UMONS
Université de Mons
EVOLUMONS

Approaches to Software Model Inconsistency Management

Tom Mens

In collaboration with J. Pinna Puissant,
R. Van Der Straeten, X. Blanc

Faculté
des Sciences

http://www.evolumons.be
http://informatique.umons.ac.be/genlog

CSMR 2010: 15th European Conference on Software Maintenance and Reengineering

March 1–4, 2011, Oldenburg, Germany

www.se.uni-oldenburg.de/csmr2011

October 1: Workshop submission deadline

October 15: Paper abstracts

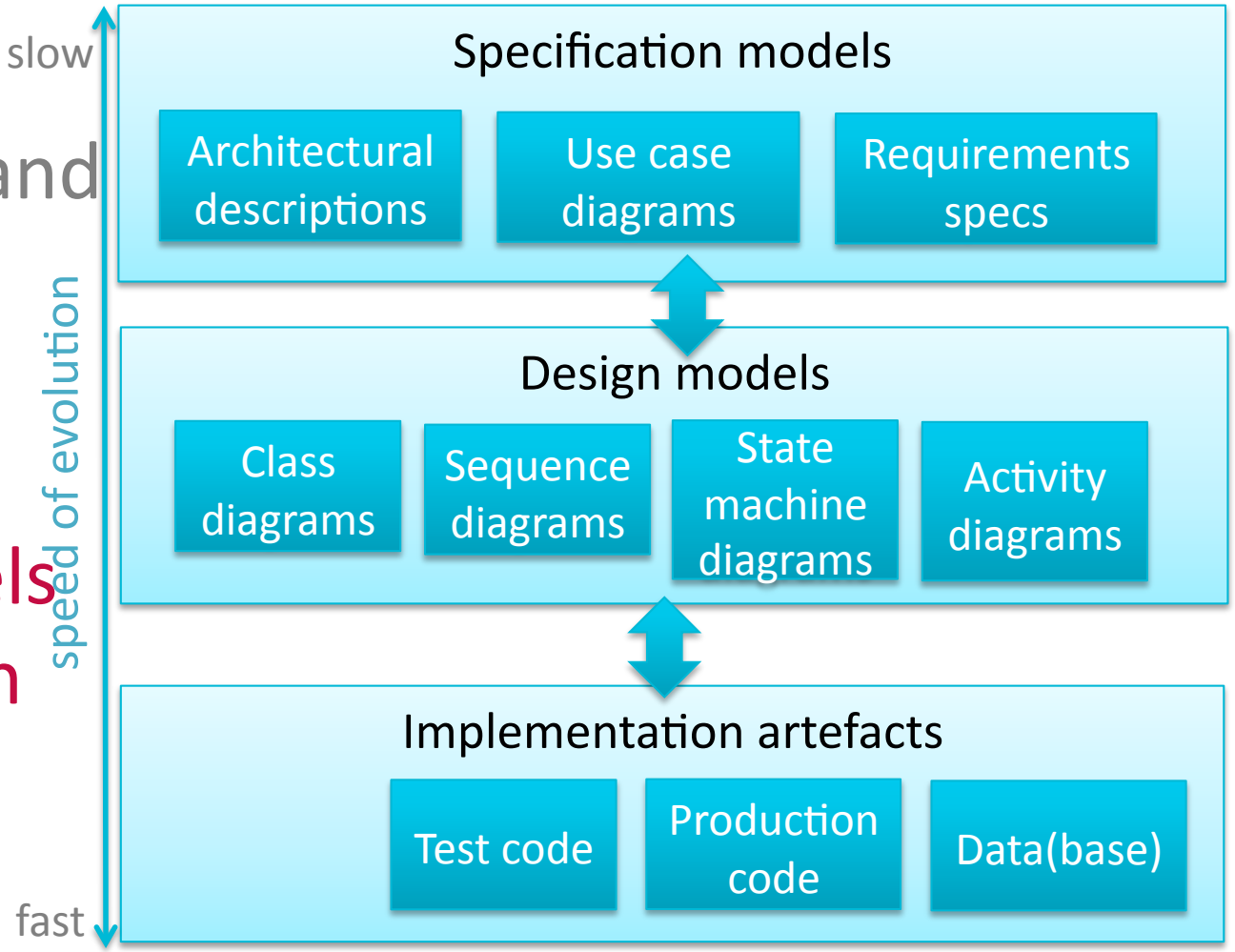
October 22: Paper submission deadline

Commercial break

CARL VON OSSIETZKY universität OLDENBURG

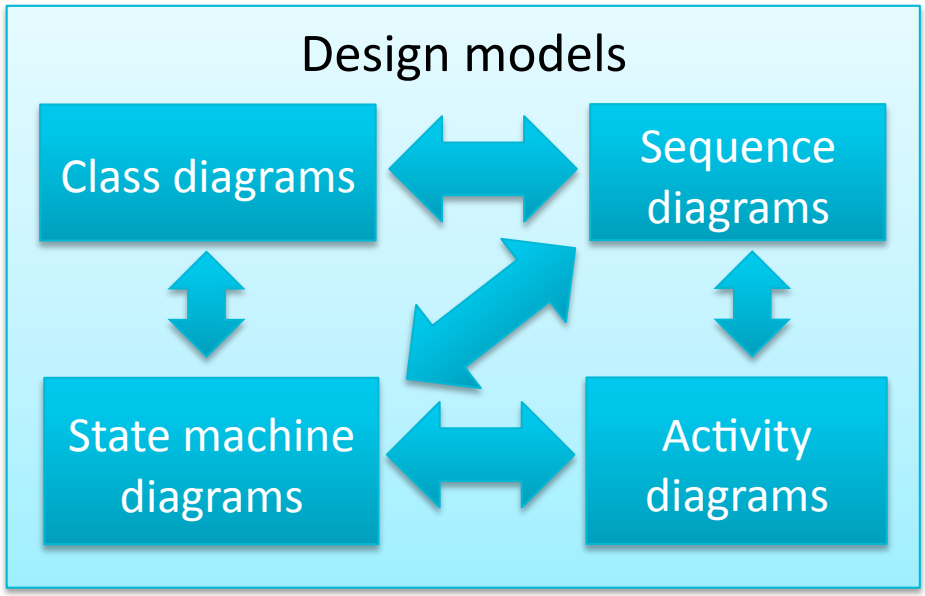
Main challenge: Software co-evolution

1. Need to synchronise and co-evolve between artefacts at different levels of abstraction



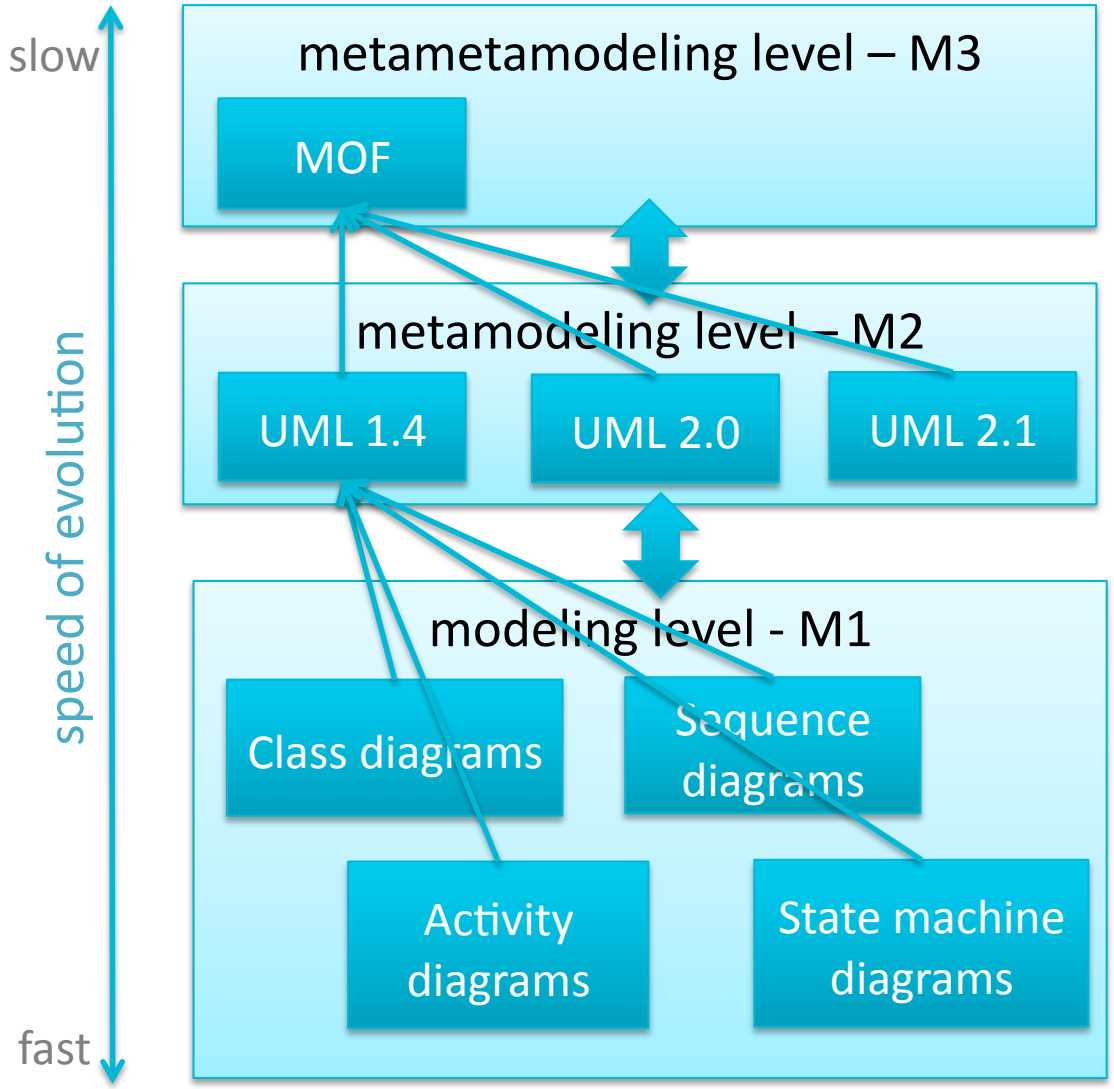
Main challenge: Software co-evolution

2. Need to synchronise and co-evolve between artefacts at same level of abstraction



Main challenge: Software co-evolution

3. Need to synchronise and co-evolve between artefacts at different layers of modelling



Observation

- The co-evolution problem arises everywhere
  - In all phases of the software life-cycle
  - In and between all levels of abstraction
  - In and between all layers of modelling
- We need generic techniques for managing co-evolution
  - Inconsistency management
  - Impact analysis, change propagation, traceability
  - Versioning, differencing, merging
  - Refactoring, restructuring

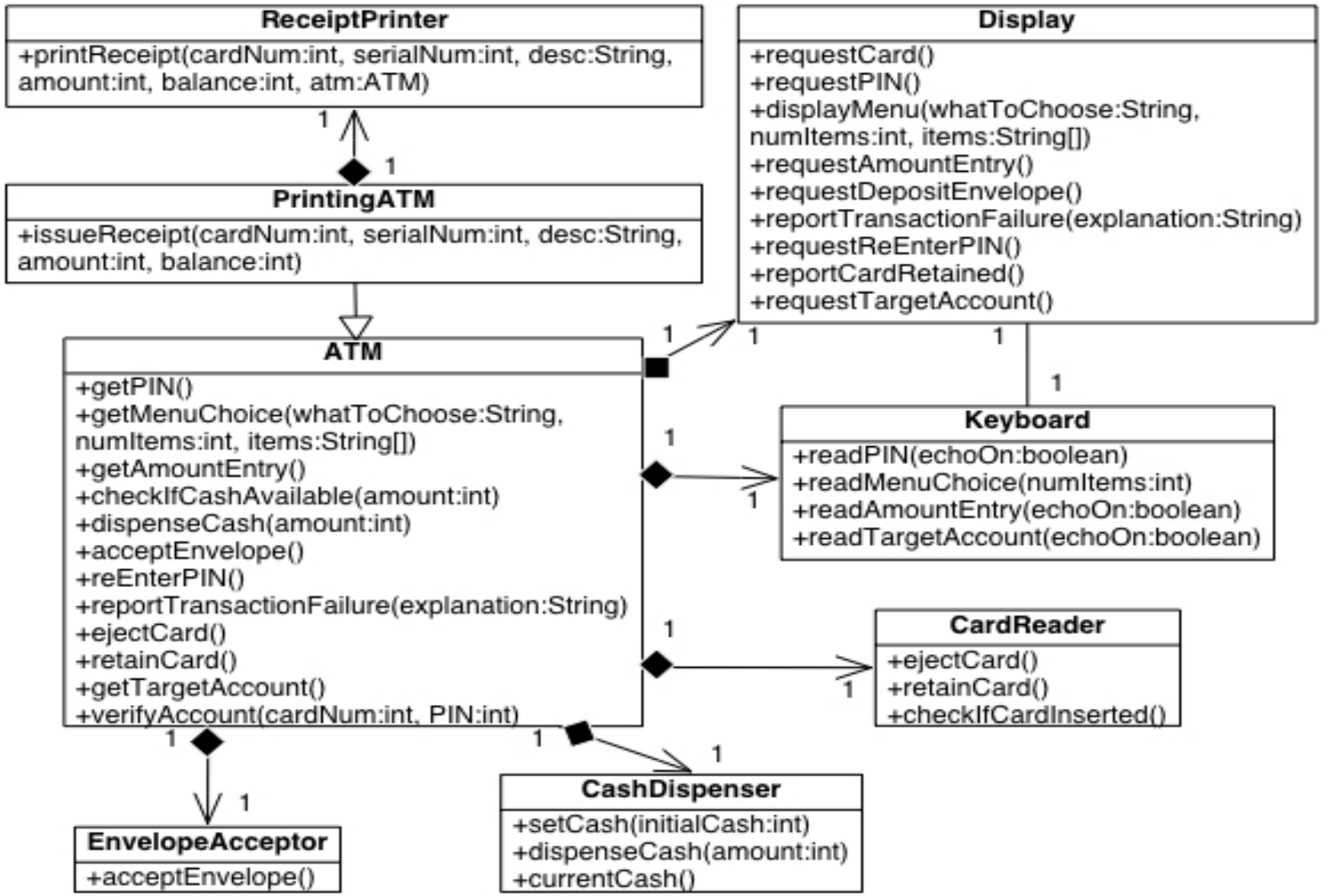
Research context

- ARC Research project (2008-2012) on
  - Model-Driven Software Evolution
- Focus on different techniques to evolve software models
  - Model refactoring
  - Model inconsistency management
- Study and develop tools and techniques to support these activities

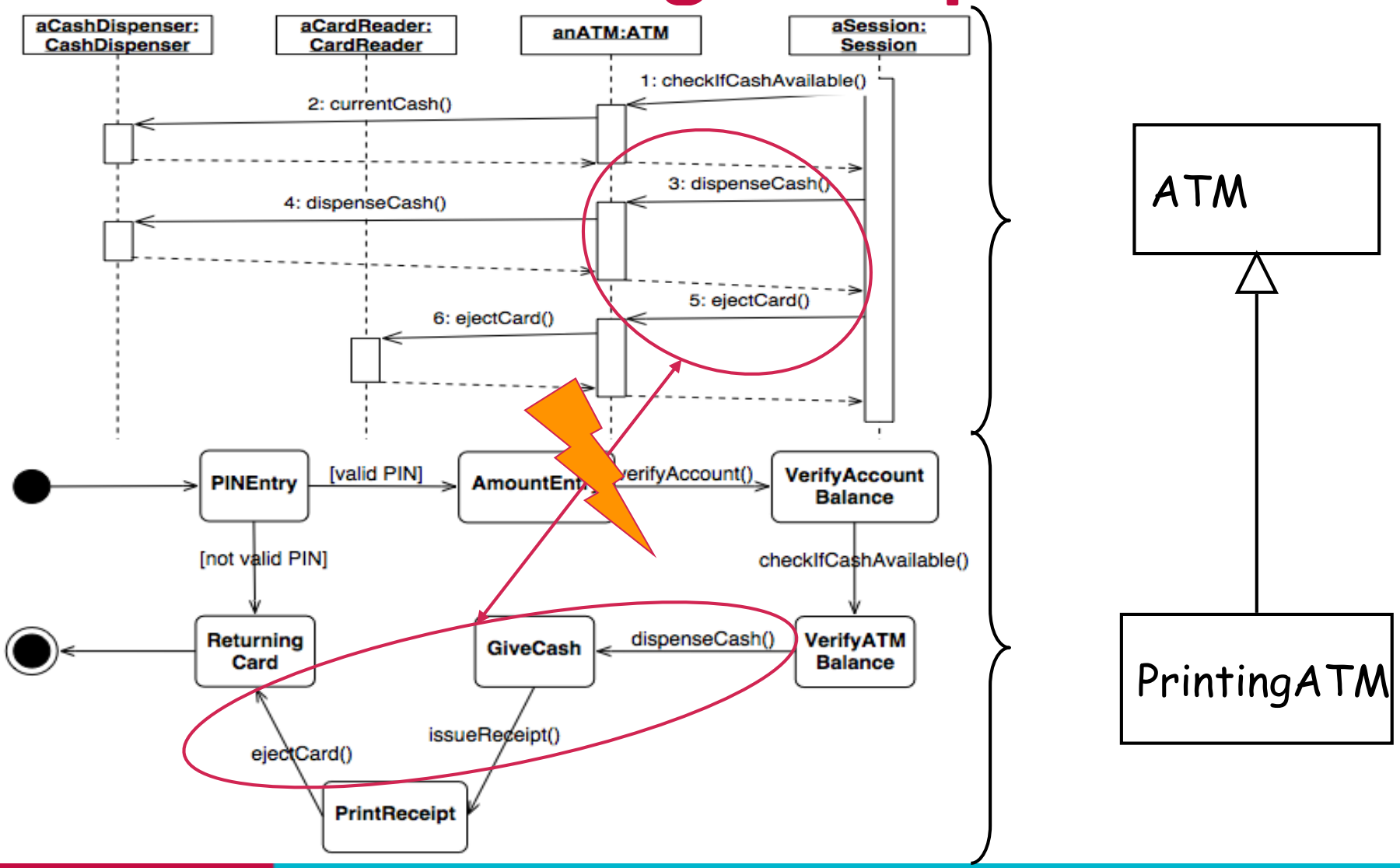
Overview of the talk

- Introduction to model inconsistency management
- Generic techniques for managing model inconsistencies
  - Using description logics
  - Using graph transformation
  - Using logic programming
  - Using automated planning

Model inconsistency management
Motivating example



Model inconsistency management
Motivating example



Model inconsistency management Introduction

Many potential sources of inconsistencies

- Poorly understood models
- Unclear, ambiguous, underspecified, incomplete models
  - due to poorly specified requirements
  - due to use of natural language
  - ill-specified semantics of modeling language
- Complexity of models with many interdependencies
- Distributed, collaborative, parallel development
- Continuous evolution of models
- …

Model inconsistency management Introduction

Many different types of inconsistencies

- lexical, structural or behavioural inconsistencies
- visual problems
- non-conformance to standards and design conventions
- design smells, bad practices, antipatterns
- incomplete models
- redundant models
- …

Overview of the talk

Generic techniques for managing model inconsistencies

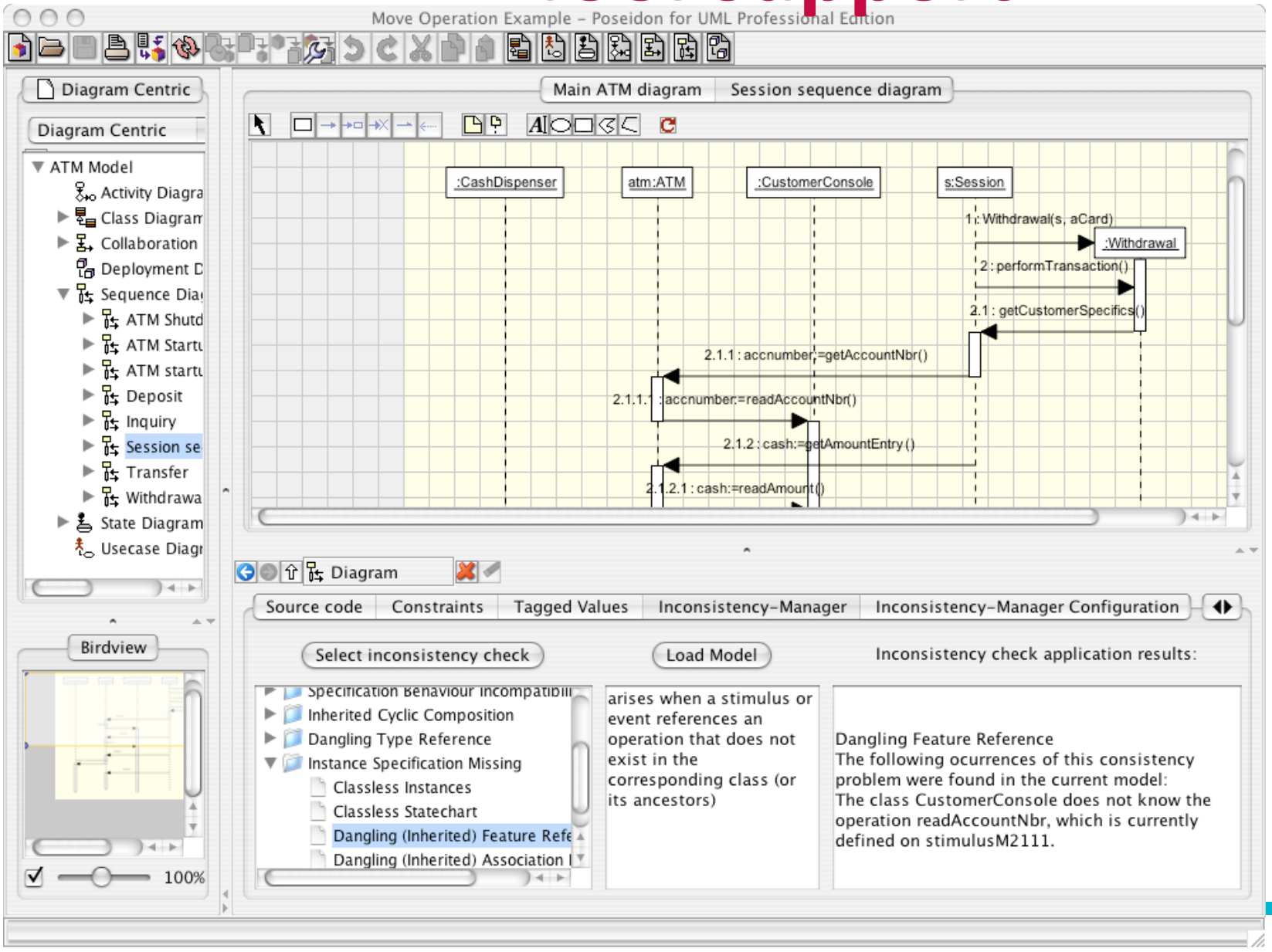
- **Using description logics**
- Using graph transformation
- Using logic programming
- Using automated planning

Model inconsistency management Using Description Logics

Ragnhild Van Der Straeten et al. (VUB and Umons, Belgium)

- PhD “Inconsistency management in model-driven engineering: an approach using description logics”, 2005
- UML 2003: *Using description logics to maintain consistency between UML models*

Model inconsistency management
Tool support



Model inconsistency management Using Description Logics

Decidable 2-variable fragment of first-order predicate logic

- Tool support: RACER

Construct Name	Syntax	Semantics	
atomic concept	A	$A^{\mathcal{I}} \subseteq \Delta^{\mathcal{I}}$	$\mathcal{S}$
universal concept	$\top$	$\top^{\mathcal{I}} = \Delta^{\mathcal{I}}$	
atomic role	R	$R^{\mathcal{I}} \subseteq \Delta^{\mathcal{I}} \times \Delta^{\mathcal{I}}$	
transitive role	$R \in \mathbf{R}_+$	$R^{\mathcal{I}} = (R^{\mathcal{I}})^+$	
conjunction	$C \sqcap D$	$C^{\mathcal{I}} \cap D^{\mathcal{I}}$	
disjunction	$C \sqcup D$	$C^{\mathcal{I}} \cup D^{\mathcal{I}}$	
negation	$\neg C$	$\Delta^{\mathcal{I}} \setminus C^{\mathcal{I}}$	
exists restriction	$\exists R.C$	$\{x \mid \exists y.\langle x,y\rangle \in R^{\mathcal{I}} \text{ and } y \in C^{\mathcal{I}}\}$	
value restriction	$\forall R.C$	$\{x \mid \forall y.\langle x,y\rangle \in R^{\mathcal{I}} \text{ implies } y \in C^{\mathcal{I}}\}$	
role hierarchy	$R \sqsubseteq S$	$R^{\mathcal{I}} \subseteq S^{\mathcal{I}}$	$\mathcal{H}$
inverse role	R^-	$\{\langle x,y\rangle \mid \langle y,x\rangle \in R^{\mathcal{I}}\}$	$\mathcal{I}$
number restrictions	$\geqslant nR$	$\{x \mid \sharp\{y.\langle x,y\rangle \in R^{\mathcal{I}}\} \geqslant n\}$	$\mathcal{N}$
	$\leqslant nR$	$\{x \mid \sharp\{y.\langle x,y\rangle \in R^{\mathcal{I}}\} \leqslant n\}$	
qualifying number restrictions	$\geqslant nR.C$	$\{x \mid \sharp\{y.\langle x,y\rangle \in R^{\mathcal{I}} \text{ and } y \in C^{\mathcal{I}}\} \geqslant n\}$	$\mathcal{Q}$
	$\leqslant nR.C$	$\{x \mid \sharp\{y.\langle x,y\rangle \in R^{\mathcal{I}} \text{ and } y \in C^{\mathcal{I}}\} \leqslant n\}$	

Model inconsistency management Using Description Logics

Class diagrams

```
(instance atm class)
(instance printingatm class)
(instance asciiprintingatm class)
(instance gen1 generalization)
(instance gen2 generalization)
(instance controls association)
(instance atmend property)
(instance asciiend property)
(related atm gen1 generalization)
(related printingatm gen1 generalization)
(related printingatm gen2 generalization)
(related asciiprintingatm gen2
  generalization)
(related atm printingatm direct-superclas
(related printingatm asciiprintingatm
  direct-superclass)
(related gen1 atm general)
(related gen2 printingatm general)
(related atm asciiend ownedAttribute)
```

Sequence diagrams

```
(define-concept m1
  (and (all op checkIfCashAvailable)
       (exactly 1 op)))
(define-concept m2
  (and (all op ejectCard)
       (exactly 1 op))))
```

Statemachine diagrams

```
(define-concept t1
  (and (all op verifyAccount)
       (exactly 1 op)))
(define-concept t2
  (and (all op checkIfCashAvailable)
       (exactly 1 op)))
(define-concept t3
  (and (all op dispensecash)
       (exactly 1 op)))
(define-concept t4
  (and (all op ejectCard)
       (exactly 1 op)))
(equivalent (and (some valid_PIN) t1)
            (some r t2)))
(equivalent t2 (some r t3))
(equivalent t3 (some r t4))
```

Overview of the talk

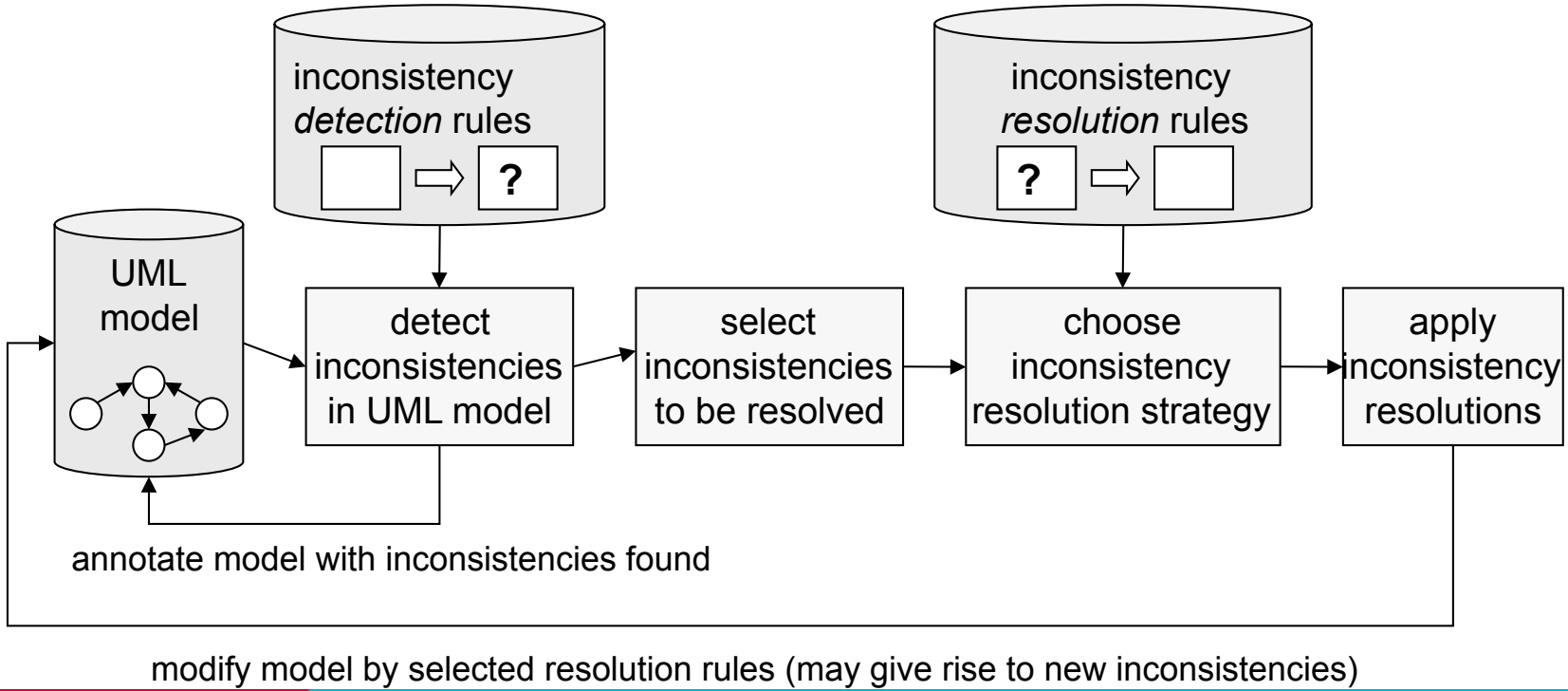
Generic techniques for managing model inconsistencies

- Using description logics
- **Using graph transformation**
- Using logic programming
- Using automated planning

Model inconsistency management using graph transformation

Mens et al. (UMONS, Belgium)

- WADT 2006: *Incremental resolution of model inconsistencies*
- MoDELS 2006: *Detecting and resolving model inconsistencies using transformation dependency analysis*



Model inconsistency management using graph transformation

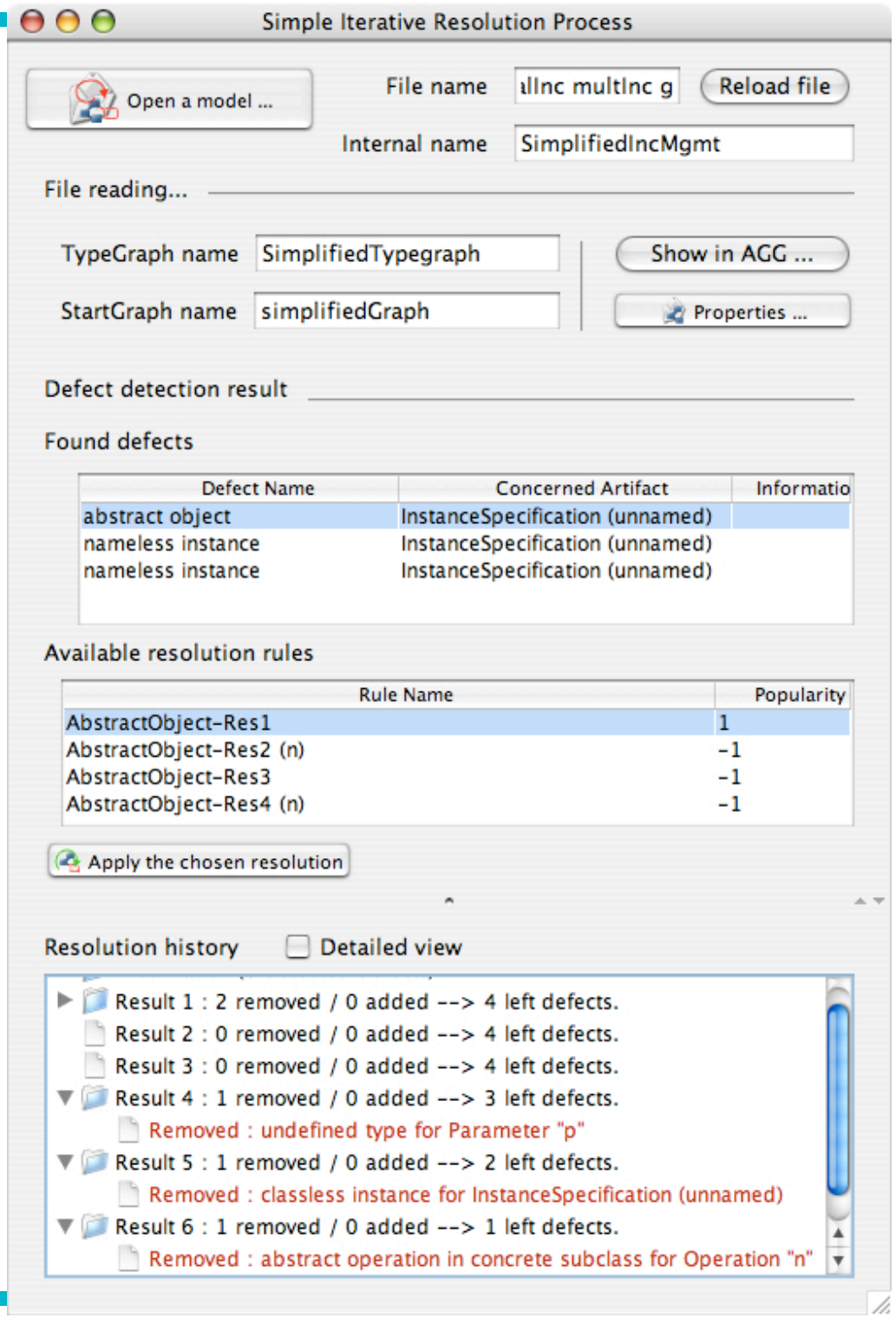
Use graph transformations to

- Specify model inconsistencies
  - Automate the *detection* of inconsistencies
- Specify inconsistency resolution rules
  - Interactively support the *resolution* of inconsistencies
- Formally analyse transformations rules
  - Detect *sequential dependencies* and *parallel conflicts* between resolution rules
  - Remove redundancy between resolution rules
  - Provide “optimal” resolution strategies

Tool support

SIRP tool

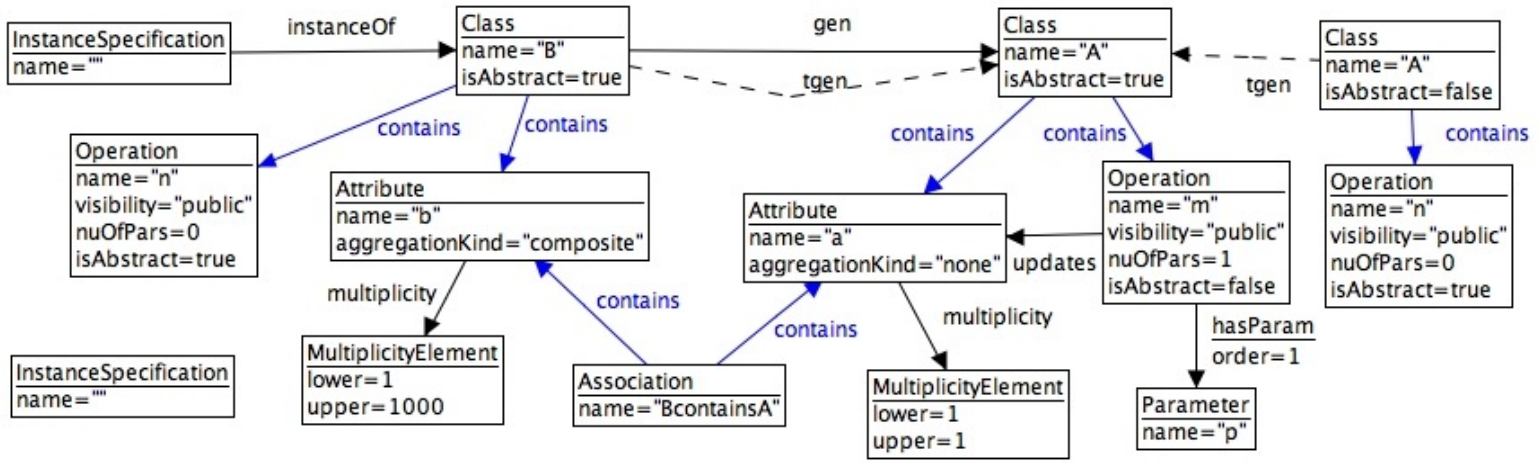
- "Simple Iterative Resolution Process"
- An interactive tool for selecting and resolving model inconsistencies
- Implemented on top of AGG graph transformation tool



Tool support

SIRP tool in action

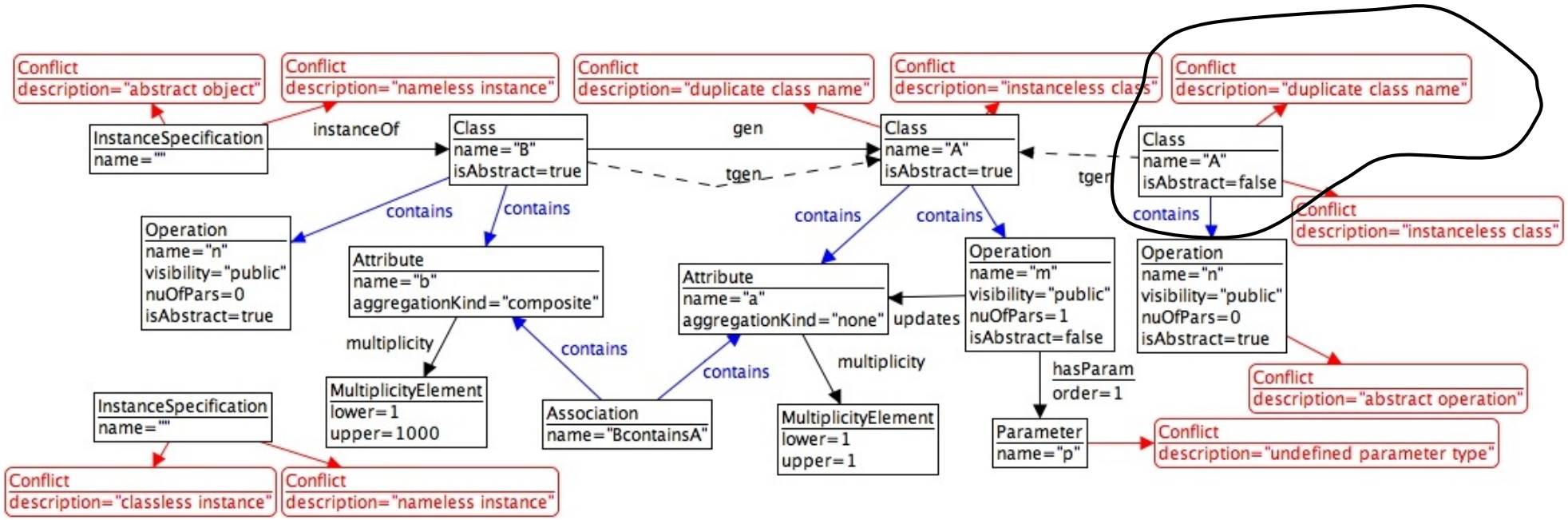
- Before detecting any inconsistency



Tool support

SIRP tool in action

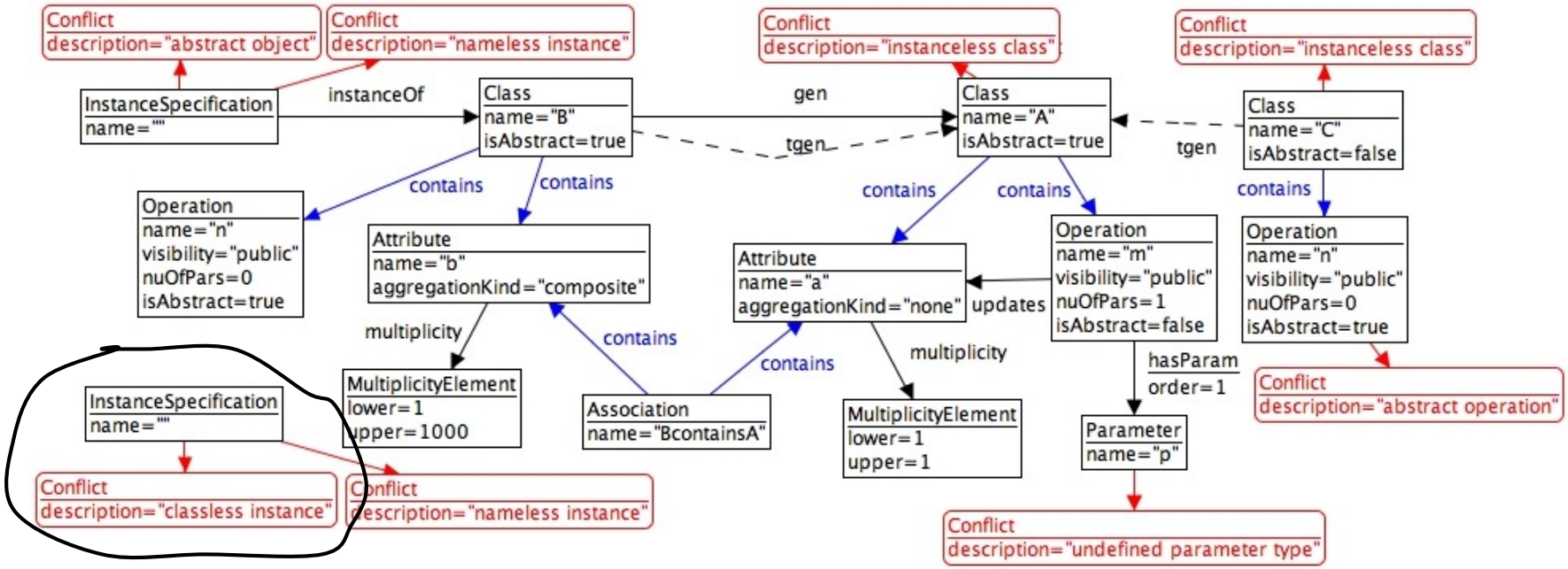
- After detecting all inconsistencies



Tool support

SIRP tool in action

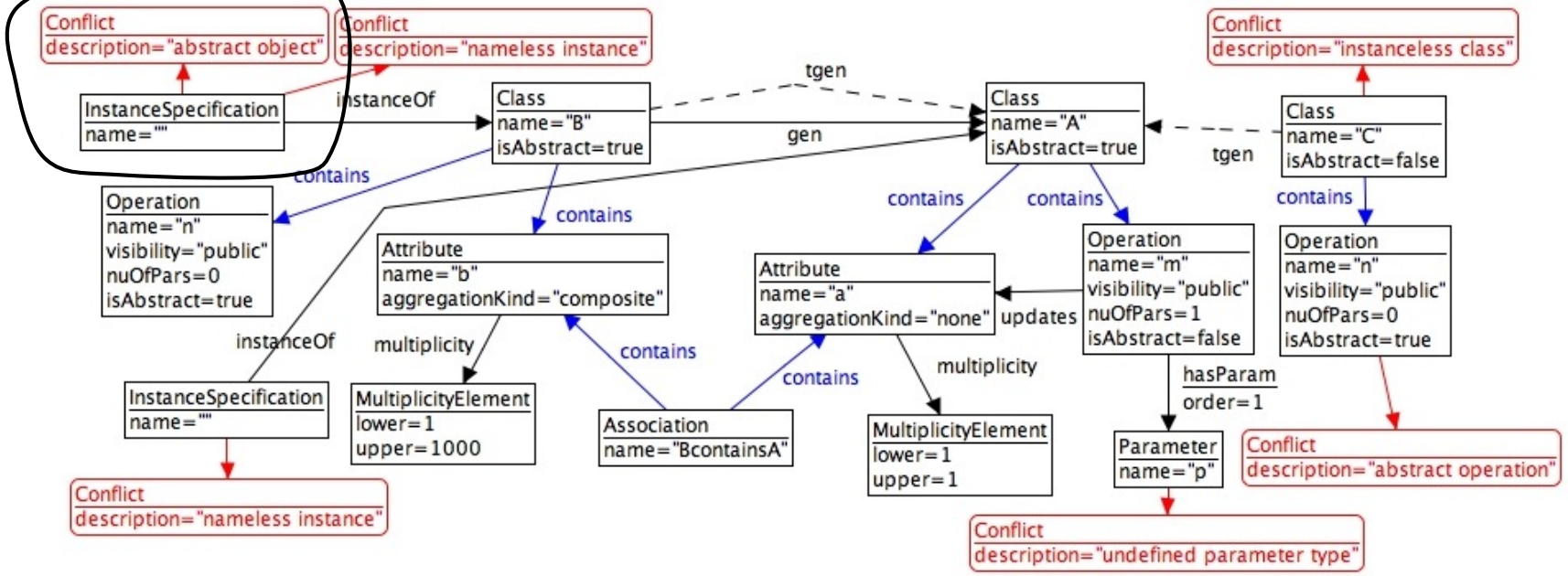
- After resolving "*duplicate class name*"
  - Two occurrences of same inconsistency removed
  - Class renamed from "A" to "C"



Tool support

SIRP tool in action

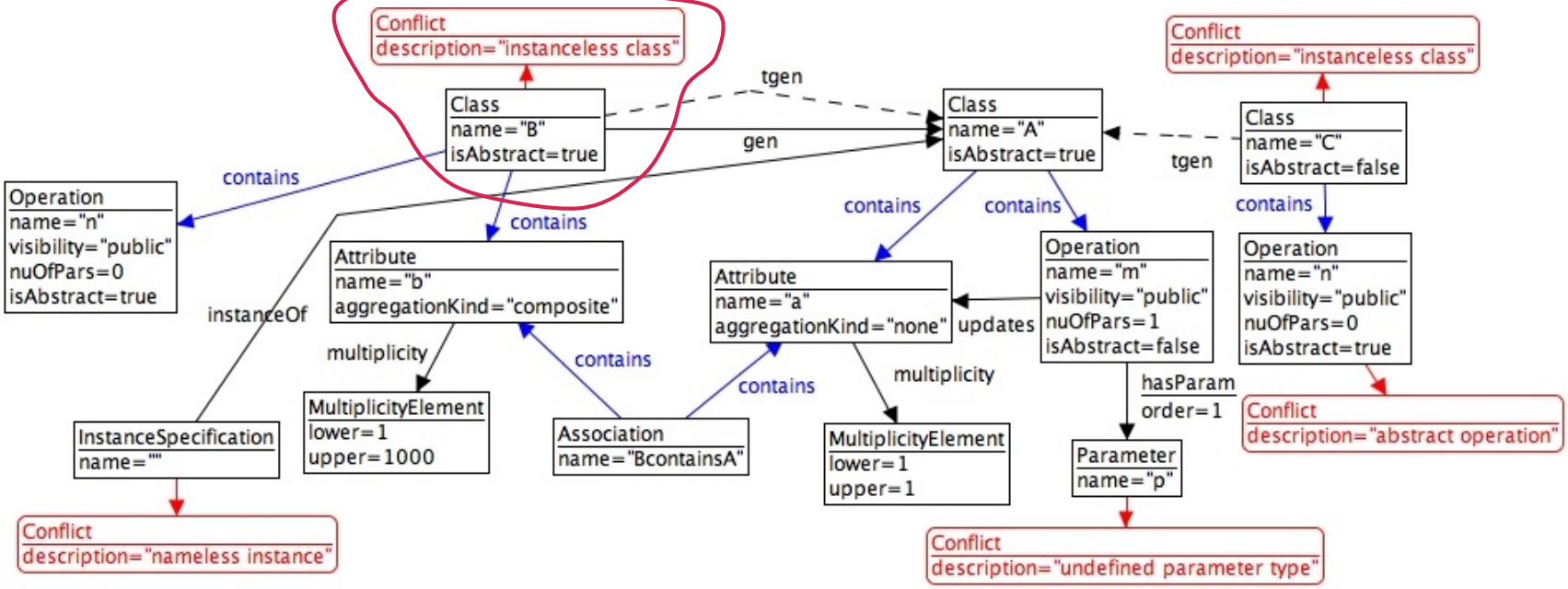
- After resolving "*classless instance*"
  - One occurrence of "classless instance" removed
  - One occurrence of "instanceless class" removed



Tool support

SIRP tool in action

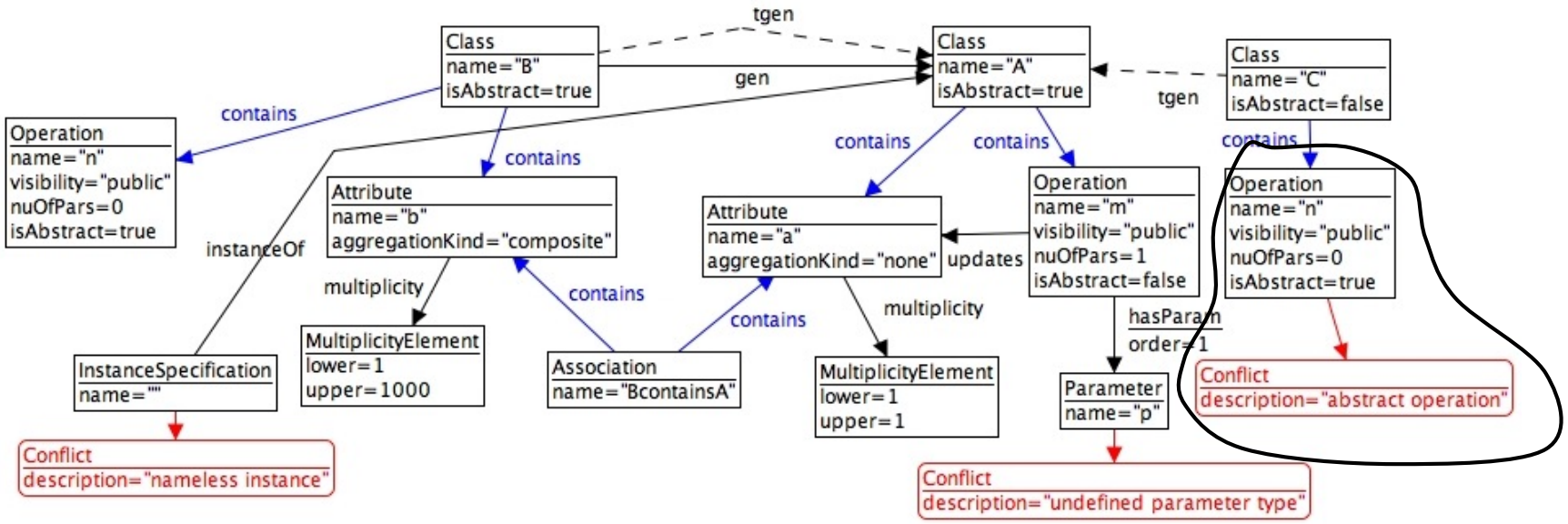
- After resolving "*abstract object*"
  - One occurrence of "abstract object" removed
  - One occurrence of "nameless instance" removed
  - Induced inconsistency: "instanceless class" !



Tool support

SIRP tool in action

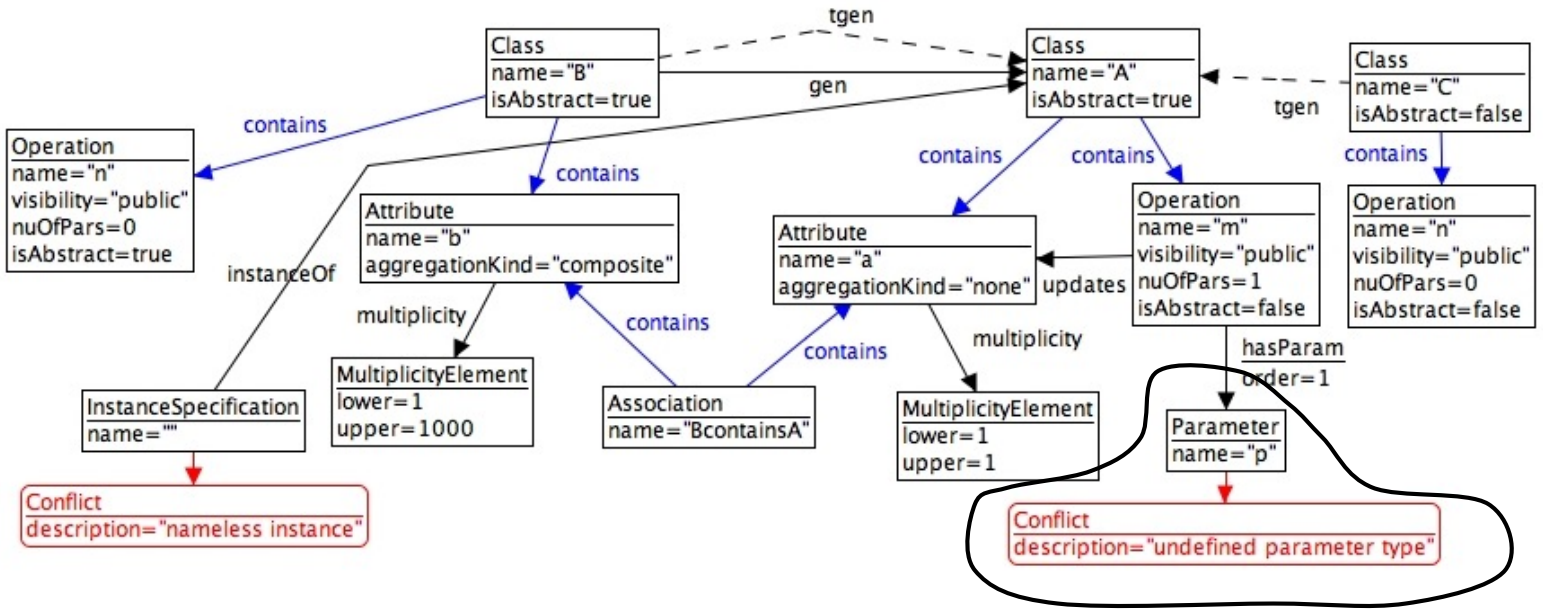
- After disabling the "*instanceless class*" rule
  - Two occurrences of "instanceless class" *ignored*



Tool support

SIRP tool in action

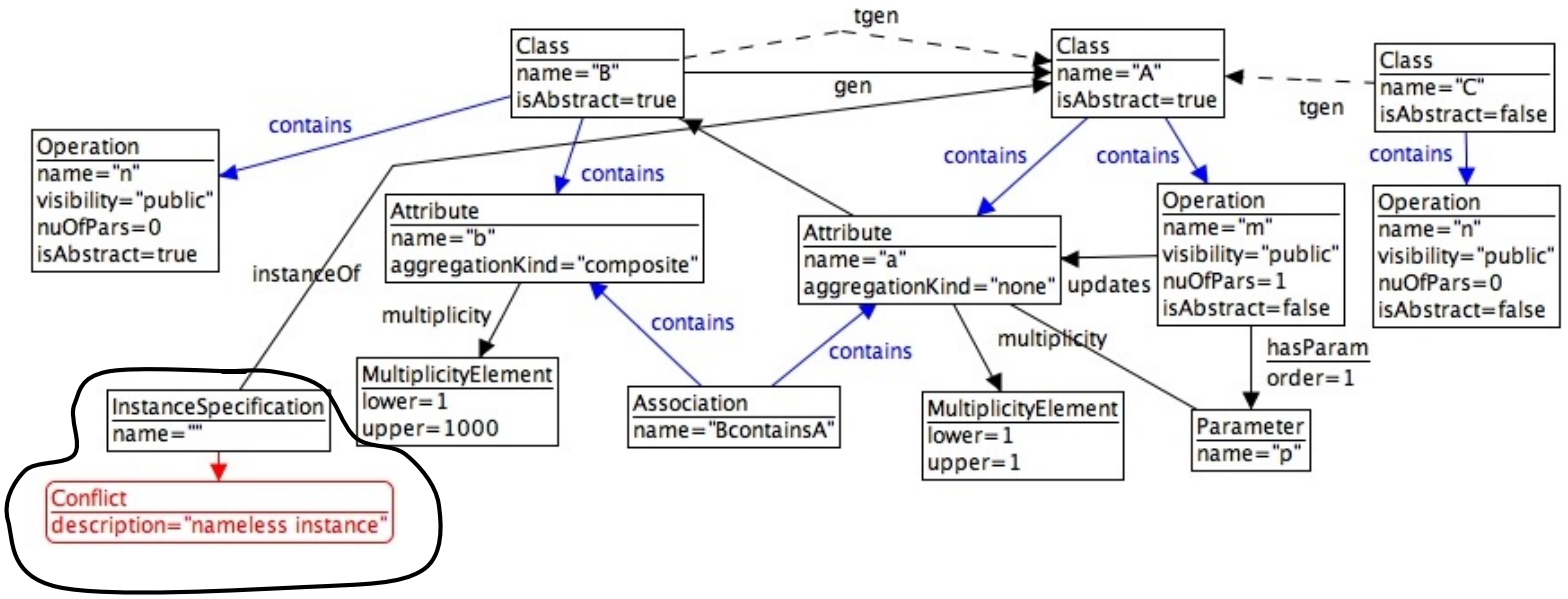
- After resolving "*abstract operation*"
  - One occurrence of "abstract operation" removed



Tool support

SIRP tool in action

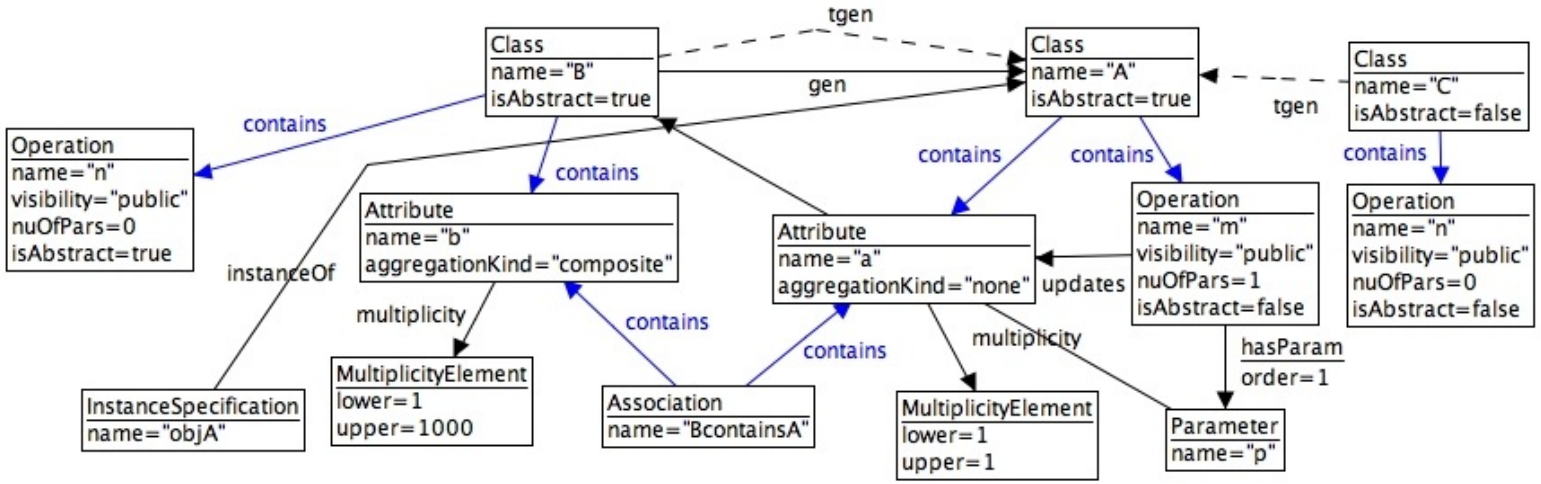
- After resolving "*undefined parameter type*"
  - One occurrence of "undefined parameter type" removed



Tool support

SIRP tool in action

- After resolving "*nameless instance*"
  - One occurrence of "nameless instance" removed
  - No more remaining inconsistencies !

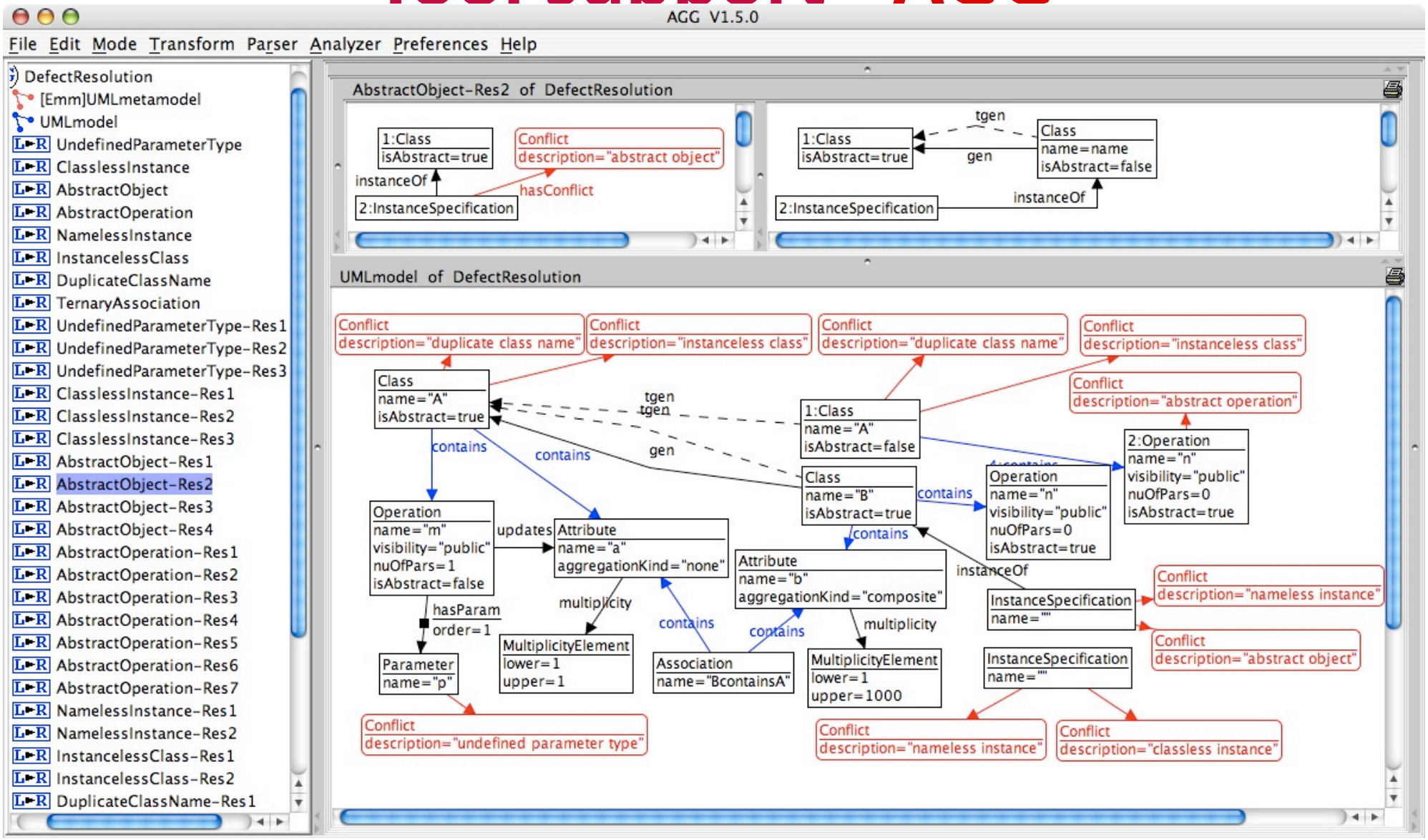


Tool support

We used AGG in the following way

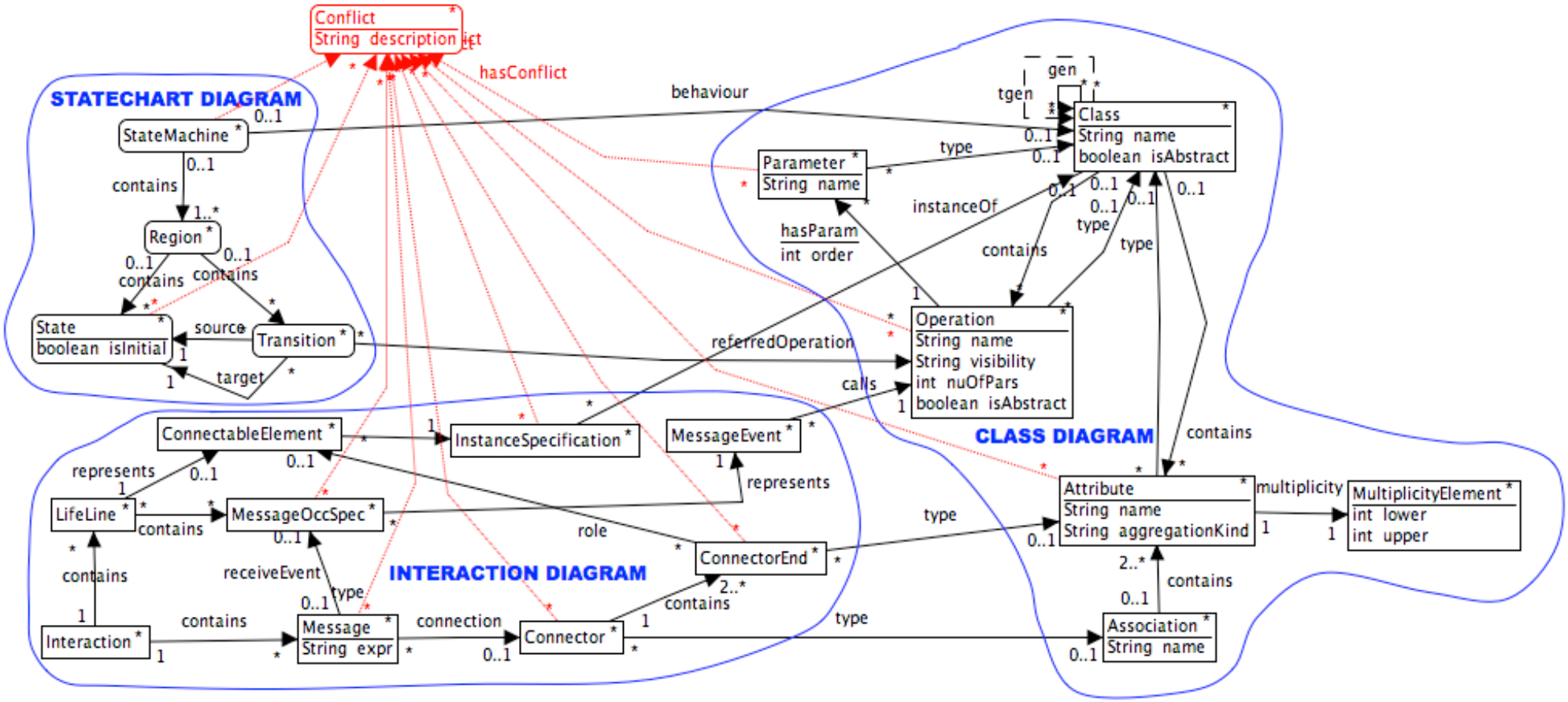
- specify the *UML metamodel* as a type graph
- specify the *models* as graphs
- *detect* and *resolve* model inconsistencies by means of graph transformation rules
- analyse *mutual exclusion* relationships and *sequential dependencies* between inconsistency resolutions by means of critical pair analysis

Tool support - AGG



Specify the metamodel

AGG type graph



Specify model inconsistencies

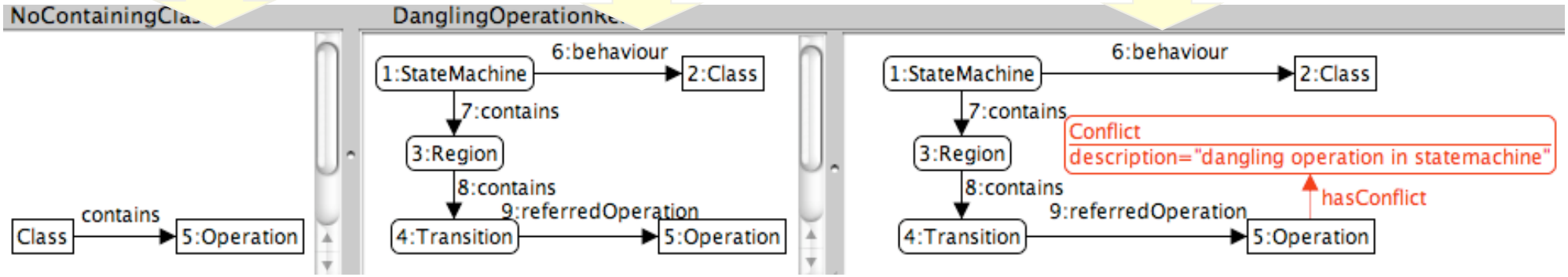
Dangling operation reference

– Graph transformation rule for detection

negative application condition (NAC)

left-hand side (LHS)

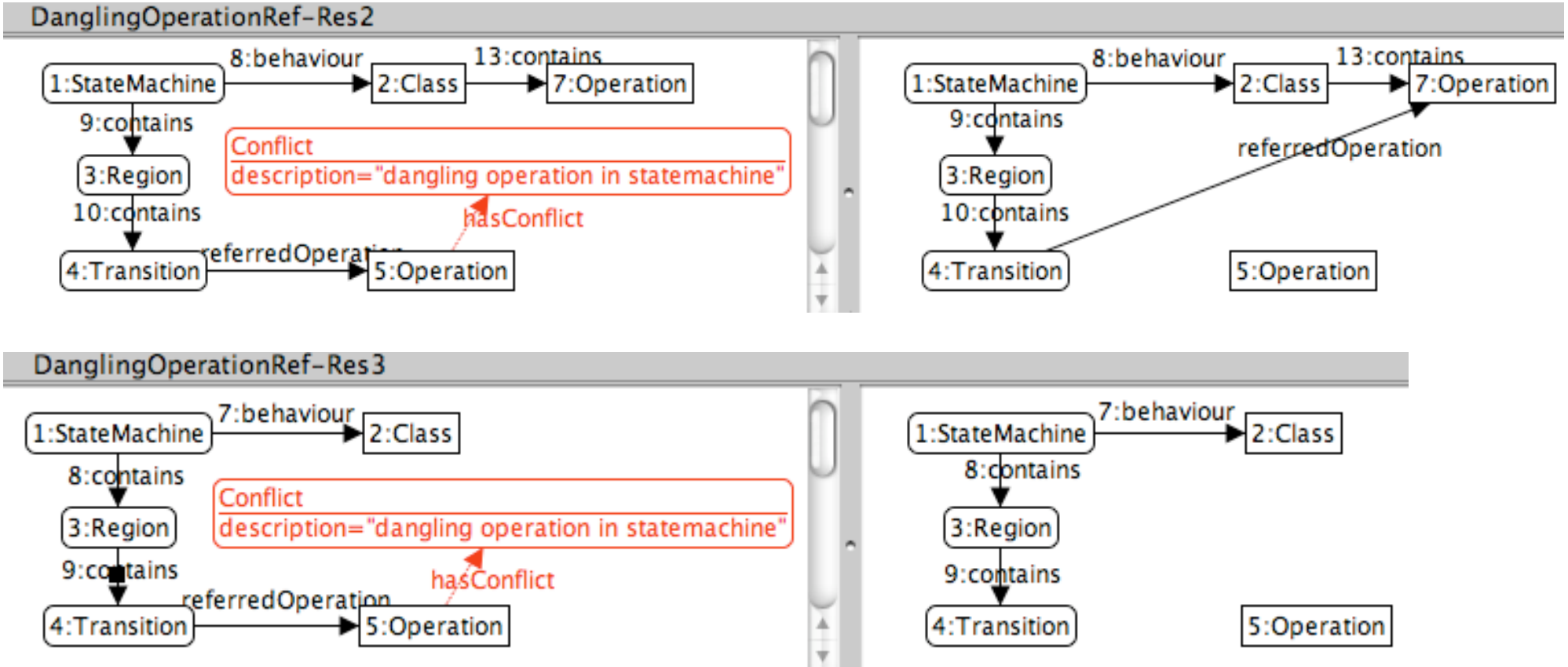
right-hand side (RHS)



Specify inconsistency resolutions

Dangling operation reference

– Graph transformation rule for resolution



Analyse mutually conflicting resolutions

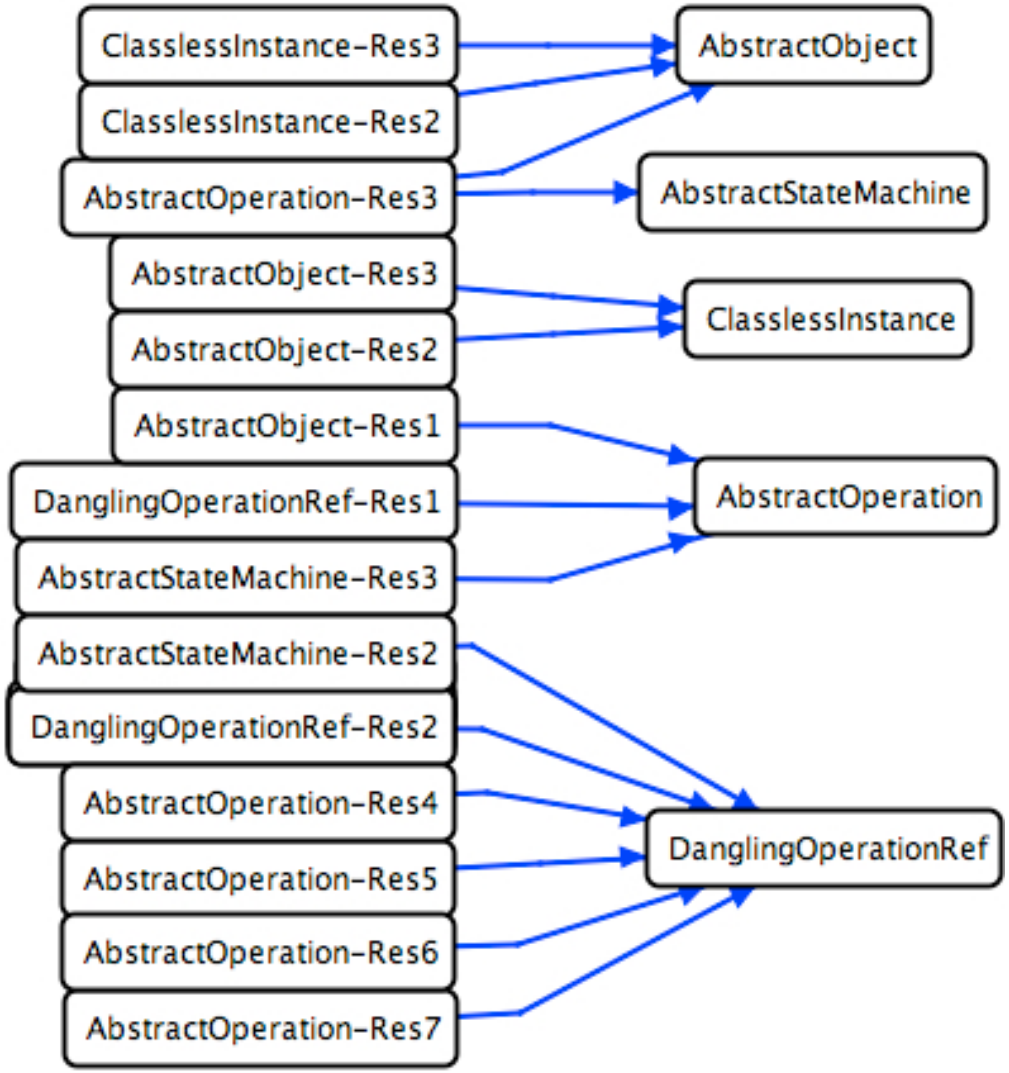
Some resolution rules cannot be jointly applied (parallel conflict!)

- Conflict graph can be generated by means of critical pair analysis

Analyse sequential dependencies

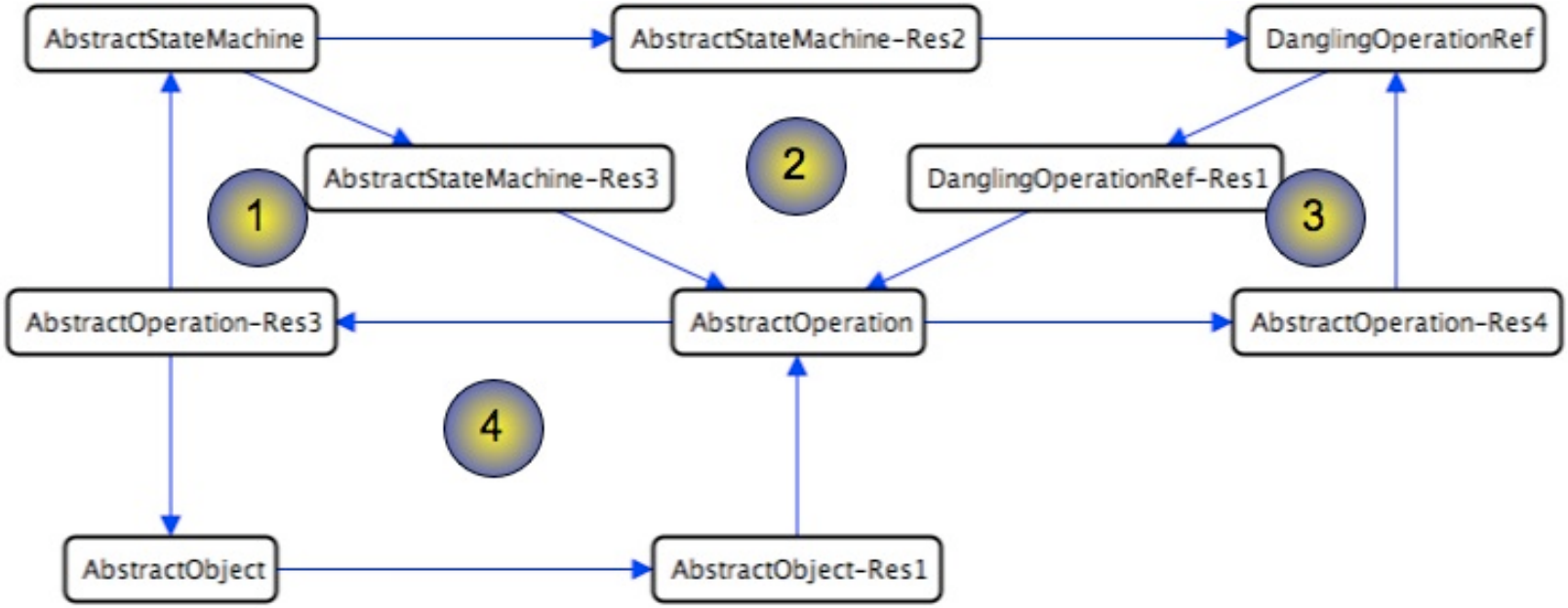
Induced inconsistencies

- Some resolution rules may give rise to new model inconsistencies
- Can be detected by analysing the dependency graph



Analyse sequential dependencies

Use the dependency graph to detect potential cycles in the resolution process



Cycles should be avoided, since this implies that the resolution process may continue forever…

Overview of the talk

Generic techniques for managing model inconsistencies

- Using description logics
- Using graph transformation
- **Using logic programming**
- Using automated planning

Model inconsistency management Using Logic Programming

Xavier Blanc *et al.* (LIP6, Paris)

ICSE 2008 : *Detecting model inconsistency through operation-based model construction.*

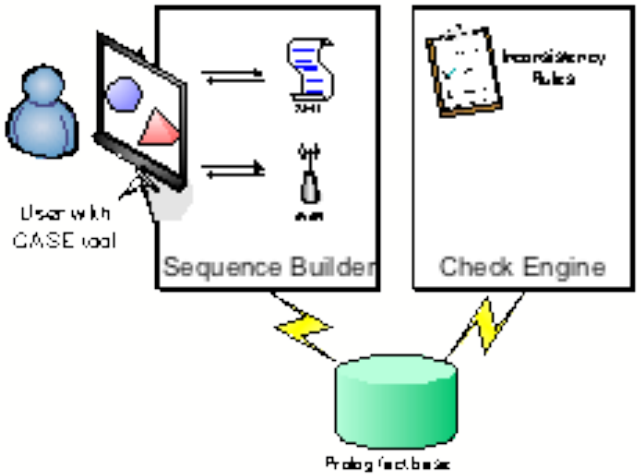
CAiSE 2009 : *Incremental detection of model inconsistencies based on model operations.*

CAiSE 2010 : *Towards Automated Inconsistency Handling in Design Models.*

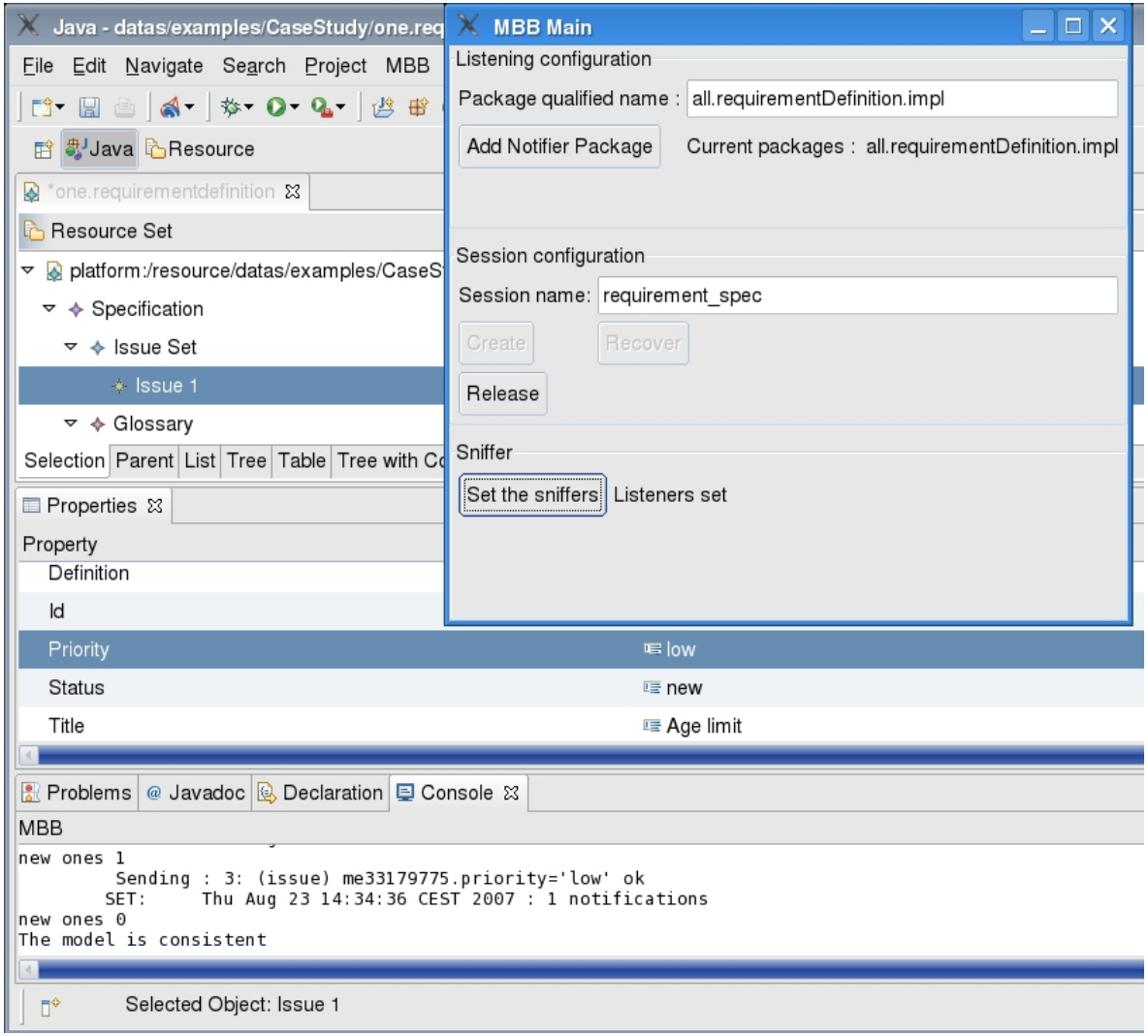
LWI 2010 : *Inconsistency Detection in Distributed Model Driven Software Engineering Environments.*

Model inconsistency management Using Logic Programming

- Tool architecture



- Sequence builder: Event listener integrated in Eclipse EMF framework
- Check engine: Implemented in Prolog



General idea

Express *models* and *model transformations* in a *uniform* and *metamodel-independent* way

- As a *sequence* of elementary operations

Express *constraints* over a model and its evolution as a *logic query* over this sequence

- Both “structural” and “temporal” constraints can be expressed in this way

Provide efficient and incremental tool support

- for checking inconsistencies
- For resolving inconsistencies

Praxis

an operation-based representation of models and their construction/evolution

`create(me,mc)`

creation of a model element instance *me* of the meta-class *mc*

Preconditions:

- *me* does not exist in the model

Postconditions:

- *me* exists in the model; does not own values; does not refer to another model element

`delete(me)`

deletion of the model element instance *me*

Preconditions:

- *me* exists in the model
- *me* is not referenced by another model element

Postconditions:

- *me* does not exist in the model

`setProperty(me,p,Values)`

assignment of a set of *Values* to property *p* of model element *me*

Preconditions:

- *me* exists in the model
- *me* is an instance of meta-class *mc*
- *p* is defined in (an ancestor of) *mc*
- *p* and Values have same signature

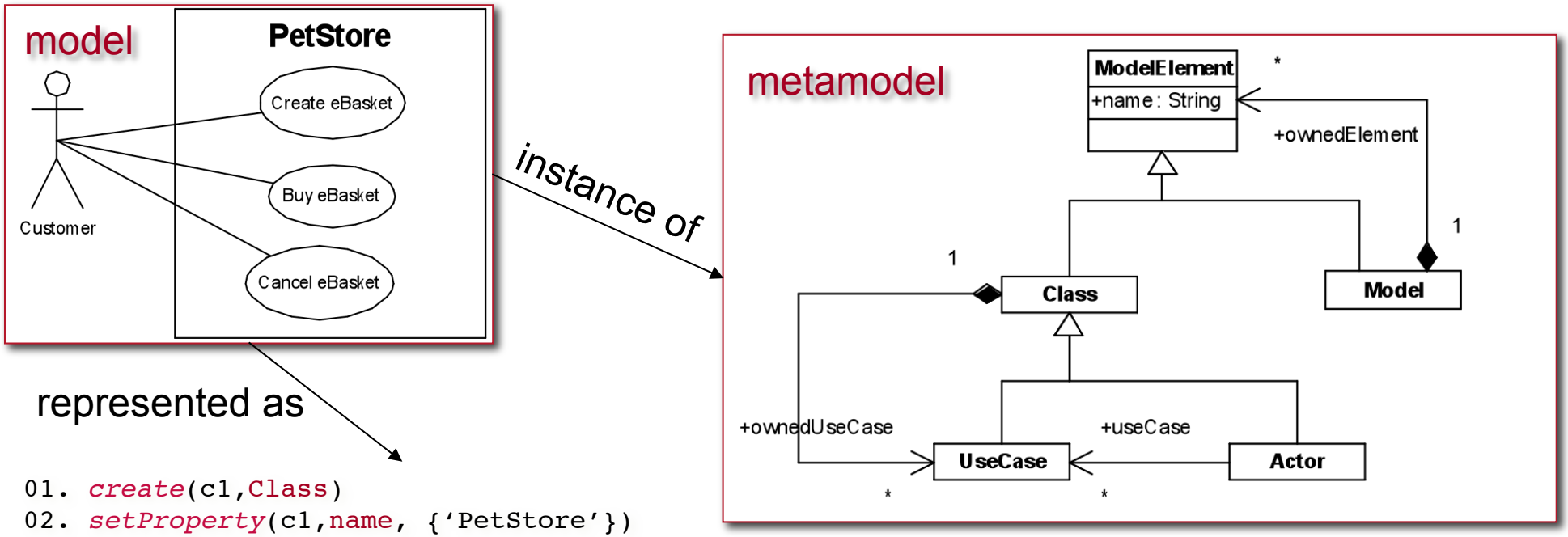
`setReference(me,r,References)`

assignment of *References* to reference *r* of model element *me*

Preconditions:

- *me* exists in the model
- *me* is an instance of meta-class *mc*
- *r* is defined in (an ancestor of) *mc*
- *r* and *References* have same signature
- All *References* exist in the model

Example



represented as

```
01. create(c1,Class)
02. setProperty(c1,name, {'PetStore'})
03. create(uc1,UseCase)
04. setProperty(uc1,name, {'Buy eBasket'}))
05. create(uc2,UseCase)
06. setProperty(uc2,name,{'Create eBasket'})
07. create(uc3,UseCase)
08. setProperty(uc3,name,{'Cancel eBasket'})
09. setReference(c1,ownedUseCase,{uc1,uc2,uc3})
10. create(a1,Actor)
11. setProperty(a1,name, {'Customer'})
12. setReference(a1, usecase, {uc1,uc2,uc3})
```

representation as a sequence of elementary construction operations

constrained by information in the metamodel

Model inconsistency management Using Logic Programming

Example of a structural inconsistency

- an actor should not own a use case

$$\text{if } a = \text{lastCreate (me, Actor) then}$$
$$!\exists\, o \in \sigma_{uc},\ o = \text{lastSetReference(me,ownedUseCase, R)}$$
$$\text{and } R \neq \varnothing.$$

Example of a temporal inconsistency

- Never unassign a use case name

$$\forall\, a \in \sigma_{uc},\ a \neq \text{setProperty(me,name,\{""\})}$$

- Assign a name just after the creation of a use case

$$\forall\, a \in \sigma_{uc},\ \text{if } a = \text{create(me,UseCase) then}$$
$$\exists\, c \in \sigma,\ c = \text{setProperty(me,name, NameVal)}$$
$$\text{and } b \in \sigma,\ a <_{\sigma} b <_{\sigma} c$$

Overview of the talk

Generic techniques for managing model inconsistencies

- Using description logics
- Using graph transformation
- Using logic programming
- **Using automated planning**

Model inconsistency management Using Automated Planning

Pinna Puissant *et al.* (UMons, Belgium)

LWI 2010: *Resolving model inconsistencies with automated planning.*

Automated planning

General idea

- given an initial state and a specific predefined goal, we can create a plan (a sequence of primitive actions) leading to the goal

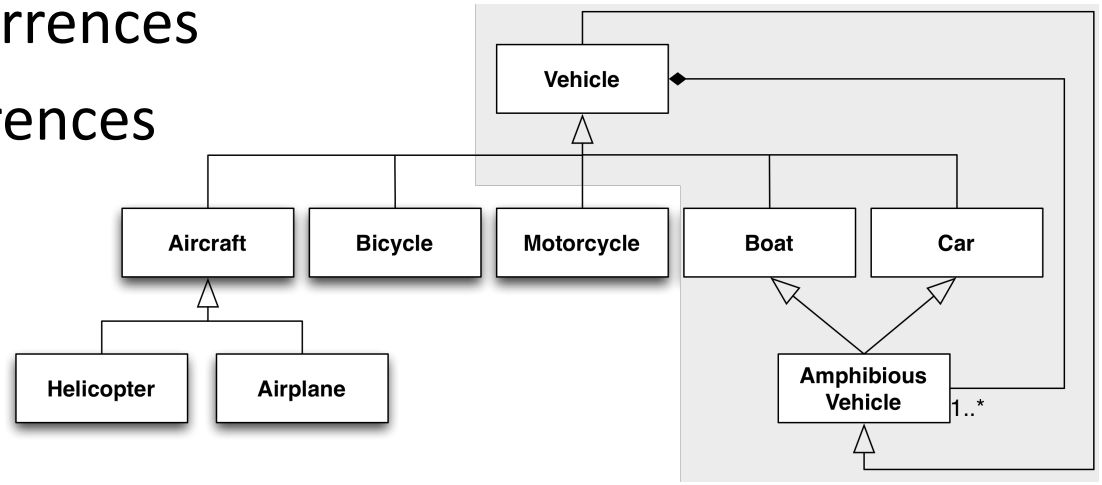
In our case:

- Initial state = inconsistent model
- Goal = consistent model
- Plan = set of actions needed to make the model consistent

Automated Planning Example

Model with 4 inconsistencies

- Cyclic composition: 2 occurrences
- Cyclic inhertance: 2 occurrences



Generated resolution plan

1. delete generalisation Vehicle → AmphibiousVehicle
2. modify multiplicity of AmphibiousVehicle association end from 1..* to 0..*

Automated planning

Different planning approaches and tools exist

- Languages: PDDL, STRIPS, ADL, …
- Tools: FF, …
- Techniques
  - Translation to a satisfiability problem (model checking)
  - Generating and traversing a search space
    - Progression planning (forward search)
    - Regression planning (backward search)

Scalability for model inconsistency resolution

- Tests are disappointing: search space explosion
- Dedicated, customised, planners should perform better

Overview of the talk

Generic techniques for managing model inconsistencies

- Using description logics
- Using graph transformation
- Using logic programming
- Using automated planning
- **Other possibilities?**

Model inconsistency management
Other possibilities?

- *Use of model checkers (e.g., SPIN, Promela,NuSMV) and model finders (e.g., Alloy Analyzer, Kodkod) to resolve model inconsistencies*
  - Van Der Straeten *et al.* (VUB & UMons, Belgium)
  - *Work in progress (submitted for publication)*
- Search-Based Software Engineering (SBSE) seems to be particularly useful for model inconsistency management. A wide variety of different techniques could be used:
  - metaheuristics
  - local search algorithms
  - automated learning
  - genetic algorithms
  - …

Conclusion