The Intensional View Environment

Kim Mens
Département d'Ingénierie Informatique
Université catholique de Louvain, Belgium
kim.mens@info.ucl.ac.be

Andy Kellens*
Programming Technology Lab
Vrije Universiteit Brussel, Belgium
akellens@vub.ac.be

Frédéric Pluquet, Roel Wuyts
Decomp
Université Libre de Bruxelles, Belgium
{frederic.pluquet | roel.wuyts}@ulb.ac.be

Abstract

This paper presents IntensiVE, a tool suite implemented in Cincom VisualWorks Smalltalk that allows for the documentation and co-evolution of high-level structural regularities in the source code of a software system.

1 Introduction

The *IntensiVE* tool suite, which is based on the underlying models of *Intensional Views* and *Intensional Relations*, allows for the documentation of high-level structural regularities in the source code of a software system. It also supports co-evolution of those regularities with the source code when either of them evolve. *IntensiVE* was implemented in VisualWorks Smalltalk (7.3) and comprises, amongst others, the following sub-tools:

- Intensional View Editor
- View Consistency Checker
- Relation Editor
- Relation Checker
- Intensional View Displayer

In the remainder of this paper we describe each of these sub-tools and how they support the co-evolution of structural regularities with source code. Along the way we introduce the underlying concepts of *Intensional Views* and *Relations*.

2 The Intensional View Editor

An *Intensional View* is a set of source-code entities (classes or methods) which are structurally similar. Instead of enumerating all elements that make up a view, it is defined by means of an *intension*: an executable description which yields, upon evaluation, the set of entities belonging to the view, also called the *extension* of the view. As languages in which to describe the intension of a view, our tools currently supports the logic meta-programming language *Soul* [2] as well as the Smalltalk language.

Figure 1 shows the *Intensional View Editor*, the main tool for creating and manipulating views, together with some structural regularities we documented for SmallWiki [3], a Wiki implementation in Smalltalk. On the screenshot, the left pane shows all defined views (and relations, which we explain in section 4) in a tree representation. The right hand side shows the Intensional View Editor opened on a view named 'Execute Methods'.

This view represents all methods responsible for executing actions on Wiki pages. As can be seen from the screenshot, the intension for this view is: `methodInProtocol(?entity,action).` This query, written in *Soul*, binds occurrences of methods in the 'action' protocol to the free logic variable `?entity`. Also notice in the screenshot (left pane) that this view is defined as a subview of the view containing 'all SmallWiki methods'. The semantics of defining a view as subview of another one is that the intension of the subview is calculated in the context of the parent view. In other words, evaluating the intension of the Execute Methods view results in all methods which belong to the extension of the view 'all SmallWiki methods' but also to an `action` method protocol.

To deal with deviating cases in the source code, the tool also supports the explicit exclusion (resp. inclusion) of an entity from a view. Figure 1 shows an example of this: the method `listActions`, implemented on the `Action` class is explicitly excluded it from the view, by putting it in the 'excludes set' of the view. Analogously, we have an 'in-

*Ph.D. scholarship funded by the "Institute for the Promotion of Innovation through Science and Technology in Flanders" (IWT Vlaanderen).

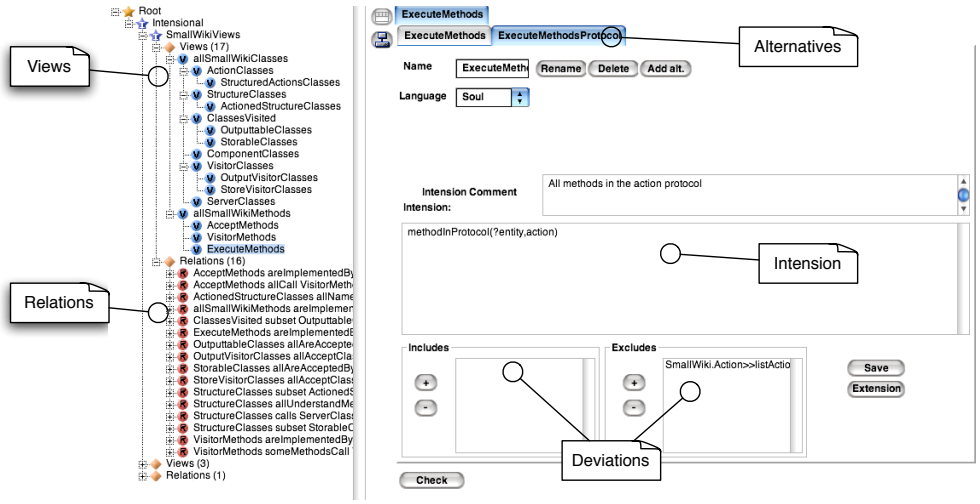

Figure 1. The Intensional View Editor at work

cludes set' of entities that should be included in a view, even though they do not satisfy the intension.

Intensional Views allow the definition of multiple alternative descriptions for the same view. This ability, together with the requirement of extensional consistency between those alternatives (explained in the next section), provides an elegant way of declaring interesting naming and coding conventions to be respected by the entities of a view.

3 The View Consistency Checker

Fig. 2 shows the *View Consistency Checker*. This tool is used to verify that the different alternative descriptions of a same view are *extensionally consistent*, meaning that they all produce the same extension. When this constraint is violated, the tool provides appropriate feedback on what entities are in conflict. A developer can use this feedback in order to fix the inconsistencies.

The tool shows the user a column per alternative description of the view. The first column contains the extension of the main alternative (by default this is the first alternative of the view, but double-clicking a column changes the main alternative); the other columns contain the delta between the extension of the main alternative and the alternative represented by the column. If an element does not exist in the main alternative it is coloured green. Elements present in the main alternative but not in the other are coloured red.

4 The Relation Editor

The *Relation Editor* allows a user to document relations between intensional views. Our model currently supports

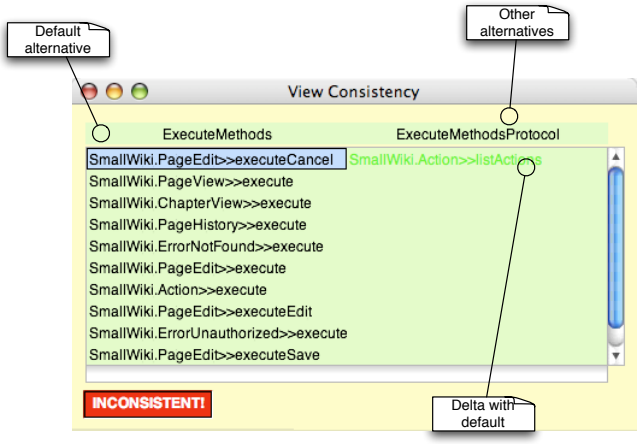

Figure 2. The View Consistency Checker

only relations of the canonical form:

$$\mathcal{Q}_1\ x \in Source : \mathcal{Q}_2\ y \in Target : x\ R\ y$$

where $\mathcal{Q}_1$ and $\mathcal{Q}_2$ are either logic quantifiers $\forall$, $\exists$, $\exists!$, $\nexists$ or more fuzzy quantifiers[1] like $some$, few, $many$ or $most$. $Source$ and $Target$ represent intensional views and R is a binary predicate over the source-code entities (denoted by x and y) contained in those views. A simple example of an intensional relation is that all 'Execute Methods' are implemented by 'Action Classes'. This relation, opened in the Relation Editor, is shown in figure 3. Expressed in the canonical form above, the relation is defined as:

[1]The fuzzy quantifiers are defined in terms of a minimum or maximum number of elements for which the condition should hold.

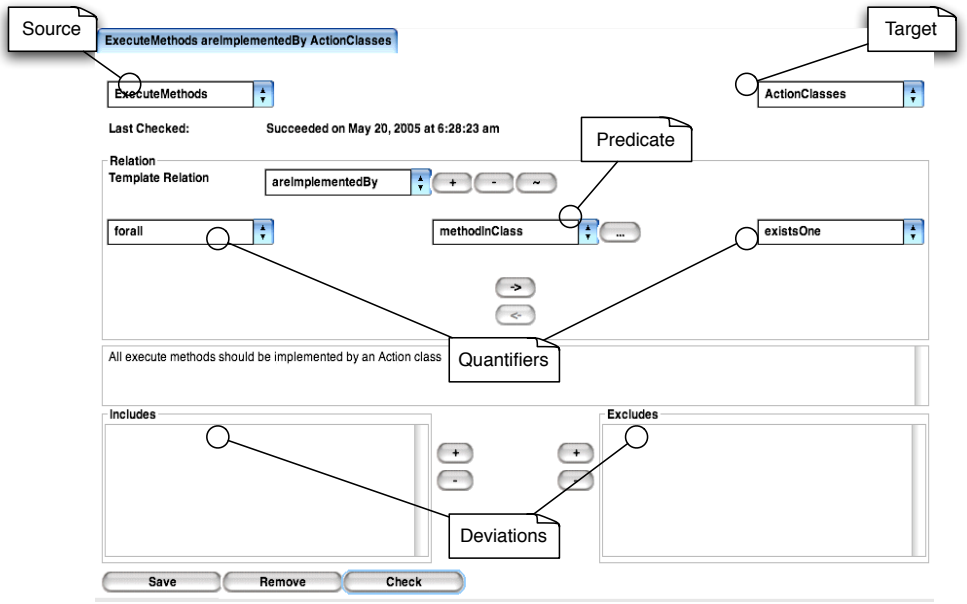

Figure 3. The Relation Editor at work

$\forall x \in$ *ExecuteMethods* :
$\exists! y \in$ *ActionClasses* :
x methodInClass y

Our tool offers two possibilities to define a binary predicate R over source-code entities, in terms of which intensional relations can be expressed. In addition to defining the predicate directly in *Smalltalk* the user can opt to use a *Soul* predicate (typically using LiCoR, an extensive library of *Soul* predicates to reason about source code). Like the Intensional View Editor, the Relation Editor also supports the explicit declaration of deviating cases. It allows a user to specify explicitly tuples of source-code entities to be included in or excluded from the relation.

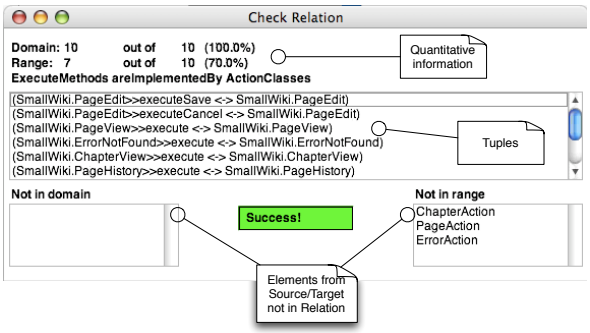

Figure 4. The Relation Checker at work

5 The Relation Checker

When pressing the 'Check' button in the Relation Editor (Fig. 3), the validity of a relation with respect to the source code is checked and the user is presented an instance of the *Relation Checker* (Fig. 4). Besides reporting whether the relation holds, the tool presents the user a list of all tuples for which the relation is valid as well as some statistics on how many elements from source and target participate in the relation. It also lists all entities from the source view which are *not in the domain* of the relation as well as all entities in the target which are not reached by the relation. When a relation does not succeed, a user can use this information to determine for which source code entities the documented relation and the source code are out of sync.

6 The Intensional View Displayer

The tools listed above support a user in manipulating (declaring, modifying, renaming, removing, verifying and saving) intensional views and relations. Our suite also includes a visualization tool that provides a user with a global and compact drawing of all defined views and relations. The *Intensional View Displayer* is depicted in Fig. 5. For a given selection of views, the displayer shows all these views, all their alternative descriptions, all subview relationships and all intensional relations in which those views take part. The views are laid out automatically in a hierarchy that reflects the view nesting, but the layout can be modified and stored manually.

The visualization tool is defined on top of *CodeCrawler*

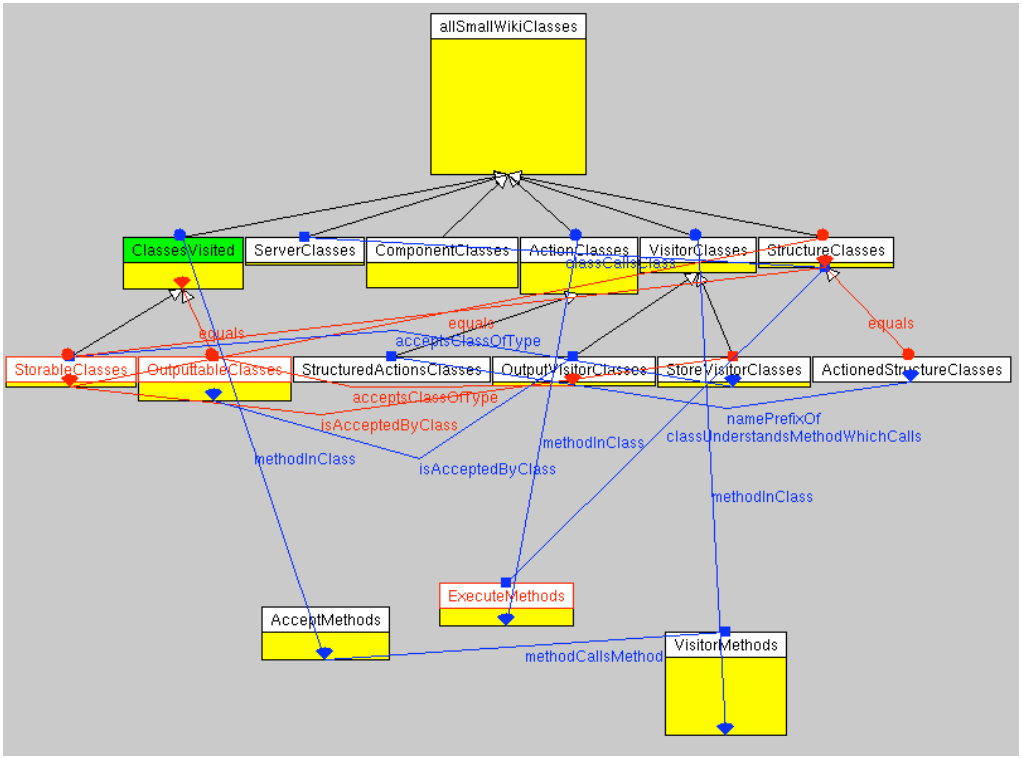

Figure 5. The Intensional View Displayer at work

[1], a reverse engineering tool which combines software metrics and visualization. This allows us tho use metrics to highlight important characteristics of intensional views or relations. For example, a simple metric for a view is the number of entities contained in its extension. In Fig. 5 this metric was used as height of the rectangular boxes representing the views.

The visualization tool also uses colors to distinguish the different kinds of objects in a drawing. By default, the name and rectangle of intensional views are drawn in black, as well as the subview edges (starting with a triangle) and edges relaying a view with its alternative descriptions (ending with a diamond). The text and rectangle of the alternative descriptions are rendered in grey and an option can be toggled to not render them at all. Finally, edges representing intensional relations, together with the relation name, are drawn in blue. The use of colours can also be used as a metric, for example to highlight inconsistencies in the documentation. The 'View Consistency' metric, for example, calculates the extensional consistency of a view and draws the view in red when inconsistent. A similar metric can be applied to the links connecting a view to its alternatives, to indicate what particular alternatives are inconsistent. In a similar way a color metric can be applied to the intensional relations, so that invalid intensional relations are highlighted in red. As shown in figure 5, this allows us to assess the possible inconsistencies between the documentation and the source-code of the system in a glimpse of the eye

7 Download

IntensiVE is available under the GNU Lesser General Public License and can be downloaded from:
http://prog.vub.ac.be/∼akellens/intensive/

References

[1] M. Lanza. Codecrawler: Lessons learned in building a software visualization tool. In *Proceedings of the 7th European Conference on Software Maintenance and Reengineering (CSMR 2003)*, pages 409–418. IEEE Computer Society, 2003.

[2] K. Mens, I. Michiels, and R. Wuyts. Supporting software development through declaratively codified programming patterns. *Elsevier Journal on Expert Systems with Applications*, 23(4):405–431, November 2002.

[3] L. Renggli. Smallwiki - collaborative content management. 2003.