Incremental inconsistency detection with low memory overhead

Jean-Rémy Falleri, Xavier Blanc, Reda Bendraou, Marcos Aurélio Almeida da Silva, Cédric Teyton

SOFTWARE—PRACTICE AND EXPERIENCE
Softw. Pract. Exper. 0000; **00**:1–22
Published online in Wiley InterScience (www.interscience.wiley.com). DOI: 10.1002/spe

Incremental inconsistencies detection with low memory overhead

Jean-Rémy Falleri[1*], Xavier Blanc[1], Reda Bendraou[2],
Marcos Aurélio Almeida da Silva[2] and Cédric Teyton[1]

[1] *University of Bordeaux, France*
[2] *Paris Universitas, France*

SUMMARY

Ensuring models' consistency is a key concern when using a model-based development approach. Therefore, model inconsistency detection has received significant attention over the last years. To be useful, inconsistency detection has to be sound, efficient and scalable. Incremental detection is one way to achieve efficiency in the presence of large models. In most of the existing approaches, incrementalization is done at the expense of the memory consumption, that becomes proportional to the model size and the number of consistency rules. In this paper, we propose a new incremental inconsistency detection approach that only consumes a small and model size-independent amount of memory. It will therefore scale better to projects using large models and many consistency rules. Copyright © 0000 John Wiley & Sons, Ltd.

Received . . .

KEY WORDS: Software engineering, Model driven engineering, Model consistency, Logic programming, Program analysis

1. INTRODUCTION

The use of large models in the design of complex systems introduces new challenges for the software industry. Ensuring their consistency along the development phases is one of them. This issue becomes more critical in a context where multiple meta-models and languages are used to design a system [1, 2]. In such a context, developers work simultaneously on complex and interdependent models. They must keep models free of inconsistencies. Indeed, editing actions that a developer performs on a model may not only introduce inconsistencies in this model but can also affect the consistency of other related models. Complex systems may contain up to 160 000 overlapping model elements spread in many models and this number is expected to increase in the near future [3]. As a consequence, without an appropriate tooling support, detecting inconsistencies quickly becomes a very tedious and time consuming task.

This situation draws some important requirements for an optimal tooling support for inconsistency management [4, 5, 6]. To be effective, inconsistency management should be *sound*, *efficient* and *scalable*. Sound means that inconsistencies detected by a tool should be correct, and not mislead the developers in their design tasks. This term is also referred to as correctness. Efficient means that detection has to be executed as fast as possible because developers always want to know the impact of their modification immediately after having performing them. Scalability is however a little more complex as a concern and usually defines specific requirements that are deemed important

*Correspondence to: falleri@labri.fr

Contract/grant sponsor: This work has been supported by the Movida ANR Project

depending on the context [7, 8]. In inconsistency management, scalability consists on the ability of the tool to handle the growing size of models, their complexity and the complexity of inconsistency rules that have to be checked on these models [4].

Many approaches dealing with inconsistency management exist in the literature. The first generation of approaches that tackle this challenge use inconsistency rules and execute them on models in batch [9, 10, 11, 12, 13, 14]. The limitations of these first generation approaches, also called batch checkers, is efficiency. The main problem is that each time a model is modified, all inconsistency rules have to be rechecked as a whole, which turns to be highly time consuming. As reported by [3], checking large models with batch checkers can then take hours to complete, which cannot be sustainable in effective development processes.

A second generation of approaches has then emerged to deal with efficiency. These approaches, also called incremental checkers, limit the set of rules to recheck (rule reduction) and/or the model elements to consider (scope reduction) after each model change [15, 16, 17, 18]. The most efficient approaches intensively use cache memory, thus introducing a memory overhead relative to the size of models and requiring a mechanism to manage the cache along the model's life cycle. An experimental study shows that one of the current fastest incremental checker has a cache that induces a linear memory overhead on several real-world UML models but that has a worst-case memory overhead that is quadratic in the number of elements [15]. With the increasing complexity and size of industrial design models, we believe that optimal memory management (scalability) in inconsistency management is as important as response time (efficiency). This is also confirmed by our industrial partners, which has realized a case study that confirms the critical need for more optimal memory management for large and complex models [19].

In this paper, we present an incremental checker that puts more emphasis on scalability rather than on efficiency. It has no cache and requires only a fixed memory overhead to perform both rule and scope reductions. Even if it cannot perform reduction for some cases (explained in Section 5) and is therefore less efficient than checkers that use cache for such cases, the intensive benchmark we have performed, using real large open source models, shows that it is fast enough (rechecks are performed in few seconds in average) to be used on several domains. We extend our previous checker [18] by adding the support of scope reduction. To that extent, it requires a *pre-instantiated impact list*, which is automatically generated and depends only on the number of inconsistency rules and on the size of the meta-model that have to be handled (i.e. few kilobytes for Java and UML case studies presented in this paper).

This paper is organized as follows. First, Section 2 explains what incremental checkers are and presents the related work. Section 3 presents our previous incremental checker. Section 4 then presents the pre-instantiation mechanism that supports scope reduction and that only requires a constant memory overhead. Section 5 presents the evaluation of our approach. Section 6 presents experiments we have done on UML and Movida models. Finally, Section 7 presents our conclusion.

2. EXISTING INCREMENTAL CHECKERS

Detection of inconsistencies consists in analysing a model to identify undesirable configurations defined by so-called inconsistency rules. If such configurations are detected in a model, then the model is said to be inconsistent. One can see an inconsistency check as a function that receives as input a model together with a set of inconsistency rules and produces as output the evaluation result for each rule. If a rule evaluates to true (i.e., the model is inconsistent), then the model elements causing the inconsistency are returned by the check function.

Inconsistency rules can be compared to the negation of well-formedness rules of [4], structural rules of [12] and syntactic rules of [20]. FindBugs defines such rules for Java.[†] For example, the `aom` rule defines that a Java class must not override an existing non-abstract method and make it abstract, and the `pcoa` rule defines that a Java constructor must not call non-final methods.

[†] `http://findbugs.sourceforge.net/bugDescriptions.html`

Prepared using speauth.cls
DOI: 10.1002/spe

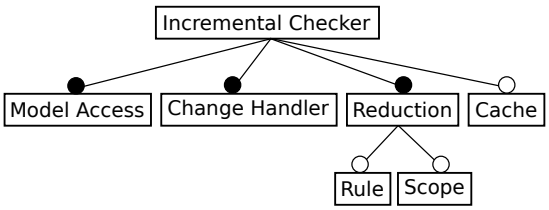


Figure 1. Feature diagram of incremental checkers.

Incremental checkers address efficiency by limiting the set of rules to recheck (rule reduction) and/or the model elements to consider (scope reduction). For example, considering the `aom` and `pcoa` rules, if a model is changed by making a method final, then needless to say that only the rule `pcoa` has to be rechecked (rule reduction) and only for constructors calling the modified method (scope reduction). Indeed, it is useless to check the `aom` rule because final methods have no impact on it, and it is useless to recheck all existing constructors while rechecking `pcoa`, as many of them do not call the modified method.

Figure 1 presents the main features of incremental checkers. For the sake of clarity, we use feature modelling to present them [21]. To perform a recheck, an incremental checker must have two mandatory components: it must have an access to the model and it must handle changes. Moreover, it must perform reductions on the rule or/and on the scope. Finally, to optimize these reductions in order to be faster, it may use a cache memory, which will irremediably introduce memory overhead.

As the purpose of our paper is not to present a deep comparison and analysis of all incremental checkers, we use Figure 1 to provide a simple classification of incremental checkers with the intent to state whether an approach supports scope and/or rule reduction, and if it either puts emphasis on efficiency (with a cache) or on scalability (without a cache).

Among existing incremental checkers, two support only rule reduction. Jouault and Tisi provide an approach for the incremental execution of transformations written in the Atlas Transformation Language (ATL) [22]. Their approach performs rule reduction by constructing an impact list that links every class of possible modification with transformation rules that should be re-executed. In [18], we provide a rule reduction approach on top of our inconsistency checker [14]. This approach is fully described in Section 3.

Six existing incremental checkers put emphasis on efficiency by using a cache memory to optimize both rule and scope reduction. In [15], Egyed provides an incremental checker for UML models. The cache includes, for each rule execution, the first model element used by the rule, and any other model elements used to check the rule. In [23], Rose et al. describe an incremental checker that is dedicated to a single inconsistency rule. This approach is similar to the Egyed's one but optimizes the cache according to the specificity of this single rule. In [24], Eichberg et al. propose an incremental checker that is based on a tabled Prolog (XSB Prolog) in order to automatically update rules evaluation. A tabled Prolog stores rule evaluations in a cache memory. In [25], Menet et al. provide an incremental checker for EMF models. This checker uses a cache that stores model elements used to check inconsistency rules. In [26], Bergmann et al. provide an incremental checker that has a cache optimized to handle modification events. Finally, in [17, 27], Xu et al. propose an incremental checker that analyses which part of a rule needs to be re-checked once a model has been modified. This approach also uses a cache memory that links model elements to the parts of the rules that use them.

It should be noted that, whatever the approach, a cache must store a number of model elements that depends on the rule and on the size of the model. For example, considering the `aom` rule, a cache will be created for each Java class contained in the model. Each cache will then contain the methods of the class, all its super-classes, and all the methods of the super-classes. The `aom` rule has therefore a memory complexity that depends on the number of classes and on the number of super-classes and contained methods. Such a rule has a low complexity as it only requires a small cache. However, a worst case occurs when a rule meets the two following conditions. First, the rule has to be checked on all elements. Second, for each element, the cache must contain all other elements.

DOI: 10.1002/spe

Approach	Rule	Scope	Cache
Jouault	✓		
Blanc	✓		
Egyed	✓	✓	✓
Rose	✓	✓	✓
Eichberg	✓	✓	✓
Menet	✓	✓	✓
Bergmann	✓	✓	✓
Xu	✓	✓	✓
Wagner	✓	✓	
Cabot	✓	✓	

Table I. Comparison of existing incremental checkers

For instance, a rule as simple as the *unique id* rule, which checks if an element has a unique id whatever its relationships with other model elements, is such a worst case rule. When this rule has to be checked, a cache that contains all the elements of the model is created for each element that has to be checked. Such a rule has therefore a $n(n-1) = O(n^2)$ memory complexity, n being the number of elements contained in the model. This can be very memory-consuming especially with a large model. If we imagine that 4 bytes pointers to the model elements are used in the cache, the memory consumption for this rule, for a model of 20000 elements, is: $\frac{4\times(20000)^2}{1024\times1024\times1024} = 1.5$ gigabytes, which is quite expensive for only one rule.

Finally, two approaches put more emphasis on scalability rather than on efficiency as they support both rule and scope reductions without any cache memory [28, 16]. These approaches are based on a static analysis of the rules to deduce reductions when a modification occurs. The provided analyses are conservative to guarantee the soundness of the approach. The main difference between these two approaches comes from the language used to express inconsistency rules and therefore on the static analysis that can be done on it. The Wagner's approach is based on graph grammar patterns [28], while the Cabot's one on OCL [16]. Even if these approaches are designed to be scalable and claimed to efficient, no performance measurements are provided on real models. Moreover, no analysis is provided to explain when reductions cannot be performed.

Table I clearly shows that there are three groups of incremental checkers. Our proposal belongs to the third one, which aims to support rule and scope reduction without any cache memory. As the Wagner's and Cabot's approaches, our approach is based on a static analysis of inconsistency rules. However, unlike these approaches, first, we use a general purpose programming language to express inconsistency rules (Prolog), and second, we provide a strong validation for efficiency and scalability.

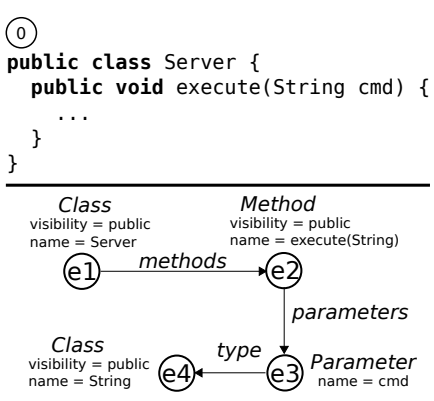
3. PRAXIS AND RULE REDUCTION

In this section, we first present our approach called Praxis that is based on what we call *editing actions* to represent any model [14]. Then, we demonstrate how Praxis can be used to detect inconsistencies. Finally, we briefly present our previous incremental checker that supports rule reduction [18]. This checker is used as a basis of our new proposal.

3.1. Praxis model

We consider a model, whatever its meta-model, as a tagged digraph of elements with a set of associated properties. For instance, Figure 2 presents a Java source code (top-left) and its corresponding digraph (bottom-left). Praxis represent the digraph in the form of a sequence of editing actions that led to its construction. To that extent, it defines six editing actions for creating or deleting the elements, properties and references that compose the digraph:

DOI: 10.1002/spe




```
1  create(e1,'Class').
2  addProperty(e1,'name','Server').
3  addProperty(e1,'visibility','public').
4  create(e2,'Method').
5  addProperty(e2,'name','execute(String)').
6  addProperty(e2,'visibility','public').
7  addProperty(e2,'abstract','false').
8  create(e3,'Parameter').
9  addProperty(e3,'name',cmd).
10 create(e4,'Class').
11 addProperty(e4,'name','String').
12 addProperty(e4,'visibility','public').
13 addProperty(e4,'abstract','false').
14 addReference(e1,'methods',e2).
15 addReference(e2,'parameters',e3).
16 addReference(e3,'type',e4).
```

Figure 2. A Java source code (top-left), its corresponding digraph (bottom-left), and its corresponding Praxis sequence (right).

- `create(e,t)` (resp. `delete(e,t)`) creates (resp. deletes) a model element e tagged t. A created (resp. deleted) element has neither properties nor references with any other elements in the model.
- `addProperty(e,t,v)` (resp. `remProperty(e,t,v)`) assigns (resp. removes) the value v to the property tagged t of the model element e.
- `addReference(e,t,r)` (resp. `remReference(e,t,r)`) assigns (resp. removes) a target model element r to the reference tagged t for the model element e.

One key advantage of Praxis is that the level of details of elements, properties and references to represent is left open. In Praxis, all elements, properties and references have a tag, but the choice of tags is left up to the chosen instantiation of the meta-model. For instance, one can decide to limit the Praxis representation of a Java code to elements such as *classes* and *methods*, or one can decide to go further in details and represent *field accesses* and *method calls* performed in methods's bodies.

Praxis editing actions used to describe a given model are totally ordered. This order is provided by the editor (the CASE Tool for example), which tracks the editing actions performed by the developers. Figure 2 (right) presents the *Praxis construction sequences* generated using the Java code of Figure 2 (top-left). This example and the UML one presented in section 6 highlight the fact that Praxis is meta-model independent.

3.2. Praxis rules

In Praxis, inconsistency rules are written in first order logic and target a construction sequence of editing actions. They are implemented as Prolog rules with the following scheme: `pr_name(`$P_1, \ldots, P_N$`) :- formula`, where `name` is the name of the rule, $P_1, \ldots, P_N$ the variables identifying inconsistent elements, and `formula` a Prolog formula. In the remainder of the article, identifiers starting with a lower case letter or a simple quote are constants (i.e. e or `'Class'`) and identifiers starting with an upper-case letter are variables (i.e. M_1). The semantics of a Praxis rule is that, if the Prolog engine can prove the formula to be true, the inconsistency `pr_name(`$P_1, \ldots, P_N$`)` is considered to be present in the model. Moreover, the elements associated with $P_1, \ldots, P_N$ returned by the Prolog engine as formula instances are the elements involved in the inconsistency. Figure 3 presents the `aom` Praxis inconsistency rule. Remember that this rule specifies that a class must not override an existing method and make it abstract. It has four variables: C_1, C_2, M_1 and M_2 that respectively represent the class, its super class, the abstract method and the non-abstract overridden one. If we execute this rule over the sequence presented in Figure 2 (right), it returns false, which means that there is no inconsistency in the model.

Praxis rules use what we call *Praxis basic predicates* to query actions of the construction sequence. To that extent, we have defined three basic predicates. The `existElement(E,T)`

DOI: 10.1002/spe

```
1  pr_aom(C1,C2,M1,M2) :-
2    existReference(C1,'methods',M1),
3    existProperty(M1,'name',N),
4    existProperty(M1,'abstract','true'),
5    subtype(C1,C2),
6    existReference(C2,'methods',M2),
7    existProperty(M2,'name',N),
8    existProperty(M2,'abstract','false').
9
10 subType(C1,C2) :- directSubType(C1,Z), subtype(Z,C2).
11 subType(C1,C2) :- directSubType(C1,C2).
12
13 directSubType(C1,C2) :- existReference(C1,'extends',C2).
```

Figure 3. The `aom` rule (abstract overridden method) for Praxis.

predicate has been defined to find a `create(E,T)` action that is not followed by a corresponding cancelling `delete(E,T)` action in the sequence. Similarly, the `existProperty(E,T,V)` and the `existReference(E,T,R)` predicates have been defined to find `addProperty(E,T,V)` and `addReference(E,T,R)` actions which have not been cancelled. The only tiny restriction we impose is that the basic predicates has to be used with a constant tag (i.e. like in `create(E,'class')`). This restriction is used to enable the static analysis explained later. The `aom` rule uses six Praxis basic predicates.

Finally, Praxis rules can use predicates that have been defined to query specific constructions. For instance, the `subType(`C_1, C_2`)` predicate, used by the `aom` rule, has been defined to query an inheritance relationships between two Java classes. This predicate makes use of Prolog recursion and calls `directSubtype(`C_1, C_2`)`, another predicate, which calls the `existReference(`C_1`,'extends',`C_2`)` Praxis basic predicate.

The execution of Praxis rules proceeds as follows [29]. The Prolog inference engine tries to prove the formula by assigning variables of the formula with facts, which correspond to the editing actions of the construction sequence. The Prolog algorithm consists in backtracking over all the possible variable assignments until it finds one solution. At this stage, one can ask to the Prolog engine to find other solutions if they exist. Since Prolog is a Turing-complete programming language, any inconsistency rule that can be described in a programming language can also be described by means of Praxis inconsistency rule.

3.3. Rule reduction

In [18], we propose a solution to reduce the set of Praxis rules that have to be re-executed after that a model has been changed. The idea is to statically define the effects a change may have regarding the inconsistencies present in a model. If a change has no impact on inconsistency, there is then no need to re-execute the Praxis rules. In other words, Praxis rules have to be re-executed only when changes may have an impact on the inconsistencies they define.

The effect that an editing action may have on inconsistency rules is described in what we call an *impact list*. This list indicates which rules need to be rechecked when a new editing action is added to an existing construction sequence. There is an infinite amount of such possible editing actions, because of argument values (imagine `addProperty(`e_1`,'name','Foo')` and `addProperty(`e_1`,'name','Bar')`). To bound the size of the impact list, we have defined a finite number of editing action equivalence classes, called *action classes*.

Two editing actions are equivalent if and only if:

- they create or delete two model elements with a same tag
- they change two references with a same tag
- they change two values of properties with the same tag

The elements of the impact list are thus ordered pairs (c, R_c) where c is an action class and R_c is the set of rules that have to be rechecked for an editing action

DOI: 10.1002/spe

Action class (c)	Rules to recheck (R_c)
referenceAction(*,methods,*)	{aom(C_1, C_2, M_1, M_2)}
propertyAction(*,name,*)	{aom(C_1, C_2, M_1, M_2)}
propertyAction(*,abstract,*)	{aom(C_1, C_2, M_1, M_2)}
referenceAction(*,extends,*)	{aom(C_1, C_2, M_1, M_2)}

Table II. The impact list corresponding to the aom rule of Figure 3.

```
(1)
public class Server {
  public void execute(String cmd) {
    ...
  }
}

public class MyServer extends Server {
  public abstract void execute(String cmd);
}
```

```
1  create(e5,'Class').
2  addProperty(e5,'name','MyServer').
3  addProperty(e5,'visibility','public').
4  addReference(e5,'extends',e1).
5  create(e6,'Method').
6  addProperty(e6,'name','execute').
7  addProperty(e6,'visibility','public').
8  addProperty(e6,'abstract','true').
9  create(e7,'Parameter').
10 addProperty(e7,'name',cmd).
11 addReference(e5,'methods',e6).
12 addReference(e6,'parameters',e7).
13 addReference(e7,'type',e4).
```

Figure 4. Evolution of the code in Figure 2 (left) and its corresponding Praxis sequence (right).

belonging to this class. The number of action classes only depends on the meta-model. We denote these action classes using elementAction(*,t), propertyAction(*,t,*) and referenceAction(*,t,*). For instance, create(e_2,'Class') belongs to action class elementAction(*,'Class'), whereas addReference(e_1,'methods',e_2) belongs to the action class referenceAction(*,'methods',*).

The *impact list* is automatically generated from the meta-model and the inconsistency rules [18]. The construction process guarantees that all possible effects that actions may have on rules are captured. Nevertheless, useless checks can be introduced in the list. Table II shows the *impact list* for the aom rule of Figure 3. Four action classes may have an impact on the rule. It should be noted that the referenceAction(*,extends,*) has an impact for the subType predicate.

The limitation of this rule reduction approach is that a rechecked rule always considers the whole model whenever an editing action triggers it. For instance, let us imagine that the code is modified as shown in Figure 4 (left). This modification is represented as the construction sequence shown in Figure 4 (right). In this construction sequence, the first action (create(e_5,'Class')) has no effect on the aom rule. On the contrary, the second action (addProperty(e_5,'name','MyServer')) will trigger the execution of the aom rule. However, checking the whole model is unnecessary since only e_5 has been modified in this case.

Alternatively, scope reduction approaches allow a rule to recheck only a subset of the whole model. In the present example, the aom(C_1, C_2, M_1, M_2) rule would need to consider only the e_5 class when triggered. This is the objective of our scope reduction approach presented in the next section.

4. SCOPE REDUCTION

To reduce the scope of the model to be rechecked, our approach proposes to pre-instantiate some variables of the Praxis rules before rechecking them. Variables to pre-instantiate depends on elements modified by editing actions triggering the rules. In this section, we first explain how we modify our previous impact list to take into account variable pre-instantiations. This new impact list is called a *pre-instantiated impact list*. Then, we describe how we automatically generate this

DOI: 10.1002/spe

$$\frac{P_a(x_1,\ldots,x_n) \quad :- \quad \Gamma\ P_b(y_1,\ldots,y_m)\ \Theta}{P_a <_{m_i} P_b, m_i : [1,m] \rightarrow [1,n], m_i(j) = k \text{ iff } y_j = x_k}$$

Table III. Call dependency formula. P_a calls P_b. Γ and Θ specify any Prolog formulae that can surround P_b.

pre-instantiated impact list and use it to recheck rules when editing actions are performed. Lastly, we explain how we keep the global consistency diagnosis of a model up-to-date.

4.1. Computing the pre-instantiated impact list

The pre-instantiated impact list extends the impact list (presented in Section 3) to support scope reduction. The first extension made by the *pre-instantiated impact list* is to make the elements modified by the editing actions explicit. Thus, we change the description of action classes by assigning identifiers to the modified model elements. Since an editing action has an impact on at least one element, called the source element, we denote `$s` this element in any action class. Additionally actions that change references also have an impact on a second element, called the referenced element. We denote `$r` this element. Action classes are then denoted by `elementAction($s,t)`, `propertyAction($s,t,*)` and `referenceAction($s,t,$r)`. For instance the action `addReference(`e_1`,'methods',`e_2`)` belongs to the action class `referenceAction($s,'methods',$r)`. This action modifies two elements `$s`$= e_1$ and `$r`$= e_1$

The second extension made by the pre-instantiated impact list states how elements modified by a change can be used to limit the scope of the recheck. The idea is to pre-instantiate rule variables by modified model elements. Thanks to such a pre-instantiation, only a sub-part of the whole model is analyzed during a recheck. A pre-instantiation is an ordered triple (r, c, b) where r is a rule, c is an action class and b is a partial function that binds modified elements of c to variables of r. For example, given the `aom` rule, we note (`aom(`C_1, C_2, M_1, M_2`)`,`propertyAction($s,'name',*)`,{`m($s)`=M_1}) a pre-instantiation for `aom` with the action class `propertyAction($s,'name',*)` that binds the modified element `$s` to the variable M_1. Such a pre-instantiation specifies that, when an editing action that belongs to the class `propertyAction($s,'name',*)` is performed, then the `aom` rule has to be rechecked by pre-instantiating its variable M_1 with `$s`. Such a pre-instantiation drastically reduces model elements to look at during the recheck.

Pre-instantiations can be automatically generated by performing a static analysis of the Prolog code that defines Praxis rules. The idea is to identify, for each Praxis rule, all the Praxis Basic predicates it may call, and how variables of the rule are transferred to them. For example, by making a static analysis of the Prolog code depicted in Figure 3, we can identify that the `aom` rule may call the Praxis basic predicate `propertyAction($s,'name',*)`, and, when it does so, its M_1 variable will be bound to `$s`. This means that an editing action belonging to this class may have an impact on the evaluation of the rule. Therefore, the rule should be rechecked each time such an action is perfomed, scoped by pre-instantiating M_1 with `$s`.

More formally, the analysis we perform aims to identify call dependencies among Prolog predicates. Let P_a be a predicate that calls another predicate P_b and let m_i be the corresponding mapping function (partial surjection from the indexes of P_b's variables to the indexes of P_a's variables). We denote by $P_a <_{m_i} P_b$ a call dependency from P_a to P_b with m_i as the binding function (we note $m_\emptyset$ the binding function that binds none of the variable). Such a behaviour is defined in Table III. The premise of the formula presents a predicate P_a that calls another predicate P_b (P_b can be defined among other predicates Γ and Θ). The conclusion of the formula then defines m_i, the mapping function such that $P_a <_{m_i} P_b$.

The identification of call dependencies in a Prolog source code is then a four step process. The first step consists in creating a call dependency for each direct call that exists in the Prolog source code. Table V presents the call dependencies identified after step one has been performed on Prolog source code of Figure 3. Note that we consider only one predicate for each

DOI: 10.1002/spe

$$\frac{P_a(x_1,\ldots,x_n) <_{m_i} P_b(y_1,\ldots,y_m) \quad P_b(y_1,\ldots,y_m) <_{m_j} P_c(z_1,\ldots,z_o)}{P_a(x_1,\ldots,x_n) <_{m_i \circ m_j} P_c(z_1,\ldots,z_o)}$$

Table IV. Transitivity of call dependencies formula.

Call dependency	Mapping function
aom $<_{m_1}$ existReference('methods')	$m_1(1) = 1, m_1(3) = 3$
aom $<_{m_2}$ existProperty('name')	$m_2(1) = 3$
aom $<_{m_3}$ existProperty('abstract')	$m_3(1) = 3$
aom $<_{m_4}$ subType	$m_4(1) = 1, m_4(2) = 2$
aom $<_{m_5}$ existReference('methods')	$m_5(1) = 2, m_4(3) = 4$
aom $<_{m_6}$ existProperty('name')	$m_6(1) = 4$
aom $<_{m_7}$ existProperty('abstract')	$m_7(1) = 4$
subType $<_{m_8}$ directSubType	$m_8(1) = 1$
subType $<_{m_9}$ subType	$m_9(2) = 2$
subType $<_{m_{10}}$ directSubType	$m_{10}(1) = 1, m_{10}(2) = 2$
directSubType $<_{m_{11}}$ existReference('extends')	$m_{11}(1) = 1, m_{11}(3) = 2$
aom $<_{m_{12}}$ directSubType	$m_{12}(1) = 1$
aom $<_{m_{13}}$ subType	$m_{13}(2) = 2$
aom $<_{m_{14}}$ directSubType	$m_{14}(1) = 1, m_{14}(2) = 2$
subType $<_{m_{15}}$ existReference('extends')	$m_{15}(1) = 1$
subType $<_{\emptyset}$ directSubType	
subType $<_{m_{16}}$ existReference('extends')	$m_{16}(1) = 1, m_{16}(3) = 2$
aom $<_{m_{17}}$ existReference('extends')	$m_{17}(1) = 1$
aom $<_{\emptyset}$ directSubType	
aom $<_{m_{18}}$ existReference('extends')	$m_{18}(1) = 1, m_{18}(3) = 2$
subType $<_{\emptyset}$ existReference('extends')	
subType $<_{m_{19}}$ existReference('extends')	$m_{18}(3) = 2$
aom $<_{\emptyset}$ existReference('extends')	
aom $<_{m_{20}}$ existReference('extends')	$m_{20}(3) = 2$
aom $<_{m_{21}}$ directSubType	$m_{21}(2) = 2$

Table V. The intitial call dependencies for the `aom` rule (top) and the ones added after the computation of the fixed point (bottom).

action class. For instance, we consider that the `existReference('methods')` predicate represents any of the `existReference($s,'methods',$t)` predicates. Regarding the `aom` predicate, it directly calls the `existReference('methods')` predicate and the third variable of `existReference('methods')` is bound to the first variable of `aom` (m_1 is defined in extenso: $m_1(3) = 1$). The second step consists in composing transitive call dependencies until we reach a fixed point. Transitive calls are computed using the formula given in Table IV. The premise of this formula presents two dependencies $P_a <_{m_i} P_b$ and $P_b <_{m_j} P_c$, while its conclusion defines the corresponding transitive dependency $P_a <_{m_i \circ m_j} P_c$. As there is a finite set of predicates and binding functions, and as composition of existing predicates generates predicates remaining in this same finite set, a fixed point exists and is always reached. Table V presents the new call dependencies identified after step 2.

The third step consists in keeping only call dependencies that exist from a Praxis rule to a Praxis basic predicate, as illustrated in Table VII. The fourth step finally consists in filtering out all subsumed binding functions. We consider that a binding function m_i subsumes another one m_j iff $m_i \sqsubseteq m_j$, as shown in Table VI. The premise of this formula defines two dependencies between two same predicates ($P_a <_{m_i} P_b$ and $P_a <_{m_j} P_b$) where $m_i \sqsubseteq m_j$. The conclusion then defines that $P_a <_{m_i} P_b$ subsumes $P_a <_{m_j} P_b$. The goal of this last step is to keep binding functions that bind fewer variables in order to be sure of the size of the scope (the less bound variables, the bigger the scope). Table VII (bottom) presents the result of this final step on the `aom` rule. The call dependencies identified after this final step can be easily translated into a pre-instantiated impact list (see Table VIII).

Prepared using speauth.cls
DOI: 10.1002/spe

$$\frac{P_a(x_1,\ldots,x_n) <_{m_i} P_b(y_1,\ldots,y_m) \quad P_a(x_1,\ldots,x_n) <_{m_j} P_b(y_1,\ldots,y_m) \quad m_i \sqsubseteq m_j}{P_a(x_1,\ldots,x_n) <_{m_i} P_b(y_1,\ldots,y_m)}$$

Table VI. Subsumption of call dependencies formula.

Call dependency	Mapping function
aom $<_{m_1}$ existReference('methods')	$m_1(1) = 1, m_1(3) = 3$
aom $<_{m_2}$ existProperty('name')	$m_2(1) = 3$
aom $<_{m_3}$ existProperty('abstract')	$m_3(1) = 3$
aom $<_{m_5}$ existReference('methods')	$m_5(1) = 2, m_4(3) = 4$
aom $<_{m_6}$ existProperty('name')	$m_6(1) = 4$
aom $<_{m_7}$ existProperty('abstract')	$m_7(1) = 4$
aom $<_{m_{17}}$ existReference('extends')	$m_{17}(1) = 1$
aom $<_{m_{18}}$ existReference('extends')	$m_{18}(1) = 1, m_{18}(3) = 2$
aom $<_{\emptyset}$ existReference('extends')	
aom $<_{m_{20}}$ existReference('extends')	$m_{20}(3) = 2$
aom $<_{m_1}$ existReference('methods')	$m_1(1) = 1, m_1(3) = 3$
aom $<_{m_2}$ existProperty('name')	$m_2(1) = 3$
aom $<_{m_3}$ existProperty('abstract')	$m_3(1) = 3$
aom $<_{m_5}$ existReference('methods')	$m_5(1) = 2, m_4(3) = 4$
aom $<_{m_6}$ existProperty('name')	$m_6(1) = 4$
aom $<_{m_7}$ existProperty('abstract')	$m_7(1) = 4$
aom $<_{\emptyset}$ existReference('extends')	

Table VII. Call dependencies to Praxis basic predicates (top) and the ones remaining after having eliminated the useless ones (bottom).

Action class (c)	Rules to recheck (R_c)
`referenceAction($s,'methods',$r)`	{`aom(`C_1`=$s,`C_2`,`M_1`=$r,`M_2`)`,`aom(`C_1`,`C_2`=$s,`M_1`,`M_2`=$r)`}
`propertyAction($s,'name',*)`	{`aom(`C_1`,`C_2`,`M_1`=$s,`M_2`)`,`aom(`C_1`,`C_2`,`M_1`,`M_2`=$s)`}
`propertyAction($s,'abstract',*)`	{`aom(`C_1`,`C_2`,`M_1`=$s,`M_2`)`,`aom(`C_1`,`C_2`,`M_1`,`M_2`=$s)`}
`referenceAction($s,'extends',$r)`	{`aom(`C_1`,`C_2`,`M_1`,`M_2`)`}

Table VIII. The pre-instantiated impact list returned for the `aom` rule.

We claim that our static analysis is sound as it ensures that no relevant recheck is missed. Indeed, it considers any potential call dependency without considering the logical connectives between them. Therefore it identifies even call dependencies that would not happen in practice. To be applied, our analysis requires the complete Prolog source of the rules. Only a few Prolog pecularities can have an effect on our analysis. Variable aliasing is possible in Prolog and will decrease the efficiency of our analysis if used extensively. Higher-order programming (in Prolog, it is possible to manipulate predicates as values) prevents our analysis from working, if used.

4.2. Using the pre-instantiated impact list

With the pre-instantiated impact list, performing a recheck is straightforward. When a new action is added to the construction sequence, the pre-instantiated rules are searched for in the list, according to the action class, and variables are substituted by modified elements if any. As editing actions never come one by one in the real world, rechecks are almost always triggered from a set of actions added to the construction sequence, i.e. after the file was saved. In this case, first we compute the rule rechecks as if actions were added one by one. Then we compute the union of the rule rechecks and remove from the resulting set the rechecks that are subsumed by any other one. A recheck is subsumed by another one if the two rechecks target the same rule and if the bound variables of the subsumed recheck are bound in a same way in the subsuming recheck, or if they are free variables. For instance `aom(`$C_1 = e_1, C_2 = e_2, M_1, M_2$`)` is subsumed by `aom(`$C_1 = e_1, C_2, M_1, M_2$`)` and by `aom(`$C_1, C_2 = e_2, M_1, M_2$`)`. `aom(`$C_1 = e_1, C_2, M_1, M_2$`)` is not subsumed by `aom(`$C_1, C_2 =$

DOI: 10.1002/spe

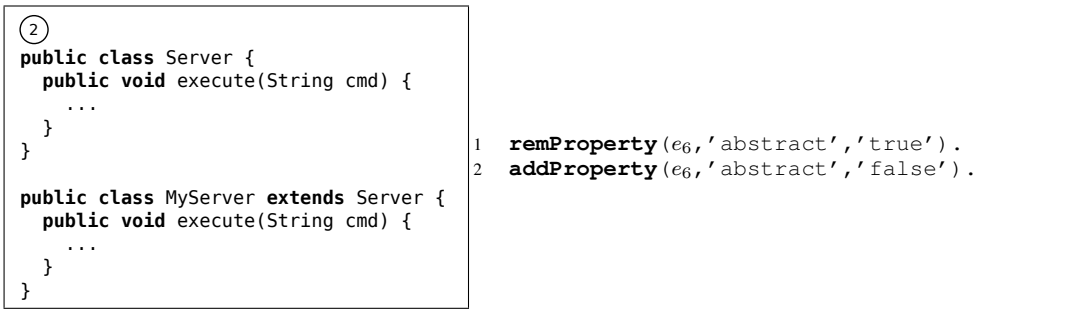


Figure 5. Evolution of the code in Figure 4 (left) and its corresponding Praxis sequence (right).

$e_2, M_1, M_2)$. aom$(C_1 = e_1, C_2 = e_2, M_1, M_2)$, aom$(C_1 = e_1, C_2, M_1, M_2)$ and aom$(C_1, C_2 = e_2, M_1, M_2)$ are all subsumed by aom(C_1, C_2, M_1, M_2).

As an example, Figure 4 presents a modification performed on our Java sample (presented in Figure 2). This modification creates a new class, named `MyServer`, that extends the `Server` class and that overrides the `execute(String)` method and makes it abstract. Four editing actions of this modification trigger the `aom` rule. Among them, the editing action `addReference(`e_5`,'extends',`e_1`)` triggers the recheck of aom(C_1, C_2, M_1, M_2). As this recheck subsumes all the other ones, it is the only recheck that is executed. It returns the inconsistency aom$(C_1 = e_5, C_2 = e_2, M_1 = e_6, M_2 = e_2)$.

4.3. Global consistency diagnostic update

Finally, a pre-instantiated inconsistency recheck only provides a partial diagnostic, related to model elements that are source or target of a change, and related to rules that have been impacted by the change. A global diagnostic considers all models elements and all the inconsistency rules. In this section, we present how to use a partial diagnostic to keep up-to-date the global diagnostic.

An inconsistency is defined in Praxis as a *fully instantiated rule*. A diagnostic, partial or global, is therefore a set of fully instantiated rules. A partial diagnostic updates a pre-existing global diagnostic. The two following steps describe the update process:

1. All the *fully instantiated rules* of the pre-existing global diagnostic subsumed by the rechecked pre-instantiated rule are removed.
2. All returned *fully instantiated rules* of the partial diagnostic are added to the global diagnostic.

As an example, Figure 5 presents a second modification performed on our Java sample. This modification gives an implementation to the `execute(String)` method to fix the inconsistency. The right part of Figure 5 highlights the fact that in Praxis, two editing actions are executed to change the value of a property. The first one removes the existing value (remProperty) and the second one adds its new value (addProperty). As these two editing actions belong to the same action class, they trigger the same two rechecks: aom$(C_1, C_2, M_1 = e_6, M_2)$ and aom$(C_1, C_2, M_1, M_2 = e_6)$. Following the two steps process, the inconsistency aom$(C_1 = e_5, C_2 = e_2, M_1 = e_6, M_2 = e_2)$ is removed from the global diagnostic as it is subsumed by the first recheck. Then, as the executions of these two rechecks return nothing, the global diagnostic is finally empty.

5. EVALUATION

This section presents the evaluation of our approach. We start by presenting our prototype. Then we describe an experiment in which we use our prototype to check *FindBugs* rules against real java project. In this experiment, we measure the time and memory performances of our approach. Additionally, we perform a fine-grained analysis of the efficiency of our scope reduction technique on the rules we implemented. Finally we present the threats to the validity of this study.

Prepared using speauth.cls
DOI: 10.1002/spe

5.1. Prototype implementation

Our prototype has been released as a set of Eclipse plugins that are publicly available[‡]. These plugins use a Prolog engine to execute the inconsistency rules. All Praxis rules are implemented as Prolog rules and Praxis actions are translated into Prolog facts. Our prototype currently uses SWI-Prolog and its Java bridge.

Our prototype supports the following meta-models: Java, UML and Movida[§]. For each meta-model, there is an Eclipse plugin that translates the models into Praxis actions and that contains the dedicated inconsistency rules in a Prolog file. The pre-instantiated impact list is specified using a XML document, automatically produced by a static analysis of the Prolog code. For example, Listing 1 presents the `aom`(C_1, C_2, M_1, M_2) rule. In this XML document, action classes that may have an impact of the rule are given using the `trigger` elements. The `sourceElementBinding` and `targetElementBinding` attributes correspond to the `$s` and `$t` elements. Their value corresponds to the index of the rule's argument bound to them.

```
1  <rules>
2  <rule name="aom" arity="4">
3    <trigger class="REFERENCE_ACTION" tag="extends"/>
4    <trigger class="PROPERTY_ACTION" tag="name" sourceElementBinding="3"/>
5    <trigger class="PROPERTY_ACTION" tag="name" sourceElementBinding="2"/>
6    <trigger class="PROPERTY_ACTION" tag="annotation" sourceElementBinding="2"/
         >
7    <trigger class="REFERENCE_ACTION" tag="methods" sourceElementBinding="0"
         targetElementBinding="2"/>
8    <trigger class="PROPERTY_ACTION" tag="abstract" sourceElementBinding="2"/>
9    <trigger class="REFERENCE_ACTION" tag="methods" sourceElementBinding="1"
         targetElementBinding="3"/>
10   <trigger class="PROPERTY_ACTION" tag="abstract" sourceElementBinding="3"/>
11 </rule>
12 </rules>
```

Listing 1: Pre-instantiated impact list for the `aom` rule

5.2. Corpus

In this section, we present the corpus we used in our experiment. This corpus contains a set of Java projects associated with modifications that have been performed to them, which have been gathered from source code repository. The corpus also contains a set of inconsistency rules, extracted from the well known FindBugs tool. Note that this is the first reproducible experiment on an incremental consistency checker that uses real data for models, rules and modifications. For example, [17], [14] and [15] uses only generated models and/or modifications.

Projects and modifications Our corpus is composed of four medium to large Java open source projects: Migen, JFreeChart, CruiseControl and Struts. Information about our project corpus is shown in Table IX. We used the version control system of these projects to extract the modifications done by developers. We chose to extract the 100 latest revisions for ensuring the gathering of medium to large sized models. Our study starts with the revision `head`-100 (`head` is the last committed revision in a repository). Then each successive revision is translated into editing actions, which are added to the sequence. The tags used when generating Praxis sequences from Java are shown in Table X. More details on the approach we propose to generate such sequences are available on our website[¶]. The data used in our experiment is available on our website[‖].

Figure 6 highlights the benefits of using real modifications to validate our approach. In particular, it shows that modifications targeting *calls* and *accesses* account for at least 50% of the whole

[‡] `http://code.google.com/p/harmony/`
[§] MOVIDA Project website: `http://movida.gforge.inria.fr`
[¶] `http://code.google.com/p/harmony/wiki/VPraxis`
[‖] `http://se.labri.fr/data/articles/ScopeReduction/`

Prepared using speauth.cls
DOI: 10.1002/spe

Project name	Total		Mean		
	#KLOC	#Elements	#Actions	#Extraction time	#Inconsistencies
Struts	65	10959	226.11	220	1103
Migen	137	26722	359.45	119	1852
CruiseControl	2368	20895	642.55	373	2453
JFreeChart	3561	18411	1837.79	193	3636

Table IX. Metrics for our selected Java projects. The first two columns show the number of line of code (#KLOC) and of elements (#Elements) of the revision `head` - 100 of the projects. The following three columns shows the mean number of actions (#Actions), the mean extraction time (#Extraction time, in milliseconds) and the mean number of inconsistencies (#Inconsistency), as computed on the 100 last revisions of each project.

Tag category	*Tags*
Element	class, interface, enum method, field
Property	abstract, static, final name, constructor, annotation visibility
Reference	extends, fields, type methods, parameters, accesses implements, catches, throws calls, inner

Table X. Tags used in our Praxis representation of Java code.

modifications, which is very important as these two classes of action have a major impact on inconsistency rules.

Rules We have used Java *FindBugs* as inconsistency rules.** FindBugs is a well known tool that aims to find bugs in Java source code. In its current version (2.0), FindBugs defines 547 rules. We have decided to consider the 207 rules of the correctness category. However, we have been able to implement only the rules that target elements supported by our Java representation. It was the case of 33 rules among the 207 (around 15%). The reason is because our Praxis representation for Java source code, shown in Table X, includes only high-level Java elements. Fine-grained information such as loops or local variable declarations in methods are not included.

The implementation of the rules has been done by a student with a basic Prolog experience and no knowledge of our scope or rule reduction approach. He only knew the description of the Praxis basic predicates and the mandatory structure of a Praxis rule. He used only the FindBugs textual documentation and Java code to produce the Prolog code. The code has been tested several times and a stable version has been established. Producing and testing this code took about one week to our student. Our static analysis has then been used on his code to automatically generate the corresponding pre-instantiated impact list. The generation of this list (Prolog code parsing, call dependency computation and XML generation) took 237 ms.

5.3. Scope reduction on FindBugs rules

Table XI shows an in-depth analysis of the pre-instantiated impact list automatically obtained from the FindBugs inconsistency rules. A row in this table represents a rule, and a column represents a class of editing actions. A square indicates how well the rule is scoped for the corresponding class. A ● indicates that all modified elements of the class are bound to variables of the rule or, when

**`http://findbugs.sourceforge.net/bugDescriptions.html`

Prepared using speauth.cls
DOI: 10.1002/spe

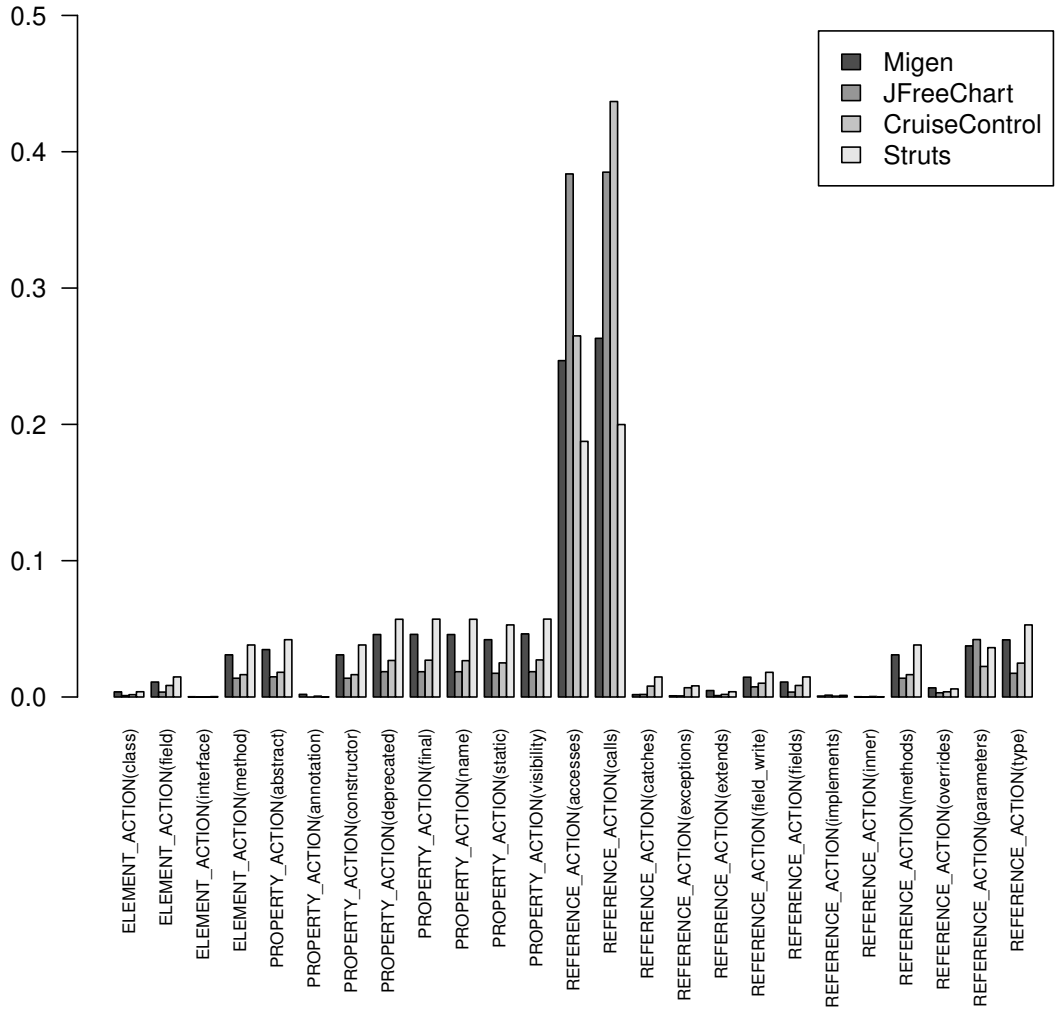


Figure 6. Distribution of the action classes in the whole Praxis sequences of our Java projects

the rule defines only one variable, that this variables is bound. A ◐ indicates that some modified elements of the class are bound to variables of the rule. A ○ indicates that the rule is not scoped at all for this class of action. Finally, an empty square indicates that the action class has no effect on the rule.

Therefore in this table, a row full of ● indicates that the rule has its scope reduced maximally for each action class. On the contrary, a row full of ○ indicates that our scope reduction technique has no effect at all. Further, a column full of ● indicates that the action class has an impact on every rule and that the rules are always scoped. A column full of ○ indicates that the action class has an impact on every rule too but that the rules are never scoped. Finally, a column full of empty squares indicates that the action class has no impact on the rule.

First of all, our scope reduction technique has an effect for each of our 33 rules, which is a very good result. Only 6 rules (around 20%) contain at least an action class that has no scope reduction. On 96 impacts, only 7 (around 7%) are not scoped at all. Finally, 16 (around 50%) rules have a maximum scope reduction. This means that our scope reduction works with a wide range of rules, and having no scope reduction is rare. Moreover, there is a fair chance to end up with maximum scope reductions.

Prepared using speauth.cls
DOI: 10.1002/spe

There are two reasons why the scope of 6 rules has not been reduced. The first one is the use of the negation. A typical example is the FindBug `he` (C, M_1, M_2), stating that if a class (C) defines a `boolean equals(Object)` method (M_1), it must defines a `int hashCode()` method (M_2) as well. Since an inconsistency for this rule is a class having the first method and not the second one, the second method cannot appear in the rule's arguments. Therefore, any use of M_2 made in the rule cannot be scoped. The second reason is the use of recursion. The `aom` rule provides an example of such a case. Recursion in Prolog leads to the introduction of new local variables (one for each recursion step for the `aom` rule) that cannot be scoped.

The columns of Table XI indicates that four action classes (*extends*, *methods*, *name* and *calls*) have a major impact on rules. Among these four action classes, only the (*extends*) one is poorly scoped. Indeed, each time an editing action of this class is performed, 5 rules have to be rechecked without any scope reduction. Fortunately, by considering the distribution of editing actions in our corpus (see Figure 6), it appears that editing actions of this class are rare.

5.4. Experimental protocol

In our experiment, we measured the time and the memory needed to perform the rechecks on the 100 revisions of each project of our corpus. Before describing the results, we provide an explanation on how the measures have been done. For each Java project, our prototype behaves as follows. First it loads in memory the pre-instantiated impact list and the initial reversion of the project (`Head`-100). The whole set of rules is then executed on this initial version, and the diagnostic is stored. Then the Praxis sequences corresponding to each of the next 100 revisions are successively loaded from the disk into the RAM. For each construction sequence, the pre-instantiated impact list is used to compute the rules to recheck. The rules are then rechecked, and the diagnostic is updated. Between each revision, the garbage collector is explicitly called. At each revision, we measure the time and the memory. This whole protocol is repeated 10 times to mitigate the differences we could obtain due the lack of precision of our measurement method. All the measures have been done on an Intel Core 2 Quad CPU Q9505 at 2.83GHz computer with 8 gb of RAM. In the remainder of this section, we explain precisely how we measure the time and the memory.

Time measurement Regarding time, we measure the time needed to 1) compute the rules to recheck, 2) recheck the rules and 3) update the diagnostic. To measure it, we have used the stopwatch technique (with the `System.currentTimeMillis()` Java method). We chose not to include the time needed to convert the Java code to Praxis construction sequences because this operation depends on the CASE tool. We can extract this sequence by listening to the IDE, as we did in [14]. This way, the conversion of the model can be done prior to apply the recheck, avoiding a time overhead. Table IX shows the mean time required for our prototype to gather a revision from a repository and to compute a diff in the VPraxis format with the previous revision (see column #Extraction time).

Memory measurement Regarding memory, we measure the memory consumed by 1) the "diff" construction sequence, 2) the diagnostic and 3) the pre-instantiated impact list. We have measured the consumed memory with the `Runtime.totalMemory()` and `Runtime.freeMemory()` Java methods. We chose not to include the memory needed to store the model as it is consumed anyway by any approach. For the sake of simplicity, our prototype currently stores the model twice (as a graph and as a sequence of editing action). However, depending on the CASE tool that will integrate our approach, we can provide a better implementation that will not store the complete sequence but only the last *diff*, which is required by our approach. The memory consumed by the rules is not measured too since any approach consumes it and further it is freed after each recheck.

5.5. Results

Time efficiency Table XII shows the mean recheck times of our incremental checker on the Java projects. The mean recheck time for a commit ranges from 1.4 to 22.45 seconds (20 times). This is

Prepared using speauth.cls
DOI: 10.1002/spe

	REFERENCE_ACTION(accesses)	PROPERTY_ACTION(visibility)	REFERENCE_ACTION(type)	PROPERTY_ACTION(anonymous)	PROPERTY_ACTION(abstract)	PROPERTY_ACTION(constructor)	REFERENCE_ACTION(field_write)	PROPERTY_ACTION(static)	REFERENCE_ACTION(extends)	PROPERTY_ACTION(final)	PROPERTY_ACTION(annotation)	REFERENCE_ACTION(methods)	REFERENCE_ACTION(fields)	PROPERTY_ACTION(name)	ELEMENT_ACTION(method)	REFERENCE_ACTION(calls)	REFERENCE_ACTION(parameters)	REFERENCE_ACTION(implements)
pr_aom					●				○			●		●				
pr_boa									○		●	●		●				◐
pr_cao						●						●					◐	
pr_co	●													●				
pr_dmi														●	●	●		
pr_eq											●			●	●		●	
pr_he			○									◐		○				
pr_iicu												●				◐		
pr_ijuBad			◐						◐			●		●			◐	
pr_ijuEmpty									◐			●				◐		
pr_ijuRun									◐			●		●		◐		
pr_ijuSetup									◐			●		●		◐		
pr_ijuSuite			◐					●	◐			●		●				
pr_ijuTearDown									◐			●		●		◐		
pr_ipu															●	●		
pr_lo						●											●	
pr_lysc																●		
pr_mdm																●		
pr_mdmExit																●		
pr_mdmSock																●		
pr_mf														●	●			
pr_mfMask									○				●	●				
pr_nff										●	●		●					◐
pr_nm									○			●		●			◐	
pr_pcoa						●				●						●		
pr_pis									○				●					◐
pr_s508c									●									●
pr_sWindowSize																●		
pr_sePriv		●										●		●				◐
pr_seRead								●				●		●				◐
pr_thr																●		
pr_umac				●					◐		●	●						
pr_uwf							●						●					

Table XI. Scoping ratio of the different action classes for each rule. ○ indicates no scope reduction, ◐ indicates a partial scope reduction and ● a complete scope reduction. An empty square indicates that the action class has no effect on the rule. For the sake of readability, action classes that have no effect at all on the rules are omitted from the table.

correlated with the mean number of actions included in a revision (see Table IX). The mean recheck time for an action ranges from 0.006 to 0.02 seconds (three times), which seems fairly stable. This is correlated with the model size. These results show that our incremental checker is fast enough to be considered as instantaneous. However, as expected, our incremental checker is slower than the one that uses a cache, although the data and experimental protocol is different [3]. Further, we can see that there is a significant improvement, from 1.64 to 4.87 times faster, over our previous approach that supports only rule reduction [18].

DOI: 10.1002/spe

Project name	per commit	per action	Ratio with our previous checker
Struts	1.41	0.006	4.87
Migen	7.18	0.02	4
CruiseControl	11.65	0.018	4.71
JFreeChart	22.45	0.012	1.64

Table XII. Mean recheck times (in seconds) for a commit and an action recheck using our incremental checker. Ratio is the improvement compared to our previous checker that supports only rule reduction.

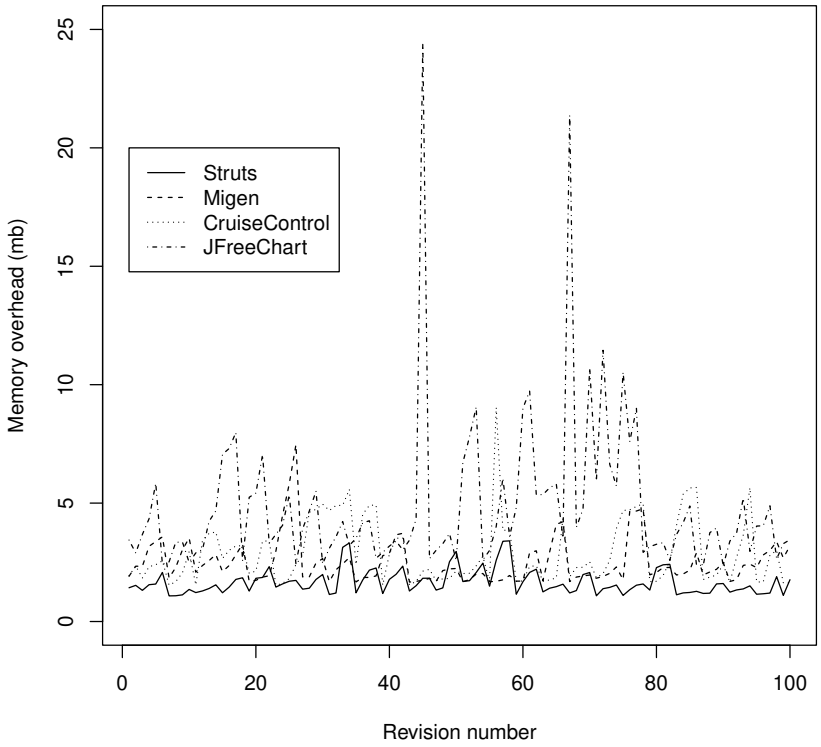


Figure 7. Memory overhead to detect the inconsistencies on the 100 revisions of the Java projects

Memory consumption Figure 7 shows that the memory oscillates around a fixed value. This value is quite the same for the four Java projects. As expected, it is very small (about 2.5 megabytes). This means that the memory overhead induced by our approach does not increase with the size of the model. The figure also highlights some peaks depending on the revision. These peaks correspond to large revisions where many modifications have been performed. As our measure includes the size of the *diff*, large revisions induce a temporary peak of memory consumption. For instance with JFreeChart, the higher memory peak (25 mb) corresponds to the biggest revision. This is not a scalability issue since these peaks do not depend on the models nor on the rules but only on the habits of the developers to commit big or small modifications.

Figure 8 shows the number of inconsistencies detected in the 100 revisions of the Java projects. As previously explained, these inconsistencies are stored in memory and are therefore included in the memory consumption shown in Figure 7. In this corpus, we can see that the variance of the number of inconsistencies is very low. It explains why the memory consumption of our approach oscillates around a fixed value. If the number of violations were growing significantly across the revisions, the

Prepared using speauth.cls
DOI: 10.1002/spe

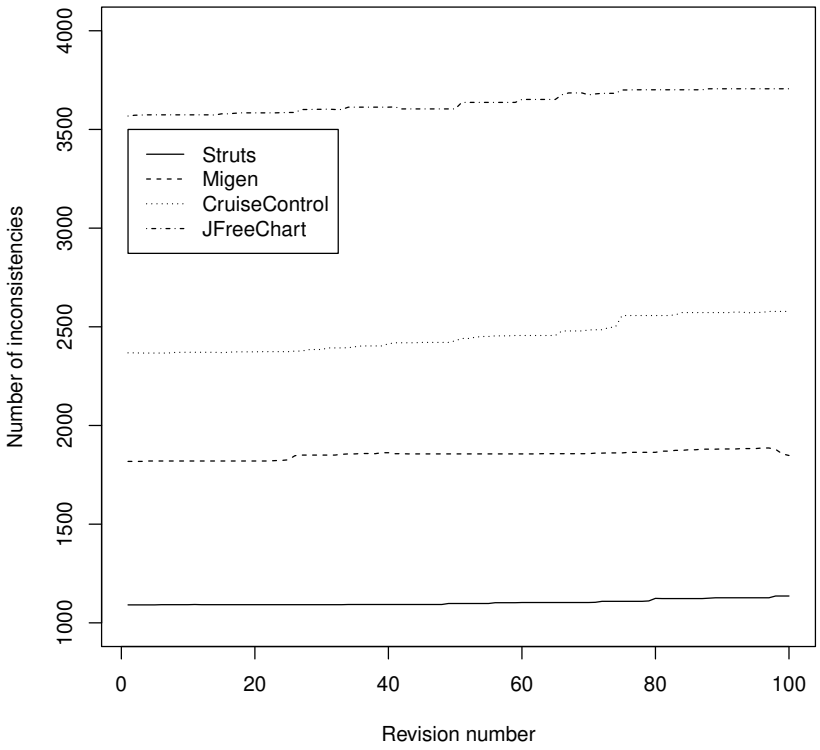


Figure 8. Number of inconsistencies on the 100 revisions of the Java projects

memory consumption would grow as well. However, this is not specific to our approach since any approach has to store the diagnostic somehow.

Finally, we have validated the fact that rule and scope reduction has a similar memory overhead. Note that the only difference on memory consumption between the two approaches is the size of the impact list versus pre-instantiated impact list. Since this size is independent of the size of the model, we have used measured the memory used by a program that only loads them. In both cases, the size was 816 kilo-bytes.

5.6. Threats to validity

Time performances can vary due to the distribution of the editing actions that are performed to the projects. Figure 6 indicates that this distribution is similar for our four Java projects. Nevertheless, our corpus is not big enough to be generalized for any Java project. Similarly, the efficiency of our checker depends on the rules. We have done our best to chose as many independently developed rules as possible, by using the open-source FindBugs project. However, this corpus cannot guarantee that our static analysis will be as efficient on another set of rules.

We measured the time performance using `System.currentTimeMillis()`, which cannot determinate the exact processor time. Similarly, the memory is measured using the `System.gc()`, `Runtime.totalMemory()` and `Runtime.freeMemory()` Java methods, which do not guarantee neither garbage collection nor precise measures. To mitigate the lack of precision of these methods, we have measured an average over ten runs. However, this is not a major threat since we do not compare our approach with any another one.

DOI: 10.1002/spe

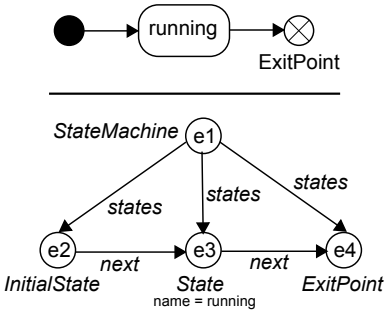


Figure 9. State machine model (top) with its corresponding digraph (bottom).

```
1  create(e1,'StateMachine').
2  create(e2,'InitialState').
3  addReference(e1,'states',e2).
4  create(e3,'state').
5  addProperty(e3,'name','Running').
6  addReference(e1,'states',e3).
7  addReference(e2,'next',e3).
8  create(e4,'ExitPoint').
9  addReference(e1,'states',e4).
10 addReference(e3,'next',e4).
```

Figure 10. Praxis construction sequences of the the state machine model

Inconsistency rules written with Prolog code may have been written to favor the efficiency of our static analysis. In particular, no rules have used variable aliasing that induces an over-estimation of the call dependencies identified by our analysis. To lower the importance of this threat, we used a student that has only a basic knowledge of Prolog, and no knowledge at all of our static analysis. Therefore, he only used on-line documentation and books to learn how to write Prolog programs. The code he produced is therefore likely to be similar to any Prolog program, and is not biased to favour our analysis.

6. BEYOND JAVA: TWO PRAXIS CASE STUDIES

This section presents how Praxis can be used to check any kind of models. It first presents examples of UML models and Praxis inconsistency rules. Second, it presents a case study that has been performed by Thales, our industrial partner as a part of the MOVIDA project. Thales wanted to try Praxis on their domain specific models, based on EMF (Eclipse Modeling Framework) [19].

6.1. Praxis for UML

As explained in Section 3, Praxis can be used to represent any model, whatever its meta-model. As an example, Figure 9 presents a UML model and its corresponding digraph. Figure 10 presents the Praxis actions performed to create this model. Figure 11 shows the `twoInitialStates` Praxis inconsistency rule that has been extracted from the UML 2.1 specification [30]. This rule specifies that a state machine `SM` can have at most one initial state.

We have implemented 48 such inconsistency rules coming from the standard OCL constraints contained in the class diagram package of the UML 2.1 specifications [30]. The rules have been implemented by a Prolog expert. We have generated the pre-instantiated impact list for these rules, and we have performed a time and memory measurement of our incremental checker with a corpus of UML models that have been automatically generated by a mathematically grounded random model sampler [31]. Similarly to the Java experiment, we measured recheck time and memory consumption on this corpus.

DOI: 10.1002/spe

```
1 twoInitialStates(SM,S1,S2) :-
2   existReference(SM,'states',S1),
3   existReference(SM,'states',S2),
4   existElement(S1,'InitialState'),
5   existElement(S2,'InitialState'),
6   S1 \= S2.
```

Figure 11. The `twoInitialStates` rule for Praxis

6.2. MOVIDA case study

In the context of the MOVIDA project, Thales has developed its own modelling language to develop complex systems. This modelling language is largely inspired from UML. To check the consistency of their models, Thales has participated to a case study to try our incremental checker.

Thirty representative inconsistency rules have been chosen by Thales engineers and implemented using Praxis by the same engineers. Then, the pre-instantiated impact list has been generated. Finally, the Thales engineers used our incremental checker together with their model editor during two months. Each time they modified and saved their model, Praxis was performing an incremental check.

Regarding the expressiveness of Praxis, Thales reported that some inconsistency rules are easier to write and understand in Praxis than in Java, while other rules are more complicated to write, requiring the intervention of experienced Prolog developers. After analysing these rules, we have concluded that rules easy to write are the ones that look for local patterns of inconsistency in closely connected elements. Such rules can be written in Praxis as simple conjunctions of connected elements. Rules that look for global patterns of inconsistency require a Prolog specialist as the Prolog recursion has to be used to implement a strategy for traversing the whole graph of elements.

Regarding efficiency, Thales has reported that our checker has been considered instantaneously by the users. The average response time was less than 100 ms. Regarding scalability, Thales has reported that the experiment has been realized with a set of models ranging from 1000 to 50000 model elements. No scalability issues were identified.

7. CONCLUSION AND FUTURE WORK

Ensuring consistency between large models is one of the challenges for the software industry. As detecting inconsistencies in a model is a very tedious and time consuming task, an appropriate tooling support, also called checker, is mandatory. To be optimal, a checker must be sound, efficient and scalable.

Most of existing checkers puts emphasis on efficiency by using a cache memory. As a cache irremediably introduce memory overhead, their main drawback is therefore scalability. In contradiction, we propose in this paper a checker that puts emphasis on scalability, which does not need any cache memory. Our checker reduces both the number of rules that need to be rechecked (rule reduction) and the models elements to be considered during a recheck (scope reduction) thanks to what we call a pre-instantiated impact list.

A pre-instantiated impact list is automatically generated by a static analysis of a set of inconsistency rules, which guarantees the soundness of the checker. It defines which inconsistency rules have to be rechecked when changes occur, and how variables of rules can be pre-instantiated using elements modified by changes.

We validated our checker against real Java projects with real modifications. The goal of the validation was to check some of the FindBugs rules, which are well-known inconsistency rules for Java. Our validation has shown that our checker is scalable as it only needs few megabytes of memory overhead to support rules and scope reductions, regardless of the models size. It is also quite efficient (few milliseconds in average) even if it does not perform any reduction for some rules and some modifications.

DOI: 10.1002/spe

Thales, our industrial partner, has experimented our checker. It has reported that it can handle large models and is efficient enough. Further, it has reported that rules searching for local patterns are easy to implement in Prolog, which is the language used to express inconsistency rules with our checker. However, it has reported that complex rules have to be written by experienced Prolog programmers.

Even if our checker is sound, scalable and quite efficient for several domain specific models and rules, the experiments we have realized have shown that our checker has limitations (rules that use negation and ones that use recursion). In the near future, we plan to work on a hybrid checker that uses a cache only for such rules, which are not scoped by our static analysis.

REFERENCES

1. Hessellund A, Czarnecki K, Wasowski A. Guided development with multiple domain-specific languages. *Model Driven Engineering Languages and Systems, 10th International Conference, MoDELS 2007, Nashville, USA, September 30 - October 5, 2007, Proceedings*, vol. 4735, Springer, 2007; 46–60, doi:10.1109/ICSE.2007.38.
2. Selic B. The pragmatics of model-driven development. *IEEE Software* 2003; **20**(5):19–25, doi:10.1109/MS.2003.1231146.
3. Egyed A. Automatically detecting and tracking inconsistencies in software design models. *IEEE Trans. Software Eng.* 2011; **37**(2):188–204, doi:10.1109/TSE.2010.38.
4. Spanoudakis G, Zisman A. Inconsistency management in software engineering: Survey and open research issues. *Handbook of Software Engineering and Knowledge Engineering*. World scientific, 2001; 329–380.
5. Egyed A. Fixing inconsistencies in UML design models. *29th International Conference on Software Engineering (ICSE 2007), Minneapolis, MN, USA, May 20-26, 2007*, IEEE Computer Society, 2007; 292–301.
6. Finkelstein S, Jacobs D, Brendle R. Principles for inconsistency. *CIDR 2009, Fourth Biennial Conference on Innovative Data Systems Research, Asilomar, CA, USA, January 4-7, 2009, Online Proceedings*, www.cidrdb.org, 2009.
7. Hill MD. What is scalability? *SIGARCH Comput. Archit. News* Dec 1990; **18**(4):18–21, doi:10.1145/121973.121975.
8. Boehm BW, In H. Identifying quality-requirement conflicts. *IEEE Software* 1996; **13**(2):25–35, doi:10.1109/52.506460.
9. Finkelstein A, Gabbay DM, Hunter A, Kramer J, Nuseibeh B. Inconsistency handling in multperspective specifications. *IEEE Trans. Software Eng.* 1994; **20**(8):569–578.
10. Grundy JC, Hosking JG, Mugridge WB. Inconsistency management for multiple-view software development environments. *IEEE Trans. Software Eng.* 1998; **24**(11):960–981, doi:10.1109/32.730545.
11. Nentwich C, Capra L, Emmerich W, Finkelstein A. xlinkit: a consistency checking and smart link generation service. *ACM Trans. Internet Techn.* 2002; **2**(2):151–185, doi:10.1145/514183.514186.
12. Straeten RVD, Mens T, Simmonds J, Jonckers V. Using description logic to maintain consistency between UML models. *UML 2003 - The Unified Modeling Language, Modeling Languages and Applications, 6th International Conference, San Francisco, CA, USA, October 20-24, 2003, Proceedings*, *Lecture Notes in Computer Science*, vol. 2863, Stevens P, Whittle J, Booch G (eds.), Springer, 2003; 326–340, doi:10.1007/978-3-540-45221-8_28.
13. Paige RF, Brooke PJ, Ostroff JS. Metamodel-based model conformance and multiview consistency checking. *ACM Trans. Softw. Eng. Methodol.* 2007; **16**(3), doi:10.1145/1243987.1243989.
14. Blanc X, Mounier I, Mougenot A, Mens T. Detecting model inconsistency through operation-based model construction. *30th International Conference on Software Engineering (ICSE 2008), Leipzig, Germany, May 10-18, 2008*, Schäfer W, Dwyer MB, Gruhn V (eds.), ACM, 2008; 511–520, doi:10.1145/1368088.1368158.
15. Egyed A. Instant consistency checking for the UML. *28th International Conference on Software Engineering (ICSE 2006), Shanghai, China, May 20-28, 2006*, Osterweil LJ, Rombach HD, Soffa ML (eds.), ACM, 2006; 381–390, doi:10.1145/1134339.
16. Cabot J, Teniente E. Incremental evaluation of OCL constraints. *Advanced Information Systems Engineering, 18th International Conference, CAiSE 2006, Luxembourg, Luxembourg, June 5-9, 2006, Proceedings*, *Lecture Notes in Computer Science*, vol. 4001, Dubois E, Pohl K (eds.), Springer, 2006; 81–95, doi:10.1007/11767138_7.
17. Xu C, Cheung SC, Chan WK. Incremental consistency checking for pervasive context. *28th International Conference on Software Engineering (ICSE 2006), Shanghai, China, May 20-28, 2006*, Osterweil LJ, Rombach HD, Soffa ML (eds.), ACM, 2006; 292–301, doi:10.1145/1134327.
18. Blanc X, Mougenot A, Mounier I, Mens T. Incremental detection of model inconsistencies based on model operations. *Advanced Information Systems Engineering, 21st International Conference, CAiSE 2009, Amsterdam, The Netherlands, June 8-12, 2009. Proceedings*, *Lecture Notes in Computer Science*, vol. 5565, Eck Pv, Gordijn J, Wieringa R (eds.), Springer, 2009; 32–46, doi:10.1007/978-3-642-02144-2_8.
19. Noir JL, Delande O, Exertier D, Silva MAAd, Blanc X. Operation based model representation: Experiences on inconsistency detection. *Modelling Foundations and Applications - 7th European Conference, ECMFA 2011, Birmingham, UK, June 6 - 9, 2011 Proceedings*, *Lecture Notes in Computer Science*, vol. 6698, France RB, Küster JM, Bordbar B, Paige RF (eds.), Springer, 2011; 85–96, doi:10.1007/978-3-642-21470-7_7.
20. Elaasar M, Briand LC. An overview of UML consistency management. *Technical Report SCE-04-18*, Department of Systems and Computer Engineering, Carleton University Aug 2004.
21. Schobbens PY, Heymans P, Trigaux JC. Feature diagrams: A survey and a formal semantics. *14th IEEE International Conference on Requirements Engineering (RE 2006), 11-15 September 2006, Minneapolis/St.Paul, Minnesota, USA*, IEEE Computer Society, 2006; 136–145, doi:10.1109/RE.2006.23.

22. Jouault F, Tisi M. Towards incremental execution of ATL transformations. *Theory and Practice of Model Transformations, Third International Conference, ICMT 2010, Malaga, Spain, June 28-July 2, 2010. Proceedings*, *Lecture Notes in Computer Science*, vol. 6142, Tratt L, Gogolla M (eds.), Springer, 2010; 123–137, doi:10.1007/978-3-642-13688-7_9.
23. Rose LM, Kolovos DS, Drivalos N, Williams JR, Paige RF, Polack FAC, Fernandes KJ. Concordance: A framework for managing model integrity. *Modelling Foundations and Applications, 6th European Conference, ECMFA 2010, Paris, France, June 15-18, 2010. Proceedings*, *Lecture Notes in Computer Science*, vol. 6138, Kühne T, Selic B, Gervais MP, Terrier F (eds.), Springer, 2010; 245–260, doi:10.1007/978-3-642-13595-8_20.
24. Eichberg M, Kahl M, Saha D, Mezini M, Ostermann K. Automatic incrementalization of prolog based static analyses. *Practical Aspects of Declarative Languages, 9th International Symposium, PADL 2007, Nice, France, January 14-15, 2007*, *Lecture Notes in Computer Science*, vol. 4354, Hanus M (ed.), Springer, 2007; 109–123, doi:10.1007/978-3-540-69611-7_7.
25. Menet L, Lamolle M, Duc CL. Incremental validation of models in a MDE approach applied to the modeling of complex data structures. *On the Move to Meaningful Internet Systems: OTM 2010 Workshops - Confederated International Workshops and Posters: International Workshops: AVYTAT, ADI, DATAVIEW, EI2N, ISDE, MONET, OnToContent, ORM, P2P-CDVE, SeDeS, SWWS and OTMA. Hersonissos, Crete, Greece, October 25-29, 2010. Proceedings*, *Lecture Notes in Computer Science*, vol. 6428, Meersman R, Dillon TS, Herrero P (eds.), Springer, 2010; 120–129, doi:10.1007/978-3-642-16961-8_28.
26. Bergmann G, Horváth Á, Ráth I, Varró D, Balogh A, Balogh Z, ökrös A. Incremental evaluation of model queries over EMF models. *Model Driven Engineering Languages and Systems - 13th International Conference, MODELS 2010, Oslo, Norway, October 3-8, 2010, Proceedings, Part I*, *Lecture Notes in Computer Science*, vol. 6394, Petriu DC, Rouquette N, Haugen O (eds.), Springer, 2010; 76–90, doi:10.1007/978-3-642-16145-2_6.
27. Xu C, Cheung SC, Chan WK, Ye C. Partial constraint checking for context consistency in pervasive computing. *ACM Trans. Softw. Eng. Methodol.* 2010; **19**(3), doi:10.1145/1656250.1656253.
28. Wagner R, Giese H, Nickel U. A plug-in for flexible and incremental consistency management. *Proc. of the International Conference on the Unified Modeling Language 2003 (Workshop 7: Consistency Problems in UML-based Software Development), San Francisco, USA*, Technical Report, Blekinge Institute of Technology, San Francisco, 2003.
29. Sterling L, Shapiro EY. *The Art of Prolog - Advanced Programming Techniques, 2nd Ed*. MIT Press, 1994.
30. OMG. Unified modeling language specification : Superstructure (Version 2.4.1). *Technical Report*, OMG Aug 2011.
31. Mougenot A, Darrasse A, Blanc X, Soria M. Uniform random generation of huge metamodel instances. *Model Driven Architecture - Foundations and Applications, 5th European Conference, ECMDA-FA 2009, Enschede, The Netherlands, June 23-26, 2009. Proceedings*, *Lecture Notes in Computer Science*, vol. 5562, Paige RF, Hartman A, Rensink A (eds.), Springer, 2009; 130–145, doi:10.1007/978-3-642-02674-4_10.