Specifying and implementing UI Data Bindings with Active Operations

Olivier Beaudoux, Arnaud Blouin, Olivier Barais, Jean-Marc Jézéquel

Specifying and implementing UI Data Bindings with Active Operations

Olivier Beaudoux
ESEO Group, GRI Team
Angers, France
olivier.beaudoux@eseo.fr

Arnaud Blouin
INRIA, Triskell Team
Rennes, France
arnaud.blouin@inria.fr

Olivier Barais
Jean-Marc Jezequel
Univ. of Rennes 1, Triskell
(barais, jezequel)@irisa.fr

ABSTRACT
Modern GUI toolkits propose the use of declarative data bindings to link the domain data to their presentations. These approaches work fine for defining simple bindings, but require an increasing programming effort as soon as the bindings become more complex. In this paper, we propose the use of active operations for specifying and implementing UI data bindings to tackle this issue. We demonstrate that the proposed approach goes beyond the usual declarative data bindings by combining the simplicity of the declarative approaches with the expressiveness of active operations.

ACM Classification Keywords
D.2.2 Software Engineering: Design Tools and Techniques—*Computer-aided software engineering (CASE), User interfaces*

General Terms
Algorithms, Design, Languages

Author Keywords
Data binding, active operation, GUI

1. INTRODUCTION

The problem of linking domain data to their presentations has appeared very early in computer science. The Model-View-Controller (MVC) design pattern has been introduced with the SmallTalk-80 language to formalize and implement such a linking [13]. The main drawback of this pattern is that the view is bound to a specific model: the view has to observe the model and refresh its state on model changes. Consequently, applications must adapt their domain data to the specific models bound to their possible views. The Java *Swing* API applies such a schema; for example, displaying a collection of elements within a *JTable* requires adapting the collection by implementing interface *TableModel*[8].

The concept of data binding can be seen as an evolution of MVC that avoids such a drawback. A data binding is a "controller" that binds the model and the view: it observes the model and updates the view whenever the model changes; conversely, it observes the view and updates the model whenever the view changes. The concept of data binding is very close to the controller of the PAC model where the Controller binds the Abstraction and the Presentation [6]. Recent GUI toolkits propose the data binding as the first-class object for linking models and views. However, as this paper will illustrate, they all suffer from two main limitations: 1) they offer a limited expressiveness so that hand-programming is often required when the complexity of the bindings grows; 2) they are platform-dependent implying that a data binding written within a given GUI toolkit must be entirely rewritten to be used within another one.

In this paper, we propose the use of active operations for specifying and implementing UI data bindings. The concept of active operation extends the usual concept of operation by allowing the result of an operation to be re-evaluated afterward. For example, usual operation $b := a.select(f)$ constructs collection b containing each element of collection a that satisfies predicate f [18]; the active version of this operation does more: it adds into b (respectively removes from b) any element newly added into a (respectively removed from a) that satisfies f. Active operations formalize and generalize our initial work on active transformations in the context of GUI [3, 2]; the mathematical definition of action operations is provided in [4].

Our proposal aims at combining the simplicity of the declarative approaches with the expressiveness of active operations, which is achieved through: a *unified* and *platform-independent* language for specifying data bindings; the use of simplified *class diagrams* (Ecore documents) for representing both the models and the views; a formalism for *implementing* the specification of bindings independently from the final implementation language; a *compatibility* schema with GUI toolkits.

The remainder of this paper is structured as follows. Section 2 presents, through three concrete examples, our language used to specify data bindings with active operations. Section 3 explains why and how the speci-

fication of active operations must be translated before being implemented. Section 4 focuses on the implementation and execution of active operations on GUI platforms. Section 5 introduces works related to data binding. Section 6 complements section 5 by comparing our proposal with the declarative data bindings of three representative Rich Internet Application (RIA) toolkits. Section 7 finally concludes on our work and its perspective.

2. SPECIFYING ACTIVE OPERATIONS

This section introduces active operations through three complementary examples. These examples have been carefully designed to illustrate the expressiveness of active operations for specifying various UI data bindings. They have also been motivated to illustrate the limitation of usual binding mechanisms (see section 6).

The following specifications of active operations are expressed using our own domain specific language (DSL) that is close to the OCL language [18].

2.1 Example 1 - A Simple XML Editor

Figure 1 gives the model of a simple XML editor. The left part represents a simplified XML document model; the right part represents the model of a tree widget; the middle part represents bindings between these two models.

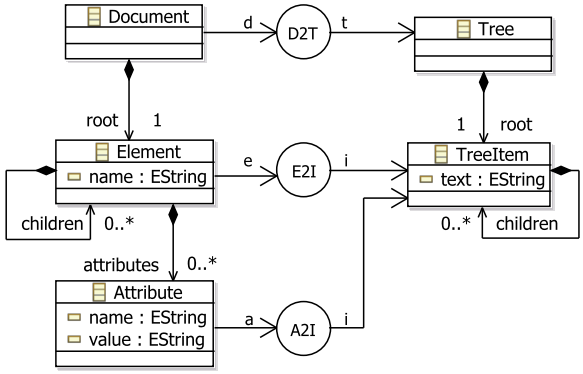

Figure 1. A Simple XML Editor

Binding *D2T* binds XML document *d*, instance of class *Document*, to tree widget *t*, instance of class *Tree*, as follows:

```
t.root := d.root.map(E2I)
```

Expression *c.map(f)* applies function *f* to each element *e* of collection *c*, and maintains a link (called a *trace*) between *e* and the element returned by *f(e)*. In the example, the root element of document *d* is bound to the root item of tree *t* through binding *E2I* defined as follows:

```
1  i.text := "<" + e.name + ">"
2  i.children :=
3      e.attributes.sortedBy(a | a.name).map(A2I) +
4      e.children.map(E2I)
```

The *text* of tree item *i* displays the *name* of element *e* surrounded by angle brackets. *Children* of item *i* result from concatenating *attributes* of element *e* sorted by their *name* and mapped to tree items (line 3), with *children* of element *e* recursively mapped to tree items (line 4). Finally, binding *A2I* displays the *name* and *value* of attribute *a* into tree item *i*:

```
i.text := "@" + a.name + "=" + a.value
```

All the previous bindings are *unidirectional*. The next example illustrates how active operations can be used to define *bidirectional* bindings.

2.2 Example 2 - A Directory Editor

Figure 2 gives the model of a directory editor. The left part represents a directory *d* of contacts; the right part represents the model of a list widget *l*; three text-fields *ff*, *lf*, and *pf*, are used to respectively edit the first-name, last-name and phone number of the contact selected from the list widget *l*; finally, a fourth text-field *tf* is used to filter the content of the list so that a user can quickly find a contact within a large directory.

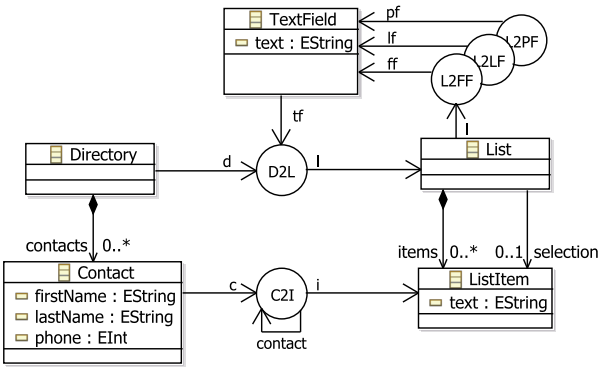

Figure 2. A Directory Editor

Binding *D2L* binds list of contacts *d*, instance of class *Directory*, to list widget *l*, instance of class *List*. The binding has a second argument *tf*, instance of class *TextField*, used to filter the list:

```
1  l.items := d.contacts
2      .sortedBy(c | c.lastName + c.firstName)
3      .select(c | tf.text.isEmpty() or
4                  c.lastName.startsWith(tf.text))
5      .map(C2I)
```

The list of contacts is first sorted (line 2); a simple string concatenation is here used to specify a sort on the *lastName* and then on the *firstName* of the contacts. The resulting list is then filtered through the *select* operation (lines 3-4): if text-field *tf* is empty, all the contacts are selected; otherwise, only contacts whose last-name starts with the text-field content are selected. Each resulting contact *c* is then mapped to a list item *i* by calling binding *C2I* (line 5), which is defined as follows:

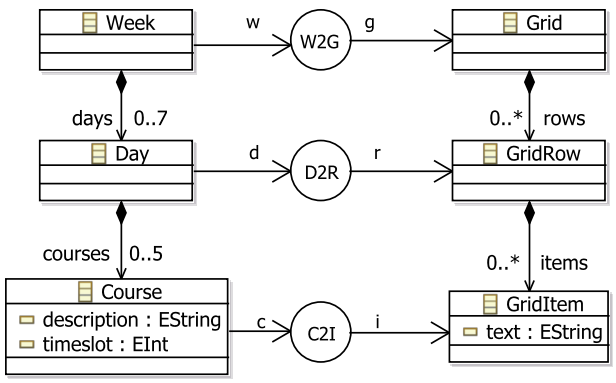

Figure 3. An academic calendar

```
1  i.text := c.firstName + " " + c.lastName
2  i.contact := c
```

This binding displays each contact in the list widget with their first and last names (line 1). A *trace* of binding *C2I,* represented by reverse arrow *contact* in figure 2, is then set (line 2).

Binding *L2PF* binds text-field *pf* used to edit the *phone* number to the selected item of the list *l*, as follows:

```
1  pf.text := l.selection.contact.phone.toString()
2  l.selection.contact.phone := pf.text.toInteger()
```

The *phone* number of the selected contact is converted into a string displayed into text-field *pf* (line 1); conversely, the *text* of text-field *pf* is converted into an integer that represents the phone number of the selected contact (line 2). Text-field *pf* must be designed to restrict the editing to integers; this must be achieved at the GUI platform level, but *not* by the binding itself. Binding *L2PF* is *bidirectional*: if the user changes the selection, the text-field is automatically updated; conversely, if the user changes the text-field content, the phone number of the selected contact is automatically updated. Infinite loop on such a bidirectional binding is avoided by only notifying changes on *new* values: if the new value does not differ from the previous one, no notification is performed by the property setter method.

Only path assignments, along with an optional conversion, can be used in a bidirectional way [4]. If no conversion is required between two properties, the double use of operator := defines a *bidirectional assignment* that can be shortened with the operator =, such as binding *L2FF* illustrates:

```
ff.text = l.selection.contact.firstName
```

Contrary to the previous binding, no conversion is required here. Binding *L2LF* is similar to *L2FF* for property *lastName*.

These bindings illustrate the bidirectionality feature of active operations; however, as within example 1, these bindings remain quite simple. The next example uses intensively active operations to achieve a complex binding.

2.3 Example 3 - An Academic Calendar

Figure 3 gives the model of an academic calendar. The left part represents a simplified model of a weekly academic calendar; the right part represents the model of an HTML-like table widget; the middle part represents bindings between these two models.

Binding *W2G* binds week *w*, instance of class *Week*, to grid widget *g*, instance of class *Grid*. It maps *days* of week *w* to *rows* of grid *g* by calling binding *D2R*:

```
g.rows := w.days.map(D2R)
```

Binding *D2R* binds day *d* to a grid row *r*, which consists of populating relation *r.items* from relation *d.courses*. Figure 4 synthesizes the principle of binding *D2R* through a simple example: collection *ts* contains all the predefined time-slots of the calendar (for example, time-slot 0 corresponds to slot 9:05-9:55am); day *d* contains two courses *a* and *b* respectively located on time-slots 1 and 4. The resulting grid-items *r.items* of grid-row *r* must contain empty grid-items (noted $i_\emptyset$) that represent the free time-slots, and non-empty grid-items (noted i_a and i_b) that represents courses *a* and *b*.

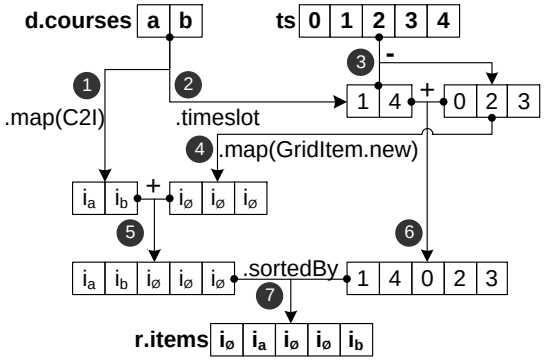

Figure 4. Principle of binding *D2R*

The collection (i_a, i_b) is computed by mapping courses *d.courses* through binding *C2I* ①. The collection of free time-slots $(0, 2, 3)$ is computed by subtracting *d.courses.timeslot*[1] equals to $(1, 4)$ ②, from collection *ts* representing all defined time-slots (0..4) ③. The collection of empty grid-items $(i_\emptyset, i_\emptyset, i_\emptyset)$ is computed by mapping collection of free time-slots with function *GridItem.new* ④. The two collections of grid-items are

[1]Expression *d.courses.timeslot* is a path, which is a specific operation detailed in section *3.1.*

then concatenated to form the collection of all grid-items $(i_a, i_b, i_\emptyset, i_\emptyset, i_\emptyset)$ ⑤. Such a collection is however not ordered accordingly to time-slots. Since time-slot numbers represent positions, the "order collection" $(1, 4, 0, 2, 3)$, resulting from the concatenation of a and b time-slots and the free time-slots ⑥, is used to sort the grid-items ⑦: operation *sortedBy* sorts collection $(i_a, i_b, i_\emptyset, i_\emptyset, i_\emptyset)$ with order $(1, 4, 0, 2, 3)$ that gives the position of each item after sorting.

The previous principle is quite complex due to the calendar model that does not divide a day into time-slots. However, this proposed model is a simplification of a calendar model that allows a time-slot to contain multiple courses and a course to span multiple time-slots. This simplified model has been designed to illustrate that active operations allow the definition of bindings between "something missing" (a time-slot without any course) to a UI object (an empty grid-item).

Binding *D2R* is specified with our DSL as follows, and illustrates the expressiveness of our DSL:

```
ts := seq(0 to 4)
courseTS := d.courses.timeslot
freeTS := ts − courseTS
items := d.courses.map(C2I) +
         freeTS.map(n | GridItem.new)
r.items := items.sortedBy(courseTS + freeTS)
```

Binding *C2I* finally binds the text of the not-empty grid-item i to the description of its corresponding course c, as follows:

```
i.text := c.description
```

3. TRANSLATING ACTIVE OPERATIONS

The previous section focuses on the specifications of bindings. However, as this section will illustrate, these specifications cannot be executed on a target platform as is: they must be *translated* into an active implementation. Such a translation uses again our own DSL by introducing new specific operations, thus making the translation process independent from the final implementation language and platform (see section 4). Moreover, some of the new operations motivated by the translation might be useful at the specification level, such as example 3 illustrates through its use of operation *sortedBy* (see section 2.3).

3.1 Paths

Paths extend the dotted notation of OOP so that the dot symbol can be applied more than once on collections [18]. For example, expression *d.contacts.firstName* represents a path that returns a *flattened* collection, as the OCL operation *collect* does [18], containing the first-name of all contacts of directory d. The dedicated operation *path* must be used for implementing path on collections on top of an OOP language; in binding *L2FF* of example 2, expression:

```
ff.text = l.selection.contact.firstName
```

must be *translated* into:

```
ff.text = l.selection
    .path(s | s.contact).path(c | c.firstName)
```

3.2 Observability

Usual loops can be used to implement usual operations on collections. For example, expression $b := a.collect(f)$ computes collection b by converting each element e of collection a into element *f(e)* within collection b; operation *collect* is easy to implement using a loop, as follows:

```
a.each(e, i | b.add(f(e), i) )
```

The previous code states that each element e at position i within collection a must be converted to the corresponding element *f(e)* at the same position i within b.

Making an operation active, such as operation *collect*, requires the *observability* of collections: an observable collection is a collection from which additions and removals can be observed [1, 9]. We have proposed in [4] to extend usual loops with *active loops* that manage collection observability and allow implementing active operations as usual loops do for implementing the usual operations. For example, the previous usual loop that implements operation *collect* can be easily translated to the following active loop:

```
1 a.eachAdded(e, i | b.add(f(e), i) )
2 a.eachRemoved(e, i | b.remove(i) )
```

This code states that: each element e *newly* added into a results in adding the converted element *f(e)* into b (line 1); each element e *newly* removed from position i within collection a results in removing the corresponding element from b (line 2).

Such a translation process from usual loops to active loops works fine for operations *collect*, *map* and *path*. However, it must be augmented for operations *select* and *sortedBy*, as explained in the next section.

3.3 Selection and sort

The following expression:

```
minors := persons.select(p | p.age < 18)
```

computes collection *minors* that contains persons p under 18. The active loop of such a *select* operation can be easily translated from a usual loop. This loop works fine whenever a person is added or removed from collection *persons*; however, it fails whenever the age of a person goes above 18: this last change is not captured by the active loop. We thus propose to *reify* predicate function f involved in expression *c.select(f)* into a *predicate collection* represented as a sequence of booleans. The predicate collection related to the previous example is defined by the following expression:

```
persons.path(a | a.age).predicate(a | a < 18)
```

If *persons* contains three people with ages 16, 42 and 12, this expression returns predicate collection $true, false, true$. By *overriding* operation *select*, the previous example should be translated into:

```
minors := persons. select (
    persons.path(a | a.age). predicate(a | a < 18)
)
```

The same problem arises with operation *sortedBy*. For example, binding *E2I* of example 1 sorts attributes of element *e* by their name:

```
e. attributes .sortedBy(a | a.name)
```

If an attribute *a* is added or removed from *e.attributes*, the active loop updates correctly the resulting collection. However, any change to *a.name* is not captured by the active loop. We thus propose to *reify* the order function *f* involved in expression *c.sortedBy(f)* into an *order collection* represented as a sequence of integers. In the previous example, the order collection is defined by the following expression:

```
e. attributes .path(a | a.name).ascendingOrder()
```

If element *e* defines three attributes named "first", "second" and "last", the previous expression returns order collection $\{0, 2, 1\}$ that gives the position of names after the sort. By *overriding* operation *sortedBy*, the previous example should be translated into:

```
e. attributes .sortedBy(
    e. attributes .path(a | a.name).ascendingOrder()
)
```

Order collections are also useful for solving various ordering problems, independently from the translation phase. For example, they have been used in the specification of binding *D2R* to bind courses and free timeslots with grid-items (see section 2.3).

3.4 Tuples

Anonymous functions *f* involved in active operations, such as *b := a.select(f)*, may require more than one argument. For example, binding *D2L* includes a selection based on two arguments *tf* and *c*:

```
d.contacts. select (c |
    tf.text.isEmpty() or
     c.lastName.startsWith(tf.text)
)
```

The reified version of this *select* operation uses a *tuple* from which a *predicate* operation is performed, as follows:

```
d.contacts. select (
    (d.contacts.lastName, tf.text). predicate(n,t |
        t.isEmpty() or n.startsWith(t)
    )
)
```

By doing so, the predicate collection is updated whenever *d.contacts.lastName* or *tf.text* changes. Tuples can thus be used for extending the initial scope of operation select to multiple arguments; a similar extension can be applied to other active operations, such as operation *collect*[2].

4. IMPLEMENTING ACTIVE OPERATIONS

The previous section explains the translation of binding specifications to their active implementation, independently from the target GUI platform. This section explains how to make the final implementation *compatible* with a given GUI platform so that its widgets can be reused. It gives theoretical and experimental results regarding performances of the final implementation.

4.1 Implementing Active Operations on a GUI Toolkit

In order to evaluate the use of active operations for UI data binding, we have implemented active operations on top of the Flex GUI platform [12]. Flex has been chosen for its own internal binding mechanism, its rich set of widgets, and its ability to pass anonymous functions as function arguments. All the three examples presented in this paper have been implemented with the resulting *ActiveFlex* project; more details about the Flex implementation can be found within the project.[3].

Figure 5 gives an overview of our *ActiveFlex* model that extends Flex classes with active operation capabilities.

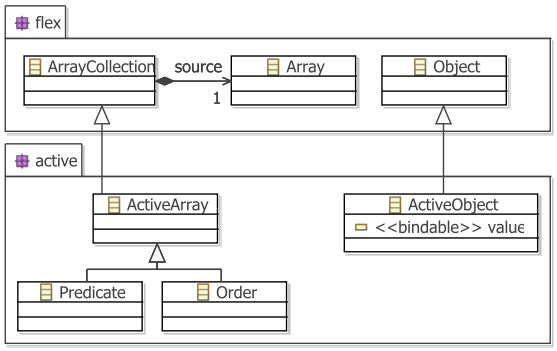

Figure 5. ActiveFlex model

The Flex API defines a built-in data binding mechanism accessible through classes *ArrayCollection* and *Object*. An *ArrayCollection* represents an observable and ordered collection of elements; it is bound to a source *Array* that holds the elements. Array collections can be bound to Flex widgets displaying collections to users, such as widget *List*. Consequently, our class *ActiveArray* inherits from this Flex *ArrayCollection* class so that instances of *ActiveArray* returned by active operations can be bound to such Flex widgets. Class *Predicate* defines predicate collections and adds usual boolean operations to class *ArrayCollection*; class *Order* defines or-

[2]Such an extension is however out of the scope of this paper.
[3]*ActiveFlex* is available under GPL license at `http://gri.eseo.fr/software/activeflex.html` .

der collections by providing operation *and* that allows sorting on multiple criteria.

Any class derived from the Flex class *Object* can mark any of its attribute as *bindable*: by doing so, the bindable attribute can be bound to Flex widgets displaying simple values, such as a text-field or a check-box. Our class *ActiveObject* thus defines its content through attribute *value* that is marked as bindable so that any active object can be bound to such Flex widgets.

The following MXML code[4] illustrates how simple the binding of the *TreeItem* of example 1 to a Flex *mx:Tree* widget is:

```
<mx:Tree dataProvider="{tree.root}" ... />
```

Object *tree* is an instance of our class *Tree*. The *mx:Tree* is bound to the *tree.root*, which is an instance of our *TreeItem* class; the *mx:Tree* then uses internally: the *children* property to recursively construct the tree content, and the bindable property *label* to display the content of each tree item. Any Flex widget can be bound to an instance of our classes *ActiveArray* or *ActiveObject* similarly.

Such a scheme can be applied to any RIA toolkit that proposes a declarative data binding mechanism, such as with WPF/Silverlight [16] or JavaFX [10]. Toolkits without data binding capabilities require more coding effort. For example, the use of active operations on top of the Swing toolkit requires implementing Java interfaces such as *TableModel*, *TreeModel* or *Document* within active collection and/or object classes.

4.2 Theoretical Complexities

Table 1 summarizes the best-case (C_{min}), average-case (C_{av}) and worst-case (C_{max}) complexities of active operations (see [4] for more details on computation of these complexities).

	C_{min}			C_{av}			C_{max}		
Operation	I	+	-	I	+	-	I	+	-
collect	n	1		n					
path	n	1		n					
select	n	1		n					
predicate	n	1		n					
not, and, or	n	1		n					
sortedBy	n	1		n					
asc.Order	n	1		$n.log_2\, n$	n		n^2	n	
and	n	1		n					
union	n	1		n					
difference	n	1		n^2	n		n^2	n	
intersection	n	1		n^2	n		n^2	n	

Table 1. Complexities of active operations

Columns labeled "I" give complexities for the initial construction of the operation result; columns labeled "+" (respectively "-") give complexities for updating the result when an addition (respectively a removal) occurs on the source collection. Initial constructions mainly cost a linear time since they globally use elementary operation *add* in the best-case (*i.e.* append); subsequent mutations also cost a linear time due the required shifting. These results only vary for sorting that requires two loops, and for difference/intersection that must check the presence with operation *contains* which costs a linear time.

The previous complexities are asymptotic. However, the *translation cost* representing the amount of time and memory used by the intermediate operations that appear during the translation (see section 3) should be also taken into account. For example, expression:

```
c. select (e | f1(e.p1) and f2(e.p2))
```

must be translated into expression:

```
c. select (c.path(e|e.p1). predicate (f1) and
           c.path(e|e.p2). predicate (f2))
```

This last expression includes 2 paths ($n_{pa} = 2$), 2 predicates ($n_{pr} = 2$), plus 1 boolean expression ($n_{pr}-1$), and thus costs 5 intermediate active collections and operations. A similar principle can be applied to operation *sortedBy* that can be based on multiple criteria (n_{cr}). Table 2 synthesizes such a cost.

Operation	Translation cost
union, diff., inter.	0
collect	0
path	1
select	$n_{pa} + 2 \times n_{pr} - 1$
sortedBy	$n_{pa} + 2 \times n_{cr} - 1$

Table 2. Translation cost

The previous complexities of *individual* operations can be used to compute complexities of an *overall* binding. Let us consider the binding defined by example 1 (the XML editor) for a source XML document with a depth d, a number of child elements per element n_e, and a number of attributes per element n_a, both common to all elements. For simplification purpose, we consider that $n_e = n_a = n$. Document thus counts $N_e = \frac{n^d-1}{n-1}$ elements, and $N_a = n \times N_e$ attributes; the tree widgets counts $N_i = N_a + N_e$ items. By considering that $N = n^d \gg 1$, we have: $N_e \simeq n^{d-1} = N/n$ and $N_a \simeq N_i \simeq n^d = N$. Number $N = n^d$ thus represents the overall and approximate number of both document nodes and tree items, while n represents the cardinality of each relation. The worst-case complexity of the associated binding is given by the complexity of the sort that binding *D2R* performs, and thus costs $\mathcal{O}(n^2)$. The initial construction of the tree widget requires N_e sort operations, and thus costs $C_{init} = N_e \times \mathcal{O}(n^2)$, *i.e.* $C_{init} = \mathcal{O}(N \times n)$. Mutations cost a linear time for all operations within the binding, *i.e.* $C_{update} = \mathcal{O}(n)$.

[4] MXML is the Flex XML dialect used to specify user interfaces.

The ratio N between these two complexities well illustrates the interest in making active binding operations, rather than recomputing all the resulting tree widget content. The next section illustrates that our experimental results match these theoretical ones.

4.3 Experimental results

Our experimental measurements focus on two objectives: verifying that our theoretical complexities match the measured performance; comparing the cost of active operations with the cost of widget rendering. They have been performed on a PC running Windows XP on top of an Intel Pentium 1.8 GHz + 1 GB RAM and an ATI FireGL 128 MB. Figure 6 gives the performance of initial construction of the tree widget for the XML document defined in the previous section with depth 3 ($N = n^3$). Axis x represents number n, and axis y gives time performance in ms.

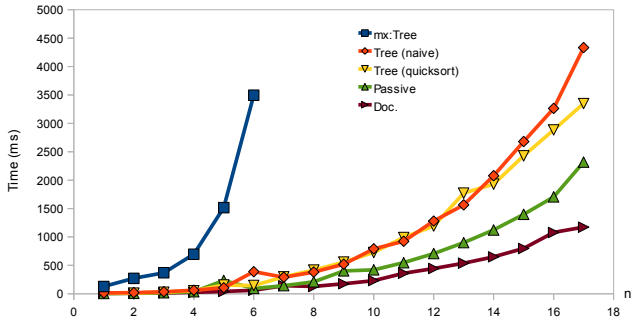

Figure 6. Construction performance

The five curves represent respectively: the XML document construction (curve "Doc.") that includes the creation of instances of classes *Document*, *Element* and *Attribute*, and the establishment of their relation; the "passive" data binding (curve "Passive") that represents the usual operations that can be used (only) for the initial construction; the tree construction (curve "Tree (naive)") that represents the binding uses to build the target instances of class *Tree* and *TreeItem* with active operations; the tree construction that uses a quicksort-based algorithm (curve "Tree (quicksort)") instead of a naive one (previous curve); finally, the tree widget rendering (curve "mx:Tree") performed by the Flex engine to display the graphical content of the widget. Curve "Doc" tends to $0,25 \times N$, thus illustrating that the document construction is linear. Curve "Tree (naive)" gives the performance of the bindings in the worst-case since it uses a naive implementation of the sort that costs $\mathcal{O}(n^2)$; it tends to $0,052 \times N \times n$, thus matching the expected theoretical result. Curve "Tree (quicksort)" tends to $0,18 \times N \times log_2\, n$; the expected logarithmic dimming effect appears for a significant value of n (here around 14). Curve "Passive" tends to $0,028 \times N \times n$, which is around half the curve "Tree (naive)": this illustrates that the translation cost is around 1, as expected (one intermediate order collection is here required). Finally, curve "mx:Tree" tends to $0,6 \times N \times n^2$, which probably results from the traversing required for computing location y of each tree-item i. This location may be recursively computed by summing heights h of the previous-sibling items of i: $y(i) = \sum_j h(i.sibling[j])$; in turn, computing the height of a node consists of summing the height of its children: $h(i) = \sum_j h(i.children[j])$. The overall layout includes N computation of y, so that the complexity tends to $N \times n^2$.

This experimentation illustrates that complexity of data bindings can often be neglected regarding the complexity of rendering complex graphics, such as a tree. Simpler widgets, such as a list widget, can however require complexity comparable to the one of the associated binding. Experiments performed on example 2 with a large list of contacts confirms this point: because the binding performs a sort, displaying the list can become faster than performing the binding, but for a large number of list items (around 10.000 in our experimentation).

Figure 7 gives the performance of the addition of an attribute; performance of its removal is very close to this curve, and is thus not shown in the figure. In both cases, updating the document (curve "Doc.") takes a constant and negligible time; data binding takes a linear time $t(ms) = 0,015 \times n$, as expected (curve "Tree"); the Flex tree widget does not have to compute new y locations: this rather consists of applying a vertical offset in the display buffer, which requires a constant time around 150 ms (curve "mx:Tree").

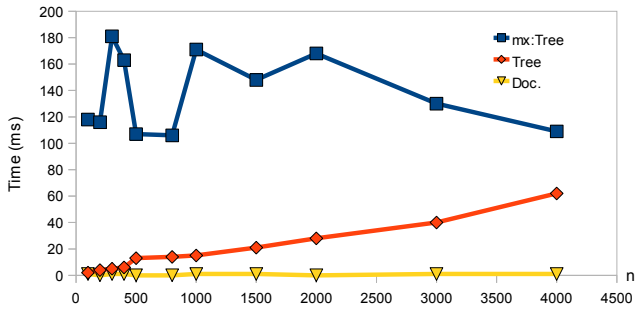

Figure 7. Addition performance

These performances illustrate that the time required to refresh a drawing is often bigger than the time for computing a binding. In the previous example, the update time for the Flex tree and the active operations become equal when n reaches $150/0,015 = 10.000$, which represents a really huge XML document (10^{12}nodes for a depth of 3).

5. RELATED WORKS

Constraint systems have been widely used for defining data bindings in the context of GUI. The Garnet toolbox proposes the use of "formula", expressed in Lisp, to bind an object value to other object values: the for-

mula is reevaluated whenever the values within the formula change, like in spreadsheets [14]. The Rendezvous architecture defines the concept of "active value": an active value is a variable used as en entry point to constraint definitions (also expressed in Lisp); when the value changes, the constraint is reevaluated to reflect the change [11]. Similarly, but within a C++ infrastructure, the Amulet environment allows the definition of constraint through "formula" that bind object values [15]. However, all these constraint-based systems are limited to the binding of object values: they do not address the binding of collections, and are thus limited to simple widgets such as text-field or combo-box.

The *JFace* toolkit defines interfaces *IObservableValue* and *IObservableCollection* that respectively allow the observation of values and collections [9]. These interfaces are implemented in classes that can be used for representing "bindable" data in both the model and the view. This approach defines the required foundation classes for data binding; however, it is a pure *programmatic* approach where bindings must be implemented rather than being declared. The *ObjectEditor* toolkit uses a more subtle approach that consists of extending the JavaBean syntax so that data bindings appear in a more declarative way [7]. Moreover, this toolkit extends the scope of data binding to action binding that allows user actions to be bound to "do", "undo" and "redo" methods. However, *ObjectEditor* offers a low expressiveness regarding data binding capabilities.

RIA toolkits, such as WPF [16], Flex [12], and JavaFX [10], have adopted a *declarative* approach to data binding. The aim of these approaches is to simplify the process of writing bindings: rather than being implemented, bindings are declared. These approaches work fine for simple bindings where the view does not differ much from the model; they however lose their simplicity as soon as the complexity of the model is increased, which often results in requiring some low level programming that contradicts the initial aim of the declarative approach. Section 6 focuses on proving these limitations.

Using rule-based incremental transformation languages solves the previous limitations of declarative data bindings. The incremental XSLT processor *incXSLT* allows the incremental execution of XSLT transformations [17]. However, XSLT concerns source XML documents and target text documents, which makes it difficult to use on various GUI platforms. Moreover, XSLT is a complex language that does not match well the required simplicity of data binding. Using a *mapping* language, such as Malan [5], combines the benefits of the simplicity of data bindings and the expressiveness of transformation languages. However, this higher level approach postpones the problem of executing the mappings: for example, active operations can be generated from a mapping so that the mapping can be executed on a target platform.

6. COMPARISON WITH BINDINGS OF RIA TOOLKITS

From our knowledge, recent RIA toolkits offer the best data binding mechanisms. This section compares our approach with three representative RIA toolkits: Flex [12], WPF[16], and JavaFX [10]. It illustrates why the three examples presented in this paper cannot be fully implemented by using their data binding capabilities.

6.1 Active Operation Capabilities

Table 3 summarizes the capabilities of active operations presented throughout this paper. The capabilities belong either to active collections, active objects, or both.

Category	Capability	Collection	Object
Math.	*union*	√	√
	intersection	√	√
	difference	√	√
	tuple	√	√
View	*select*	√	√
	sort	√	n/a
	multiple views	√	√
Transf.	*collect*	√	√
	path	√	√
	map	√	√
Bidir.	bidir. assignment	n/a	√
	+ conversion	n/a	√

Table 3. Active operation capabilities

Category "Math" includes the usual mathematical set operations, and the definition of tuples; tuples are useful in operations such as *select(f)* and *collect(f)* with more than one parameter in *f*. Category "View" (in the database sense) includes operations *select* and *sort*; it also defines the ability for an active collection or object to be treated more than once by an active operation (such as *select* and *sort*), thus allowing *multiple views* on a common model. Category "Transformation" concerns operations *collect*, *path* and *map*, that allow a progressive transformation from the source to the target. Finally, category "Bidir." defines the bidirectional assignment used to perform two-way bindings on active objects; this functionality is supplemented by a reversible conversion.

All these capabilities have been illustrated in the 3 examples of section 2. The next section discusses if and how these capabilities are present within the data binding mechanisms of the three RIA toolkits Flex, WPF and JavaFX.

6.2 Comparison with RIA Toolkits

Table 4 summarizes the data binding capabilities of the three RIA toolkits Flex, WPF and JavaFX.

As mentioned in section 4.1, instances of the Flex class *ArrayCollection* can be bound to collection widgets, and bindable properties defined within instances of class *Object* can be bound to simple widgets [12]. These two capabilities are equivalent to operation *map* on collections

	Flex		WPF		JavaFX	
	Coll.	Obj.	Coll.	Obj.	Coll.	Obj.
Math op.						
select	(√)		√			
sort	(√)	n/a	√	n/a		n/a
m. views			√	√	√	√
collect		√		√	√	√
path		√		√		√
map	√	√	√	√	√	√
assign.	n/a	√	n/a	√	n/a	√
+ conv.	n/a		n/a	√	n/a	

Table 4. binding capabilities of RIA toolkits

and on objects. Bidirectional assignment is possible through property binding, but without conversion. A path can be specified when binding two properties; however, paths cannot be defined on collections. It is also possible to define a "transformation function" within a property binding, which is analogous to operation *collect* for a single object. All the other capabilities are not available with Flex. Operations *select* and *sort* exist on class *ArrayCollection*, but they do not return new collections: they rather modify the source model directly; this does not separate well the model from the GUI. Moreover, these operations cannot use "active" parameters (for example, through tuples).

WPF offers richer binding capabilities than Flex [16]. Operations *select* and *sort* return new bindable collections, and thus allow the definition of multiple views; bidirectional conversion is also possible. However, math operations and transform operations *collect* and *path* remain undefined for collections.

JavaFX has adopted a radically different strategy for defining bindings: rather that proposing a bunch of binding artifacts, such as with Flex or WPF, JavaFX introduces new language *keywords* dedicated to the definition of bindings [10]. Consequently, JavaFX bindings are clearer than that of Flex or WPF since they are based on very few constructs, while Flex and WPF bindings use many different artifacts that often require low level *ad hoc* programming. However, as table 4 illustrates, important features are unavailable in JavaFX thus making it not as expressive as expected.

6.3 Implementing the Examples using RIA Toolkits

Regarding example 1, Flex data binding cannot be used to bind XML documents to the tree widget since this requires *union* operations. The example could be built by coding a specific data provider that implements Flex interface *ITreeDataDescriptor*. This practice is similar to the implementation of Swing models, and thus does not respect the initial philosophy of binding. Filtering contacts in example 2 is impossible with Flex binding: this requires the programming of an *ad hoc* code that recomputes the filtering whenever the user changes the filtering text. Moreover, sorting and filtering the contacts must be specified by altering the code of the model: the design of the GUI thus impacts the source model, which contradicts the fact that binding clearly separates the model and its GUIs. Finally, example 3 cannot be implemented by using Flex binding. This is due to the free-time slots objects that are not present in the source model, and thus cannot be bound to grid-items.

As with Flex, example 1 cannot be implemented using WPF binding due to the lack of operation *union*. Example 2 is easier to implement than with Flex due to WPF view capabilities; however, the active filtering of contacts again requires the development of *ad hoc* code. But most of all, implementing example 2 with WPF requires a bunch of data binding artifacts: "bidirectional converters" must be programmed by implementing interface *IValueConverter*; "data templates" must be defined with the XAML document[5] for implementing functions that define the text of the tree items; and "observable collection" must specify the filtering and sorting of contacts in a programmatic way. The developer thus needs to juggle with multiple artifacts, and with both programmatic and declarative paradigms. Example 3 cannot reap benefit from WPF binding and must be implemented from scratch.

Due to the lack of view capabilities, examples 1 and 2 cannot be implemented using JavaFX binding. As with Flex and WPF binding, example 3 is not feasible with JavaFX.

6.4 Discussion

Data binding of RIA toolkits are simple to use for simple bindings. Their binding mechanisms remain sufficient as long as the source models do not differ much from the target models. Moreover, these mechanisms allow binding to existing relational databases and/or XML documents directly, thus making them attractive in an enterprise context. In counter part, the offered binding capabilities become insufficient to address various and/or complex problems. These toolkits often require multiple artifact juggling that alter the initial simplicity of the declarative approach. Moreover, each RIA toolkit proposes its own data binding mechanism. Consequently, a binding written in WPF cannot always be translated to a Flex or JavaFX binding: they are all platform dependent and do not share a common model.

The use of active operations for specifying and implementing data binding removes such limitations. A single DSL is used for specifying the bindings, and translating them before being implemented; it is thus platform-independent. The resulting implementation performances are good for the initial construction of operation results, and very good for updating these results. In counter part, active operations have to be implemented through a dedicated compiler; however, RIA toolkits also require some hidden code generation

[5]XAML is the XML dialect used by WPF to specified user interfaces.

(for example, to implement paths between two bound properties). Active operations currently do not provide mechanisms for binding UI directly with databases or XML documents. However, such an issue is not specific to the domain of active operations: it concerns the serialization and deserialization of models on different data platforms.

7. CONCLUSION

This paper presents how to reap benefits from active operation expressiveness and easiness for specifying and implementing UI data bindings. Active operations offer the ability to reevaluate the result of the operations in real-time while the user interacts with the system. The proposed approach overcomes the limitations of data binding mechanisms provided by modern UI toolkits. These limitations are: GUI bindings are platform-dependent and lack at defining complex bindings. It is based on a single DSL, mainly inspired by OCL, that allows both the specification and the platform-independent implementation of data bindings. Active operations have been implemented on top of the Flex RIA toolkit; the resulting ActiveFlex API offers good performance for the initial construction of active operation results, and very good ones regarding their updates.

The next step of our work is to implement a compiler that will generate active implementations from any active operation specification. Such a compiler would generate the implementations on various GUI platforms. Since our approach offers a great expressiveness, we will use and evaluate it in the context of "Information Visualization" in order to confront its performance to large models.

8. REFERENCES

1. D. H. Akehurst. *Model Translation: A UML-based specification technique and active implementation approach*. PhD thesis, University of Kent, 2000.
2. O. Beaudoux. XML active transformation (eXAcT): Transforming documents within interactive systems. In *DocEng '05: Proceedings of the 2005 ACM symposium on Document engineering*, pages 146–148. ACM, 2005.
3. O. Beaudoux and A. Blouin. Linking data and presentations: from mapping to active transformations. In *DocEng '10: Proceedings of the 2010 ACM symposium on Document engineering*, pages 107–110. ACM, 2010.
4. O. Beaudoux, A. Blouin, O. Barais, and J. M. Jezequel. Active operations on collections. In *MoDELS '10: Proceedings of the 13th ACM/IEEE International Conference on on Model Driven Engineering Languages and Systems (LNCS 6394)*, pages 91–105. Springer, 2010.
5. A. Blouin, O. Beaudoux, and S. Loiseau. Malan: A mapping language for the data manipulation. In *DocEng '08: Proceedings of the 2008 ACM symposium on Document engineering*, pages 66–75. ACM, 2008.
6. J. Coutaz. PAC, on object oriented model for dialog design. In *Interact'87*, 1987.
7. P. Dewan. Increasing the automation of a toolkit without reducing its abstraction and user-interface flexibility. In *EICS '10: Proceedings of the 2nd ACM SIGCHI symposium on Engineering interactive computing systems*, pages 47–56. ACM, 2010.
8. R. Eckstein, M. Loy, and D. Wood. *Java Swing*. O'Reilly, 2002.
9. Eclipse Foundation. JFace data binding. `http://wiki.eclipse.org/index.php/JFace_Data_Binding`.
10. R. Field. JavaFX language reference (chapter 7 - Data binding). `http://openjfx.java.sun.com/current-build/doc/reference/ch07s01.html`.
11. R. D. Hill. The Rendezvous constraint maintenance system. In *UIST'93: Proceedings of the 6th annual ACM symposium on User interface software and technology*, pages 225–234. ACM, 1993.
12. C. Kazoun and J. Lott. *Programming Flex 2*. O'Reilly, 2007.
13. G. E. Krasner and S. T. Pope. A description of the model-view-controller user interface paradigm in smalltalk80 system. *Journal of Object Oriented Programming*, 1:26–49, 1988.
14. B. Myers, D. Giuse, R. Dannenberg, B. Zanden, D. Kosbie, E. Pervin, A. Mickish, and P. Marchal. Garnet: comprehensive support for graphical, highly interactive user interfaces. *Computer*, 23(11):71–85, 1990.
15. B. Myers, R. McDaniel, R. Miller, A. Ferrency, A. Faulring, B. Kyle, A. Mickish, A. Klimovitski, and P. Doane. The Amulet environment: new models for effective user interface software development. *IEEE Transactions on Software Engineering*, 23(6):347–365, 1997.
16. C. Sells and I. Griffiths. *Programming Windows Presentation Foundation*. O'Reilly, 2005.
17. L. Villard and N. Layaida. An incremental XSLT transformation processor for XML document manipulation. In *WWW '02: Proceedings of the 11th international conference on World Wide Web*, pages 474–485. ACM, 2002.
18. J. B. Warmer and A. G. Kleppe. *The object constraint language: getting your models ready for MDA*. Addison-Wesley.