Towards an Application of Update Propagation on Logic Programs Representing Java Source Code

Richard Tantius, Daniel Speicher, Andreas Behrend

University of Bonn, Computer Science III,
Römerstraße 164, 53117 Bonn, Germany
{tantius, dsp, behrend}@cs.uni-bonn.de

Abstract. Logic programs are now used as a representation of object-oriented source code in academic prototypes for about a decade. This representation allows a clear and concise implementation of analyses of the object-oriented source code. The full potential of this approach is far from being explored. In this paper, we report about an application of the well-established theory of update propagation within logic programs. Given the representation of the object-oriented code as facts in a logic program, a change to the code corresponds to an update of these facts. We demonstrate how update propagation provides a generic way to generate incremental versions of such analyses.

Keywords: logic program, update propagation, application, logic meta programming, refactoring impact

1 Introduction

In this paper, we show how update propagation can be employed for efficiently computing-derived information within the domain of logic fact-bases representing object-oriented source code.

1.1 Logic Meta Programming

Logic Meta Programming approaches build a detailed representation of a program (typically) written in another programming language in a logic fact-base [14], [8], [10]. This representation allows analysing the orginal program by means of a logic program built on those facts. This approach has been used to detect code locations that need design improvement [13]. It provides a suitable basis for different static analyses from the implementation of code quality metrics to the implementation of type constraints. Recently, we have been arguing that logic meta programming should be used to integrate the knowledge about good design structures and suspicious design structures, creating a database of code quality knowledge, which can be evolved over time [13].

1.2 Update Propagation

Update propagation (UP) is an established database research topic, which has been studied over the last 40 years mainly in the context of integrity checking and materialized views maintenance, e.g., [4], [2], [9], [7], [11], [12]. Therefore, UP is known from the SQL and Datalog world. UP makes a contribution to efficiently compute implicit changes of derived relations resulting from explicitly performed updates of extensional facts of a logic fact-base. SQL view specifications and Datalog rules, as well as Prolog rules are related forms of deductive rules. This supports the idea to adapt UP to the Prolog world, and also use sets of deltas, together with specialized update statements to incrementally maintain derived predicates (which can be a software analysis, implemented in Prolog). The original predicates are required only once for materializing their initial answers, the specialized delta versions are used in update statements afterwards for continuously updating the materialized results. Assuming that a great portion of the materialized content of the original logic rules remains unchanged, the application of such update statements may considerably enhance the efficiency of computing the state of such relations after an update of the fact-base.

1.3 Refactoring Impact Prediction

UP enables computing the results of an analysis after an update of the fact-base without actually executing the change. Such an update of the fact-base can be induced for example by applying a structural improvement like a refactoring [6] to the source code. This allows to efficiently execute a "what-if"-style analysis. The approach enables us to evaluate a potential refactoring of Java programs, by computing quality attributes like software metrics before the refactoring and simulate via UP how the metric result changes due to the refactoring.

1.4 Automation

Figure 1 introduces our implementation which automates several aspects of the refactoring impact prediction via UP. The picture shows the different components of the system. Those components may consist of several Prolog modules. The application covers the derivation of a suitable abstract model, on which we build the software analysis we intend to employ. We use the Logic Meta Programming approach JTransformer presented in [8], to derive such a model. The implementation also handles the refactoring simulation on the level of that abstract model, metric computation and UP rule generation. We have used SWI-Prolog (in the version $6.0.2^1$) for the implementation. Beside the metric definitions that had to be implemented in a strictly declarative syntax, so that we can apply UP, we used the full feature set of the SWI-Prolog environment for the other parts of the implementation.

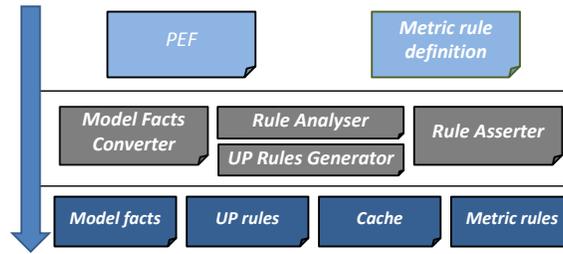[1] The Project homepage of SWI-Prolog: `http://www.swi-prolog.org/` (accessed 17.08.12).

Fig. 1. Overview Update Propagation

2 Model and Analysis

Software analyses like software metrics are a frequently-studied approach to detect lack of quality and are also capable of making improvements of quality measurable. Logic Meta Programming Approaches provide the capability to represent software systems as logic programs. A refactoring in this context, therefore, can be understood as a transformation imposing changes on an extensional fact-base. We discuss the structural cohesion metric Lack of Cohesion in Methods $LCOM1$ [5], as an example of such a software analysis. Cohesion can be defined as the degree of how closely module components are related to each other. A unified framework for structural metrics was presented by Briand in [3], who created a common model for existing metrics. The model unified the syntactical representation and operational semantic of those metrics. In order to provide the information about the source code the metric relies on, we also present a simplified abstract model as basis for $LCOM1$. This meta model will be directly derived from the Logic Meta Programming fact-base.

2.1 Abstract Cohesion Model

The Logic Meta Programming approach [8], which we use to derive our abstract cohesion model, is based on Prolog. For this reason, we represent the relevant information for the $LCOM1$ metric as Prolog facts. We will also present the $LCOM1$ metric itself as a logic program in the following. To sufficiently describe the information relevant for cohesion metric, we need to take the following information into account: *Which class contains a certain method or field? Which methods are called and which fields are accessed by a method?* The presented model is based on the cohesion model as presented in Briand [3] and was adapted to Prolog. We consider the following Prolog predicates:

```
c(M).        class
cm(C, M).    class contains method
cf(C, F).    class contains field
mf(M, F).    method accesses field
mm(M, N).    method invokes method
```

The related Prolog facts are ground versions of the predicates from above and the variables `C,M,F,...` are bound to unique identifiers for the corresponding elements. In the next subsection, we demonstrate how we extract this model from the JTransformer fact-base. In the following section, we build the $LCOM1$ metric on top of those facts as a logic program.

2.2   Model Fact Derivation

We derive the abstract cohesion model directly from the fact-base created by the JTransformer Logic Meta Programming approach. JTransformer builds a so called *Abstract Syntax Tree* (AST) that already is a full model representation in Prolog of the Java language. The JTransformer facts are called *Program Element Facts* **PEF**, they are the starting point for creating the cohesion model.

Figure 1 gives an overview of the various components of our Implementation. The architecture consists of three layers. In the first layer, we have the metric rules and the facts created by JTransformer. The layer in the middle is the meta programming layer, in which we process the rules and facts. Here we compile the UP rules. In the layer at the bottom, we have the subsystems which actually contain the executable code. Each subsystem may consist of several Prolog modules.

Based on generator predicates, the **Model Facts Converter** component creates cohesion model facts (from the JTransformer facts) and asserts them to a prolog module (`cohesion_model`). Additionally, we perform checks, if an element should be included at all. In the case of classes, we do not consider the three following class types. *Interface classes* do not provide method calls and attribute references. Cohesion cannot be examined here. *Classes from external dependencies* are supposed to be examined elsewhere. *Anonymous classes* are not supposed to be analysed standalone.

The implementation of the class generator predicates is as follows:

```
generate(FactsModule, c, [ClassId]) :-
  % Here we only use the class id
  classT(ClassId, _, _, _),
  source_class(ClassId).
```

For the free variable `FactsModule`, we use the module `cohesion_model` mentioned above as a default value. The implementation of the class analysis considers different JTransfomer facts to determine the class type:

```
source_class(ClassId) :-
  % JTransformer facts
  not(externT(ClassId)),
  not(interfaceT(ClassId)),
  % See below
  not(anonymous_class__(ClassId)).
```

```
anonymous_class__(Class):-
  classT(Class, _, ClassName, _),
  string_concat('ANONYMOUS$', _, ClassName).
```

On top of the provided model facts, we create our software analyses, for example the cohesion metric presented before. The deductive **Metric rule definition** is the starting point (also shown in Figure 1). We define the deductive part of the metric in a Prolog module:

```
:- module( metricName_deductive_ruleset, []).
    rule₁(A, B, ...) :- ...
```

For example:

```
:- module(lcom1_deductive_ruleset, []).
    ...
```

2.3 Declarative Metric Implementation

The $LCOM1$ metric definition we present in the following is based on the definition given by Briand in [3].

Query and Mapping The computation of structural metrics can be divided into two steps. First, a query step collects the elements or relations that are relevant for the metric. Second, a mapping maps the result of the query to a number. Separating these steps has the benefit that we can discuss both steps on their own. A query result may be evaluated with different mappings. A mapping may be applied to the result of different queries. The separation of query and mapping has the advantage, that we can apply the update propagation approach to the deductive rules defining the query.

Query $LCOM1$ counts the number of method pairs within a class that do not access even one common field. We split this definition into two predicates. Each predicate will be first described in natural language, then as a logic program.

M and N in C are connected, if M accesses a field F, that belongs to the class C, and N accesses that field F as well.

```
cp(C, M, N) :-
    mf(M,F), cf(C, F), mf(N,F).
```

M and N in C are a pair of methods lacking cohesion, if M is a method in C, N is as well a method in C and M and N are not a connected pair in C:

```
lp(C, M, N) :-
    cm(C, M), cm(C, N), not(cp(C, M, N)).
```

Mapping To complete the $LCOM1$ computation rule, we need to perform some additional steps after the deductive part.

```
lcom1(C, R):-
 findall([M,N], (cp(C, M, N), not(M=N)), E),
 length(E, T),
 R is T/2.
```

3 Refactoring as Cohesion Model Update

We model a refactoring as an update on our cohesion model. Performing the refactoring only on the abstract level of the cohesion model, helps to concentrate the computation only on aspects relevant for the metric computation. Because we use update propagation in the following, we do not directly perform model updates, rather every update will generate a so called delta fact, which depicts the actual change and will be discussed in detail in Section 4. In the following, we briefly discuss the refactorings we use and describe their effects on the level of the model. Both refactorings assume in our setting, that the moved element will be extracted into a new class, creating the following delta fact: `add_c(#newClassId)`

Since we exclude constructor methods from our model and do not consider modificators as public, private and protected, the following refactorings require no special preconditions to be applied.

3.1 Move Method, Move Field

The move method refactoring moves a method from one class to another. Though in a real world refactoring, we would need to adjust the code in several ways, so that it remains functioning and compileable we add two simple delta facts to our model:

```
add_cm(C, M)
del_cm(C, M)
```

Similar to the move method refactoring the move field refactoring moves a field from one class to another, the resulting delta facts are as follows:

```
add_cf(C, F)
del_cf(C, F)
```

4 Update Propagation

In this section, we show how to apply update propagation in Prolog. Because of the different evaluation mechanisms for SQL views (set-oriented, bottom-up) and Prolog rules (instance-oriented, top-down), however, the transformation techniques from UP could not be applied directly. Instead, the specific properties of Prolog rules have to be taken into account in order to achieve a complete and sound update propagation based on delta predicates.

```
% Derived from: lp(C, M, N) :- cm(C, M), cm(C, N), not(cp(C, M, N)).

add_lp(C, M, N) :- add_cm(C, M),                     nwd_cm(C, N), not(nwi_cp(C, M, N)), not(   lp(C, M, N)).
add_lp(C, M, N) :- add_cm(C, N),     nwd_cm(C, M),                 not(nwi_cp(C, M, N)), not(   lp(C, M, N)).
add_lp(C, M, N) :- del_cp(C, M, N), nwd_cm(C, M), nwd_cm(C, N),                          not(   lp(C, M, N)).
del_lp(C, M, N) :- del_cm(C, M),                      cm(C, N), not(   cp(C, M, N)), not(nwi_lp(C, M, N)).
del_lp(C, M, N) :- del_cm(C, N),         cm(C, M),             not(   cp(C, M, N)), not(nwi_lp(C, M, N)).
del_lp(C, M, N) :- add_cp(C, M, N),      cm(C, M),      cm(C, N),                    not(nwi_lp(C, M, N)).
nwi_lp(C, M, N) :-                    nwd_cm(C, M), nwd_cm(C, N), not(nwi_cp(C, M, N)).
nwd_lp(C, M, N) :-        lp(C, M, N), not(del_lp(C, M, N)).
nwd_lp(C, M, N) :- add_lp(C, M, N).


% Derived from: cp(C, M, N) :- mf(M, F), cf(C, F), mf(N, F).

add_cp(C, M, N) :- add_mf(M, F),                     nwd_cf(C, F), nwd_mf(N, F), not(   cp(C, M, N)).
add_cp(C, M, N) :- add_cf(C, F), nwd_mf(M, F),                     nwd_mf(N, F), not(   cp(C, M, N)).
add_cp(C, M, N) :- add_mf(N, F), nwd_mf(M, F), nwd_cf(C, F),                     not(   cp(C, M, N)).
del_cp(C, M, N) :- del_mf(M, F),                      cf(C, F),      mf(N, F), not(nwi_cp(C, M, N)).
del_cp(C, M, N) :- del_cf(C, F),      mf(M, F),                      mf(N, F), not(nwi_cp(C, M, N)).
del_cp(C, M, N) :- del_mf(N, F),      mf(M, F),      cf(C, F),                 not(nwi_cp(C, M, N)).
nwi_cp(C, M, N) :-                   nwd_mf(M, F), nwd_cf(C, F), nwd_mf(N, F).
nwd_cp(C, M, N) :-        cp(C, M, N), not(del_cp(C, M, N)).
nwd_cp(C, M, N) :- add_cp(C, M, N).


% Direct transition rules for the model facts mf/2 and cf/2

nwd_mf(C, M) :- mf(C, M), not(del_mf(C, M)).
nwd_mf(C, M) :- add_mf(C, M).

nwd_cf(C, F) :- cf(C, F), not(del_cf(C, F)).
nwd_cf(C, F) :- add_cf(C, F).
```

Fig. 2. The derived update propagation and indirect and direct transition rules for the $LCOM1$ definition.

4.1 Rule Transformation in Prolog

The task of UP is to systematically compute the set of all induced changes, starting from the physical changes of base data. Technically, this is a set of delta facts for any affected predicate which may be stored in corresponding delta relations. For each predicate symbol `p`, we will use a pair of delta predicates `<add_p, del_p>` representing the insertions and deletions induced on `p` by an update. The initial set of delta facts represents the so-called *UP seeds*.

In the following, we briefly review a transformation-based approach to UP where the Prolog rules and the UP seeds are employed to derive *propagation rules* for computing delta relations. A propagation rule refers to at least one delta predicate in its body in order to provide a focus on the underlying changes when computing induced updates. For showing the effectiveness of an induced update, however, references to the state of a predicate before and after the base update has been performed are necessary.

For each predicate `p` we use `old_p` to refer to its old state before the changes given in the delta sets have been applied (technically the rule behind `old_p` is the unmodified version of `p`). We use `new_p` to refer to the new state of `p`. These state relations are never completely computed, but are queried with bindings from the delta sets in the propagation rule body, and thus act as a test of effectiveness. An induced insertion or induced deletion can be simply represented by the difference between the two consecutive database states. We consider the following Prolog rule:

```
p(X) :- q(Y),r(Z),not s(C).
```

The difference rules may look as follows:

```
add_p(X) :- add_q(Y), new_r(Z), not(new_m(C)), not(old_p(X)).
add_p(X) :- new_q(Y), add_r(Z), not(new_m(C)), not(old_p(X)).
add_p(X) :- new_q(Y), new_r(Z),    del_m(C),  not(old_p(X)).
```

The propagation rules basically perform a comparison of the old and new versions of the predicates. While providing a focus on insertions into q and r, all necessary combinations of delta and state predicates are considered. Because of the negative referenced predicate s, an additional rule has to be considered, which covers new derivations for p due to a deletion from s.

All propagation rules also contain the additional effectiveness test `not old_p(X)`, to check for the effectiveness of the induced insertions in case of alternative derivations of the same fact p in the old state. As an optimization we can drop the test, in case there are no alternative derivations, or if the set of, insertions can be overestimated. To avoid the full determination of state predicates, we should move the delta predicates as far left as possible in the rule body. This way the bindings provided by the delta facts can be used for restricting the evaluation of state predicates. This leads to the following rules:

```
add_p(X) :- add_q(Y), new_r(Z), not(new_m(C)).
add_p(X) :- add_r(Z), new_q(Y), not(new_m(C)).
add_p(X) :- del_m(C), new_q(Y),    new_r(Z).
```

For simulating the new predicate state from a given update and the old state, so called *transition rules* [12] can be used. The transition rules of a derived predicate infer its new state from the new states of the underlying predicates. Thus, for a rule $A : -L_1, \ldots, L_n$ a transition rule of the form $new\_A : -new\_L_1, \ldots, new\_L_n$ is considered. In contrast, for every extensional predicate A so-called incremental transition rules are used:

```
new_A :- old_A, not(del_A).
new_A :- add_A.
```

which explicitly refer to the computed changes to `A`.

For a rule $A : -L_1, \ldots, L_n$ we may also use the direct transition rules if there are no mutual dependencies between the predicate and the predicates in the body. In any case, the indirect transition rules of the form
`new_A :- new_`L_1`,...,new_`L_n
are used in the effectiveness test of negative propagation rules.

As an example, consider the following Prolog program

```
p(X) :- q(X,Y),r(Y),not(s(Y)).
        q(1,2). r(3). s(4).
        q(2,3). r(4). s(5).
        q(3,4). r(5). s(6).
```

and the insertion `r(2)` into relation r. The following propagation and transition rules

```
p(X) :- add_r(Y),new_q(X,Y),not(new_s(Y)).
new_q(X,Y) :- q(X,Y), not(del_q(X,Y)).
new_q(X,Y) :- add_q(X,Y).
new_s(X,Y) :- s(X,Y), not(del_s(X,Y)).
new_s(X,Y) :- add_s(X,Y).
```

were derived using the scheme described above. These rules allow for efficiently computing the induced insertion p(1) (represented by the fact $add\_p(1)$) by avoiding any redundant recomputations.

5 Propagation of Cohesion Model Updates

In this section, we apply UP as presented in Section 4 to our cohesion model and the metric rules from Section 2.

Before we employ UP to simulate the model state and metric results after the refactoring, we first need to generate delta facts as described in Section 3. At definition time of the metric rules, we may also derive the propagation and transition rules for UP. Figure 2 shows the result of the rule derivation process. We can see that for each body literal of a rule that appears in the metric definition, we create a positive and a negative propagation rule, so that for the cohesive pair rule `cp` we obtain six propagation rules in total. We replace negated delta literals simply by their opposite versions, for example: `not(del_cp)` ⇔ `add_cp`. For both rules `lp` and `cp`, we also see two derived transition rules `nwi_lp` and `nwi_cp`, simulating the versions of those predicates in the new state, which operate on the propagation rules.

5.1 Rule Generation

The UP rule generator component consists of two subcomponents, which can be seen in the middle of Figure 1. The **Rule Analyser** component collects meta information about the metric rules from the deductive rule modules and the **UP Rules Generator**, which creates the UP rules, based on the collected meta information. SWI-Prolog provides various meta predicates to examine loaded programs. It is important that UP ensures that the augmented rule set (which includes UP rules as shown in Figure 2 and the original rules like those for $LCOM1$) generated for a set of logic rules which is guaranteed to terminate, still keeps this property. This was shown for the language set of Datalog [1], [7], which only allows straight forward declarative rules, in comparison to Prolog. We also need to assure this in Prolog, it would be unfavourable if the UP rules got stuck in infinite loops. Prolog allows a broad variety of syntax constructs. For the metric definition, therefore, we only allow the model predicates and predicates defined in the template module itself and those from the cohesion model. We also allow a narrow list of built-in predicates, namely `=/2`, `member/2`

and the negation predicate `not/1`. We also do not allow any complex terms in the head of the rules, like: `cp(a(A),B,[H|T]) :- ....` Though this is a sharp restriction of Prolog, we were able to describe several structural cohesion metrics, as long as they contained a deductive part.

Rule Analyser First, the **Rule Analyser** determines all predicates defined in the template module which contains the $LCOM1$ metric rules we presented before. For $LCOM1$ those are `cp` and `lp`. The analyser collects various meta information, which we need to build the UP rules. An overview of the collected meta information:

`head_of_rule`($headId$, $groundHead$, [$name$, $arity$])
`body_predicate_of_rule`($headId$, $bodyId$,
                $positonInBody$, $groundPredicate$)
`rule_variables`($headId$, $bodyId$, $headBodyPrefix$,
                $positonInBody$, $groundPredicate$)

 `predicate_dependencies_transitive_closure`(
                [$name$, $arity$], $dependencies$)

We need to determine the free variables in the head and the body of the rules. We also need to check, if the rules contain self references. This is relevant to determine the positioning of the delta terms (`del_` and `add_` in the body of propagation rules). Second, in order to create the indirect transition rules `nwi_P` of a predicate `P`, we need to analyse, if there are transitive mutual dependencies between `P` and its body predicates L_i (where $0 \leq i$ ¡ #number of body predicates of `P`). The result will determine if we rather use `nwi_L`$_i$ or `nwd_L`$_i$ in `nwi_P`, this is important to ensure that the rules still terminate. We therefore create a predicate dependency graph.

UP Rules Generator Based on the information collected by the rule analyser, the **UP Rules Generator** creates the UP rules. As mentioned before, we create negative and positive Propagation Rules (for the metric), the direct Transition Rules (metric and model facts) and indirect Transition Rules (metric only). The rules created are asserted to a module. After completing the generation process, we also compile all rules to static predicates by using the Prolog `compile_predicates` predicate. The rule creation is based on string concatenation, therefore we convert all predicates to atoms, and prepend the necessary augmentations to each predicate.

6   Conclusions and Future Work

Earlier research had shown that the representation of object-oriented source code as a fact-base in a logic program allows for clear and concise implementation of static analyses of the object-oriented code. We explored the potential of this

approach further by applying the well-established theory of update propagation to it. Update propagation gives us a generic way to transform any analysis represented as a (sufficiently well-formed) logic program into an incremental version. Given an actual or hypothetical small change to the fact-base, this incremental version of the analysis provides an efficient way to caculate the new result of the analysis after the (hypothetical) change.

We implemented a transformation, generating the propagation and transition rules based on the original rules. Besides the applicability of update propagation to our setting, we already conducted several experiments to explore the performance benefits. In our experiments, update propagation was significantly faster than actually transforming our model and re-evaluating the original definition. As a future work, we intend a detailed study of the performance benefits.

We focused on the precalculation of the refactoring impact on metrics as this is in line with our research interest. Nevertheless, there is no reason to limit update propagation for logic meta programming to metrics or not even to refactoring. In the context of refactoring, update propagation could for example be used to verify that certain constraints will still hold true after the refactoring. This would be in line with the original motivation of update propagation.

A precise definition of the pairs of abstract refactorings on the model and the corresponding refactorings on the source code, could ease and clarify the nature of the induced updates model updates. More sophisticated refactorings also need complex preconditions, before they may be applied legally. We should be able to check those conditions on the level of sour model.

References

1. A. Behrend. Soft Stratification for Transformation-Based Approaches to Deductive Databases, University of Bonn: PhD thesis, 2004
2. A. Behrend, R. Manthey: Update Propagation in Deductive Databases Using Soft Stratification, ADBIS 2004: pp. 22-36, Springer.
3. L. C. Briand: A Unified Framework for Cohesion Measurement in Object-Oriented Systems, Empirical Software Engineering 3 (1), pp. 65-117, 1998
4. F. Bry, H. Decker, and R. Manthey: A uniform approach to constraint satisfaction and constraint satisfiability in deductive databases. In Proceedings of the International Conference on Extending Database Technology (EDBT '88), volume 303 of LNCS, pp. 488-505, Venice, Italy, March 1988. Springer.
5. Shyam R. Chidamber, Chris F. Kemerer: A metrics suite for object oriented design. IEEE Transactions on Software Engineering 20 (6), pp. 476-493, June 1994.
6. M. Fowler. Refactoring: Improving the Design of Existing Code. Addison-Wesley, 1999.
7. U. Griefahn. Reactive Model Computation, University of Bonn: PhD thesis, 1997
8. G. Kniesel, J. Hannemann, and T. Rho.: A Comparison of Logic-Based Infrastructures for Concern Detection and Extraction. In Proceedings of the 3rd workshop on Linking aspect technology and evolution (LATE 2007), New York, NY, USA, 2007.
9. V. Küchenhoff: On the efficient computation of the difference between consecutive database states, DOOD 1991: pp. 478-502, volume 566 of LNCS, Springer.

10. Elnar Hajiyev, Mathieu Verbaere, and Oege de Moor. CodeQuest: Scalable Source Code Queries with Datalog. In Dave Thomas, editor, ECOOP, volume 4067 of Lecture Notes in Computer Science, pages 227. Springer, 2006.
11. R. Manthey: Reflections on some fundamental issues of rule-based incremental update propagation, DAISD 1994: pp. 255-276, September 19-21.
12. A. Olivé: Integrity Constraints Checking In Deductive Databases, VLDB 1991: 513-523.
13. D. Speicher: Evolving Software Quality Knowledge, IC3K 2nd International Workshop on Software Knowledge - SKY 2011, Paris, France, 2011.
14. Roel Wuyts. A logic meta-programming approach to support the co-evolution of object-oriented design and implementation. PhD thesis, Vrije Universiteit Brussel, 2001.