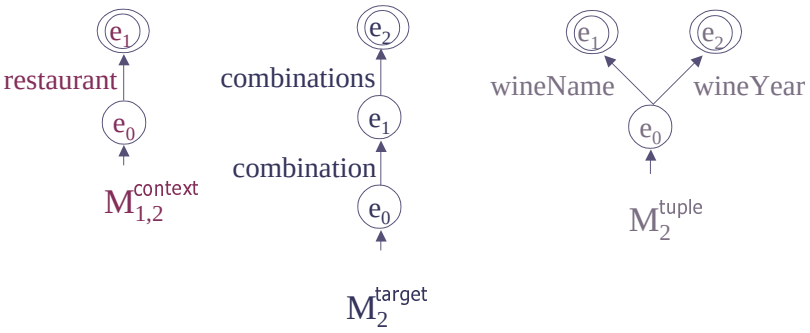
As you can see, the first one represents the context path in reverse: when in initial state e0, encountering the label node « restaurant » it fires the transition to the final state e1;

In a same way, the second one represents the target path, and the last one represents key tuple paths.

These automata allow to know, for each node, its potential role in the key.

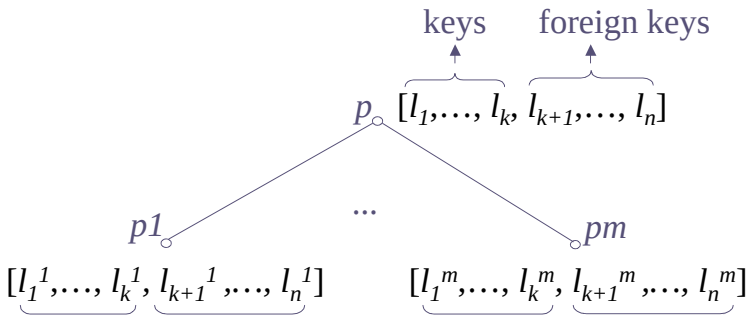
FSA for foreign keys

FK_2= (/restaurant, (./combinations/combination, {./wineName, ./wineYear})) $\subseteq K_1$



It is exactly the same for a foreign key: we will use succesively the foreign key tuple automaton, then he target automaton and lastly the context automaton, to select at each node the values to be carried up in order to perform referential verification.

Output lists, computed at each node



$l_1 = f_1(p, [l_1^1,\ldots,l_1^m])$

...

$l_n = f_n(p, [l_n^1,\ldots,l_n^m])$

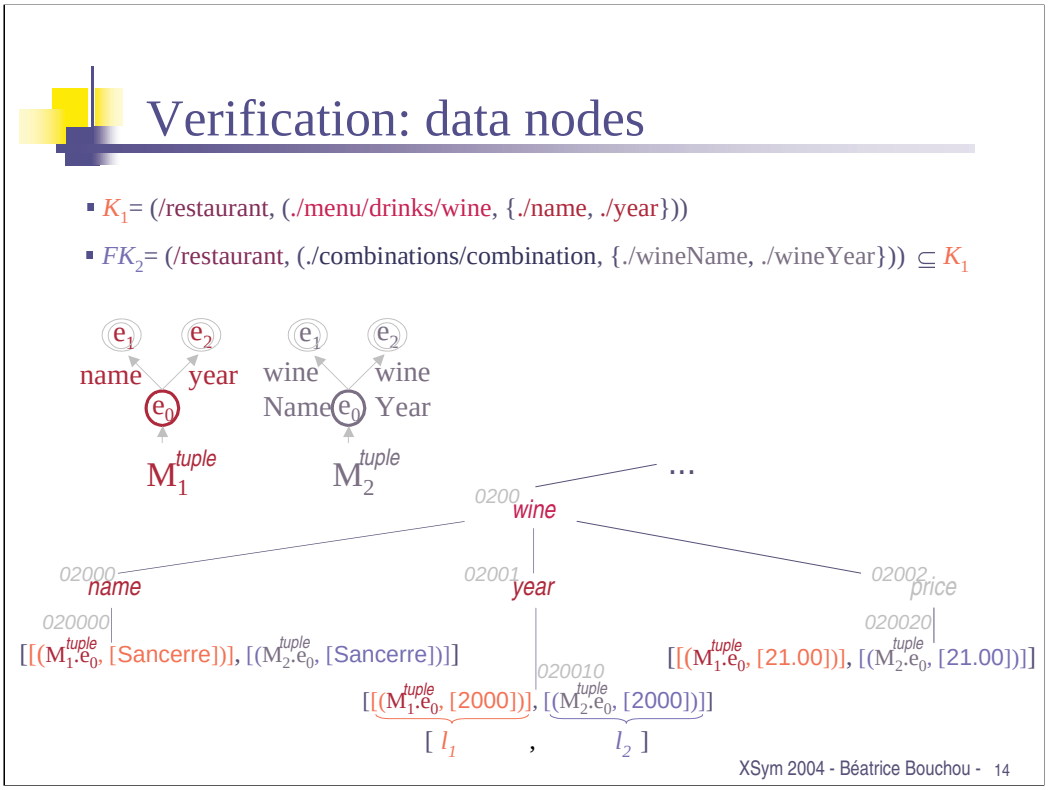
XSym 2004 - Béatrice Bouchou - 13

Our validator runs bottom up: for each position it computes its state (in the tree automaton, which represents schema constraints), together with its output list, which represents integrity constraints.

The output list is composed by one list for each key and each foreign key. First lists are for keys and last ones are for foreign keys.

Each list is computed by applying a function on values coming from children lists: for instance here, l1 is computed with the function of the first key, f1, applied on values coming from l11, l12, l1m.

If a node does not play any role in a key then the corresponding list in its output is empty.



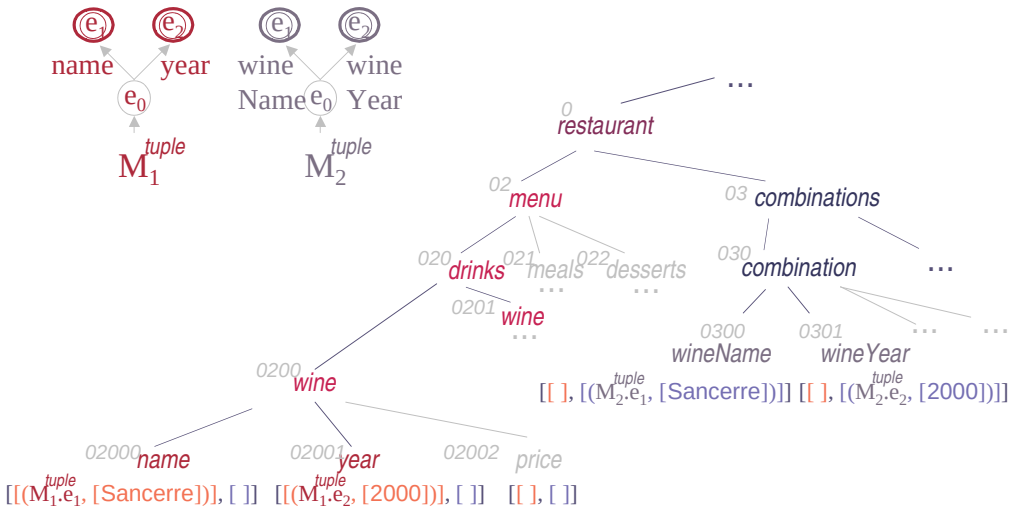
Concerning our key and foreign key example, list l1 contains information associated to the key and list l2 contains information about the foreign key.

Notice that each value list is preceded by an automaton configuration: this is to select those values which play a role in key or foreign key.

Here you can see that tuple automata are used, and that they are in their initial state.

At the beginning of the validation, all data nodes transmit their value, as illustrated here.

Verification: key and foreign key items

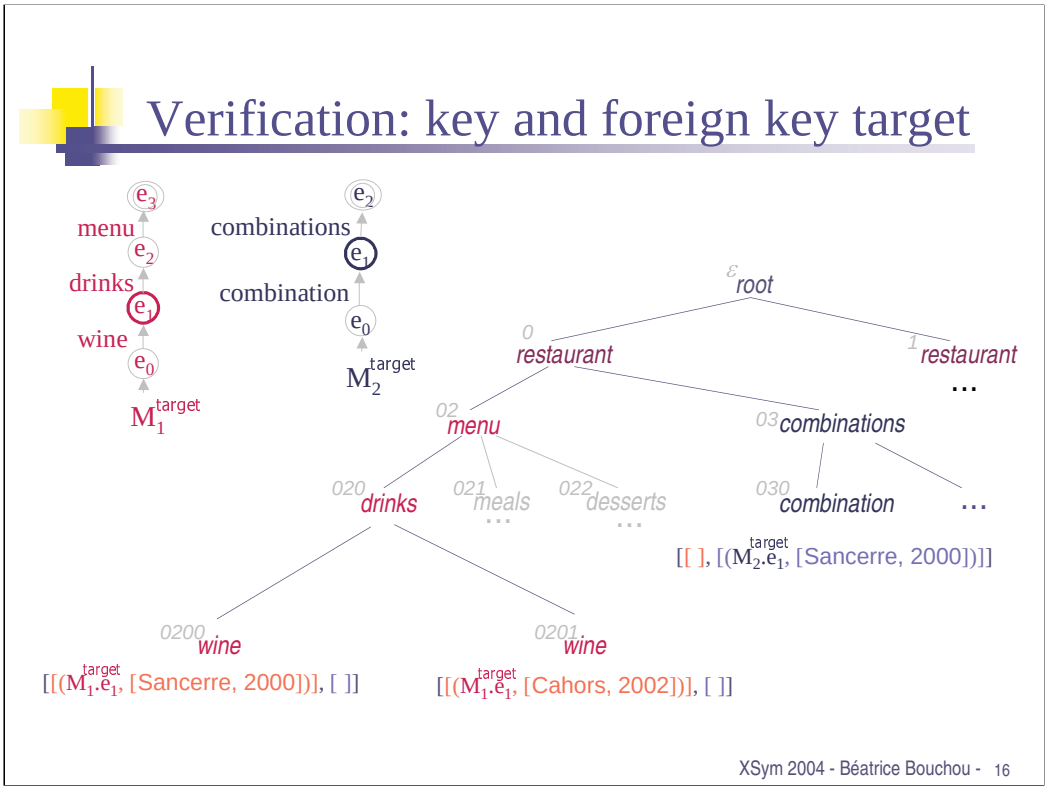


On top of data nodes, only key item values are kept:
they are selected in pairs whose (key) automaton can apply a transition.

Node that don't play any role nor in key neither in foreign key have only empty lists in their output.

Here you can see that wine name and year values are collected for the key, and also for the foreign key.

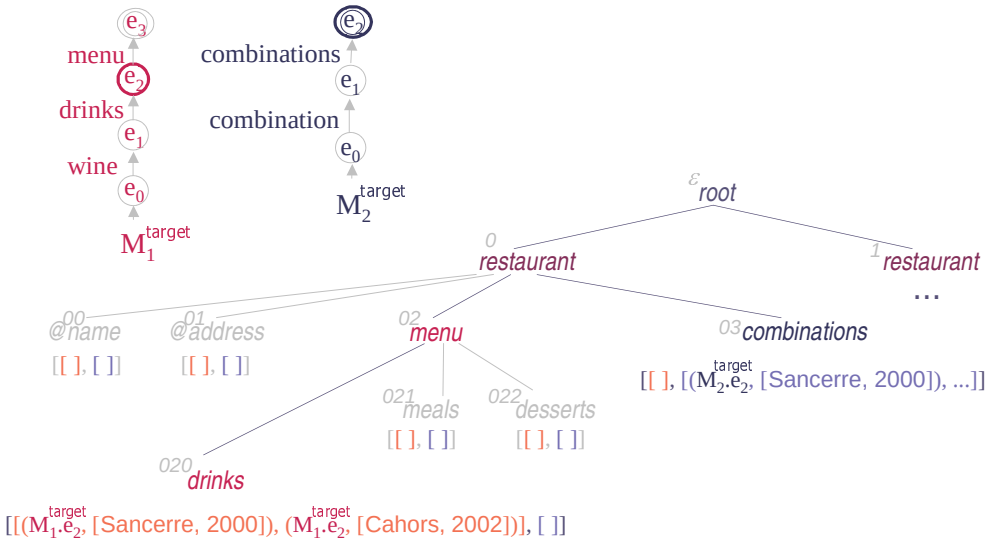
Notice that now configurations are tuple automata in their final state, which means that key nodes (and foreign key nodes) have been reached.



At target nodes, values composing keys are selected from pairs whose (key tuple) automaton is in a final state. These values are merged to form tuple of key values.

Configurations associated with target nodes are target automata in their second state (as the target label has been read).

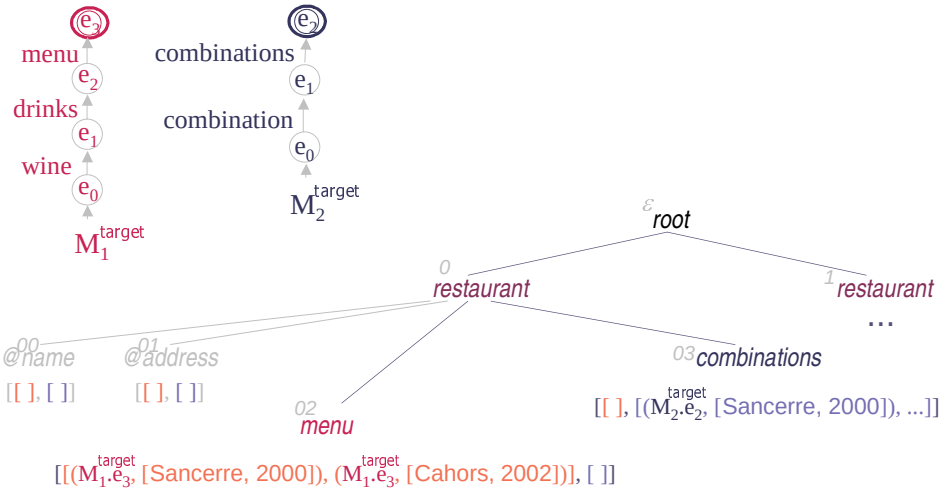
Verification: going up



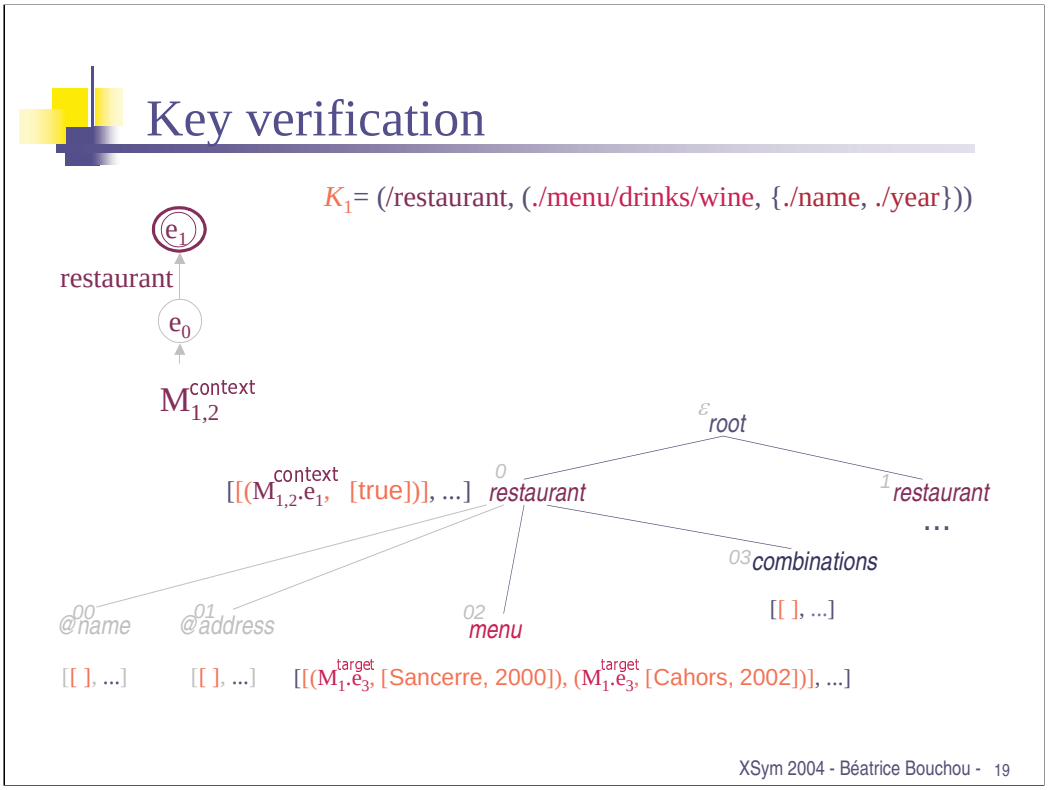
Values are carried up along the path, always selecting pairs whose automaton can apply a transition.

Here, the foreign key target automaton is already in its final state.

Verification: at context level



When all children output of a context node are computed, the output for this node can be built testing values preceded by target finite state automata in final state.



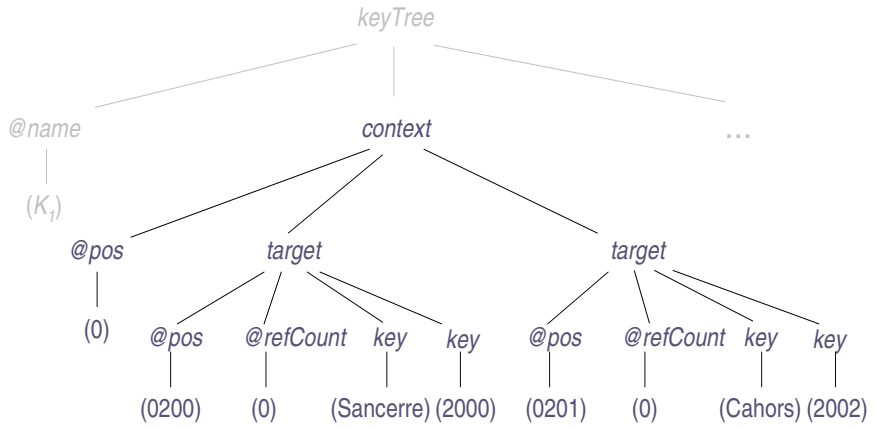
First, the configuration is the context automaton in its second state (« restaurant » have been read).

Then, keys are verified: if each tuple value has the good lentgh and is unique then the output value is true, otherwise it is false.

For our example key, in that context, the output value is true.

In parallel with the whole validation process, keyTrees are built:

Index KeyTrees



they are auxiliary data structures where key tuple values are stored, so as in an index.

They contain the key name and the list of context nodes for this key in the document.

For each context we store its position together with the list of targets in this contex.

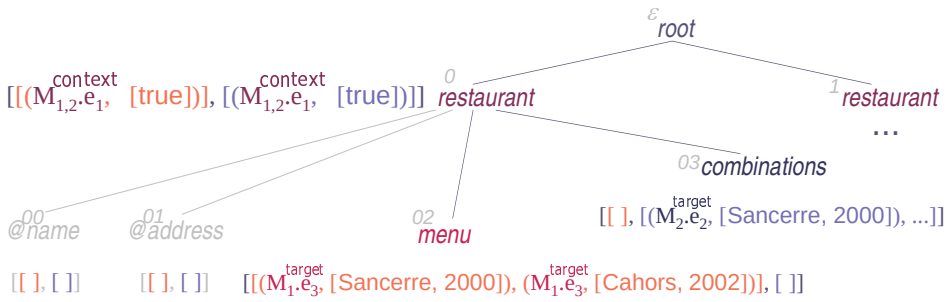
For each target we store its position, its key tuple values and a reference counter.

KeyTrees are used to verify foreign keys and, moreover, to perform incremental checking.

In our example, this context element is generated for the key.

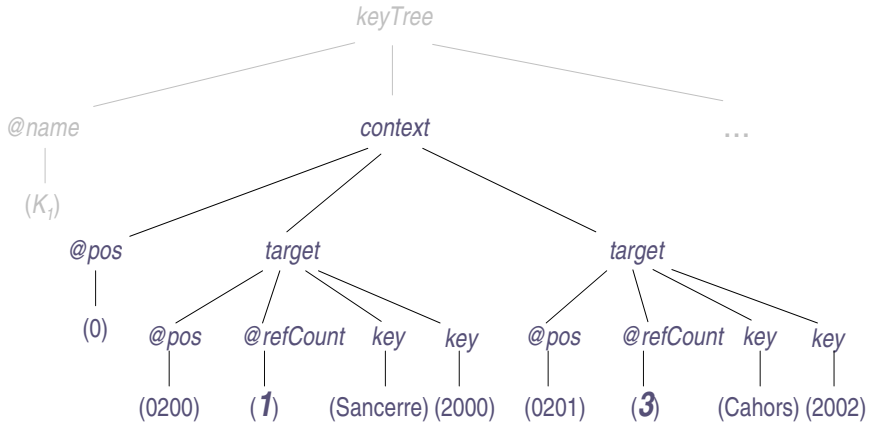
Foreign key verification

FK_2= (/restaurant, (./combinations/combination, {./wineName, ./wineYear})) $\subseteq K_1$



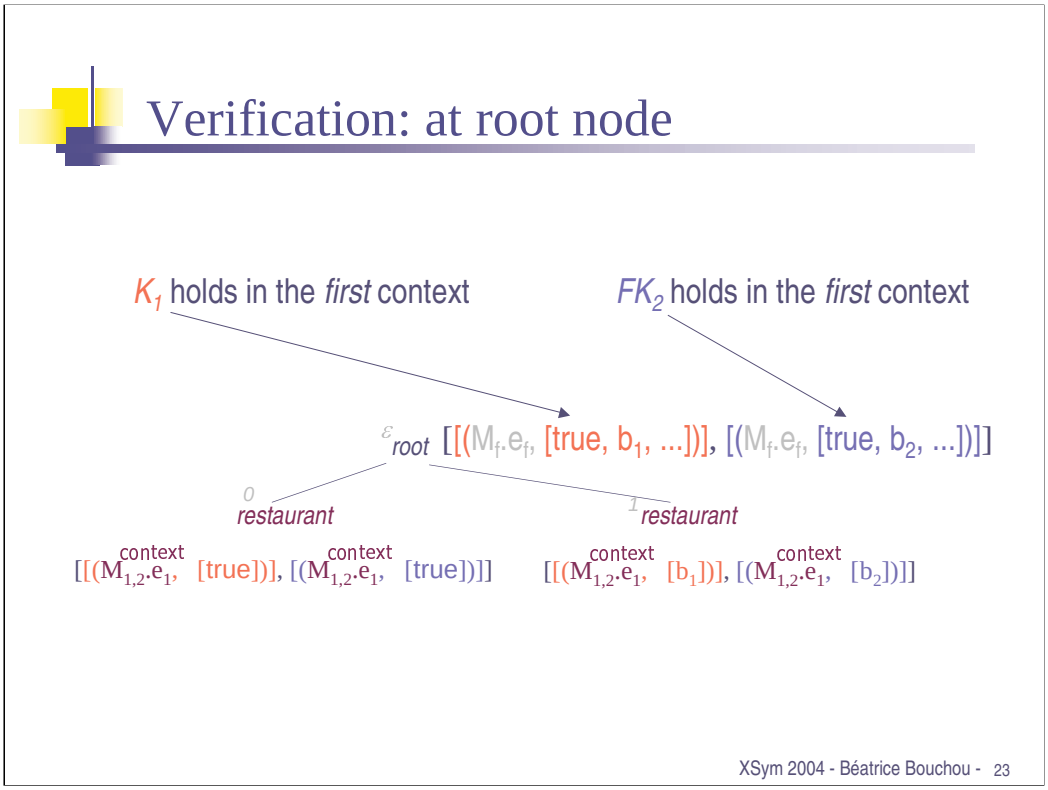
Once keys have been treated, foreign keys are checked: each tuple foreign key value must exists in the same context as a key tuple value.
For our example, the output value is true, once again.

Index KeyTree K_1: *FK* references



This is checked using corresponding keyTree: if the condition holds then reference counter for that key tuple value is incremented.

For instance here, you can see that there are 1 foreign key Sancerre 2000 and 3 foreign keys Cahors 2002 in the context 0.

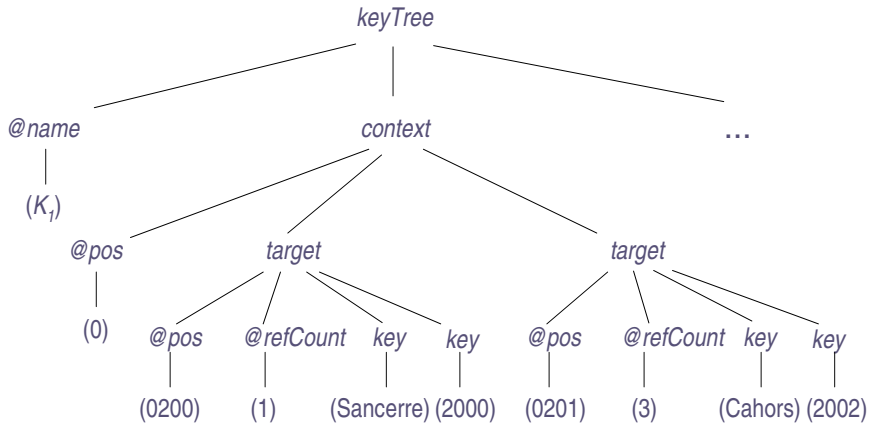


Boolean values generated at context nodes are carried up to root node: output list of root node is illustrated here for our key and our foreign key.

As usual, accurate values are selected considering configurations in each children output list.

First item in each list denotes information on first context, second item is for second context, and so on.

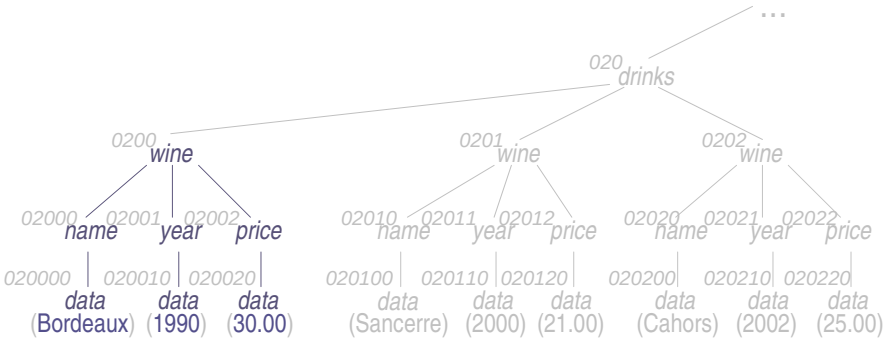
Finalized index KeyTree K_1



The keyTree is finalized at root node, for each key.

Insertion concerning a key

- Insertion of a new *wine* as first child of node *drinks*: if insertion is valid...



We will see now how these auxiliary structures are used in incremental checking.

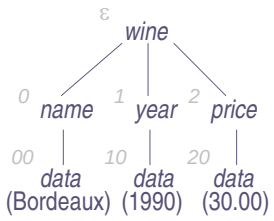
I ’ll focus on two update operations : insertion and deletion.

First, let ’s consider an insertion of a new wine as first child of node drinks in the first restaurant’s menu:

This operation is performed only if it keeps the document valid wrt schema and integrity constraints.

Incremental check of insertion (key)

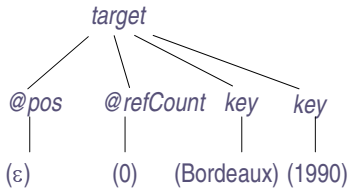
- local validation of *t'*:



State at ε: q_{wine}

Output list at ε: $[[(M_1.\dot{e}_1^{target}, [Bordeaux, 1990])], [\,]]$

keyTree K_1:



XSym 2004 - Béatrice Bouchou - 26

This condition is checked incrementally, that is, only the part concerned by insertion is revalidated.

The first step is to run the validator on t': this local validation ensures that a state of the tree automaton can be associated to the root.

Several conditions are also checked for keys and foreign keys, depending on the root position wrt to key, target and context:

(i) If the root of t' is a target node, local validation verifies wether the number of values in key lists corresponds to the number of key nodes,

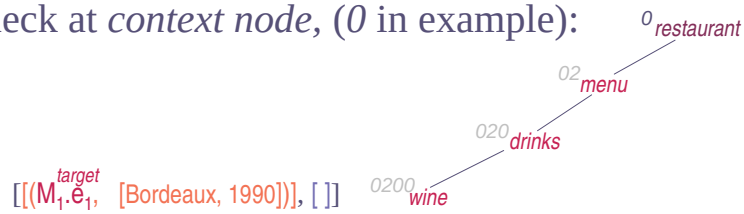
(ii) If the root of t' is a context node, or above, then local validation verifies that there is no key tuple duplicates in t' and that foreign key tuple values reference existing key tuple values.

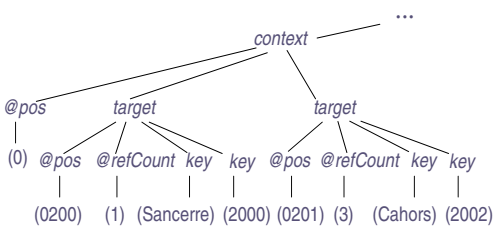
Moreover, local parts of keyTrees are built during local validation.

In our example, the root of t' is a target for the key, so you can see that we get two values, (Bordeaux, 1990) in K1 output list and that a target element has been built for keyTree K1.

Incremental Check of Insertion (key)

- Check at *context node*, (*0* in example):





In *keyTree* K_1, *at context* 0, there is no key tuple value equal to [Bordeaux, 1990].

The last step, if conditions of local validity hold, is to consider the update done, without performing it yet, and to verify local implications in the original document.

As tests must be performed at context level, we first determine, *for each key* the context of the update position.

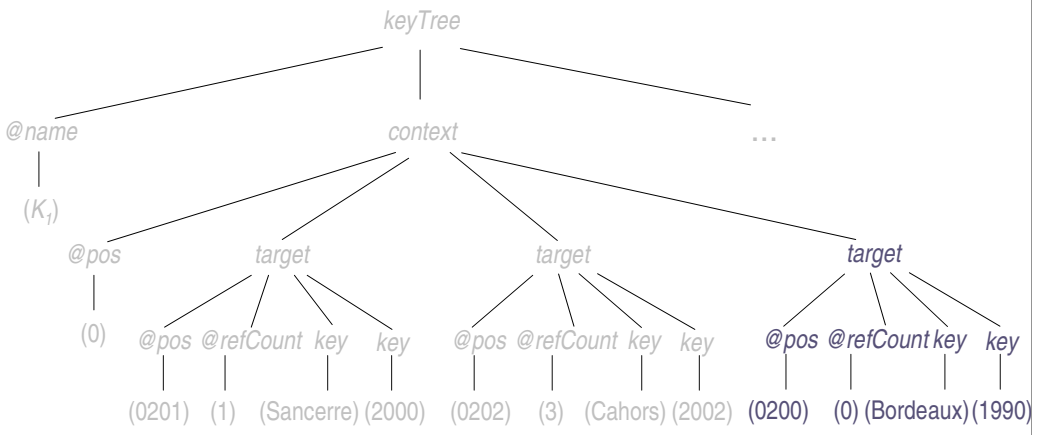
If insertion position is a context or above a context then the local validation implies that the insertion is ok concerning this key and its foreign keys.

Otherwise output lists coming from insertion position are carried up to the context, where they are checked: in particular key tuple values should not have duplicates in the corresponding keyTree.

In this example the insertion is valid, so it can be performed.

Performing Insertion (key)

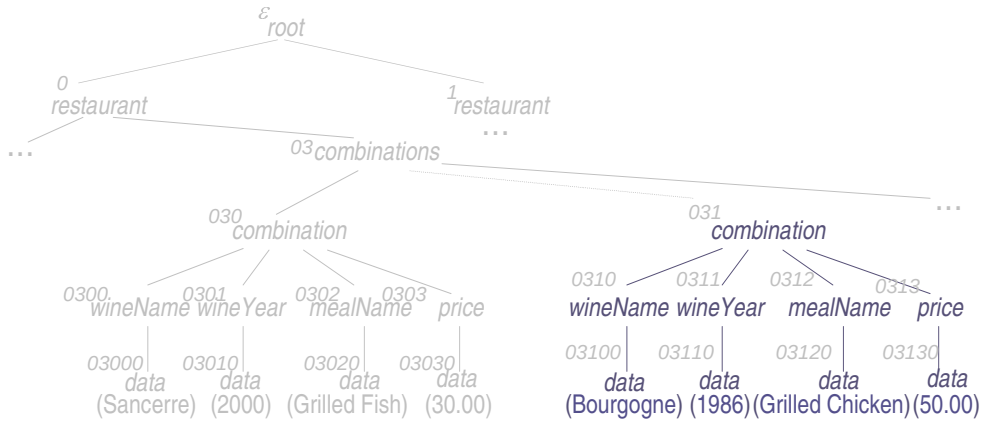
- Updated *keyTree* K_1:



In that case, the document is updated as well as keyTrees. You see here the new target.

Insertion concerning a foreign key

- insertion of a new *combination* in the first *restaurant* (if insertion is valid):

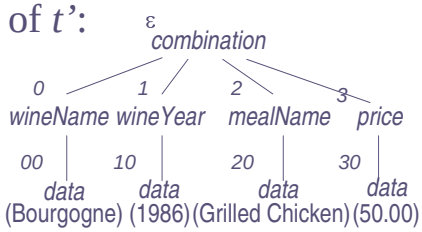


Now for another case of insertion, concerning foreign key:

Suppose we want to add this new combination in the first restaurant, having as drink a Bourgogne 1986.

Incremental Check of Insertion (foreign)

- local schema validation of *t'*:



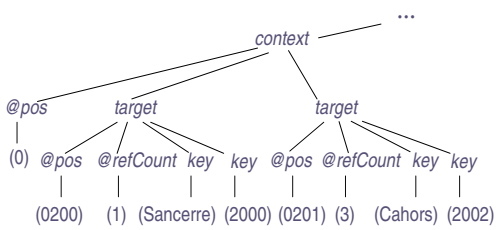
State at ε: $q_{combination}$

Output list at ε: [[ ], [($M_2^{target}.e_1$, [Bourgogne, 1986])]]

The inserted subtree is valid for the schema, and local validation gives these output lists at its root: nothing concerning the key, and one tuple for our foreign key.

Incremental Check of Insertion (foreign)

- Check at *context node*:



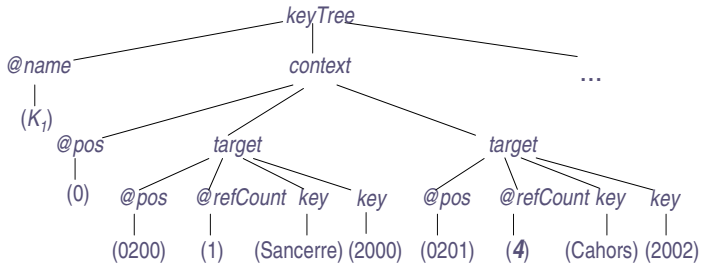
In *keyTree* K_1, *at context* 0, there is no key tuple value equal to [Bourgogne, 1986].

As in the preceding example, the context node of this update is at position 0.

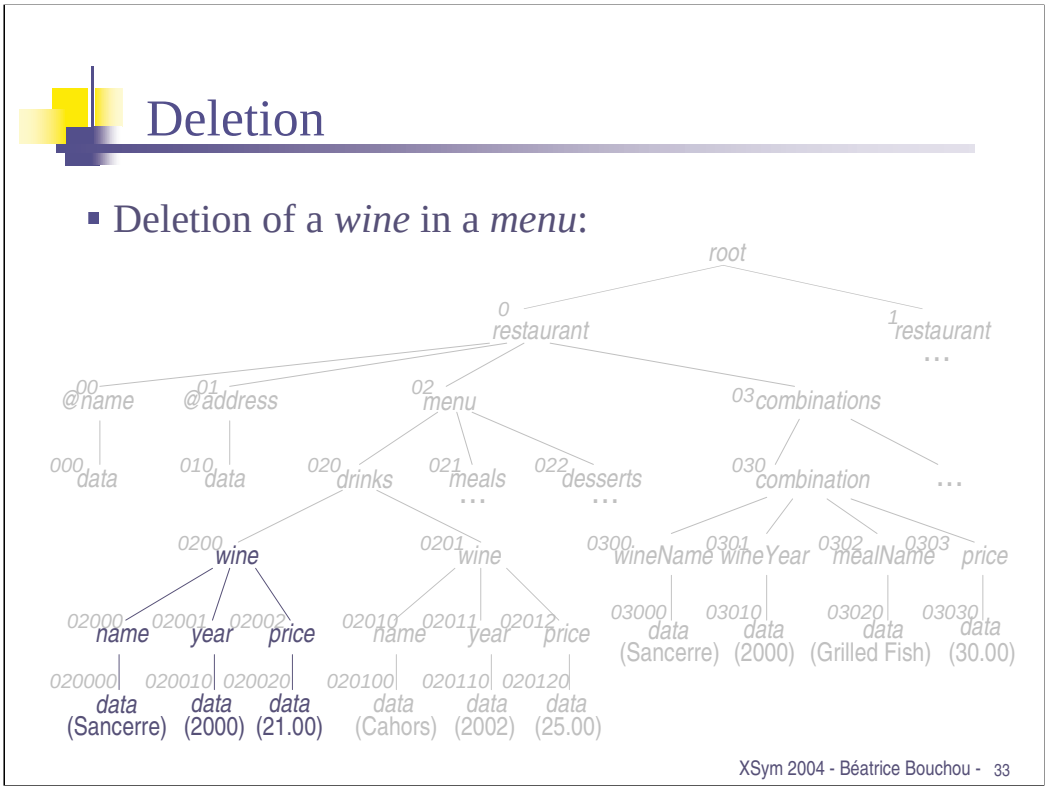
Now we can see that there is no key tuple values in keyTree K1 corresponding to the inserted foreign key tuple values, and then this insertion will be rejected.

If performing insertion (foreign key)

- If wine tuple values would have been [Cahors, 2002] then the insertion would have been accepted and
- *refCount* for target *0201* would have been incremented in *keyTree* K_1.



Notice that if foreign key tuple values would have been existing key tuple values, then auxiliary structures are also updated.

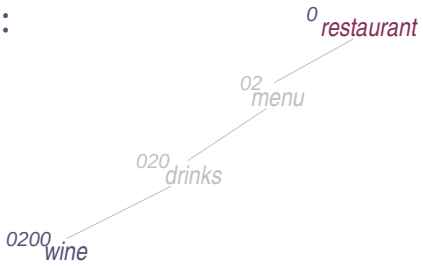


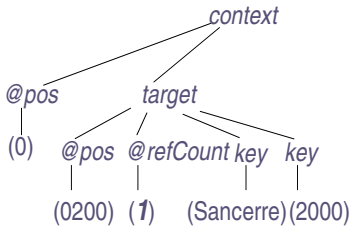
The last example is a case of deletion:

Suppose we want to suppress the wine Sancerre 2000 from drinks in the first restaurant.

Incremental Check of Deletion

- Check at *context node*:





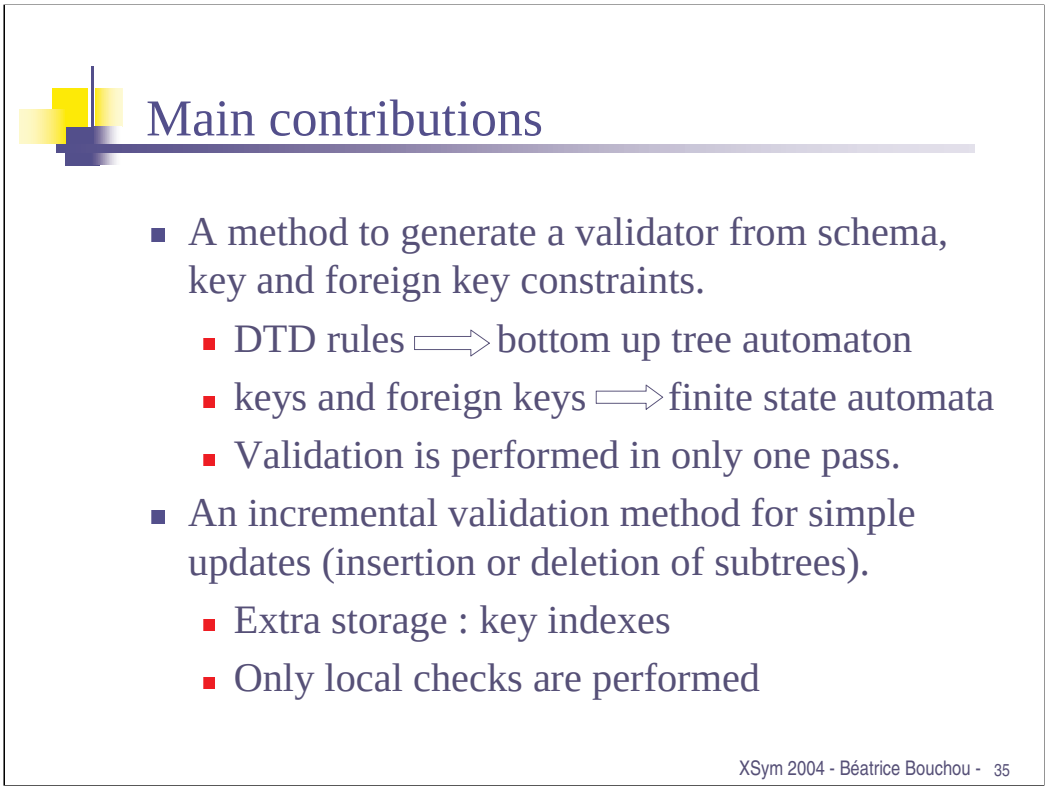
The delete position (*0200*) is a target in context *0* of *keyTree* K_1:
its *refCount* is not 0 so update is not valid.

We find the context node: here it is 0, and we scan the corresponding keyTree to see if the deletion concerns the key. A key tuple can be deleted if and only if it is not referenced by a foreign key tuple.

In our example, the delete position is a target in context 0.

As its refCount is not equal to 0, this update will be rejected.

Notice that a foreign key tuple can always be deleted but we have to update the corresponding key refCount.



- We have a method to build a validator from a DTD and a set of keys and foreign keys.

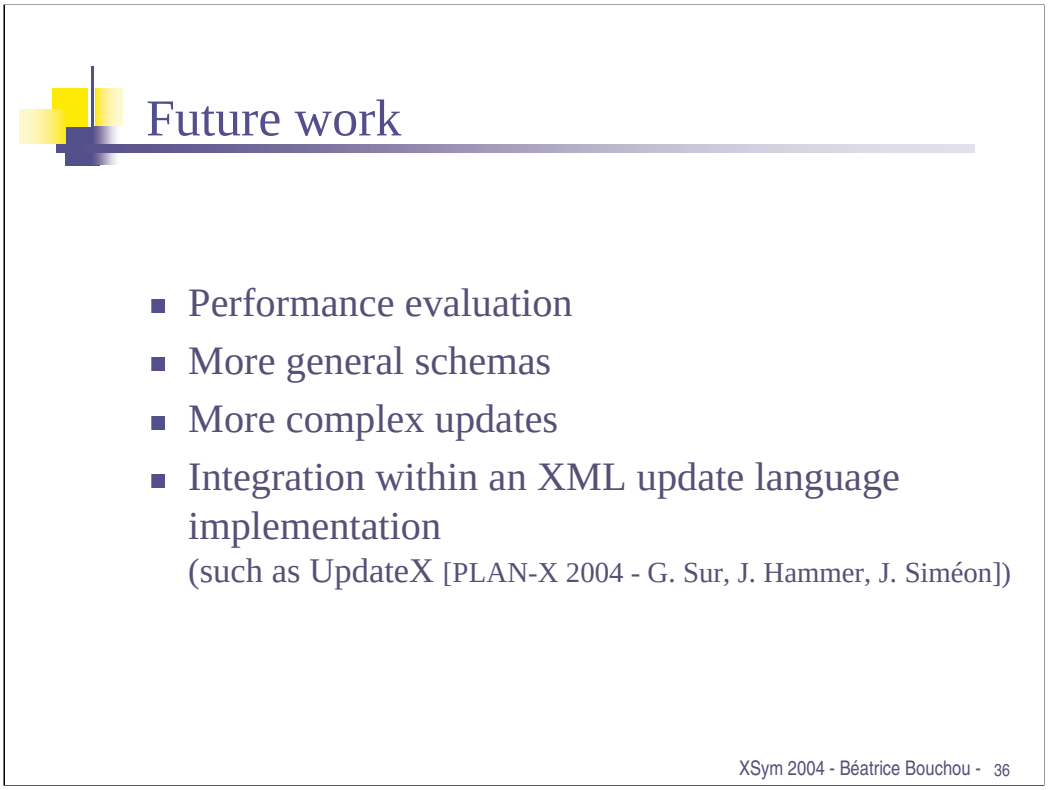
- The schema is translated into a bottom up tree automaton, while keys and foreign keys are represented with finite states automata, used to compute output values.

- The validator reads the whole document once.

- The validator is used to incrementally check updates on valid documents.

- For that purpose it builds and maintains key indexes.

- Incremental checks for keys and foreign keys are always bounded at context levels.



We'll soonly be able to present performance evaluation using existing XML benchmarks.

The extension to any kind of schema specification is done (and implemented in Java) for schema validation.

We are also working on more complex update transactions, including several simple updates, as proposed by an update language such as UpdateX.

Indeed, our incremental validator may be useful to XML update processors.