ISSN 1346-5597

National Institute of Informatics

NII Technical Report

Contradiction Finding and Minimal Recovery for UML Class Diagrams using Logic Programming

Ken Satoh, Ken Kaneiwa, Takeaki Uno

NII-2006-009E
June 2006

URL: http://www.nii.ac.jp/ Tokyo 101-8430, JAPAN +81-3-4212-2000 FAX: +81-3-3556-1916

Contradiction Finding and Minimal Recovery for UML Class Diagrams using Logic Programming

Ken Satoh[1] Ken Kaneiwa[2] Takeaki Uno[1]
[1]National Institute of Informatics
2-1-2 Hitotsubashi, Chiyoda-ku, Tokyo 101-8430, Japan
[2]National Institute of Information and Communications Technology
4-2-1 Nukui-Kitamachi, Koganei, Tokyo 184-8795, Japan
ksatoh@nii.ac.jp, kaneiwa@nict.go.jp, uno@nii.ac.jp

Abstract

UML (Unified Modeling Language) is the de facto standard model representation language in software engineering. We believe that automated contradiction detection and repair of UML become very important as UML has been widely used. In this paper, we propose a debugging system using logic programming paradigm for UML class diagram with class attributes, multiplicity, generalization relation and disjoint relation.

We propose a translation method of a UML class diagram into a logic program, and using a meta-interpreter we can find (set-inclusion-based) minimal sets of rules which leads to contradiction. Then, we use a minimal hitting set algorithm developed by one of the authors to show minimal sets of deletion of rules in order to avoid contradiction.

1 Introduction

UML [12, 6] is a model representation language used in a design of software system and has been widely used in software industry and now considered as the de facto standard language for software modeling. To make software developed by UML more reliable, we believe that a formal approach for verification of the consistency of UML diagrams and repair of inconsistent UML diagrams become of much importance as UML is applied to more complex systems.

This paper is toward this direction of research. We consider automated contradiction detection and minimal recovery of UML class diagrams which consist of class attributes, multiplicity, generalization relation and disjoint relation. In order to reason about contradiction, we firstly need a rigorous semantics of UML class diagrams. We follow the works of Berardi et al. by [2, 3, 4] where they define a semantics of UML class diagrams in terms of description logic and first-order logic with counting quantifier. In this paper, we use a semantics of first-order logic formulation since it is easier for us to consider the translation from this semantics to a logic program. Berardi et al. showed that inconsistency in a UML class diagram corresponds to inconsistency in a description logic formula translated from the diagram, and used an existing description logic theorem prover to check consistency. However, in their framework, they can only detect inconsistency in a UML class diagram, and cannot explain which part is the cause of inconsistency whereas this paper considers not only contradiction detection but also minimal recovery of UML class diagrams.

In this paper, we translate a UML class diagram into a logic program in order to check inconsistency directly. It turns out by our previous results [9] that, for the restricted class of UML class diagram with class attributes, multiplicity, generalization relation and disjoint relation, only multiplicity mismatch and violation of disjointness lead to contradiction. Therefore, all we have to do is to check whether these cases happen and this gives an efficient checking of contradiction detection.

We believe that well-developed compilation techniques of logic programming help to get an efficient implementation. Moreover, by attaching an annotation (rule number) to rules in logic programming, we can use a meta-interpreter to detect which rules are blamed for inconsistency. Our translation actually has one-to-one correspondence between each rule and a part of the diagram, so we can directly point out the blamed part in the UML diagram itself.

Then, using inconsistent rule sets, we can show a possible candidate of deletion of rules to avoid contradiction. We have already proposed various methods of formalizing minimal update for software requirements represented as a logic program [13, 14, 15, 16]. This minimal update corresponds to computing maximal consistent set of rules from augmented requirements with a new requirement.

The above methods mainly concern computing maximal consistent set of a logic program directly without computing a cause of contradiction. In this paper, however, we have information about contradiction set of rules by contradiction detection and we use this information to get a set-inclusion-based minimal deletion of rules in the current logic program as follows. In order to avoid contradiction, it is sufficient to delete one rule from each inconsistent set of rules. However, this deletion might not be minimal since we could delete the same rule from multiple inconsistent sets. To avoid this redundant deletion, we compute a minimal hitting set for each inconsistent sets (a minimal set which has a common element with each inconsistent set). We use a new algorithm of minimal hitting set computation proposed by one of the authors [17] which is very simple and efficient in an average case.

Then, thanks to one-to-one correspondence between a rule and a part of a UML class diagram, we can point out which part of a UML class diagram should be deleted on the diagram directly.

The structure of the paper is as follows. We firstly give a semantics of UML class diagram in terms of first-order logic with counting quantifier according to [9, 3]. Using this semantics, we provide a sound and complete translation method from a UML class diagram to a logic program in order to check consistency. Then, we provide a method to compute minimally updated UML class diagram using minimal hitting set computation. We also show an example how such contradiction finding and minimal recovery are represented in our implemented system.

2 Semantics of UML Class Diagrams by First-Order Logic with Counting Quantifiers

We give a semantics of UML class diagrams in terms of first-order logic with counting quantifiers according to [9, 3].

- Let c and t be a class, a an attribute and $[i..j]$ a multiplicity. Class c with an attribute $a[i..j]\colon t$ has the following semantics:

  $$\forall x \forall y(c(x) \rightarrow (a(x,y) \rightarrow t(y)))$$
  $$\forall x(c(x) \rightarrow \exists_{\geq i} z(a(x,z)))$$
  $$\forall x(c(x) \rightarrow \exists_{\leq j} z(a(x,z)))$$

  where $\exists_{\geq i} z$ and $\exists_{\leq j} z$ are counting quantified variables and $\exists_{\geq i} z$($\exists_{\leq j} z$ respectively) means that there exists some elements z whose cardinality is more than or equal to i (less than or equal to j respectively).

- A class c generalizing $c_1, ..., c_n$ is captured by the following formula.

  $$\forall x(c_1(x) \rightarrow c(x)), \ldots, \forall x(c_n(x) \rightarrow c(x))$$

- Meaning of disjointness between classes $c_1, \ldots, c_n$ is shown by the following formula.

  $$\forall x(c_i(x) \rightarrow \neg c_{i+1}(x) \wedge \cdots \wedge \neg c_n(x))$$
  for $i \in \{1, \ldots, n-1\}$

The inconsistency of a UML class diagram is defined as inconsistency of the above translated formulas plus an existential formula $\exists x(c(x))$ for every class c.

3 Translating Counting Quantified Formula into Logic Program

We translate a class c with an attribute $a[i..j]\colon t$ into the following rules in logic programming.

```
t(f_a(X)) :- c(X).
at_least(f_a(X),i) :- c(X).
```
(where $i > 0$)
```
contradiction :-
    c(X), at_least(f_a(X),I), j<I.
```
(where $j \neq *$)

where `f_a` is a Skolem function for an attribute a which expresses a function which maps an element X of class c to an element of the attribute a and a predicate `at_least` expresses the minimum number of elements of a with respect to X and `contradiction` expresses contradiction. Intuitive meaning of the above rules are as follows:

- The first rule means that if X is an instance of class c, then the attribute a of X is in the class t.
- The second rule means that if X is an instance of class c, then the minimum number of instances of attribute a of X is at least i.
- The third rule means that if X is an instance of class c and the minimum number of instances of attribute a of X derived from some rules must be less than or equal to j. (Otherwise contradiction occurs).

If c is a generalization of classes $c_1, ..., c_n$, we give the following translation.

```
c(X) :- c1(X).
      ⋮
c(X) :- cn(X).
```

The disjointness between classes $c_1, \ldots, c_n$ are translated as follows.

For every $i = 1, \ldots, n-1$,

```
contradiction :- ci(X),ci+1(X).
      ⋮
contradiction :- ci(X),cn(X).
```

We add the following fact for every class c.

```
c(e_c).
```

where $\mathtt{e_c}$ is a new symbol expressing for an element of each class.

We check contradiction of a UML class diagram by checking whether `contradiction` is derived or not in the translated program.

Theorem 1 *A set of first-order formula with counting quantifier which gives a semantics of a UML class diagram is inconsistent if and only if* `contradiction` *is derived in the above translated logic program.*

Proof: See the Appendix.

This result is very important since this guarantees not only correctness of our method (if `contradiction` is derived from a logic program then the UML class diagram has inconsistency), but also completeness of our method (if the diagram has inconsistency, we can always detect it). This shows the power of our method.

4 Consistency Check in Logic Programming

This section provides a method of computing a inconsistent set of rules using meta-interpreter. For the purpose, we attach a rule number `N` for each fact or rule such as `N@G` (in the case of a fact) and `N@(G:-B)` (in the case of a rule). Then, by using the following meta-interpreter, we can calculate all the inconsistent sets by calling `solve(contradiction,U,[])`.

```
solve(G,Used,NewUsed):-
    \+G=(_,_),
    !,
    solve1(G,Used,NewUsed).
solve((G1,G2),Used,NewUsed):-
    !,
    solve1(G1,Used,Used1),
    solve(G2,Used1,NewUsed).

solve1(G,[N@G|Used],Used):-
    N@G.
solve1(G,[N@(G:-B)|Used],NewUsed):-
    N@(G:-B),
    solve(B,Used,NewUsed).
```

Intuitive meaning of the above interpreter is as follows:

- The first rule means that to solve a singleton goal `G`, we call `solve1`.
- The second rule means that to solve a compound goal `(G1,G2)`, we call `solve1` for `G1` and we solve rest (`G2`) by calling `solve` recursively.
- The third rule means that to solve a singleton goal `G`, we match an annotated fact of the form `N@G` and the fact is recorded in the list of the second argument of `solve1`.
- The forth rule means that to solve a singleton goal `G`, we match an annotated rule of the form `N@(G:-B)` and solve `B` and the rule is recorded.

Then, we can calculate minimal inconsistent sets of rules by checking minimality of these inconsistent sets.

5 Minimal Contradiction Recovery

After detecting inconsistency, we would like to derive a maximal consistent set which keeps the previous UML class diagram as much as possible. Note that there might be multiple sources of contradiction. Therefore, we need to delete one rule from these sources. But there might be the same rules in multiple contradiction sources so we should pick up such a rule in order to avoid contradiction and simultaneously keep the previous diagram as much as possible[1].

In order to pick such overlapped rules, we use a *minimal hitting set*[2] computation defined as follows:

Definition 1 *Let Π be a finite set and $\mathcal{H}$ be a subset family of Π. A* **hitting set** *HS of $\mathcal{H}$ is a set s.t. for every $S \in \mathcal{H}$, $S \cap HS \neq \emptyset$. A* **minimal hitting set** *HS of $\mathcal{H}$ is a hitting set s.t. there exists no other hitting set HS' of $\mathcal{H}$ s.t. $HS' \subset HS$ (HS' is a proper subset of HS).*

We use an algorithm to compute minimal hitting sets shown in Fig. 1. This algorithm was proposed by [17]. Let $\mathcal{H}$ be $\{S_0, ..., S_n\}$ where i of S_i means the order of the input set. The algorithm in 1 incrementally computes minimal hitting sets of each $\{S_0, ...S_i\}(1 \leq i \leq n)$ without redundant enumeration. The behavior of the algorithm is as follows:

1. We start from choosing one element from S_0.
2. If we get a minimal hitting set mhs up to $S_0, ..., S_{i-1}$ then, we consider a new minimal hitting set up to $S_0, ..., S_{i-1}, S_i$ as follows:
   - If mhs and S_i have any common element, mhs itself is a hitting set (and also minimal) for $S_0, ..., S_{i-1}, S_i$ as well. Therefore, we continue this process for further sets.
   - Otherwise, we have to add one element e from S_i to mhs. We, however, must check whether the addition still preserves minimality. We continue this process for further sets only if the resulting set $mhs \cup \{e\}$ is a minimal hitting set for

[1] Note that minimal recovery is based on the set-inclusion of contradiction sources.

[2] Note that the minimality is based on the subset ordering.

$S_0, ..., S_{i-1}, S_i$. Then, we continue this process for further sets.

This algorithm requires exponential time with respect to the number of sets (n in the algorithm) in the worst case, but it is empirically shown to be efficient (See [17] for the detail).

6 Execution Example

We show an inconsistent UML class diagram in Fig. 2. This actually has two sources of contradiction.

- The class $c4$ has the superclasses $c5$ and $c6$ which are disjoint. Since $c4$ has an instance, it leads to contradiction.
- The class $c1$ has an attribute $a1$ whose class is $c2$ which has an attribute $a2$ whose instances' maximum number is 5. On the other hand, the class $c1$ has a superclass $c3$ which also has the same name attribute $a1$ as $c1$ whose class is $c4$ which has a superclass $c5$ which has the same name attribute $a2$ as $c2$ whose instances' minimum number is 7. Therefore, $c1.a1.a2$ has contradictory multiplicity information; the number of instances are at most 5 and at least 7. This causes contradiction.

We translate this diagram into the following logic program to detect such contradiction [3]:

```
c4(a1(X)):-c3(X).                          (1)
at_least(a1(X),1):-c3(X).                  (2)
c2(a1(X)):-c1(X).                          (3)
at_least(a1(X),1):-c1(X).                  (4)
d(a2(X)):-c2(X).                           (5)
at_least(a2(X),1):-c2(X).                  (6)
contradiction:-
    c2(X), at_least(a2(X),I), 5<I.         (7)
e(a2(X)):-c5(X).                           (8)
at_least(a2(X),7):-c5(X).                  (9)
contradiction:-
    c2(X), at_least(a2(X),I), 10<I.       (10)
c3(X):-c1(X).                             (11)
c3(X):-c2(X).                             (12)
c5(X):-c4(X).                             (13)
c7(X):-c5(X).                             (14)
c7(X):-c6(X).                             (15)
c6(X):-c4(X).                             (16)
contradiction:-c5(X), c6(X).              (17)
c1(ec1).                                  (18)
c2(ec2).                                  (19)
c3(ec3).                                  (20)
```

[3]Note that in an actual setting, we attach a unique rule number for each rule for the input to the meta-interpreter, but we omit the notation for simplicity.

```
c4(ec4).                                  (21)
c5(ec5).                                  (22)
c6(ec6).                                  (23)
c7(ec7).                                  (24)
d(ed).                                    (25)
e(ee).                                    (26)
```

We obtain the following two minimal inconsistent sets by running the meta-interpreter. Note that contradiction arises when we add the facts that each class has an element (such as `c1(ec1)`) into the translated program.

Minimal Inconsistent Set 1:

```
c5(X):-c4(X).                             (13)
c6(X):-c4(X).                             (16)
contradiction:-c5(X), c6(X).              (17)
```

Minimal Inconsistent Set 2:

```
c4(a1(X)):-c3(X).                          (1)
c2(a1(X)):-c1(X).                          (3)
contradiction:-
    c2(X), at_least(a2(X),I), 5<I.(7)
at_least(a2(X),7):-c5(X).                  (9)
c3(X):-c1(X).                             (11)
c5(X):-c4(X).                             (13)
```

Then, we calculate minimal hitting sets for the above minimal inconsistent sets and obtain the following candidate sets of deletion of rules: $\{(13)\}$, $\{(16),(1)\}$, $\{(16),(3)\}$, $\{(16),(7)\}$, $\{(16),(9)\}$, $\{(16),(11)\}$, $\{(17),(1)\}$, $\{(17),(3)\}$, $\{(17),(7)\}$, $\{(17),(9)\}$, and $\{(17),(11)\}$.

7 UML Debugging System

We made a prototype system which directly reflects the results from consistency check and minimal update of UML class diagrams. We make a UML class diagram using UML class editor and the system translates a UML class diagram into a logic program to check consistency. If there is inconsistency, the system shows a minimal source of contradiction one by one. The system also shows a set of minimally deleted rules to avoid contradiction one by one. For the check of the example in the last section, there are two sources of inconsistency and the system displays them one by one (Fig. 3 and 4) where the parts corresponding to each inconsistency source are shown by changing its color or underlining the corresponding part (a1). In Fig. 5, we show one example of a set of deleted rules (in this example, only the line from class `c4` to `c5`) changes its color. One to one correspondence between a part of a UML class diagram and a rule of a translated logic program enables us to perform such display.

```
global S_0, ..., S_{n-1};
compute_mhs(i, mhs) /* mhs ∈ MHS({S_0, ..., S_{i-1}}) */
begin
  if i == n then output mhs and return;
  elseif S_i ∩ mhs ≠ ∅ then compute_mhs(i + 1, mhs);
  else for every e ∈ S_i s.t. mhs ∪ {e} is a minimal hitting set of S_0, ..., S_i do
    compute_mhs(i + 1, mhs ∪ {e});
  return;
end
```

Figure 1. Algorithm to Compute Minimal Hitting Sets

8 Related Work

In software engineering, there are several proposals of logical treatment of "inconsistency" of software specification such as [5, 19]. A survey of inconsistency handling is found in [7]. Finkelstein et al.[5] use non-collapsible "quasi-classical logic" even in the existence of inconsistency and formalizes consistency management between multiple specifications defined by several users. Zowghi et al. [19] propose an application of default reasoning, belief revision and epistemic entrenchment to model requirements evolution. However, these works more focus on logical aspects of inconsistency handling rather than computation.

More computation-approaches for consistency management are found in [8, 18]. Nuseibeh and Russo [8] use abductive logic programming to implement "quasi-classical logic". In their work, they detect rules which should be deleted to restore consistency using abduction. Zisman and Kozlenkov [18] represent the UML specifications in terms of knowledge base and then compile this knowledge base into a logic program which can be used for verification.

The technique of finding a minimal deletion of rules using minimal hitting set computation was already proposed in [11] in consistency-based diagnosis and in [10, 1] in logic programming. So, this paper can be seen as an application of their techniques as well.

9 Conclusion

In this paper, we propose a translation method of a UML class diagram with attributes, multiplicity, generalization and disjoint relation into a logic program in order to check consistency and propose a minimal deletion of a part of the diagram. As for the future work, we should extend our work not only on contradiction detection and minimal recovery for a wider class of UML class diagrams but also on contradiction detection between UML class diagrams and other UML diagrams.

Acknowledgements

This work was supported by the project "Research Priority Area on Informatics Studies for the Foundation of IT Evolution" by MEXT.

References

[1] Aravindan, C. and Baumgartner, P., A Rational and Efficient Algorithm for View Deletion in Databases, *Proc. of ILPS-97*, pp. 165 – 179 (1997).

[2] Berardi, D., Calvanese, D., De Giacomo, G., Reasoning on UML class diagrams is exptime-hard, *Proc. of the 2003 International Workshop on Description Logics (DL2003)* (2003).

[3] Berardi, D., Cali, A., Calvanese, D., De Giacomo, G., Reasoning on UML class diagrams, *Artificial Intelligence*, **168** (1-2), pp. 70 – 118 (2005).

[4] Cali, A., Calvanese, D., De Giacomo, G., Lenzerini, M., A formal framework for reasoning on UML class diagrams, *LNCS 2366*, pp. 503 – 513 (2002).

[5] Finkelstein, A. C. W., Gabbay, D., Hunter, A., Kramer, J., Nuseibeh, B., Inconsistency Handling in Multiperspective Specifications, *IEEE Transactions on Software Engineering*, 20, pp. 569 – 578 (1994).

[6] Fowler, M., *UML Distilled: A Brief Guide to the Standard Modeling Object Language*, Object Technology Series. Addison-Wesley, third edition (2003).

[7] Nuseibeh, B., To Be and Not to Be: On Managing Inconsistency in Software Development, *Proc. of 8th IEEE International Workshop on Software Specification and Design (IWSSD-8)*, pp. 164 – 169 (1996).

[8] Nuseibeh, B., Russo, A., Using Abduction to Evolve Inconsistent Requirements Specifications, *Australian Information Systems Journal, Special Issue on Require-*

ments Engineering, 7(1), ISSN: 1039-7841, pp. 118 – 130 (1999).

[9] Kaneiwa, K., Satoh, K., *Consistency Checking Algorithms for Restricted UML Class Diagrams*, Proc. of FoIKS2006, *LNCS 3861*, pp. 219 – 239 (2006). An extended version with a proof can be found at `http://research.nii.ac.jp/~kaneiwa/uml-ex.pdf`

[10] Pereira, L.M., Damásio, C. V., Alferes, J. J., Diagnosis and Debugging as Contradiction Removal, *LPNMR-93*, pp. 334 – 348 (1993).

[11] Reiter, R., A Theory of Diagnosis from First Principles, *Artificial Intelligence* **32**, pp. 57 – 95 (1987).

[12] Rumbaugh, J., Jacobson, I., Booch, G., *The Unified Modeling Language Reference Manual*, Addison-Wesley, Reading, Massachusetts, USA, 1 edition (1999).

[13] Satoh, K., Minimal Revision of Logical Specification Using Extended Logic Programming: Preliminary Report, *Proc. of the AAAI-99 Workshop on Intelligent Software Engineering*, pp. 61 – 65, Orlando, Florida, USA (1999).

[14] Satoh, K., Consistency Management in Software Engineering by Abduction, *Proc. of the ICSE-2000 Workshop on Intelligent Software Engineering*, pp. 90 – 99, Limerick, Ireland (2000).

[15] Satoh, K., Computing Minimal Revised Specifications by Default Logic, *Proc. of Workshop on Intelligent Technologies in Software Engineering (WITSE2003)*, pp. 7 – 12, Helsinki, Finland (2003).

[16] Satoh, K., Uno, T., Enumerating Minimal Revised Specification using Dualization, *New Frontiers in Artificial Intelligence, Joint JSAI 2005 Workshop Post-Proceedings*, *LNAI 4012*, pp. 182 – 189 (2006).

[17] Uno, T., A Practical Fast Algorithm for Enumerating Minimal Set Coverings, *SIGAL83*, Information Processing Society of Japan, pp. 9 – 16 (in Japanese) (2002).

[18] Zisman, A., Kozlenkov, A., A Knowledge based Approach to Consistency Management of UML Specifications, *Proc. of the 16th Conference on Automated Software Engineering*, pp. 359 – 363 (2001).

[19] Zowghi, D., Ghose, A., Peppas, P., A Framework for Reasoning about Requirements Evolution, *Proc. of PRICAI'96*, pp. 157 – 168 (1996).

Appendix: Proof of Theorem 1

It is straightforward on the equivalence on translation of formulas except class attributes. We prove here the equivalence on translation of formulas related with class attributes.

The following formulas with counting quantifiers:

$$\forall x \forall y(c(x) \rightarrow (a(x,y) \rightarrow t(y)))$$
$$\forall x(c(x) \rightarrow \exists_{\geq i} z(a(x,z)))$$
$$\forall x(c(x) \rightarrow \exists_{\leq j} z(a(x,z)))$$

is equivalent to the following formulas without counting quantifiers.

$$\forall x \forall y(c(x) \wedge a(x,y) \rightarrow t(y)))$$
$$\forall x(c(x) \rightarrow \exists z_1 ... \exists z_i(a(x,z_1) \wedge ... \wedge a(x,z_i) \wedge distinct(z_1,...,z_i)))$$
$$\forall x(c(x) \rightarrow \neg\exists z_1 ... \exists z_{j+1}(a(x,z_1) \wedge ... \wedge a(x,z_{j+1}) \wedge distinct(z_1,...,z_{j+1})))$$

where $distinct(z_1,...,z_i)$ is an abbreviation of:

$$\wedge_{k=1}^{i-1} \wedge_{m=i+1}^{i} (z_k \neq z_m)$$

We introduce distinct i Skolem functions into the second formula to eliminate existential quantifiers:

$$\forall x \forall y(c(x) \wedge a(x,y) \rightarrow t(y)))$$
$$\forall x(c(x) \rightarrow (a(x,f_a^1(x)) \wedge ... \wedge a(x,f_a^i(x)) \wedge distinct(f_a^1(x),...,f_a^i(x))))$$
$$\forall x \forall z_1 ... \forall z_{j+1}(c(x) \rightarrow (\neg a(x,z_1) \vee ... \vee \neg a(x,z_{j+1}) \vee \neg distinct(z_1,...,z_{j+1})))$$

We divide the conclusion part in the second formula as follows:

$\forall x(c(x) \rightarrow a(x,f_a^1(x))) \wedge \ldots \wedge$
$\forall x(c(x) \rightarrow a(x,f_a^i(x))) \wedge$
$\forall x(c(x) \rightarrow distinct(f_a^1(x),...,f_a^i(x)))$

Note that each function symbol $f_a^k(k = 1,...,i)$ is associated only with an attribute a. That means that if $a(t,f_a^k(t))$ is derived for k and t then, for other $a(t,f_a^l(t))$ for any $l(l = 1,...,i)$ can be derived as well since we can get a proof of $a(t,f_a^l(t))$ from the proof of $a(t,f_a^k(t))$ by replacing of occurrences of f_a^k in the proof of $a(t,f_a^k(t))$ by f_a^l.

This means that it is sufficient to consider one representative rule $\forall x(c(x) \rightarrow a(x,(f_a(x)))$ in stead of $\forall x(c(x) \rightarrow a(x,(f_a^k(x)))(k = 1,...,i)$. Moreover, by abbreviating $distinct(f_a^1(x),...,f_a^i(x)))$ as $at\_least(f_a(x)),i)$, we can translate the second formula into the following Horn clauses:

$$\forall x(c(x) \rightarrow a(x,(f_a(x)))$$

$$\forall x(c(x) \rightarrow at\_least(f_a(x), i))$$

Then, the third formula leads to contradiction if and only if formulas of the form $a(x, z)$ with $j + 1$ distinct terms in the second argument z are derived. This is equivalent to the fact that $at\_least(f_a(x), i')$ s.t. $j < i'$ is derived. Therefore, the third formula is equivalent to the derivation of $contradiction$ in the following Horn clause:

$$\forall x \forall i'(c(x) \wedge at\_least(f_a(x), i') \wedge (j < i') \rightarrow contradiction)$$

Thus, the formulas with counting quantifiers can be translated into the following logic program:

```
t(Y) :- c(X),a(X,Y).
a(X,f_a(X)) :- c(X).
at_least(f_a(X),i) :- c(X).
contradiction :-
  c(X), at_least(f_a(X),I), j<I.
```

By partial evaluation of `a(X,Y)` in the first rule by the second rule, we get:

```
t(f_a(X)):-c(X),c(X)
```

This is equivalent to `t(f_a(X)):-c(X)` which is the first rule in the resulting logic program. If there is a formula like

$$\forall x(d(x) \rightarrow \exists_{\geq i} z(a(x, z))),$$

it becomes `a(X,f_a(X)) :- d(X).` and we could use this rule for partial evaluation of `a(X,Y)`. However, if we do so, we get:

```
t(f_a(X)):-c(X),d(X)
```

which is subsumed by `t(f_a(X)):-c(X)`. So, we do not have to consider this partial evaluation.

Therefore, contradiction in formula with counting quantifiers is equivalent to the derivability of `contradiction` in the following logic program:

```
t(f_a(X)) :- c(X).
at_least(f_a(X),i) :- c(X).
contradiction :-
    c(X), at_least(f_a(X),I), j<I.
```

□

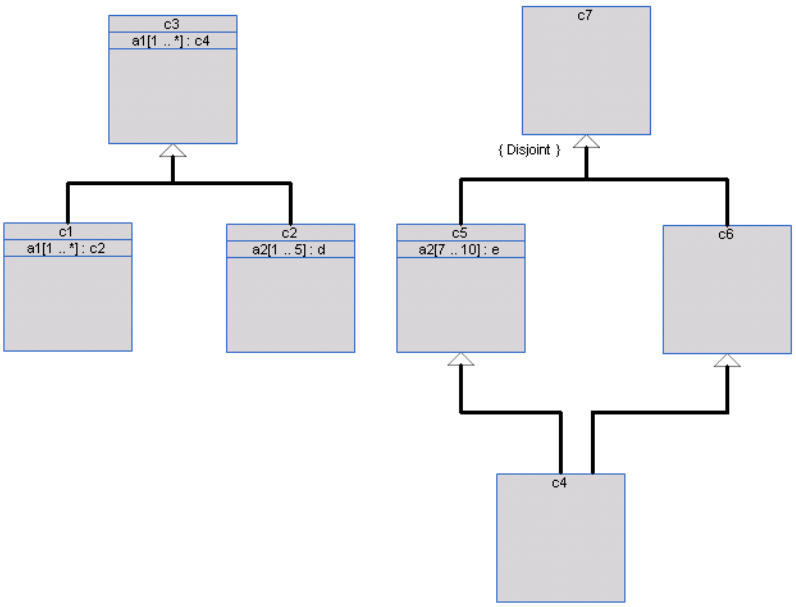

Figure 2. Inconsistent UML Class Diagram

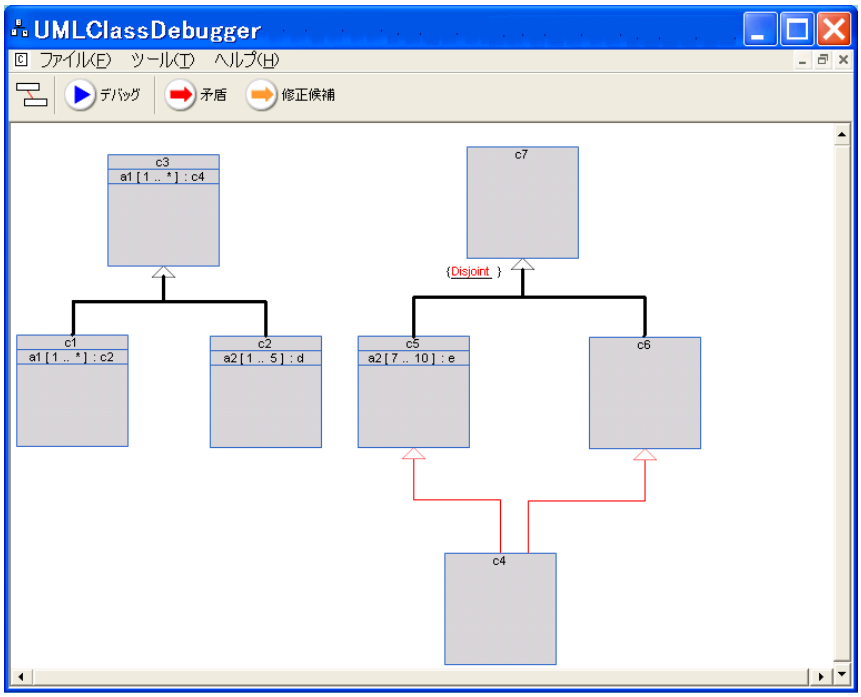

Figure 3. Minimal Inconsistent Set 1

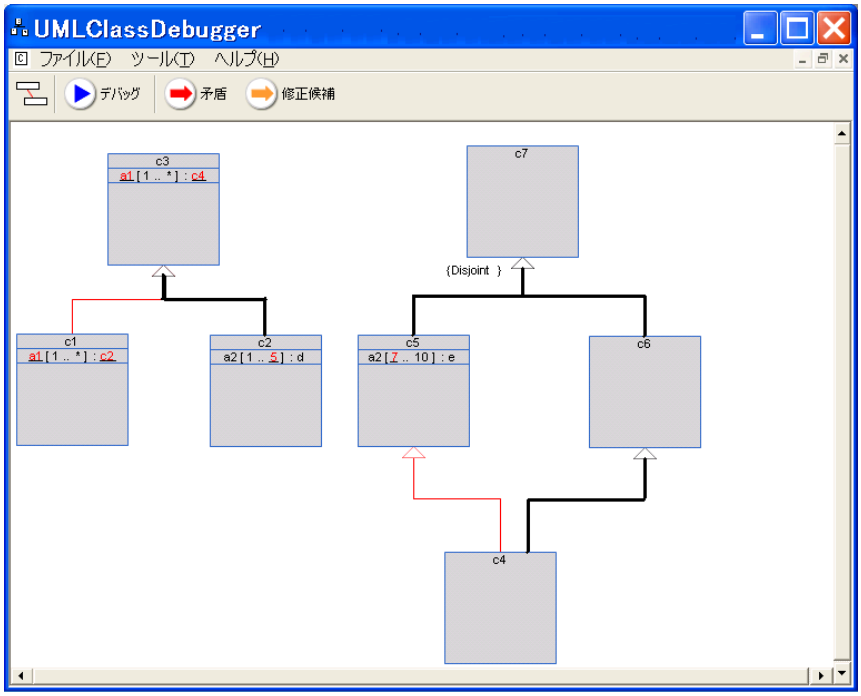

Figure 4. Minimal Inconsistent Set 2

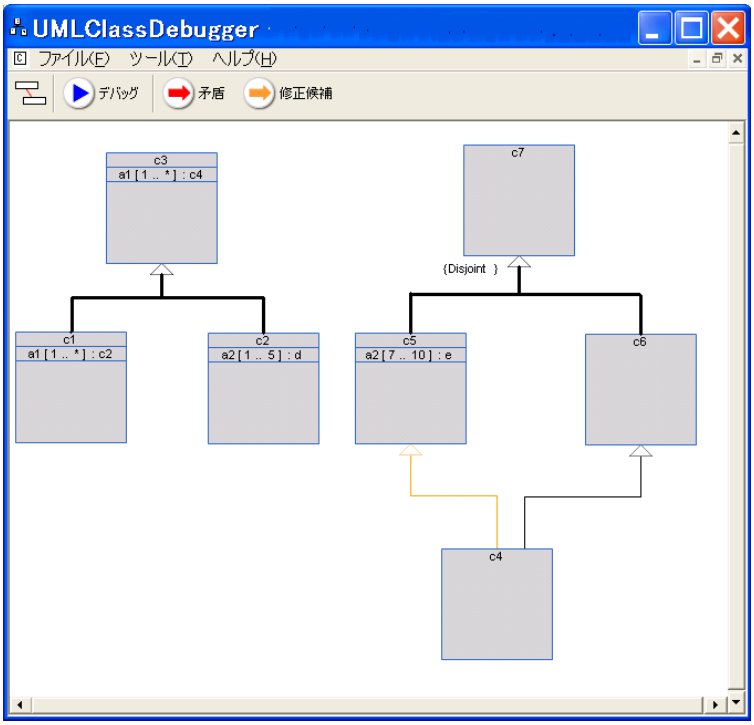

Figure 5. Example of Candidate Set of Deleted Rules