Efficient Analysis of Pattern-Based Constraint Specifications

Michael Wahler[1]**, David Basin**[2]**, Achim D. Brucker**[3]**, Jana Koehler**[4]

[1] ABB Corporate Research, Industrial Software Systems, 5405 Baden-Dättwil, Switzerland, e-mail: `michael.wahler@ch.abb.com`
[2] ETH Zurich, 8092 Zurich, Switzerland, e-mail: `basin@inf.ethz.ch`
[3] SAP Research, 76131 Karlsruhe, Germany, e-mail: `achim.brucker@sap.com`
[4] IBM Zurich Research Laboratory, 8803 Rüschlikon, Switzerland, e-mail: `koe@zurich.ibm.com`

Abstract Precision and consistency are important prerequisites for class models to conform to their intended domain semantics. Precision can be achieved by augmenting models with design constraints and consistency can be achieved by avoiding contradictory constraints. However, there are different views of what constitutes a contradiction for design constraints. Moreover, state-of-the-art analysis approaches for proving constrained models consistent either scale poorly or require the use of interactive theorem proving.

In this paper, we present a heuristic approach for efficiently analyzing constraint specifications built from constraint patterns. This analysis is based on precise notions of consistency for constrained class models and exploits the semantic properties of constraint patterns, thereby enabling syntax-based consistency checking in polynomial-time. We introduce a consistency checker implementing these ideas and we report on case studies in applying our approach to analyze industrial-scale models. These studies show that pattern-based constraint development supports the creation of concise specifications and provides immediate feedback on model consistency.

Key words UML, OCL, constraints, patterns, consistency

1 Introduction

Model-Driven Engineering (MDE) [39] comprises the generation of executable code from specifications given by models. In MDE, a model of the intended system and its various aspects (e. g., security requirements [4]) can be initially specified using graphical models at a high level of abstraction. During the development process, these models are incrementally refined and eventually transformed into code in some programming language that describes a partial implementation. In particular, graphical models can be refined by annotating model elements with constraints in a textual constraint language. This increases the models' precision because model developers can express details of the system not expressible with graphical languages and thus specify a system that conforms to the intended domain semantics.

In order to use textual constraints, model developers must have three different kinds of expertise. First, they must be knowledgeable about the application domain in order to understand the requirements behind each constraint. Second, they must understand the graphical modeling language employed. Third, they must be able to correctly formulate constraints on their models in the textual constraint language used. These are nontrivial requirements and developers often fall short of the task. In particular, due to the quantity and complexity of the model elements involved, constraint specifications can contain constraints that are inadvertently contradictory, resulting in an inconsistent model. Such inconsistencies violate the validity of the model and can lead to the generation of erroneous code, which will only be detected during testing, if at all. Since detecting and remedying inconsistencies requires costly development iterations, it is important that inconsistencies are detected as early as possible in the development process.

Whereas the topic of constraint consistency has recently attracted some attention in the context of MDE, existing approaches either consider the consistency of models without general constraints, fail to precisely define the underlying consistency notions, or abstract from intricacies such as infinitely-large model states. Moreover, we have not found a comprehensive survey of formal consistency notions in the MDE literature and it is unclear which consistency notions are relevant for MDE-based processes. As we will see, different consistency notions are relevant for different development phases.

Detecting inconsistencies in object-oriented specifications with arbitrary constraints is undecidable. Currently, there are two kinds of tools tackling this problem. The first kind is based on interactive theorem proving and has the advantage of supporting the analysis of arbitrary specifications. Successfully applying such tools, however, requires expertise in interactive theorem proving. In contrast, the second kind is based on automated model-checking techniques [15]. Their use requires simplifying the problem by restricting the

specification to a decidable, usually NP-complete, subset of the specification language. Subsequently, classical model-checking or SAT-solving [22] techniques can be applied. Hence, current tools require a trade-off between expressiveness and automation.

1.1 Motivation

We motivate the problem of consistency for constrained class models with an example. Figure 1 shows a class model of a company in the Unified Modeling Language (UML) [50]. The model defines the concepts of employee, manager (a kind of employee), office, and two kinds of offices: single offices and cubicles. Employees and managers are related by an association that represents an employment relationship. Employees and offices are related by another association with the requirement that each employee can work in at most one office.
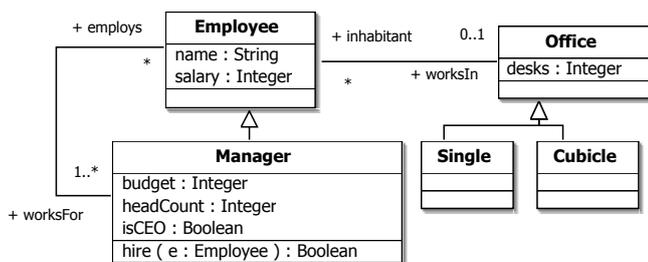

Figure 1 Class model of a company.

In UML, model states (i. e., instances of a model) can be constrained in only limited ways using graphical elements of class models. For example, the multiplicity of association ends can be constrained by specifying natural numbers as the lower and upper bound. Multiplicities cannot, however, be constrained to stand in some given relation to an attribute value. For example, it is impossible in class models to specify without textual constraints that the number of employees of each manager cannot be more than the value of the manager's headCount attribute. As a result, multiplicity constraints of association ends are often too general to be used in code generation because they allow the instantiation of states that do not conform to the intended domain semantics.
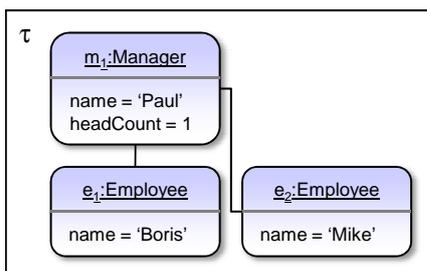

Figure 2 Invalid company state.

Figure 2 provides an example of a state where a manager employs two employees, despite having a headcount of one. Although this state *does* conform to the model in Figure 1, it does not conform to the domain semantics where the number of employees under a manager must be at most the manager's head count. Excluding such states can be achieved by augmenting class models with textual constraints. Such constraints provide a means to reduce the possible states of the models so that they conform to the intended domain semantics.

The standard constraint language for UML class models is the Object Constraint Language (OCL) [49][1]. In this paper, we assume readers to be familiar with OCL and refer to [40] for an introduction. OCL is a textual constraint language based on first-order logic (FOL). For example, the following OCL invariant on the model specifies that the number of employees of each manager must not exceed the manager's headcount.

```
context Manager
inv headCountRestriction:
    self .employs−>size() <= self.headCount
```

A state of a class model is defined in terms of an object model. Informally, the semantics of a class model M is defined as the set of valid states of M. A *valid state* satisfies all constraints defined on the elements of the model. An example state that satisfies the constraint headCountRestriction is shown in Figure 3.
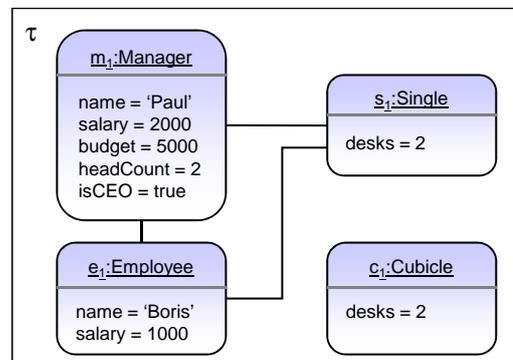

Figure 3 Example state of the company model.

In order to simplify the definition of constraints, the concept of specification patterns [24] has recently been introduced as *constraint patterns* in MDE [1,17,21,47,59]. Constraint patterns are parameterized constraint expressions that can be instantiated to specific constraints.

Definition 1 (Constraint Pattern) *A constraint pattern Π with respect to a meta-model $\mathcal{M}$ is a function that maps a set of meta-model elements to a constraint.*

---

[1] We base our work on [49] as more recent versions of the OCL 2.0 standard fail to provide a consistent semantic, e. g., see [8] for details.

An example of a constraint pattern is the No Cyclic Dependency pattern, which can be used to prohibit cyclic links between objects of a class. It has one parameter property of type Sequence(Property), denoting an OCL navigation path x.y.z in the model. In general, all argument types in pattern definitions are defined in the UML or OCL specification. The definition of the No Cyclic Dependency pattern is based on the closure pattern, which we define as follows.

```
pattern NoCyclicDependency(property: Sequence(Property)) =
  self .closure(property,Set{})−>excludes(self)


pattern closure(property : Sequence(Property),S:Set(Class)) =
  property−>union((property − S)−>
    collect (o:Class | o.closure(property,S−>including(self)))−>
    asSet())
```

By instantiating the No Cyclic Dependency pattern, we can specify the constraint noCycles, which forbids managers to manage themselves.

```
context Manager
inv noCycles: NoCyclicDependency(worksFor)
```

In MDE tools, pattern instantiation can be implemented by adding the required pattern definitions to the context class in the form of OCL operation definitions. To this end, it is sufficient to replace the keyword **pattern** by the OCL keyword **def** and to replace the parameter type Sequence(Property) by the actual type of the corresponding navigation path x.y.z. Consequently, the respective MDE tool can be used to check the syntax and type correctness of the constraint statement.

Even if constraints are correct with respect to syntax and types, they can be inconsistent and thus there is no valid state of the model. This often occurs in industrial-scale models, which possess dozens or even hundreds of constraints. Here model developers can easily lose track of how their constraints interact and inadvertently specify contradictory constraints.

Some inconsistencies are easy to spot and therefore to avoid, for example, contradictory constraints on the value of an attribute. However, other inconsistencies are difficult for model developers to detect because they require a thorough understanding of the semantics of the modeling language and experience in using it. For example, the multiplicity constraint on the worksFor association end requires that every employee has at least one manager. As a consequence, management hierarchies must be cyclic to ensure finitely large and non-empty states. But, if cycles in management hierarchies were undesired (which they are in most companies) and ruled out by the noCycles constraint, it is not obvious whether the model would remain consistent.

1.2 Contributions and Results

Our first contribution in this paper is a survey of consistency notions for UML/OCL models, which provides a uniform representation and comparison of these notions. Based on this comparison, we recommend a consistency notion for real-world MDE development processes. In addition, we use the results from this survey as a foundation for our consistency analysis.

Our second contribution is an automatic polynomial-time analysis approach for constraint specifications based on *constraint patterns*. Whereas constraint patterns have primarily been introduced to simplify and accelerate constraint development, we show how consistency analysis can also benefit from constraint patterns. Efficient, automated consistency analysis is essential for the model developer to receive immediate feedback during model development. However, existing analysis tools scale poorly to large input models because automatically analyzing the consistency of OCL specifications requires exponential complexity.

Our analysis consists of two parts. In the first part, we show how the semantic properties of the patterns can be approximated by syntactic properties and formalize them as consistency lemmas. These lemmas state sufficient conditions under which adding an instance of the respective pattern to the model preserves the model's consistency. As an example, we present one assumption of the consistency lemma for the No Cyclic Dependency pattern. When constraining a path, i. e., a sequence $p_1. \cdots .p_n$ of association ends, with the No Cyclic Dependency pattern, the model remains consistent if single edges can be deleted from instances of this path without violating any multiplicity constraints. In particular, there must be one association end in this path whose lower multiplicity bound is zero and, in addition, there is one association end whose opposite end also has a lower multiplicity bound of zero. If this condition does not hold, there can only be *infinitely long* noncyclic instances of this path, which rules out finitely large states of the respective model.

In the second part of the analysis, we use these lemmas to check the consistency of constrained class models. Here it is sufficient to check for each pattern instance whether the assumptions of the consistency lemma of the respective pattern are satisfied. It will turn out that these assumptions, which are syntactic properties of the model and the pattern instances, can be checked in polynomial time. Hence our analysis is efficient. However, since there is no decision procedure for the consistency of a set of OCL formulas, there is a price to be paid: We sacrifice a complete procedure for the sake of a (quickly) decidable one. As a result, our analysis can produce false positives, i. e., consistent constraints can be displayed as potentially inconsistent.

Our third contribution is to validate that our analysis is applicable to real-world constraints. To this end, we have carried out extensive case studies on industrial-scale models. The results show that our pattern-based analysis approach is applicable to the majority of constraints and has a good runtime performance. Moreover, our approach offers immediate feedback about model consistency to model developers without interrupting their workflow. Thus, we provide an effective means for domain specialists to develop large parts of their constraint specifications in a concise and consistent way, without the need to acquire knowledge of formal specification languages and deduction machinery.

1.3 Organization

This paper is structured as follows. In Section 2, we survey different consistency notions in the literature and recommend a practically relevant notion of consistency for class models. In Section 3, we introduce an efficient approach to checking the consistency of pattern-based constraint specifications. In Section 4, we present an implementation of our analysis for a Computer Aided Software Engineering (CASE) tool. In Section 5, we validate our approach by analyzing the consistency of industrial-scale models that are used in commercial software. In Section 6, we discuss related work and we summarize our findings in Section 7.

2 Consistency Notions for UML/OCL

In MDE, the term *consistency* is used with two different meanings: *inter-model* consistency and *intra-model* consistency. Inter-model consistency with respect to a predicate P denotes that two or more model states satisfy P. For example, an activity model (visualized as an activity diagram) $\mathcal{A}$ and a state model (visualized as a state chart) $\mathcal{S}$ are consistent if the state transitions of the data items in $\mathcal{A}$ are defined in $\mathcal{S}$ [42]. Inter-model consistency is the subject of numerous publications, e. g., [25,35,41,42,54]. In contrast, intra-model consistency of a model M denotes that M can be instantiated, i. e., there exists at least one state for M. This corresponds to consistency in classical logic. When we refer to consistency in this paper, we mean intra-model consistency.

Different notions of consistency for class models have been proposed in the literature, but there are still open questions that require answers. Although there is an intuitive understanding of consistency, not all of the notions given have been precisely defined. Moreover, there is no comprehensive study of the relationships between the notions and their practical relevance. For example, it is unclear which of the different consistency notions are best suited for MDE development projects.

In this section, we survey different consistency notions for UML/OCL models. In doing so, we provide a uniform representation and a comparison of these notions. Before we begin, we review consistency in first-order logic (FOL) to highlight the basic concepts behind consistency and derive a first notion of consistency for UML/OCL. Focusing on the specifics of UML/OCL, we subsequently obtain alternative notions of consistency. Finally, we recommend a consistency notion for real-world MDE development processes.

2.1 Consistency in First-Order Logic

We first set the stage for UML/OCL consistency by reviewing consistency in classical first-order logic (FOL). Our motivation for choosing FOL is twofold: First, we view OCL to be an extension of FOL, adding recursion. Second, most readers will be familiar with FOL, which makes it a suitable starting point from a didactic perspective. In the following, we assume that the reader is familiar with the syntax and terminology of FOL as represented for instance in [27].

To simplify our presentation, we focus on FOL sentences, that is, formulas without free variables. The semantics ϕ_σ of a first-order sentence ϕ is defined as a function $\phi_\sigma : \phi \to \{T, F\}$, where the *state* $\sigma = (\mathcal{U}_\sigma, \mathcal{I}_\sigma)$ is a structure in which $\mathcal{U}_\sigma$ is a set, called the *universe*, and $\mathcal{I}_\sigma$ is a mapping from syntax to semantics, called the *interpretation*. We require that the universe is nonempty, as otherwise universally quantified formulas trivially hold.

A sentence ϕ is called *satisfiable* if there exists a structure σ such that $\phi_\sigma = T$, also denoted by $\sigma \models_{\mathsf{FOL}} \phi$. In this case, σ is called a *valid state* (or a *model*) of ϕ. The notion of satisfiability is related to the notion of *consistency*, which is defined as follows.

Definition 2 (Consistency in FOL) *A set* $\Phi = \{\phi_1, \ldots, \phi_n\}$ *of FOL sentences is* consistent *if and only if there exists a state* σ *such that*

$$\sigma \models_{\mathsf{FOL}} \phi_1 \wedge \ldots \wedge \phi_n \text{ and } \mathcal{U}_\sigma \neq \emptyset,$$

i. e., $\phi_1 \wedge \ldots \wedge \phi_n$ *is satisfiable.* Φ *is* inconsistent *otherwise.*

In contrast to FOL, OCL is a typed logic. Fortunately, FOL can be extended with types. In addition, OCL is a three-valued logic, which entails that all datatypes, including Boolean, are extended with an additional exception element OclUndefined. OCL reasons over an object-oriented data model, that is, OCL formulas can contain path expressions. Thus, we need to adopt and extend the concepts introduced for FOL.

As a prerequisite, we must clarify the semantics of the OCL operation ::allInstances(). Informally, C::allInstances() returns a set containing all instances (i. e., objects) of class C in a given state. The following features of the OCL semantics in general and C::allInstances() in particular are important:

1. OCL considers valid states only [49, Appendix A], i. e., states in which the invariants of all objects evaluate to *true*. Thus, C::allInstances() contains only objects that satisfy their invariant.
2. The set C::allInstances() contains all objects of *kind* C, i. e., instances of class C *and* all subclasses thereof.
3. Following [7], we assume that C::allInstances() may return a set containing infinitely many elements. Operators such as size() are undefined on infinite sets.

Analogous to $\models_{\mathsf{FOL}}$, we introduce the relation $\models_{\mathsf{OCL}}$ between states and OCL formulas. This relation holds for a pair (τ, ϕ) if and only if τ is a state (represented by an object model) that satisfies the formula ϕ, i. e., evaluating the formula ϕ over the object model τ results in true (see [7] for formal definitions). Note that τ in $\models_{\mathsf{OCL}}$ plays the role of a first-order structure in $\models_{\mathsf{FOL}}$.

Now, based on Definition 2 of consistency for FOL and the semantics of UML/OCL as introduced above, we define a first notion of consistency for UML/OCL models. In the following, we write $\exists\_.\_$ for unbounded existential quantification and $\forall\_ \in \_.\_$ for bounded universal quantification.

Definition 3 (Consistency in UML/OCL (i)) *A constrained UML model M is* consistent *if and only if there exists a valid state τ in which there exists an instance of all classes C in M. Formally,*

$$\exists\tau.\ \forall C \in M.\ \tau \models_{\mathsf{OCL}} C\text{::allInstances()}{-}{>}\text{notEmpty()}\ .$$

This definition takes into account that the universe of each type must be nonempty and that all objects satisfy the constraints of their class. However, the type system of UML/OCL is more complicated than the type systems typically used in (sorted) FOL [37]. For example, UML possesses abstract classes, which cannot be instantiated by definition. In the following subsections, we investigate these characteristics of UML/OCL and provide alternative consistency definitions.

2.2 Subtype Consistency and Abstract Classes

The notion of subtype (or *specialization* in UML terminology) is an important concept in object-oriented modeling as it allows model developers to extend and thereby specialize concepts. The invariants specified for a superclass are typically not added explicitly to its subclasses. Instead, subclasses are usually annotated with *additional* invariants that their instances must satisfy, as in the following example.

```
context Employee
  inv positiveSalary : salary > 0

context Manager
  inv positiveBudget: budget > 0
```

These constraints require that the salary of all employees is positive and, in addition, the budget of all managers must be positive for the company model.

In UML, an instance of a class is also considered an instance of each of its superclasses [50, Sect. 7.3.20]. Thus, if C' is a subclass of C, instances of C' need not only satisfy the invariants of C', but also of C and each of its superclasses. This principle is known as *Liskov's substitution principle* [45]. It requires that if ϕ is a property that holds for objects of type T, then ϕ should hold for objects of type S, where S is a subtype of T ($S \leq T$). We call the property that instances of subclasses must satisfy both their own invariants and the invariants of their superclasses *subtype consistency*, which we define as follows.

Definition 4 (Subtype Consistency) *A model M is* subtype consistent *if and only if for all C, $C' \in M$, where C' is a subclass of C, a valid instance of C' is also a valid instance of C. Formally,*

$$\forall C, C' \in M.\ \forall \tau.\ \tau \models_{\mathsf{OCL}} C'\text{::allInstances()}{-}{>}\text{forAll}(x|$$
$$x.\text{oclIsTypeOf}(C')\ \text{and}\ x.\text{oclIsKindOf}(C)$$
$$\text{implies}$$
$$\text{not}\ x.\text{oclAsType}(C).\text{oclIsUndefined()})\ .$$

The above definition expresses the subtype relation between some class C' and its superclass C. If each instance x of some class C' is also an instance of some class C, then it must be possible to cast x to its superclass C.

Subtype consistency ensures that subclasses inherit the invariants from their superclasses. Subtype consistency is also known as *class subsumption* [5] for class models without OCL invariants or *structural subtyping* [2] for models with OCL invariants, which requires that the invariant of a subclass implies the invariant of its superclass. Since subtype consistency is an important requirement for class models, we refine our definition of consistency as follows.

Definition 5 (Consistency in UML/OCL (ii)) *A constrained UML model M is* consistent *if and only if it is subtype consistent and there exists a valid state τ in which there exists an instance for all classes C in M.*

A special case of a subtyping relationship between a superclass C and a subclass C' occurs when C is an *abstract* class. At first glance, any model that contains abstract classes cannot be consistent because, according to the UML specification [50], an abstract class(ifier) "*does not provide a complete declaration and can typically not be instantiated.*" Interfaces are treated similarly in the UML specification: "*Since interfaces are declarations, they are not instantiable.*" Does this mean that any model that contains at least one abstract class is inconsistent by definition?

To clarify the role of abstract classes and interfaces for model consistency, we must investigate their roles in the model. We interpret these roles as follows. Model developers usually create an abstract class or interface C assuming that there will be at least one concrete class C' that specializes (or implements) C, since C would be superfluous otherwise. Thus, if C' can be instantiated, then C can also be instantiated because an instance of a subclass C' is considered an instance of all of its superclasses in UML. Thus, the classical notion of consistency *is* generally applicable to UML with its concepts of interfaces and abstract classes, provided that subtype consistency is additionally required.

Interestingly, subtype consistency distinguishes between classes that cannot be instantiated because they are defined as abstract and those that cannot be instantiated because they have contradictory invariants. Suppose the following invariant is added to the company model.

```
context Employee
inv noEmployee: false
```

The invariant noEmployee excludes valid instances of Employee because no object can satisfy this invariant. This invariant therefore renders the model inconsistent. In contrast, if Employee was defined as an abstract class without the noEmployee invariant, the model would remain consistent because there is at least one subclass of Employee (Manager, which is also an instance of Employee according to the UML standard [50]) that can be instantiated. Thus, we can specify a state containing one object o of type Manager, for

which o.oclIsKindOf(Employee) holds. For this state, it holds that Employee::allInstances()−>notEmpty() and the model is therefore consistent according to our definition.

2.3 Finitely Large Model States

In Definition 5, we defined a model M as consistent if there is a state of M that contains at least one object for each class of M. According to the classical notion of consistency, such a state can be infinitely large. Moreover, it is possible that *only* infinitely large states satisfy all the constraints of a given model. Since states with infinitely many objects are rarely desirable in practice, we discuss this problem in this subsection.

The requirement for models to have finitely large states is well-known in database theory [10] and has recently been explored in the context of UML/OCL [13,46,53]. Consider the following constraint, which forbids cycles in the management hierarchy of a company, i. e., managers must not be their direct or indirect (i. e., via transitivity) manager.

```
context Manager
inv noCycles: NoCyclicDependency(worksFor)
```

This constraint is an instance of the previously defined pattern No Cyclic Dependency. Since the multiplicity constraints in the UML model require that each employee is associated with at least one manager, noCycles can only be satisfied by a state in which there are either no employees at all or infinitely many managers. However, states with no employees violate consistency and states with infinitely many managers are undesirable.

In this situation, the model developer has two choices: either drop the constraint noCycles or change the model to make it consistent. Suppose the developer changes the multiplicity of the worksFor association end from 1..∗ to ∗ and leaves the rest of the company model as is. This results in the model company2, displayed in part in Figure 4, which has finitely large states. We show one of these states in Figure 5, named τ.
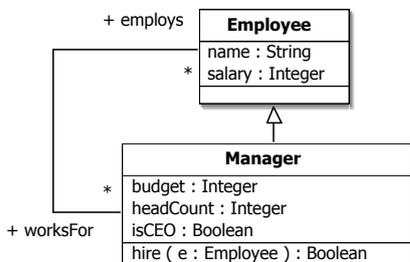

Figure 4 Extract of the modified company model.

The updated model company2 has a valid state with a finite number of objects that satisfies the constraint noCycles. In general, we can formalize the requirement that a model M must have at least one finitely large state as follows.
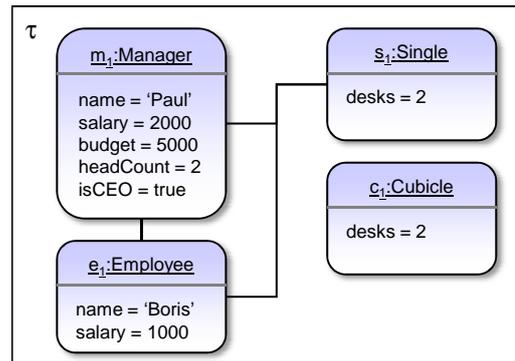

Figure 5 Valid state of the company2 model.

Definition 6 (Finitely large states) *A model M has at least one finitely large state if and only if for all classes C of M, there exists a valid state τ and a natural number $n > 0$ such that*

$$\tau \models_{\text{OCL}} C\text{::allInstances}()\text{−>size}() \leq n\,.$$

Having investigated the characteristic features of UML/OCL, we can check whether our company model is consistent. It will turn out that the classical notion of consistency is too strong for the company model. In the following subsection, we investigate the reason for this and incrementally weaken the classical notion of consistency in order to obtain more fine-grained notions of consistency for UML/OCL.

2.4 Weaker Notions of Consistency

We have seen in the previous subsection that the updated model company2 is consistent and has at least one state with a finite number of objects. However, it is not always possible to find a single state in which all classes of a given model can be instantiated. Consider the following invariant, which allows companies to have either offices of type Single or of type Cubicle, but not both.

```
context Office
inv workConditions:
    Single :: allInstances()−>isEmpty() or
    Cubicle :: allInstances()−>isEmpty()
```

The company2 model with the constraint workConditions is inconsistent with respect to the classical notion of consistency because no instance of the model exists in which all classes are instantiated *and* the workConditions constraint is satisfied. However, the model developer may have deliberately created this scenario because it reflects company policy. In contrast, the requirement from the classical consistency notion that each class must be instantiated (or, in our scenario, that both types of office are instantiated) does not conform to the company policy. Therefore, the classical notion of consistency is not applicable in this scenario because it is too strong.

Another example that illustrates that the classical notion of consistency can be too strong in practice is the following. In early phases of system development, there may be an abstract class C for which no concrete subclasses have been defined yet. As a consequence, there is no state that contains objects of type C and thus, the model is inconsistent. However, this contradicts the intention of the model developer for whom the presence of C is desired. To reconcile the developer's intention with a formal notion of consistency, we must weaken the classical notion of consistency. To this end, we investigate which notions of consistency a model developer may be concerned about.

Can each class be instantiated in the same state? As shown, if a set of constraints is consistent in the classical sense, there exists at least one state in which all classes can be instantiated. This is a strong requirement, but it can be useful for UML/OCL models with strong dependencies between the classes in the model, i. e., an instance of a class can only exist if instances of all other classes also exist.

The classical notion of consistency is presented in [38, 46]. We refer to the classical notion of consistency as *strong consistency*, which we define as follows.

Definition 7 (Strong Consistency) *A UML/OCL model M is* strongly consistent *if and only if M is subtype consistent and there exists a valid state in which all classes of M are instantiated. Formally,*

$$\exists\tau.\ \forall C \in M.\ \tau \models_{\mathsf{OCL}} C\text{::allInstances()}{-}{>}\text{notEmpty()}\,.$$

We have seen that the model company2 is strongly consistent because there exists a state with the required properties (cf. Figure 5).

Can each class be instantiated in some state? Since strong consistency can be too strong for certain desired constraints, we follow [5, 46, 52] and weaken the previous requirement by asking for the existence of a *set of states* for M such that each class in M is instantiated in at least one of these states. We call this weaker notion of consistency *class consistency* and define it as follows.

Definition 8 (Class Consistency) *A UML/OCL model M is* class consistent *if and only if M is subtype consistent and for each class $C \in M$, there exists a valid state that contains an instance of class C. Formally,*

$$\forall C \in M.\ \exists\tau.\ \tau \models_{\mathsf{OCL}} C\text{::allInstances()}{-}{>}\text{notEmpty()}\,.$$

We have seen that the company model is not strongly consistent if it is annotated with the constraint workConditions. However, the model is class consistent and we show witness model states in Figure 6. In this figure, the constraint workConditions is satisfied by both the state τ_1, which contains objects of type Single only, and by the state τ_2, which contains objects of type Cubicle only. Thus, each class of the company model can be instantiated, although in different states. Every strongly consistent model is also class consistent, but not vice versa.

Can any class of the model be instantiated? In the early phases of system development, some classes in a model may not yet be implemented. It can also happen that some classes are still in the model, but using them is discouraged or they no longer have an implementation.

According to the previously defined notions of consistency, such models are neither strongly consistent nor class consistent because not all classes can be instantiated. However, as explained above, such a situation may be desired. Thus, the developer may want to know whether there is a nonempty subset of all classes in M that can be instantiated. Such a weak notion of consistency is used in [5, 46, 52], which we define as follows.

Definition 9 (Weak Consistency) *A UML/OCL model M is* weakly consistent *if and only if M is subtype consistent and a nonempty subset of the classes of M can be instantiated in some valid state. Formally,*

$$\exists\tau.\ \exists C \in M.\ \tau \models_{\mathsf{OCL}} C\text{::allInstances()}{-}{>}\text{notEmpty()}\,.$$

Every class consistent model is also weakly consistent, but not vice versa. Given a UML/OCL model that is weakly consistent, the model developer can incrementally modify the model to achieve a stronger notion of consistency.

The weakest notion of all is inconsistency. In an inconsistent class model, not a single class can be instantiated in any state. This, of course, is undesirable in practice.

Definition 10 (Inconsistency) *A UML/OCL model M is* inconsistent *if and only if it is not weakly consistent.*

As mentioned before, strongly consistent models and class consistent models are also weakly consistent and therefore, they are *not* inconsistent.

2.5 Practically Relevant Consistency Notions

We have investigated different notions of consistency based on the characteristic features of UML/OCL. We now assess which notions are most relevant for model development.

We believe that UML/OCL models in MDE should be subtype consistent and have at least one finitely large state. We consider subtype consistency a necessary requirement for object-oriented models [45], and if a model is not subtype consistent, it should be revised. In certain scenarios, such as code re-use, subtype consistency can be ignored. We also propose that models should have finitely large states when used in MDE processes.

At the end of the development process, when code is generated from the models, we propose that class models should be class consistent because every class should be instantiable in at least one system state or be removed otherwise. Earlier in the process, the models may be weakly consistent, but they should be refined such that they are eventually at least class consistent. To this end, the next section covers consistency analysis, that is, a method for determining whether a model is consistent.
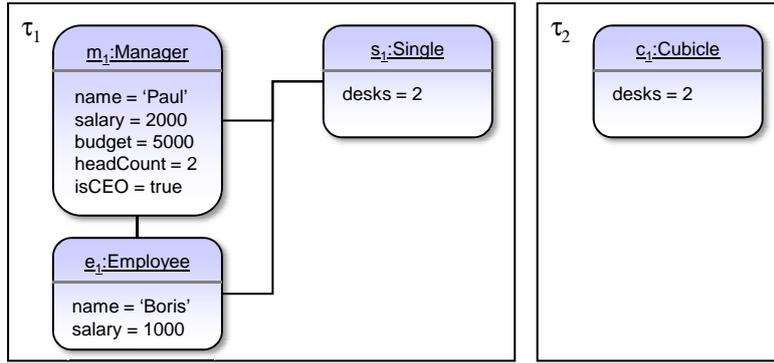
Figure 6 Witnesses for the class consistency of company2.

3 Pattern-Based Consistency Analysis

In this section, we introduce a novel approach to the consistency analysis of pattern-based constraint specifications. Our approach consists of proving general consistency properties of the constraint patterns *once* and subsequently using these properties for an efficient, automatic, heuristic analysis of pattern instances. The basis for our analysis is the previously defined consistency notions.

We make the following assumptions. First, we assume that graphical constraints on the class model, e.g., cardinality constraints, are removed from the model and specified as textual constraints as presented in [30]. We further assume that these textual constraints are represented using our library of constraint patterns. Finally, we assume that there are only binary associations in the UML class model and no association ends that are derived unions. Note that we make this restriction only to keep this paper concise. Our approach can be generalized to n-ary associations with $n > 2$.

3.1 Introducing Consistency Lemmas

We capture the general consistency properties of a constraint pattern Π as a set of assumptions under which adding an instance of Π to a constraint specification preserves the specification's consistency. This analysis must be done just once for each constraint pattern in a given pattern library. As a result, we formulate and prove a *consistency lemma* for each pattern. We define consistency lemmas in Definition 11. In this definition, we use a general notion of consistency that must be replaced by a specific notion as defined in the previous section.

Definition 11 (Consistency Lemma) *A consistency lemma for a constraint pattern Π states sufficient conditions for the consistency of instances of Π. It has the following structure:*

- *Let $\langle M, \Phi_M \rangle$ be consistent, where M is a class model and Φ_M is its constraint specification. Let ψ be an instance of the pattern Π.*
- *Assume the syntactic properties $P_1(M, \Phi_M)$, ..., $P_n(M, \Phi_M)$ hold for the model and its constraint specification.*

- *Then $\langle M, \Phi_M \cup \{\psi\} \rangle$ is consistent.*

When the patterns are used, it is sufficient to check whether the assumptions P_1, ..., P_n hold. Since these assumptions are syntactic properties of the UML/OCL model $\langle M, \Phi_M \rangle$, they can be checked in polynomial time. In our approach, we first must prove that the initial class model without any textual constraints is consistent, which is a decidable property that can be computed in linear time [46].

In the following, we analyze our pattern library for potential contradictions. The main challenge is identifying which patterns can potentially contradict a given pattern Π. To this end, we classify constraint patterns into three kinds: (1) those that constrain the value of attributes, e.g., Unique Identifier, (2) those that constrain the structure of the object graph spanned by objects and links between them, e.g., Surjective Association, and (3) those that constrain both an attribute value and the object graph. The only pattern of the third kind is the Multiplicity Restriction pattern, which relates the multiplicity of an association to an attribute value.

We use the No Cyclic Dependency pattern as an example to illustrate a consistency lemma. The following lemma defines sufficient assumptions under which a constrained model remains strongly consistent after instantiating the pattern. We use strong consistency in the consistency lemmas in this paper because the proofs for this notion of consistency are more compact and illustrative. However, the lemmas can be easily adapted to the other notions of consistency. Following the proof sketch for the example lemma below, we provide a variant for class consistency. In our lemmas and proofs, we use the notation p^{-1} to denote the opposite association end of an association end p in a binary association.

Lemma 1 (Strong Consistency of No Cyclic Dependency) *Let M be a model and Φ_M be a constraint specification based on patterns. Let $\langle M, \Phi_M \rangle$ be strongly consistent and have at least one finitely large state, and let ϕ be an instance of the No Cyclic Dependency pattern with class C as context and navigation path $p_1.p_2. \cdots .p_n$. If there are $j, k \in \{1, \ldots, n\}$ such that*

(i) the lower multiplicity bound of p_j and p_k^{-1} in M is zero,

(ii) there is no instance of the Surjective Association or Bijective Association pattern in Φ_M with parameter value p_k,

(iii) there is no instance of the Multiplicity Restriction pattern in Φ_M with p_j and p_k^{-1} as values for any parameter, and
(iv) there is no instance of the Object In Collection pattern in Φ_M on any element of the path $p_1.p_2. \cdots .p_n$,
then $\langle M, \Phi_M \cup \{\phi\} \rangle$ is strongly consistent and has at least one finitely large state.

In the following, we explain the assumptions of this lemma and we give the proof in the subsequent section. Because of the requirement that the initial model is strongly consistent, there is a state τ in which each class of M is instantiated. If τ does not satisfy the new constraint ϕ, we construct a state τ' from τ as follows. First, we delete the jth link from the path. Now, $\tau' \models_{\mathsf{OCL}} \phi$, but τ' may not satisfy the multiplicity constraints in Φ_M. Thus, we walk along the path backwards starting from position j and instantiate and link new objects until the multiplicity constraints hold. This will eventually be the case because of the existence of p_k and its assumed properties. Thus, $\tau' \models_{\mathsf{OCL}} \Phi_M$ and M and $\langle M, \Phi_M \cup \{\phi\} \rangle$ is strongly consistent and has at least one finitely large state.

The assumptions (i)-(iv) are crucial for the correctness of above construction. Assumption (i) requires the existence of an association end p_j on the constrained path that has a lower multiplicity bound of zero, and there must be an association end p_k whose opposite association end also has a lower multiplicity bound of zero. Generally speaking, this assumption allows us to delete existing links in the above construction and ensures its termination. However, it is not sufficient that the lower multiplicity bound of these association ends is zero. In addition, there must be no constraints in the constraint specification Φ_M of the model M that constrain the lower multiplicity bound of these association ends. There are three constraint patterns in our library that can be used to constrain lower multiplicity bounds: Surjective Association, Bijective Association, and Multiplicity Restriction. Therefore, we added assumptions (ii) and (iii) to the consistency lemma. In addition, we forbid that the reflexive path is constrained by an instance of the Object In Collection pattern (iv), which would otherwise affect these association ends.

In the presence of constraint-pattern instances and the absence of general OCL constraints, the consistency lemmas implicitly guarantee subtype consistency because *all* other pattern instances are examined for contradictory statements. This includes all pattern instances defined on superclasses of the analyzed classes. For example, assumption (iii) of Lemma 1 requires the absence of instances of the Multiplicity Restriction pattern on the respective association ends. This includes the case that a contradictory instance of the Multiplicity Restriction pattern is specified for a superclass of the class for which an instance of the No Cyclic Dependency pattern is specified.

To support analysis for different kinds of consistency, we must establish additional consistency lemmas. For example, we show the consistency lemma for the No Cyclic Dependency pattern for class consistency. This lemma is almost identical to the lemma for strong consistency, with the exception that the term "strongly consistent" is replaced by "class consistent".

Lemma 2 (Class Consistency of No Cyclic Dependency)
Let M be a model and Φ_M be a class consistent constraint specification based on patterns that has at least one finitely large state. Let ϕ be an instance of the No Cyclic Dependency pattern with class C as context and navigation path $p_1.p_2. \cdots .p_n$. If there are $j, k \in \{1, \ldots, n\}$ such that
(i) [. . . same as in Lemma 1 . . .],
then $\langle M, \Phi_M \cup \{\phi\} \rangle$ is class consistent and has at least one finitely large state.

Whereas the lemma statement for class consistency is almost identical to that of strong consistency, the proof of class consistency requires some changes. Since our basic assumption is now the *class consistency* of the constraint specification, there exists a *set of states* $T = \{\tau_1, \ldots, \tau_n\}$ in which each class of M is instantiated. For each $\tau_i \in T$ that does not satisfy ϕ, we apply the construction from the proof for strong consistency for τ_i. We thereby obtain a set of states T' in which each τ_i satisfies both Φ_M and ϕ. Thus, T' is a witness for the class consistency of $\langle M, \Phi_M \cup \{\phi\} \rangle$.

Note that we are interested in *sufficient* conditions for pattern instances to preserve the consistency of the model. Thus, the lemmas typically have the form $\mathcal{P} \Rightarrow Q$, where $\mathcal{P}$ is a set of syntactic assumptions and Q states that the refined model is consistent. In general, it is not possible to have consistency lemmas of the form $\mathcal{P} \Leftrightarrow Q$ because the consistency of Q is, in general, undecidable, whereas the conditions in $\mathcal{P}$ can be checked in polynomial time.

3.2 Library of Consistency Lemmas

In this section, we present the library of constraint patterns previously defined in [56] and investigate under which assumptions instances of these constraint patterns preserve model consistency. Starting from a model M and a strongly consistent constraint specification Φ_M, we add instances of our constraint patterns and analyze potential conflicts. We assume that all constraints in the constraint specification are pattern instances and no literal OCL expressions are present. An overview of the patterns from [56] is shown in Figure 7.

We use the example model in Figure 8, which represents a simple class model with classes, attributes, associations, and generalization, to illustrate our findings. Needless to say, our consistency lemmas make statements about *arbitrary* class models.

3.2.1 Association-Constraining Patterns. In this subsection, we introduce the association-constraining patterns shown in Figure 7. For each pattern, we provide its definition, a consistency lemma, its proof, and an example of inconsistent pattern instances.
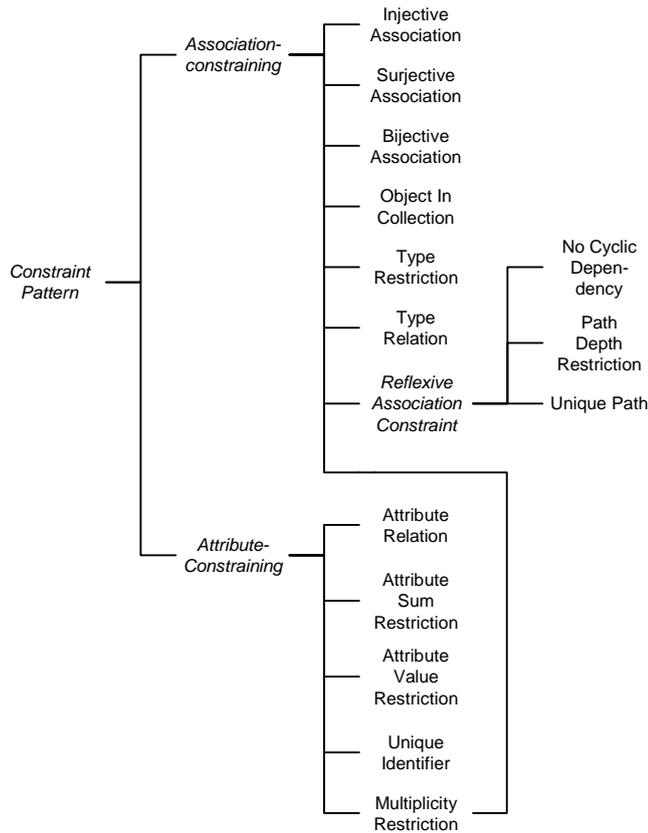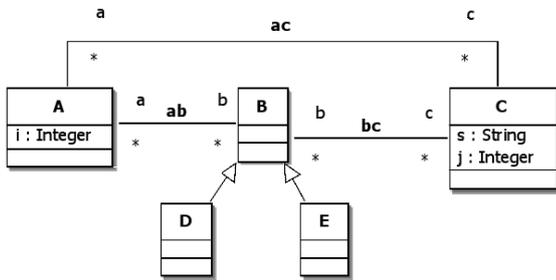
Figure 7 Library of constraint patterns.

Figure 8 Generic class diagram.

Injective Association. The Injective Association pattern can be instantiated to make an association end injective.

```
pattern InjectiveAssociation (property:Sequence(Property)) =
  self .property−>size() = 1 and
  self .class :: allInstances()−>forAll  (x,y  |  x.property = y.property
                                             implies x=y)
```

Note that the shorthand .class is replaced with the type name of the actual class with which the pattern is instantiated by the pattern instantiation mechanism. We define the consistency lemma for the Injective Association pattern as follows.

Lemma 3 (Consistency of Injective Association) *Let* $\langle M, \Phi_M \rangle$ *be a strongly consistent model that has at least one finitely large state. Let* ϕ *be an instance of the Injective Association pattern with class* C *as context and navigation path* $P = p_1.p_2.\ \cdots\ .p_n.$ *If*

(i) the upper multiplicity bound of the opposite association end of each $p_i \in \{p_1, \ldots, p_n\}$ *is either one or * and*

(ii) there is no instance of the Multiplicity Restriction pattern in Φ_M *that constrains the opposite association end of any* $p_i \in \{p_1, \ldots, p_n\},$

then $\langle M, \Phi_M \cup \{\phi\} \rangle$ *is strongly consistent.*

Proof (for Lemma 3) Because $\langle M, \Phi_M \rangle$ is strongly consistent, there exists a state τ in which each class of M is instantiated. Based on τ, we construct a state τ' that witnesses the strong consistency of $\langle M, \Phi_M \cup \{\phi\} \rangle$. In the following, we do not consider the trivial case and thus assume $\tau \not\models_{\text{OCL}} \phi$.

In this case, there are two or more objects of class C in τ that are connected to the same object of class $type(p_n)$ as illustrated in Figure 9. We construct τ' from τ as follows. For every object o_n of class $type(p_n)$, we walk the path P backwards. For all $i, 1 \leq i < n$, if there is more than one link from o_i to o_{i+1}, we arbitrarily select and delete all but one link. This preserves the invariants of the classes on the "right-hand side" of the link, because exactly one link will be left and the multiplicity of this association is either one or unlimited (*) (i) and not further constrained (ii). After this construction, there is at most one link from an object of class C to an object of class $type(p_n)$ and thus, $\tau' \models_{\text{OCL}} \phi$.

However, the multiplicities of the classes on the "left-hand side" of the deleted links may have been violated by the previous construction. We repair the multiplicities as follows. For each object of class C, we walk the path P from $1 \leq i < n$. If the multiplicity invariants of class $type(p_i)$ are violated, we create objects of class $type(p_{i+1})$ until the invariants are satisfied. Since each newly created object o_i is connected to exactly one object o_{i-1}, $\tau' \models_{\text{OCL}} \phi$ still holds. After this second part of the construction, all the multiplicity constraints hold and thus, $\tau' \models_{\text{OCL}} \Phi_M$. Because $\tau' \models_{\text{OCL}} \phi$ also holds, $\langle M, \Phi_M \cup \{\phi\} \rangle$ is strongly consistent. $\quad\square$
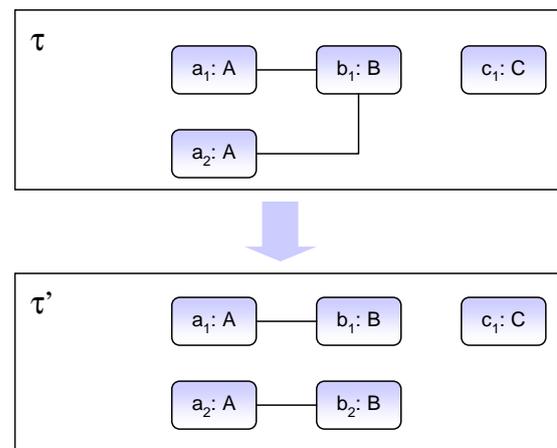

Figure 9 Illustration of the proof for Lemma 3.

The following is an example of an *inconsistent* constraint.

```
context A
    inv:  InjectiveAssociation (b)
context B
    inv:   MultiplicityRestriction  (a,>,1)
```

Whereas the first invariant states that no object of class *B* may be connected to more than one object of class *A*, the second invariant states the exact opposite.

Surjective Association.   The Surjective Association pattern can be instantiated to make an association end surjective.

```
pattern SurjectiveAssociation(property:Sequence(Property)) =
    self .property.class :: allInstances()−>forAll  ( y |
        self .class :: allInstances()−>exists( x |
            x.property−>includes(y)
            and x.property−>size()=1))
```

We define the consistency lemma for the Surjective Association pattern as follows.

Lemma 4 (Consistency of Surjective Association) *Let $\langle M, \Phi_M \rangle$ be a strongly consistent model that has at least one finitely large state. Let ϕ be an instance of the Surjective Association pattern with class C as context and navigation path $P = p_1.p_2. \cdots .p_n$. If*
  (i) for all i, $1 \leq i \leq n$, the upper multiplicity bound of property p_i^{-1} is greater than zero,
  (ii) there is no instance of the Multiplicity Restriction pattern in Φ_M that constrains any opposite association end p_i^{-1} for all $i, 1 \leq i \leq n$, and
  (iii) there is no instance of the No Cyclic Dependency pattern in Φ_M with the navigation path P,
then $\langle M, \Phi_M \cup \{\phi\} \rangle$ is strongly consistent.

Proof (for Lemma 4) Because $\langle M, \Phi_M \rangle$ is strongly consistent, there exists a state τ in which each class of M is instantiated. Based on τ, we construct a state τ' that witnesses the strong consistency of $\langle M, \Phi_M \cup \{\phi\} \rangle$. In the following, we do not consider the trivial case and thus assume $\tau \not\models_{\mathsf{OCL}} \phi$.

In this case, there is an object y of class $type(p_n)$ in τ that is not linked to an object of class C via the path P. We establish a link between an object of class C and y as follows. From y, we navigate for all i, $1 \leq i < n$, backwards along the path P. For each part i of the path for which there does not exist a link, we either create an instance of p_i between object o_{i+1} and an existing object o_i if the multiplicity constraints of class $type(o_i)$ allow o_i be connected to o_i or, otherwise, we create a new object o'_i of class $type(o_i)$. Creating such a link is possible because objects of class $type(o_{i+1})$ can be connected to objects of $type(o_i)$ since the multiplicity of association end p_i^{-1} is at least one (i) and not further constrained by an Multiplicity Restriction (ii). Since we try to connect to an existing object, a cycle can be introduced, but because cycles on this part are not forbidden (iii), this construction does not violate Φ_M. This construction terminates after the nth step connecting an object of type C to a path that leads to y. Therefore, $\tau' \models_{\mathsf{OCL}} \phi$, and since this construction has not violated any constraint in Φ_M, $\langle M, \Phi_M \cup \{\phi\} \rangle$ is strongly consistent.   □
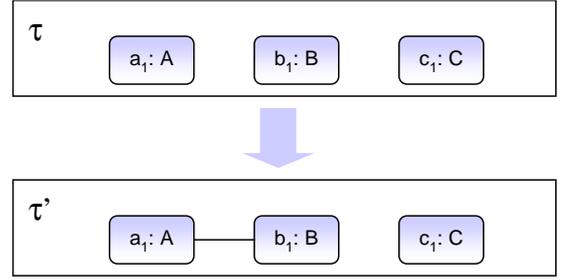

Figure 10 Illustration of the proof for Lemma 4.

The following is an example of an *inconsistent* constraint.

```
context A
    inv:  SurjectiveAssociation(b)

context B
    inv:   MultiplicityRestriction  (a,<,1)
```

Whereas the first invariant requires every object of class *B* to be connected to an object of class *A*, the second invariant forbids this by stating that no objects of class *A* may be connected to objects of class *B*.

Bijective Association.   The Bijective Association pattern can be instantiated to make an association bijective.

```
pattern BijectiveAssociation (property:Sequence(Property)) =
    InjectiveAssociation (property)  and
    SurjectiveAssociation(property)
```

We define the consistency lemma for the Bijective Association pattern as follows.

Lemma 5 (Consistency of Bijective Association) *Let $\langle M, \Phi_M \rangle$ be a strongly consistent model that has at least one finitely large state. Let ϕ be an instance of the Bijective Association pattern with class C as context and navigation path $P = p_1.p_2. \cdots .p_n$. If*
  (i) for all i, $1 \leq i \leq n$, the upper multiplicity bound of property p_i^{-1} is greater than zero,
  (ii) there is no instance of the Multiplicity Restriction pattern in Φ_M that constrains any opposite association end p_i^{-1} for all i, $1 \leq i \leq n$,
  (iii) there is no instance of the No Cyclic Dependency pattern in Φ_M with the navigation path P,
  *(iv) the upper multiplicity bound of the opposite association end of each $p_i \in \{p_1, \ldots, p_n\}$ is either one or *, and*
  (v) there is no instance of the Multiplicity Restriction pattern in Φ_M that constrains the opposite association end of any $p_i \in \{p_1, \ldots, p_n\}$,
then $\langle M, \Phi_M \cup \{\phi\} \rangle$ is strongly consistent.

Note that assumptions (i)-(iii) are the assumptions from the Surjective Association pattern and assumptions (iv) and (v) are the assumptions from the Injective Association pattern.

Proof (for Lemma 5) Because $\langle M, \Phi_M \rangle$ is strongly consistent, there exists a state τ in which each class of M is instantiated. Based on τ, we construct a state τ' that witnesses the

strong consistency of $\langle M, \Phi_M \cup \{\phi\}\rangle$. In the following, we do not consider the trivial case and thus assume $\tau \not\models_{\mathsf{OCL}} \phi$.

We construct a state τ' from the state τ by making the navigation path P both surjective and injective. Making P surjective is possible because of the assumptions (i)-(iii) (cf. Lemma 4) and (iv)-(v) (cf. Lemma 3). After this construction, P is both surjective and injective. Thus, P is bijective and $\langle M, \Phi_M \cup \{\phi\}\rangle$ is strongly consistent. $\quad\square$

The following is an example of an *inconsistent* constraint.

```
context A
    inv:  BijectiveAssociation (b)
context B
    inv:   MultiplicityRestriction  (a,>,1)
```

These invariants are inconsistent because the first invariant specifies a one-to-one relation between objects of classes *A* and *B* whereas the second invariant imposes a one-to-many relation between objects of *B* and objects of *A*.

Object In Collection.   The Object In Collection pattern can be instantiated to require objects of a class to be contained in a set of related elements.

```
pattern ObjectInCollection(set:Sequence(Property),
                           element:Sequence(Property)) =
    self .set−>includesAll(self.element)
```

In Lemma 1, we stated dependencies between this pattern and the No Cyclic Dependency pattern. In addition, there are further dependencies, as stated in the following lemma.

Lemma 6 (Consistency of Object In Collection) *Let* $\langle M, \Phi_M \rangle$ *be a strongly consistent model that has at least one finitely large state. Let* ϕ *be an instance of the Object In Collection pattern with context class* C, *set* $= p_1.p_2.\ \cdots\ .p_m$, *and element* $= p_1.p_2.\ \cdots\ .p_n$. *If*
  (i) there is no instance of the No Cyclic Dependency pattern in Φ_M *with parameter property* = *set and*
  (ii) the upper multiplicity bound of p_m *is at least one, and*
  (iii) there is no instance of the Multiplicity Restriction pattern in Φ_M *with any* $p_i \in \{p_1, \ldots, p_m\}$ *as a value for the navigation parameter,*
then $\langle M, \Phi_M \cup \{\phi\}\rangle$ *is strongly consistent.*

Proof (for Lemma 6) Because $\langle M, \Phi_M \rangle$ is strongly consistent, there exists a state τ in which each class of M is instantiated. Based on τ, we construct a state τ' that witnesses the strong consistency of $\langle M, \Phi_M \cup \{\phi\}\rangle$. In the following, we do not consider the trivial case and thus assume $\tau \not\models_{\mathsf{OCL}} \phi$.

For each object o_0 of type C for which the instance of the Object In Collection pattern does not hold, we perform the following construction, illustrated in Figure 11. Starting from o_0, we navigate along the path $p_1.p_2.\ \cdots\ .p_m$. This path ends at an object o_k because of $\tau \not\models_{\mathsf{OCL}} \phi$. For all i, where $i > k$ and $i < m$, create an object o_i of class $type(p_i)$ and link it to the previous object. This does not violate any constraints because at least one relation between these objects can exist because their multiplicities are not constrained (iii).

Connect the last object of $type(p_{m-1})$ to o_0. This does not violate any constraints because arbitrary many objects of type C can be connected to objects of type $type(p_{m-1})$ because the multiplicity of p_m is at least one (ii). The last step creates a cyclic link between o_0 and itself, which does not violate any constraint because cycles are not forbidden (i). Due to this cycle, $\tau' \models_{\mathsf{OCL}} \phi$ and thus $\langle M, \Phi_M \cup \{\phi\}\rangle$ is strongly consistent. $\quad\square$
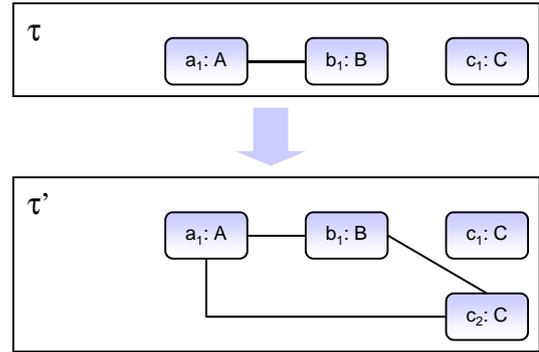

Figure 11 Illustration of the proof for Lemma 6.

The following is an example of an *inconsistent* constraint.

```
context A
inv: ObjectInCollection(Sequence{}, b.c.a)
inv: NoCyclicDependency(b.c.a)
```

These invariants are inconsistent because the first invariant requires each object of class *A* to be in the set of objects reachable via path *b.c.a*, which implies a cycle. However, the second invariant explicitly forbids such cycles.

Type Restriction.   The Type Restriction pattern can be used to constrain an association that is defined between a class and a superclass by limiting the allowed subclasses.

```
pattern TypeRestriction(property:Property,
                        allowedClasses:Set(Class)) =
    self .property−>forAll(x |  allowedClasses−>exists(t |
      x.oclIsTypeOf(t )))
```

We define the consistency lemma for the Type Restriction pattern as follows.

Lemma 7 (Consistency of Type Restriction) *Let* $\langle M, \Phi_M \rangle$ *be a strongly consistent model that has at least one finitely large state. Let* ϕ *be an instance of the Type Restriction pattern with class* C *as context, a navigation path* $P = p_1.p_2.\ \cdots\ .p_n$ *and a set* $S = \{C_1, \ldots, C_n\}$ *of allowed classes. If*
  (i) for all i, $1 \le i \le n$, *the lower multiplicity bound of property* p_i^{-1} *is zero,*
  (ii) there is no instance of the Multiplicity Restriction pattern in Φ_M *that constrains any of the opposite association end* p_i^{-1} *for all* i, $1 \le i \le n$,

(iii) there is no instance $\psi(p', S')$ of the Type Relation pattern in Φ_M where p' is a suffix of p and $S' \backslash S \neq \emptyset$, and

(iv) there is no other instance $\psi(p', S')$ of the Type Restriction pattern in Φ_M where p' is a suffix of p and $S' \cap S \neq \emptyset$,

then $\langle M, \Phi_M \cup \{\phi\} \rangle$ is strongly consistent.

Proof (for Lemma 7) Because $\langle M, \Phi_M \rangle$ is strongly consistent, there exists a state τ in which each class of M is instantiated. Based on τ, we construct a state τ' that witnesses the strong consistency of $\langle M, \Phi_M \cup \{\phi\} \rangle$. In the following, we do not consider the trivial case and thus $\tau \not\models_{\mathsf{OCL}} \phi$.

In this case, there exists a path between an object o_1 of class C to an object o_n, where $class(o_n) \notin \{C_1, \ldots, C_n\}$. We construct τ' by deleting all links between objects of class $type(p_n^{-1})$ and o_n as shown in Figure 12. This does not violate the multiplicity constraints of $class(o_n)$ because this class can be related to zero objects of class $type(p_n^{-1})$ because the lower multiplicity bound of p_n^{-1} is zero (i) and not further constrained (ii). The deletion of the link does not violate any Type Relation constraint because no object of $class(o_n)$ is required to be on path P (iii). Now, $\tau' \models_{\mathsf{OCL}} \phi$.

If the multiplicity constraints of class $type(p_n^{-1})$ are violated, we create objects of any allowed class $C_i \in \{C_1, \ldots, C_n\}$ until the multiplicity constraints of class $type(p_n^{-1})$ are satisfied. Furthermore, the newly created objects do not violate any Type Restriction constraint in Φ_M because there is no other type restriction that requires any class not in $\{C_1, \ldots, C_n\}$ (iv). Now, also $\tau' \models_{\mathsf{OCL}} \Phi_M$ holds and thus $\langle M, \Phi_M \cup \{\phi\} \rangle$ is strongly consistent. □
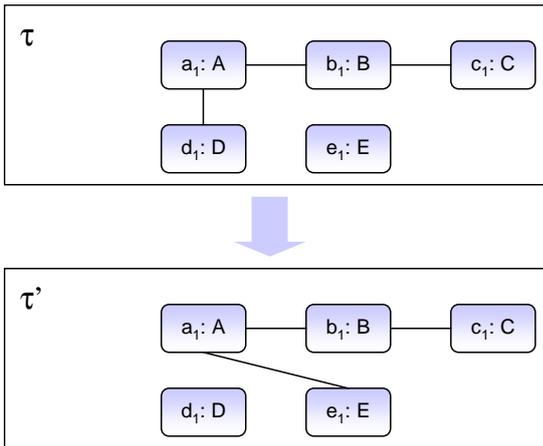

Figure 12 Illustration of the proof for Lemma 7.

The following is an example of an *inconsistent* constraint.

```
context A
   inv: TypeRestriction(b,E)
context D
   inv:  MultiplicityRestriction (a,>,0)
```

These invariants applied to the model in Figure 8 are not strongly consistent because no instance of D can be created. In particular, the first invariant requires that only objects of the subclass E of B may be connected to objects of class A on the association end b. However, the second invariant requires that objects of class D, the other subclass of B, must be connected to at least one object of class A, which contradicts the first invariant.

Type Relation. The Type Relation pattern can be used to enforce that instances of certain subclasses $C_1, \ldots, C_n$ of C_0, the requiredClasses, must participate in some relation.

```
pattern TypeRelation(property:Sequence(Property),
                     requiredClasses:Set(Class)) =
  requiredClasses−>forAll(c | self .property−>exists(p |
    p.oclIsTypeOf(p)))
```

We define the consistency lemma for the Type Relation pattern as follows.

Lemma 8 (Consistency of Type Relation) *Let $\langle M, \Phi_M \rangle$ be a strongly consistent model that has at least one finitely large state. Let ϕ be an instance of the Type Relation pattern with class C as context, a navigation path $P = p_1.p_2. \cdots .p_n$ and a set $S = \{C_1, \ldots, C_n\}$ of required classes. If*

(i) there exists a $p_i \in \{p_1, \ldots, p_n\}$ for which the upper multiplicity is greater than or equal to $|S|$,

(ii) there is no instance of the Multiplicity Restriction pattern in Φ_M that constrains the above-mentioned association end p_i, and

(iii) there is no instance $\psi(p', S')$ of the Type Restriction pattern in Φ_M, where p' is a suffix of p and $S' \backslash S \neq \emptyset$,

then $\langle M, \Phi_M \cup \{\phi\} \rangle$ is strongly consistent.

Proof (for Lemma 8) Because $\langle M, \Phi_M \rangle$ is strongly consistent, there exists a state τ in which each class of M is instantiated. Based on τ, we construct a state τ' that witnesses the strong consistency of $\langle M, \Phi_M \cup \{\phi\} \rangle$. In the following, we do not consider the trivial case and thus, $\tau \not\models_{\mathsf{OCL}} \phi$.

In this case, there is an object $o_1 : C$ that is not linked to an object of class $C_j \in S$. We walk the path P from o_1 to p_i. At p_i, we create a new object o_i' of class $type(p_i)$ and link it to the previous object in the path as illustrated in Figure 13. After this construction, $\tau' \models_{\mathsf{OCL}} \Phi_M$ still holds because the unlimited multiplicity of p_i (i), which is not further constrained (ii), allows one to connect an unlimited number of elements to the previous objects in the path.

From o_i', we continue to walk the path P, creating a new object in each step. The last object in the path must be of type C_j. This is possible because objects of this class are not forbidden to connect to this path by any instance of the Type Restriction pattern (iii). Then, $\tau' \models_{\mathsf{OCL}} \phi$, and thus $\langle M, \Phi_M \cup \{\phi\} \rangle$ is strongly consistent. □

The following set of invariants is inconsistent.

```
context A
   inv: TypeRelation(b,{D,E})
   inv:  MultiplicityRestriction (b,<=,1)
```
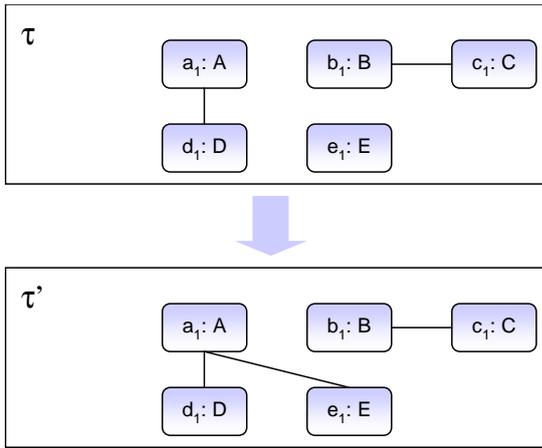
Figure 13 Illustration of the proof for Lemma 8.

The first invariant requires every object of class *A* to be related to objects of class *D and* objects of class *E*, whereas the second invariant allows objects of class *A* to be related to at most one object of class *B*, the superclass of *D* and *E*.

No Cyclic Dependency. The No Cyclic Dependency pattern can be instantiated to disallow cyclic links between objects on a given navigation path. We defined it as follows in Section 1.2.

```
pattern NoCyclicDependency(property: Sequence(Property)) =
  self .closure(property,Set{})−>excludes(self)


pattern closure(property : Sequence(Property),S:Set(Class)) =
  property−>union((property − S)−>
    collect (o:Class | o.closure(property,S−>including(self)))−>
    asSet())
```

The No Cyclic Dependency pattern has a parameter property, which denotes a path in the model that must be non-cyclic in all model states. The pattern invokes the closure pattern, which denotes the transitive closure of a path property in the model. The closure pattern has an additional parameter S, which we use to ensure termination of the computation on finite model states. Since S grows with each recursive invocation, the set property − S (where '−' is the OCL syntax for the set difference operator '\') will eventually be empty and the recursion will terminate.

Lemma 9 (Termination of the Closure Pattern) *Let M be a class model and τ be an arbitrary, but finite, state of M. Let $C \in M$ be a class and p be a path in M from C to C. If $o \in \tau$ is an object of class C, the computation of* o.closure(p,Set{}) *terminates.*

Proof (for Lemma 9) The closure pattern recursively invokes closure on all objects in the set (property − S). On each invocation, the context object self is added to the set S. This prevents closure from being invoked more than once for each object $o \in \tau$. Since τ is a finite state, there can be only finitely many recursive invocations of closure and thus closure terminates. □

If the pattern is instantiated on a navigation path p, it must be checked that the multiplicities of the association ends included in p allow for noncyclic instantiations. We expressed this property in Lemma 1, which we defined in Section 3.1. We now prove this lemma.

Proof (for Lemma 1) Because $\langle M, \Phi_M \rangle$ is strongly consistent, there exists a state τ in which each class of M is instantiated. Based on τ, we construct a state τ' that witnesses the strong consistency of $\langle M, \Phi_M \cup \{\phi\} \rangle$. We distinguish two cases:

Case 1: $\tau \models_{\mathsf{OCL}} \Phi_M \wedge \phi$. In this case, τ does not contain a cyclic link between objects of class C on path $p_1.p_2. \cdots .p_n$. Thus, $\tau' = \tau$.

Case 2: $\tau \not\models_{\mathsf{OCL}} \Phi_M \wedge \phi$. In this case, there is an object $o_1 : C \in \tau$ and a sequence $(o_1, \ldots, o_m, o_1)$ of objects that represent a cyclic link in which the link from object o_{i-1} to o_i is an instance of association end p_i. We construct τ' from τ by deleting the link from o_{j-1} to o_j. This deletion does not violate any invariant of C_{j-1} because our assumptions state that no relation between objects of class C_{j-1} and C_j is required by an instance of the Object In Collection pattern (iv), and objects of class C_{j-1} are not required to relate to objects of class C_j because the lower multiplicity bound of association end p_j is zero (i) and not further constrained (iii). Now, $\tau' \models_{\mathsf{OCL}} \phi$.

If, after deleting the link, $\tau' \models_{\mathsf{OCL}} \Phi_M$ holds, the construction is finished. If not, the deletion has violated the multiplicity constraints of at least one class of which an object participates in the cycle. In this case, we initially create an object o'_{j-1} of type C_{j-1} and link it to o_j. Subsequently, we iterate the index i from $j - 1$ down to 1 (and potentially from m to $j + 1$ afterwards) and create an object of type C_{i-1}. We link this object to o_i only if class C_i requires a link to class C_{i-1}; the algorithm terminates otherwise. It eventually terminates because there exists a class C_k that does not require a link to class C_{k-1}. This is the case because the lower multiplicity bound of the opposite association end of p_k can be zero, it is not further constrained (i-iii), and there is no instance of the Object In Collection pattern on p_k (iv). After this construction, the multiplicity constraints hold that were violated by the deletion and thus $\tau' \models_{\mathsf{OCL}} \Phi_M$ and, as shown before, $\tau' \models_{\mathsf{OCL}} \phi$. Thus, $\langle M, \Phi_M \cup \{\phi\} \rangle$ is strongly consistent. □

Figure 14 shows the construction that we use in the proof on an example. In this example, there exists a state τ in which there is a cyclic link between the objects. We construct a state τ' according to the construction in the proof; the association end b is the required p_j and c is the required p_k.

Path Depth Restriction. The Path Depth Restriction pattern can be used to limit instances of reflexive associations to a given length.
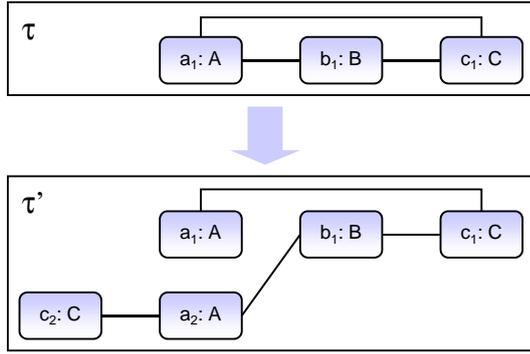
Figure 14 Illustration of the proof for Lemma 1.

```
pattern PathDepthRestriction(property: Sequence(Property),
                             maxDepth:Integer) =
  self .pathDepthSatisfied(property,maxDepth−1,0)


pattern pathDepthSatisfied(property: Sequence(Property),
                           max:Integer, counter:Integer) =
  if  (counter > max or max < 0) then false
  else  if ( self .property−>isEmpty()) then true
      else  self .property−>forAll(m|
           m.pathDepthSatisfied(property, max, counter+1))
      endif
  endif
```

We define the consistency lemma for the Path Depth Restriction pattern as follows.

Lemma 10 (Consistency of Path Depth Restriction) *Let* $\langle M, \Phi_M \rangle$ *be a strongly consistent model that has at least one finitely large state. Let ϕ be an instance of the* Unique Path *pattern with class C as context, a navigation path $P = p_1.p_2. \cdots .p_n$, and a maximum depth of n. $\langle M, \Phi_M \cup \{\phi\} \rangle$ is strongly consistent.*

Proof (for Lemma 10) Because $\langle M, \Phi_M \rangle$ is strongly consistent, there exists a state τ in which each class of M is instantiated. Based on τ, we construct a state τ' that witnesses the strong consistency of $\langle M, \Phi_M \cup \{\phi\} \rangle$. In the following, we do not consider the trivial case and thus, $\tau \not\models_{\text{OCL}} \phi$.

In this case, there is an instance of the path P in τ of length m, with $m > n$. We know that the length of the path is finite because $\langle M, \Phi_M \rangle$ is strongly consistent. Thus, we know that instances of this path with finite length can exist.

We create a state τ' from τ as follows. Starting from the first element o_0 on the path of class C, we follow the path instance n times, ending at another object o_i of class C. We cut the path by deleting the link between o_i and o_{i+1}, as shown in Figure 15. Next, we restore the head of the remainder of the path by instantiating a new object of class C and linking it to o_{i+1}. We recursively apply these steps until the end of the path is reached. With this construction, we have split the instance of the path into parts with a maximum size of n each. Thus $\langle M, \Phi_M \cup \{\phi\} \rangle$ is strongly consistent. □
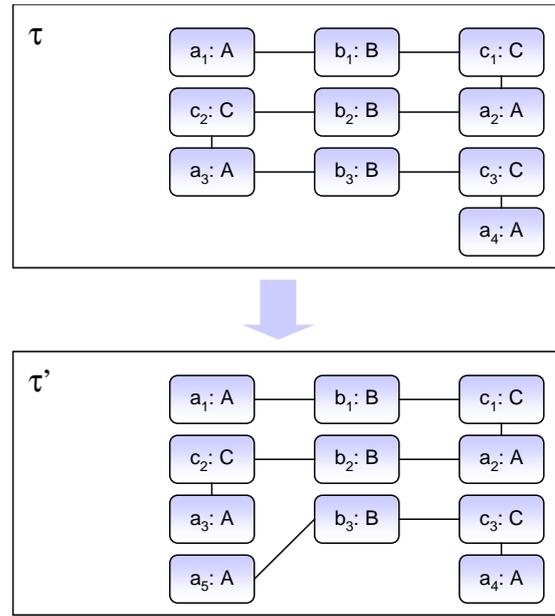

Figure 15 Illustration of the proof for Lemma 10.

3.2.2 Attribute-Constraining Patterns. Whereas the previous constraint patterns can be used to constrain the structure of the object graph, the following patterns constrain the values of attributes. Since the values of two or more attributes can be mutually dependent, instances of attribute-constraining patterns can also be contradictory. Consider the following example.

```
context A
  inv:  self .b−>forAll( b |  self .x > b.y )   −− Attribute Relation
context B
  inv:  self .y = self .c−>sum(z)   −− AttributeSumRestriction
context C
  inv:  self .z = self .a.x   −− AttributeValueRestriction
```

In every model state, each summand of the sum restriction for class B is greater than the sum. This is a contradiction and thus no satisfying instance exists. We reflect this in the following consistency lemmas in which we do not ensure consistency if attributes are constrained more than once.

Attribute Relation. Using the Attribute Relation pattern, attributes can be related to other attributes.

```
pattern AttributeRelation (navigation:Sequence(Property),
                           remoteAttribute:Property,
                           operator: OclExpression,
                           contextAttribute :Property) =
  self .navigation−>forAll( x |
     x.remoteAttribute operator  contextAttribute )
```

We define the consistency lemma for the Attribute Relation pattern as follows.

Lemma 11 (Consistency of Attribute Relation) *Let* $\langle M, \Phi_M \rangle$ *be a strongly consistent model that has at least*

one finitely large state and let $\phi(navigation, remote$-$Attribute, op, contextAttribute)$ *be an instance of the Attribute Relation pattern with class C as context. If*

    (i) there is no instance of the Attribute Sum Restriction, Attribute Value Restriction, Multiplicity Restriction, or another Attribute Relation pattern in Φ_M in which context-Attribute is used as a parameter,

   (ii) $remoteAttribute \neq contextAttribute$, and

  (iii) there is no instance of the Unique Identifier pattern in Φ_M in which contextAttribute is one of the unique properties,

then $\langle M, \Phi_M \cup \{\phi\}\rangle$ is strongly consistent.

Proof (for Lemma 11) Because $\langle M, \Phi_M\rangle$ is strongly consistent, there exists a state τ in which each class of M is instantiated. Based on τ, we construct a state τ' that witnesses the strong consistency of $\langle M, \Phi_M \cup \{\phi\}\rangle$. In the following, we do not consider the trivial case and thus, $\tau \not\models_{\mathsf{OCL}} \phi$.

    In this case, there exists an object o of class C in τ for which the contextAttribute violates ϕ. We set the value of contextAttribute such that it satisfies ϕ as shown in Figure 16. Now, $\tau' \models_{\mathsf{OCL}} \phi$. Furthermore, no constraint in Φ_M is violated because contextAttribute is not related to another property (i), to itself (ii), and its value need not be unique (iii). Thus it holds that $\tau' \models_{\mathsf{OCL}} \Phi_M$ and therefore $\langle M, \Phi_M \cup \{\phi\}\rangle$ is strongly consistent. $\square$
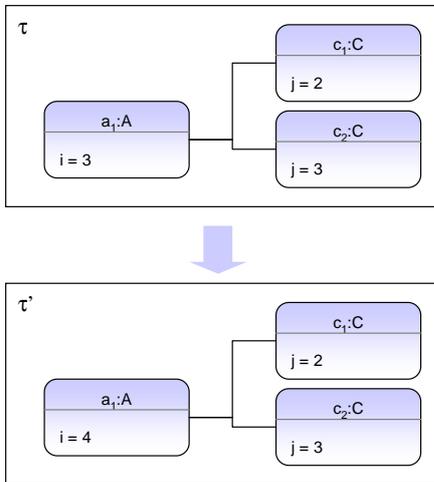

Figure 16 Illustration of the proof for Lemma 11.

The following constraints are inconsistent.

```
context A
   inv: AttributeRelation (b.c.a, i ,>, i)
   inv: AttributeSumRestriction(i ,b.c.a, i)
```

Whereas the first invariant requires that for each object of class A, the value of the attribute i must be less than the value of the same attribute of all related A objects, the second invariant requires that the value of attribute i must be equal to the sum of the values of attribute i of all related A objects.

Attribute Sum Restriction.   The Attribute Sum Restriction pattern can be used limit the value of an integer attribute to the sum of the values of related attributes.

```
pattern AttributeSumRestriction(navigation: Sequence(Property),
                     summand: Property,
                     summation: Property) =
   self .navigation.summand−>sum() <= summation
```

We define the consistency lemma for the Attribute Sum Restriction pattern as follows.

Lemma 12 (Consistency of Attribute Sum Restriction)
Let $\langle M, \Phi_M\rangle$ be a strongly consistent model that has at least one finitely large state and let $\phi(navigation, sum$-$mand, summation)$ be an instance of the Attribute Sum Restriction pattern with class C as context. If

    (i) there is no instance of the Attribute Relation, Attribute Value Restriction, Multiplicity Restriction, or another Attribute Sum Restriction pattern in Φ_M in which summation is used as parameter and

   (ii) if navigation is reflexive and summation $=$ summand, there is no instance of the Unique Identifier pattern in Φ_M in which summation is one of the unique properties,

then $\langle M, \Phi_M \cup \{\phi\}\rangle$ is strongly consistent.

Proof (for Lemma 12) Because $\langle M, \Phi_M\rangle$ is strongly consistent, there exists a state τ in which each class of M is instantiated. Based on τ, we construct a state τ' that witnesses the strong consistency of $\langle M, \Phi_M \cup \{\phi\}\rangle$. In the following, we do not consider the trivial case and thus $\tau \not\models_{\mathsf{OCL}} \phi$.

    In this case, there exists an object o of class C for which the *summation* attribute does not have the correct value. We therefore set the value of *summation* to the sum of the *summand* properties of all objects related to o via *navigation* as shown in Figure 17. Now $\tau' \models_{\mathsf{OCL}} \phi$ holds. Furthermore, the value of *summation* is not related to any other property (i) and it is not related to itself (ii). Thus, no existing constraint in Φ_M is violated, $\tau' \models_{\mathsf{OCL}} \Phi_M$, and $\langle M, \Phi_M \cup \{\phi\}\rangle$ is strongly consistent. $\square$

The following constraints are inconsistent.

```
context A
   inv: AttributeSumRestriction(i ,b.c.a, i)
   inv: UniqueIdentifier (i )
```

The first invariant requires that the value of attribute i must be equal to the sum of the values of all i attributes of related A objects, which leaves zero as the only possible value for i for two or more related objects. In contrast, the second invariant requires i to have a unique value for all objects of class A.

Attribute Value Restriction.   The Attribute Value Restriction pattern represents a common kind of constraint, namely simple value restrictions for attributes.

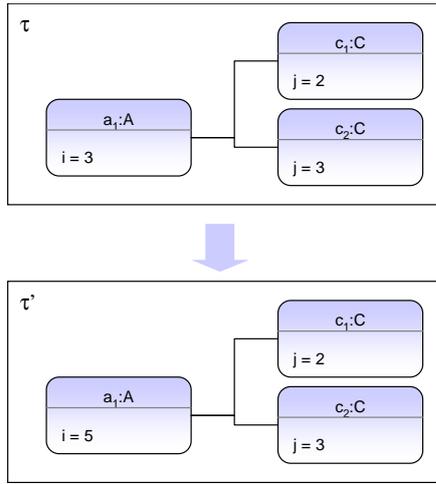**Figure 17** Illustration of the proof for Lemma 12.

```
pattern AttributeValueRestriction (property:Property,
                                   operator:OclExpression,
                                   value:OclExpression) =
    self .property operator value
```

We define the consistency lemma for the Attribute Value Restriction pattern as follows.

Lemma 13 (Consistency of Attribute Value Restriction)
Let $\langle M, \Phi_M \rangle$ *be a strongly consistent model that has at least one finitely large state and let* $\phi(p, op, v)$ *be an instance of the Attribute Value Restriction pattern with class* C *as context. If*
  (i) there is no instance of the Attribute Sum Restriction, Attribute Relation, Multiplicity Restriction, or another Attribute Value Restriction pattern in Φ_M *in which* p *is used as a parameter,*
 (ii) if op is "=", there is no instance of the Unique Identifier pattern in Φ_M *in which* p *is one of the unique properties,*
then $\langle M, \Phi_M \cup \{\phi\} \rangle$ *is strongly consistent.*

Proof (for Lemma 13) Because $\langle M, \Phi_M \rangle$ is strongly consistent, there exists a state τ in which each class of M is instantiated. Based on τ, we construct a state τ' that witnesses the strong consistency of $\langle M, \Phi_M \cup \{\phi\} \rangle$. In the following, we do not consider the trivial case and thus, $\tau \not\models_{\mathsf{OCL}} \phi$.

In this case, there exists an object o of class C in τ for which the property p violates ϕ. We set the value of p such that it satisfies ϕ as shown in Figure 18. Now, $\tau' \models_{\mathsf{OCL}} \phi$. Furthermore, no constraint in Φ_M is violated because p is not related to another property or to itself (i), and its value need not be unique (ii). Thus, $\tau' \models_{\mathsf{OCL}} \Phi_M$, $\langle M, \Phi_M \cup \{\phi\} \rangle$ is strongly consistent. $\square$

The following constraints are obviously inconsistent.

```
context A
   inv: AttributeValueRestriction (i,<,0)
   inv: AttributeValueRestriction (i,>,0)
```
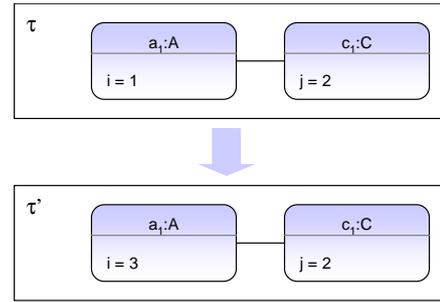

Figure 18 Illustration of the proof for Lemma 13.

Unique Identifier. Using the Unique Identifier pattern, a tuple of properties can be specified that must be unique for each object of the context class.

```
pattern UniqueIdentifier (property:Tuple(Property)) =
    self .class :: allInstances()−>isUnique(property)
```

We define the consistency lemma for the Unique Identifier pattern as follows.

Lemma 14 (Consistency of Unique Identifier) *Let* $\langle M, \Phi_M \rangle$ *be a strongly consistent model that has at least one finitely large state. Let* ϕ *be an instance of the Unique Identifier pattern with class* C *as context, and a set* P *of properties. If for all* $p \in P$,
  (i) the domain of the type of p *is infinite,*
 (ii) there is no instance $\psi(p, \_, p)$ *of the Attribute Sum Restriction pattern in* Φ_M,
(iii) there is no instance $\psi(\_, \_, =, p)$ *of the Attribute Relation pattern on* C *in* Φ_M, *and*
(iv) there is no instance $\psi(p, =, \_)$ *of the Attribute Value Restriction pattern in* Φ_M,
then $\langle M, \Phi_M \cup \{\phi\} \rangle$ *is strongly consistent.*

Proof (for Lemma 14) Because $\langle M, \Phi_M \rangle$ is strongly consistent, there exists a state τ in which each class of M is instantiated. Based on τ, we construct a state τ' that witnesses the strong consistency of $\langle M, \Phi_M \cup \{\phi\} \rangle$. In the following, we do not consider the trivial case and thus, $\tau \not\models_{\mathsf{OCL}} \phi$.

In this case, there exist two objects o_1 and o_2, both of class C, for which $p_i(o_1) = p_i(o_2)$, for all $p_i \in P$. We arbitrarily choose some i and change the value of $p_i(o_2)$ such that there is no other object o of class C with $p_i(o) = p_i(o_2)$, as shown in Figure 19. Now, $\tau' \models_{\mathsf{OCL}} \phi$. This is possible because there are infinitely many possible values for p_i (i). Furthermore, there would be only one possible value for p_i in the presence of a reflexive Attribute Sum Restriction constraint, which we exclude in the assumptions (ii) and objects of class C are not required to have the same value (iii and iv). After this change, it is possible that $\tau' \not\models_{\mathsf{OCL}} \Phi_M$ if p_i is the parameter of an instance of the Attribute Sum Restriction, Attribute Relation, or Attribute Value Restriction pattern. In this case, the attribute values in τ' must be changed such that $\tau' \models_{\mathsf{OCL}} \Phi_M$. This is possible because each attribute value is constrained by at most one constraint because of

the assumption that $\langle M, \Phi_M \rangle$ is strongly consistent and because of the Lemmas 11, 12, and 13. Now, $\tau' \models_{\mathsf{OCL}} \Phi_M$ and $\langle M, \Phi_M \cup \{\phi\} \rangle$ is strongly consistent. $\square$
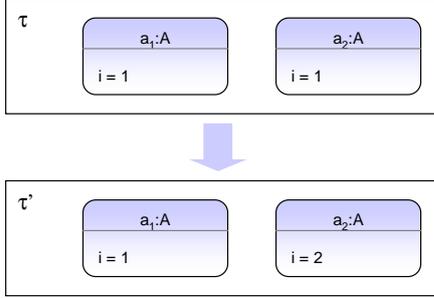

Figure 19 Illustration of the proof for Lemma 14.

The following constraints are inconsistent as explained above for the Attribute Sum Restriction pattern.

```
context A
    inv: UniqueIdentifier(i)
    inv: AttributeSumRestriction(i,b.c.a,i)
```

3.2.3 Other Patterns. The next pattern is the only pattern that constrains both the structure of the object graph and attribute values.

Multiplicity Restriction. The Multiplicity Restriction pattern can be instantiated to limit the multiplicity of an association to a given attribute value.

```
pattern  MultiplicityRestriction (navigation: Sequence(Property),
                                  operator: OclExpression,
                                  value:OclExpression) =
self.navigation−>asSet()−>size() operator value
```

Since the Multiplicity Restriction pattern constrains both the structure of the object graph and attribute values, it is related to almost all other patterns as shown in the previous lemmas. We take this into account in the following consistency lemma.

Lemma 15 (Consistency of Multiplicity Restriction) *Let $\langle M, \Phi_M \rangle$ be a strongly consistent model that has at least one finitely large state and let $\phi(P, op, v)$ be an instance of the Multiplicity Restriction pattern with class C as context. If*

(i) v is not a property that is used as parameter for an instance of the Attribute Sum Restriction, Attribute Relation, Attribute Value Restriction, or another Multiplicity Restriction pattern in Φ_M,

(ii) the tuple (op, v) is not one of $(<, 1)$, $(=, 0)$, or $(\leq, 0)$, where the lower multiplicity bound of the last element of P is greater than zero,

(iii) for all $p \in P$, the multiplicity is unbounded (*),

(iv) there is no other instance of the Multiplicity Restriction pattern on any $p \in P$ in Φ_M, and

(v) p is not used as part of a parameter value for any instance of the No Cyclic Dependency, Object In Collection, Surjective Association, Injective Association, or Type Relation pattern in Φ_M, and p^{-1} is not used in any instance of the Type Restriction pattern in Φ_M,

then $\langle M, \Phi_M \cup \{\phi\} \rangle$ is strongly consistent.

Proof (for Lemma 15) Because $\langle M, \Phi_M \rangle$ is strongly consistent, there exists a state τ in which each class of M is instantiated. Based on τ, we construct a state τ' that witnesses the strong consistency of $\langle M, \Phi_M \cup \{\phi\} \rangle$. In the following, we do not consider the trivial case and thus, $\tau \not\models_{\mathsf{OCL}} \phi$.

In this case, there exists an object o of class C that has either too few or too many links to objects of class $type(P)$. We therefore create or delete objects of class $type(P)$ until $\tau' \models_{\mathsf{OCL}} \phi$, as shown in Figure 20. Creating or deleting such objects does not violate any constraint in Φ_M because P is not constrained by any other constraint (v); furthermore, at least one object of class $type(P)$ will remain because the case is excluded that zero objects of class $type(P)$ are connected to o. This follows because the value of v is not constrained if v is a property (i), the multiplicity of P is unbounded (iii), there is no other instance of Multiplicity Restriction on p (iv), and the parameters of ϕ allow for at least one link (ii). Thus $\tau' \models_{\mathsf{OCL}} \Phi_M$, and $\langle M, \Phi_M \cup \{\phi\} \rangle$ is strongly consistent. $\square$
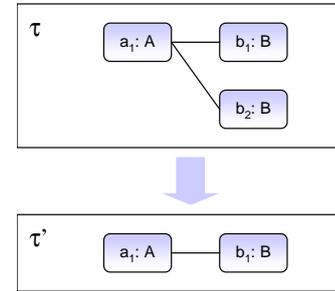

Figure 20 Illustration of the proof for Lemma 15.

The following constraints are inconsistent.

```
context A
    inv:  MultiplicityRestriction (b,<=,i)
    inv:  AttributeValueRestriction (i,=,0)
    inv:  MultiplicityRestriction (b,>=,1)
```

Whereas the first invariant limits the number of associations between an A object and a B object to the value of i, the second invariant determines this value to be zero. The third invariant contradicts the previous two by requiring at least one B object to be related to each A object.

3.3 Summary of Constraint Pattern Dependencies.

In the previous subsection, we have stated consistency lemmas for each constraint pattern in our library by exploring
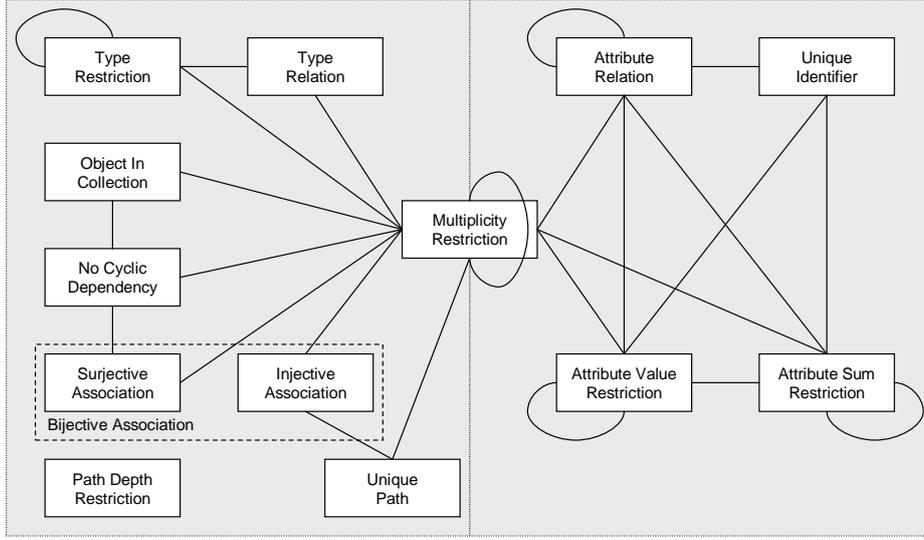
Figure 21 Conflict matrix of constraint patterns.

dependencies between the patterns. Figure 21 helps visualize the results. In the graph shown in this figure, solid edges between two patterns denote that instances of the respective patterns can potentially contradict each other.

The figure is split into two halves. The patterns that constrain the object graph are in the left half and the patterns that constrain attribute values are in the right half. The Multiplicity Restriction pattern sits in the middle because it belongs to both groups. Note that there is no edge, and thus no immediate contradiction, between a pattern in the left half and a pattern in the right half, with the exception of the Multiplicity Restriction pattern. As a consequence, instances of pattern in the left half of the figure and the right half of the figure can contradict each other only if an instance of the Multiplicity Restriction pattern is present. This motivates the following corollary.

Corollary 1 *Two pattern instances $\phi_1, \phi_2 \in \Phi$ can be inconsistent iff there exists a path between their patterns in the graph in Figure 21 and there exists a pattern instance in Φ for each node on this path.*

3.4 Automating the Analysis

In this section, we explain how the consistency lemmas can be used to implement an automatic, efficient analysis. To analyze pattern-based constraint specifications for consistency, it is sufficient to check for each pattern instance whether the assumptions defined in the lemma of the respective pattern hold. This consists of syntactic checks that can be performed in polynomial time. For each pattern instance in the constraint specification, the consistency analysis can either result in *consistent* if the assumptions in the respective theorem hold or *unknown* otherwise. In the latter case, the consistency of the constraint specification must be proven by using alternative analysis approaches. We will explain how to deal with the *unknown* case in Section 3.5.4 and present alternative analysis approaches in Section 6.

In the following, we explain our analysis algorithm in a Java-like language. We assume that constraint patterns are represented by classes that implement the respective consistency assumptions in a method isConsistent(). The consistency analysis for a model $\langle M, \Phi_M \rangle$ first iterates over all pattern instances in the constraint specification Φ_M and invokes the method isConsistent() on each pattern instance ψ.

```
void validateModel (Model M, Set<ConstraintPattern> Φ_M) {
  boolean modelConsistent = true;

  foreach (ψ ∈ Φ_M) {
    if  (! ψ.isConsistent(M, (Φ_M\{ψ})))
      modelConsistent = false;
  }

  if  (modelConsistent)
    System.out.println ("Model is consistent." );
  else
    System.out.println ("Consistency of model is unknown.");
}
```

The method isConsistent() evaluates all consistency assumptions for a given pattern instance. If one of these assumptions does not hold, the pattern instance is not consistent with the rest of the pattern specification.

```
abstract class ConstraintPattern {
  Set<ConsistencyAssumption> A;

  boolean isConsistent (Model M, Set<ConstraintPattern> Φ_M) {

    foreach (assumption ∈ A)
      if  (assumption.evaluate(this, M, Φ_M) == false)
        return false;

    return true;
  }
}
```

}

The following code snippet shows an implementation of Consistency Assumption (i) of the No Cyclic Dependency pattern (cf. Lemma 1). Consistency assumptions are implemented as specializations of the abstract class ConsistencyAssumption and implement the evaluate() method. For Assumption (i) of No Cyclic Dependency, this method checks whether the parameter property contains association ends p_j and p_k^{-1} whose lower multiplicity bounds are zero. If both such association ends are found, the assumption holds.

```
class Assumption1 extends ConsistencyAssumption {

  public boolean evaluate (ConstraintPattern ψ, Model M,
                           Set<ConstraintPattern> Φ_M) {
    boolean foundJ = false;
    boolean foundK = false;

    List<Property> P = ψ.getParameterValue("property");

    foreach (p ∈ P) {
      if  (p.getLowerBound() == 0)
        foundJ = true;

      if  (p.getOpposite().getLowerBound() == 0)
        foundK = true;
    }

    return (foundJ && foundK);
  }
}
```

We compute the time complexity of our algorithm as follows. In our approach, the method validateModel() iterates over each pattern instance ψ in the constraint specification Φ_M and invokes its isConsistent() method. This method iterates over the consistency assumptions A of the respective constraint pattern and invokes the evaluate() method for each assumption. We analyze below the complexity of this check for each assumption.

There are two types of consistency assumptions. Assumptions of the first type are syntactic checks in the model regarding multiplicity bounds (e. g., Assumption i of Lemma 1) or the reflexivity of navigation paths (e. g., Assumption ii of Lemma 12). Such checks have linear complexity in the length of the path, assuming that accessing model information (e. g., values of multiplicity bounds) has constant complexity. The second type of assumptions are checks that instances of certain patterns exist. Such checks have linear complexity in the number of pattern instances in the specification. As a consequence, evaluate() always has linear complexity.

Checking the consistency assumptions for *one* pattern instance ψ hence requires time $\mathcal{O}(|A| \cdot |\Phi_M| \cdot |M|)$ in the worst case, where $|A|$ is the number of consistency assumptions of ψ and $|\Phi_M|$ is the total number of pattern instances in the constraint specification. Since the number of association ends constitutes an upper bound for the length of each path P in the class model, $|M|$ is the size of the input model in

terms of the number of association ends contained. The analysis of a *whole* constraint specification Φ_M hence takes time $\mathcal{O}(|\Phi_M|^2 \cdot |A_{max}| \cdot |M|)$ in the worst case, where A_{max} is the maximum number of consistency assumptions of all pattern instances in Φ_M. Thus, this analysis has polynomial-time complexity.

3.5 Characteristics of the Algorithm

Our analysis is *incomplete* because it can return *unknown* even for a consistent UML/OCL model. In contrast, a consistency analysis is *complete* if for every consistent UML/OCL model, the algorithm returns *consistent*.

Our analysis approach is *sound*. A consistency analysis is *sound* if whenever it returns *true*, then the UML/OCL model is consistent. Since our analysis algorithm only returns *true* if the consistency assumptions hold (and these have been proven to be sufficient conditions for consistency), it is sound.

In the following, we point out four limitations of our analysis approach: arbitrary OCL constraints cannot be analyzed, the return values of methods cannot be used as parameter values in patterns, adding new constraint patterns is non-trivial, and providing sufficient (but not necessary) conditions in the consistency lemmas generates warnings.

3.5.1 Arbitrary OCL Constraints   Our analysis only analyzes the consistency of constraints that are instances of constraint patterns. Thus, the consistency of models that are additionally annotated with arbitrary OCL constraints cannot be analyzed.

We see two options if arbitrary OCL constraints appear in a constraint specification. First, each OCL constraint can be abstracted into a constraint pattern and a consistency lemma for this new pattern can be established. However, this is often infeasible because adding new constraint patterns to a pattern library requires significant effort in updating the existing consistency lemmas. Second, the development process can be split into two phases. In the first phase, the model is only augmented with pattern instances and subsequently analyzed for consistency. This allows model developers to receive immediate feedback about the consistency of one part of the constraint specification because of the polynomial complexity of our approach. In the second phase, the model is augmented with the remaining constraints in OCL. The model developer must then analyze the consistency of the fully constrained model using other analysis approaches. We describe other approaches, which have a significantly higher complexity than our approach, in Section 6.

3.5.2 Method Calls in Pattern Instances   In our constraint patterns, the types of the pattern parameters are typically properties or sets thereof, classes, or integer. In typical constraint languages, it is also possible to call methods in invariants. For example, the fact that managers should not be able to hire themselves could be expressed as an invariant not self.hire(self)—although such a condition should clearly

be stated in the pre-condition of hire(). When analyzing the consistency of this invariant, the definition of hire() must be taken into account. Since methods can be recursive and potentially nonterminating, we do not allow methods as parameter types for our patterns, following [20].

3.5.3 Adding New Constraint Patterns   Clearly, the library of constraint patterns presented in Section 3.2 is extensible. However, after each extension of the library, two tasks must be carried out. First, a new consistency lemma for the newly added constraint pattern must be proven. Second, the consistency lemmas of all existing patterns must be revised. In particular, for each lemma, it must be analyzed whether instances of the newly added pattern can cause inconsistencies with instances of the respective pattern. Not all consistency lemmas must be analyzed though: If the newly added pattern is a pure attribute-constraining pattern, it will not contradict pure association-constraining patterns and vice versa. We further elaborate on extending the pattern library in Section 5.2.

3.5.4 Handling "Unknown" Consistency   Since the assumptions in our consistency lemmas represent *sufficient* conditions and not *necessary* conditions, there are models that are consistent, but our analysis cannot determine the consistency because the assumptions are overly restrictive. In this case, the model developer is confronted with pairs of pattern instances that can *potentially* contradict each other.

Consider the following two pattern instances, which cannot be shown to be consistent by our analysis.

```
context Manager
inv c1:   TypeRelation(worksIn,{Single})
inv c2:   MultiplicityRestriction  (worksIn,=,1)
```

The analysis warns that these instances may be inconsistent because assumption (v) of the consistency lemma for the Multiplicity Restriction pattern is violated for constraint c2. An instance ϕ of the Multiplicity Restriction pattern and an instance ψ of the Type Relation pattern can be contradictory if ψ requires objects of n different classes to be in a relation, but the multiplicity of this relation is constrained by ϕ to some m, where $m < n$.

Since a consistency assumption is violated, our analysis cannot determine the consistency of c1 and c2. In this case, the model developer must use alternative analysis approaches to determine whether the warning generated by the analysis indicates an actual inconsistency. We present several alternative analysis approaches in Section 6. However, these approaches typically have exponential complexity and cannot directly benefit from the pattern representation of the constraints. Thus, an interesting question is whether the formalization of the constraints using patterns can help to accelerate consistency analysis with alternative approaches? Using the dependency graph shown in Figure 21, one could divide the set of constraints to obtain smaller inputs for consistency analysis. Analyzing such a divide-and-conquer approach remains as future work.

4 Tool Support

The success of MDE depends on tool support. Hence, we have implemented a tool, which we call **Co**nstraint **Pa**tterns and **C**onsistency **A**nalysis (Copacabana), as a collection of plug-ins for the MDE tool IBM Rational Software Architect (RSA). This tool supports model developers during the consistency-preserving refinement of UML class models with constraint patterns and guides them through the different phases of a pattern-based development process. As illustrated in Figure 22, this process comprises the phases *constraint elicitation* (the tool automatically analyzes the model and proposes a set of constraints that should be added to the model), *constraint specification* (the developer selects constraints from the proposed set and adds them to the model by instantiating constraint patterns), *consistency analysis* (described in this paper), and *code generation*. This process is explained in detail in [57].

Figure 23 shows a screenshot of RSA with the Copacabana plug-ins. The largest view (1) contains the company model and an instance of the No Cyclic Dependency pattern. In this view, models can be edited and patterns can be instantiated via drag-and-drop. Below this view, the *constraint elicitation* view (2) shows the results of the constraint elicitation component. The bottom part of the window contains two additional views. The results of the consistency analysis are shown in the bottom-left part in the *problems* view (3) of RSA. In the bottom-right is the *pattern explorer* (4), which displays the available patterns along with their description. The pattern explorer also displays all available model transformations in the RSA transformation framework.

In the following, we further describe the component for consistency analysis and provide examples based on our company model.

4.1 Consistency Analysis

Copacabana implements the pattern-based consistency analysis introduced in Section 3. To this end, we store with each pattern the assumptions from the respective consistency lemma, i. e., the assumptions under which the pattern can be instantiated in a consistency-preserving way. The current implementation contains the lemmas for strong consistency.

The analysis uses these consistency assumptions and checks for each pattern instance whether the assumptions hold. If they hold, the constraint specification is consistent. Otherwise, no statement about consistency can be made, as explained in Section 3.5. In this case, the consistency analysis component issues a *warning* that the pattern instance is potentially inconsistent. Such a warning can be seen in Figure 24, which displays the analysis results for the company model and the constraints that we have presented in this paper. As can be seen, a warning is displayed that the constraint noCycles cannot be shown to be consistent because the constraint noCycles disallows cyclic management hierarchies, whereas the company model requires every manager to have at least

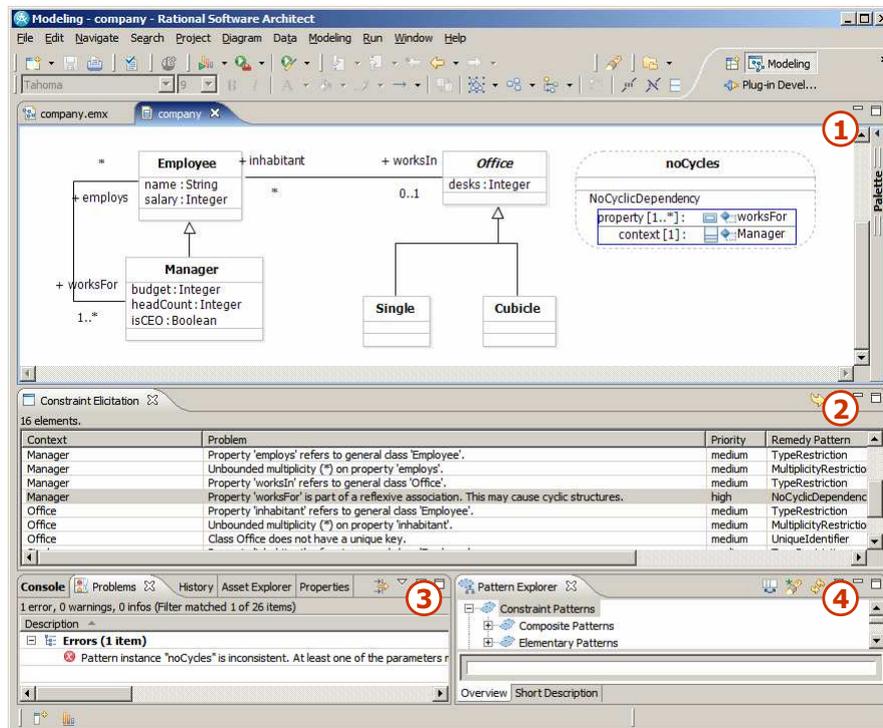**Figure 22** Process for pattern-based constraint development.

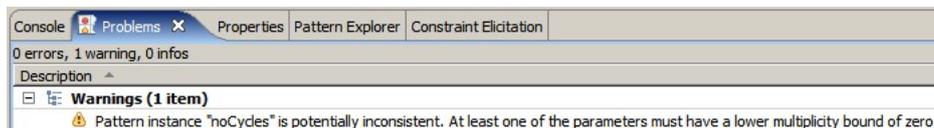**Figure 23** Screenshot of the Copacabana prototype in RSA.

Figure 24 Consistency analysis results of the constrained company model.

one superior. As a result, there is no state of the company model with a nonempty, finite set of managers.

In cases where models cannot be shown to be consistent, the model developer must examine the warnings and understand if they indicate a consistency problem. This can be either done by an alternative consistency analysis (cf. Section 6) or by manually inspecting the model and its constraint specification. Subsequently, the model developer can either adapt the model or correct the incorrect constraint if the warning represents an actual error.

4.2 Summary

Copacabana extends IBM Rational Software Architect (RSA) to support developers in using our pattern approach to developing consistent constraint specifications. The resulting tool supports, besides consistency analysis for pattern-based constraint specifications, constraint elicitation, constraint speci-

fication using constraint patterns, and code generation. Using Copacabana, model developers can create consistent OCL specifications without writing a single line of OCL.

Since we built Copacabana on top of various frameworks in RSA, the code base for Copacabana only contains the application logic and is thus rather small. In total, Copacabana consists of 101 classes, together containing around 4,200 lines of code.

Copacabana has been published as *IBM Constraint Patterns and Consistency Analysis* on the IBM developerWorks website [32]. On this website, there is also a tutorial [58] that guides model developers, step-by-step, through the installation and the development process. The plug-ins are bundled as a *reusable asset* for RSA; using the Reusable Asset Specification (RAS) explorer in RSA, the plug-ins can be downloaded and used for free.

5 Case Studies

We have validated our pattern-based analysis approach in three case studies. In these studies, we used our tool support to formalize the constraint specifications of three different industrial-scale models using constraint patterns and we subsequently analyzed their consistency. In this section, we summarize the results of these case studies. In Section 5.1, we give an overview of the models involved and introduce the metrics of interest. Next, we describe how we developed pattern-based constraint specifications for the respective models in Section 5.2. In Section 5.3, we report on the results of our consistency analysis.

5.1 Subject Models and Evaluation Metrics

We used constraint patterns to develop consistent constraint specifications for three different models. In each case, a class model and a constraint specification were given in advance. Our objective in each case study was to formalize the constraints using constraint patterns and to evaluate their consistency. We used the following models for our case studies.

Business monitors. We formalized the constraint specification of a meta-model for business monitors used in an IBM product. Business monitor models describe the workflow of messages created during the execution of business processes. The model comprises 25 classes and the specification document comprises 71 informally specified constraints.

Business processes. We formalized constraints on business process models that originate from an IBM-internal prototype for merging process models. This prototype requires input models to satisfy five constraints in addition to the standard constraints on process models. The meta-model for business process models, which is used in an IBM product, comprises 33 classes.

Royal & Loyal. We formalized the Royal & Loyal model, a model of a customer bonus program. This model is frequently used in publications on MDE and as an example in MDE tools. It can thus be considered a benchmark model for MDE approaches. The model as presented in [40] comprises 11 classes and 26 constraints, which are specified in OCL.

The main metric of interest is the fraction of constraints that can be analyzed with our algorithm presented in Section 3.4. This metric is the product of the fraction of constraints that can be formalized using our constraint patterns and the fraction of these constraints for which our algorithm returns a result. In addition, we are interested in the fraction of false positives, i.e., consistent constraints that cannot be shown to be consistent by our analysis.

5.2 Developing Pattern-based Specifications

In this subsection, we report on specification coverage results, that is, the fraction of constraints that can be expressed using constraint patterns. Specification coverage is an important metrics for our analysis approach because if only a low percentage of constraints could be expressed using patterns, our analysis would be applicable only to a minority of constraints.

Fortunately, around 75% of constraints in our case studies could be expressed using constraint patterns. The remaining 25% of constraints must thus be formalized using plain OCL and fall outside of our automated analysis. This raises two questions. First, what are the reasons why there are no suitable constraint patterns for 25% of the constraints and second, should our pattern library be extended to account for the remaining 25%?

The constraints not expressible using our constraint patterns can be divided into two groups. First, constraints that invoke user-defined functions cannot be expressed using our constraint patterns. In the course of constraint development, one must sometimes define functions such as transitive closure operations or domain-specific parsing of recursive expressions, and such functions must be hand-written in OCL. In fact, approximately 66% of the constraints that could not be expressed using patterns required user-defined functions. Second, there are constraints that share a similar structure with a constraint pattern, but they do not match any existing constraint pattern. An example is the requirement that the budget of managers must *exactly* cover the salaries of their employees. This constraint, when formalized, differs from the Attribute Sum Restriction pattern only in the binary operator.

```
context Manager
inv salariesExactlyCovered:
    self.employs.salary−>sum() = self.budget
```

In such cases, one has two choices: specify the constraints using plain OCL or change the pattern library. As mentioned previously, hand-written OCL constraints are error-prone and cannot be analyzed using our analysis.

The second choice, changing the constraint-pattern library, has the advantage that the constraint can subsequently be checked for consistency by our analysis. The library can be changed in two ways: an existing pattern can be generalized or a new constraint pattern can be added. Generalizing an existing pattern requires consistency lemmas to be re-proved. The consistency matrix (cf. Figure 21) helps one to identify those patterns whose lemmas are affected by the pattern update.

Adding *new* constraint patterns, in contrast, has two substantial disadvantages. First, when too many constraint patterns are offered to the model developer, it becomes difficult to find the right pattern. In the worst case, if the pattern library offers a specification coverage of 100%, it merely represents a different syntax for the underlying specification language. Second, extending the pattern library with a new pattern entails that both a new consistency lemma for the new pattern must be proven and the consistency lemmas of the existing patterns must be checked whether they still hold.

Due to these reasons, the developer of the constraint-pattern library must carefully evaluate whether new con-

straint patterns should be added to the library. An important metrics is the number of constraints that can be expressed using the new pattern. In our case studies, several constraints occur that forbid a certain class to be instantiated. Since there are several such constraints, we can add a new pattern to our library, which we define as follows.

```
pattern NoInstances(class:Class) =
  class :: allInstances()−>isEmpty()
```

Using this pattern increases the coverage, i. e., the fraction of constraints that can be expressed using patterns, by a few percent. However, this pattern has important consequences for the consistency of the model: Since the constrained class cannot be instantiated in *any* model state, the model can be weakly consistent at best. This supports our hypothesis that weak consistency is an appropriate notion of consistency in early development phases. In our case, weak consistency of input models was sufficient for the prototype version of the tool.

5.3 Analyzing Pattern-based Specifications

Our second task in each case study was to analyze the consistency of the pattern-based constraint specification. Among others, two evaluation criteria were especially important in our investigation. First, the *performance* of the analysis is crucial because one of the goals of our approach is to offer an efficient consistency analysis that does not interrupt the model developer's workflow. Second, the *quality* of the analysis is important. Since our analysis can either return "consistent" or "unknown", we define the quality of our analysis as the fraction of consistent constraints that our analysis can show to be consistent.

On average, 75% of the constraints in our case studies can be shown to be consistent. For the remaining 25%, our analysis returns "unknown". Thus, we must prove or refute their consistency as described in Section 3.5.4.

Analyzing the consistency of the models in our case study had an interesting side effect. Consider the following two constraints, which the analysis could not show to be consistent.

```
context StructuredActivityNode
inv explicitControlFlow :
   MultiplicityRestriction (StructuredActivityNode,
                               inputControlPin, <=, 1) and
   MultiplicityRestriction (StructuredActivityNode,
                               outputControlPin, <=, 1) and
   MultiplicityRestriction (StructuredActivityNode,
                               inputObjectPin, <=, 1) and
   MultiplicityRestriction (StructuredActivityNode,
                               outputObjectPin, <=, 1)
```

```
context Action
inv noObjectFlow:
   MultiplicityRestriction (Action,inputObjectPin,=,0) and
   MultiplicityRestriction (Action,outputObjectPin,=,0)
```

The reason why these constraints cannot be shown to be consistent is that they violate an assumption in the consistency lemma for the Multiplicity Restriction pattern (Lemma 15). The following is an extract from this lemma.

Let $\langle M, \Phi_M \rangle$ be a strongly consistent model and let $\phi(p, op, v)$ be an instance of the Multiplicity Restriction pattern. If

(i) …

(ii) there is no other instance of the Multiplicity Restriction pattern in Φ_M that constrains p,

then $\langle M, \Phi_M \cup \{\phi\} \rangle$ is strongly consistent.

As can be seen, assumption (ii) is violated because both constraints restrict the association ends inputObjectPin and outputObjectPin. However, the above constraints explicitControlFlow and noObjectFlow are obviously consistent. The reason why assumption (ii) is violated is because the consistency assumptions are *sufficient* conditions for consistency and not *necessary* conditions, as explained in Section 3.5.4.

The case studies contained several other consistent constraints for which the consistency assumptions are too strong. In general, such information helps the developer of the pattern library to weaken the consistency assumptions such that they can show a larger fraction of constraints consistent. Consequently, this will help to decrease the number of warnings in consistency analysis.

5.4 Quantitative Results

We used two measures to assess our analysis: execution time and the fraction of constraints that can be analyzed. The execution time of the implementation of our analysis algorithm in IBM Rational Software Architect (RSA) is very small. Our analysis is integrated in the validation framework of RSA. Thus, when the developer validates a model, numerous constraints from the UML meta-model are checked in addition to our consistency assumptions. Nevertheless, full model validation, including consistency analysis on our fully constrained models, takes between 2 and 4 seconds. Thus, the execution time of our analysis does not significantly interrupt the workflow of the model developer. Regarding the quality of the analysis, our approach was able to show 75% of the formalized constraints consistent. Since we were able to formalize 75% of all constraints using patterns, we compute a quality measure of $(75\%)^2 \approx 56\%$.

coverage	75%
execution time	4 sec
quality	56%

Table 1 Results from our case studies.

In our case studies, we have seen that a majority of constraints could be formalized using our library of con-

straint patterns. Furthermore, the majority of each pattern-based constraint specification could be shown to be consistent within a few seconds. In addition, we have seen that our approach is capable of handling industrial-scale models with large constraint specifications.

6 Related Work

Previous work on intra-model consistency of class models either examines the consistency of models without OCL constraints, lacks uniform representation, or only defines individual, specialized notions of consistency. Until now, a comprehensive study of consistency for UML/OCL models has been missing as well as a detailed analysis of which aspects of consistency are relevant in practice. In particular, there is no prior work that contrasts different strong notions of consistency within the development process.

Several publications cover the consistency of class models without general OCL constraints. In [46], an algorithm is given for computing the consistency of class models without general OCL constraints. While the authors present definitions for strong consistency, class consistency, and weak consistency, they do not cover abstract classes and infinitely large model states. Their algorithm can be integrated in our approach to show the consistency of unconstrained class models before pattern instances are added. In [38], class models without general OCL constraints are transformed into a consistency problem in FOL. Although no explicit notion of consistency is given, the notion implicitly used is strong consistency. In [5], class models without general OCL constraints are transformed into a consistency problem in Description Logic (DL). The paper provides an informal definition of weak consistency. A different approach translates a subset of UML to the B specification language [55], which can then be automatically checked for consistency [44]. Since B is not object-oriented, only a subset of class models can be used in this approach. Furthermore, OCL constraints are not translated to B.

Different researchers have considered the problem of establishing the consistency of class models with OCL constraints. In [52], the authors show how to transform UML class models and a subset of OCL into a constraint-solving problem for the CQC method [26], a semi-decision procedure for the finite satisfiability of deductive database schemes. The authors provide informal definitions for class consistency and weak consistency.

OCL evaluation tools such as USE [29] or OCLE [14] are not geared towards automatic validation. Nevertheless, they provide limited support for consistency analysis of UML/OCL models. For example, USE provides a means for programmatically generating snapshots on which arbitrary OCL formulas can be evaluated. This can be used for manually testing specific model instances for consistency. In this approach, it is the responsibility of the model developer to specify an input script to find a witness for whatever notion of consistency is desired.

The UMLtoCSP tool [11] transforms a class diagram with OCL invariants to a constraint satisfaction problem. As such, UMLtoCSP allows for an automated validation (bounded verification) of strong consistency and weak consistency of UML/OCL models. For analyzing larger models efficiently, UMLtoCSP requires the user to precisely specify the search space (i. e., the range in which a solution for the underlying CSP is searched for) because the runtime of UMLtoCSP grows exponentially with the size of the searched result space. In contrast, our approach achieves efficient analysis without such an additional user input.

In [23], an approach is described to translate UML/OCL into a model checking problem. The paper provides a mapping of UML and a subset of OCL into a temporal logic called BOTL. Consistency notions can be expressed in BOTL, although doing so is not covered by this approach.

The Alloy Analyzer [34] is a consistency checker for specifications in the Alloy specification language [33], which is less expressive than OCL. However, this approach can be applied to UML/OCL models because there exists a method and tool to transform UML and a subset of OCL into Alloy [6].

The work that is most closely related to ours is [18], which presents a pattern-based constraint specification approach that uses stereotypes on UML constraints to implement the pattern concept. Analogous to our consistency lemmas, the authors define *consistency rules* for their patterns, which are used in a tool for automatic reasoning over the constraints. Furthermore, *redundancy rules* are presented that help to identify superfluous constraints. In contrast to our work, their consistency rules are used to check whether a given constraint specification is *inconsistent*. Analogous to our approach, nothing can be concluded if these rules are violated, e.g., the specification cannot then be assumed to be consistent. By combining our approach with theirs, more detailed (yet still incomplete) consistency information could be given to model developers.

In contrast to the *automatic* approaches summarized above, *interactive* analysis approaches have also been proposed. In these approaches, consistency must be manually encoded as a proof obligation and interactively proven using a theorem-proving system. In [57], we show how this can be done in the HOL-OCL system [9], which provides a formal semantics for UML/OCL and a proof environment for Isabelle/HOL [48]. Another example is given by [43], who provide a proof environment for UML/OCL in higher-order logic within the PVS proof environment [51]. The KeY tool, an interactive reasoning environment for UML/OCL that translates constrained models into dynamic logic, is presented in [2]. Such approaches are more powerful than the automated approach we take here and they do not suffer from the problem of incomplete analysis due to heuristics, although the human who guides the prover may still fail to find a consistency proof. Interactive theorem proving, however, requires considerable interaction as well as specialized knowledge and is therefore not amenable for use by non-specialists, such as typical model designers.

Consistency is, of course, also important in other formal development approaches. VDM++ is an object-oriented extension of the formal modeling language Vienna Development Method (VDM) [36]. Different consistency properties of VDM are considered in [19]. [3] explains how proof obligations can be generated that ensure the consistency of VDM specifications. However, their approach abstracts from the object-oriented aspects of VDM++ and focuses on the functional aspects only, which can be analyzed more easily.

Summarizing the related work, we can thus group the various approaches as follows: interactive theorem proving (e. g., HOL-OCL), witness creation (e. g., USE), SAT and model checking (e. g., BOTL), and pattern-based analysis. In Table 2, we summarize these different kinds of approaches and compare them according to the following criteria. *Coverage* denotes the subset of UML/OCL that is supported. All approaches in our investigation support class models. Approaches that do not support full OCL are typically pattern approaches such as ours or exclude recursive expressions. Since we have identified different notions of consistency, we highlight the *flexibility* of each approach, i. e., to what extent and with what effort it is possible to analyze a given model for different notions of consistency. Whereas analysis approaches in which a certain consistency notion is "hard-coded" have low flexibility, approaches such as witness creation allow developers to easily switch to stronger or weaker consistency notions. We classify our approach as having medium flexibility because different consistency notions can be used, but it requires some effort to state and prove consistency lemmas. *Complexity* denotes the computational effort required to perform consistency checking in the respective approach. The degree of *automation* ranges from "low" for interactive approaches to "high" for fully automatic approaches.

As explained in Section 1.2, we have developed our approach with a focus on efficiency and automation. Besides SAT-based approaches, our approach is the only one that offers full automation to model developers. In contrast to SAT-based approaches, our approach has polynomial complexity, which allows consistency analysis to scale to large models, as shown in Section 5.

7 Conclusion

In this paper, we provided precise notions of consistency for constrained class models and identified practically relevant notions for different software engineering problems. Based on this, we presented a polynomial-time approach for analyzing pattern-based constraint specifications. We have implemented a consistency checker based on these ideas and used it to carry out case studies on industrial-scale models. Overall, the consistency analysis presented in this paper complements the pattern-based constraint elicitation and specification process presented in [56].

Our case studies highlighted two substantial benefits of our approach over other approaches. First, our analysis offers immediate feedback to model developers, which is beneficial because development can proceed without significant delay. Second, it is currently the only approach that can handle large models with comprehensive constraint specifications.

Since our approach is limited to pattern-based constraint specifications, it can be used in a two-phase development process where pattern-based constraints are added to the model before the remaining constraints in OCL are added. This has the advantage that model developers can learn about the consistency of the majority of their constraints without invoking other analysis approaches, which are time-consuming or require theorem proving expertise.

We see several directions for future work. First, the conditions could be explored under which the different notions of consistency can be composed. For example, how is consistency affected if a strongly consistent and a class consistent model are merged? Second, the number of warnings in consistency analysis can be reduced by appropriately weakening the assumptions in the patterns' consistency theorems. For example, assumption (iii) of Lemma 1 can be weakened such that it does not exclude instances of Multiplicity Restriction with the operators $\leq$ or $<$. Third, our approach can be combined with existing approaches to increase the fraction of constraints that can be analyzed. For example, the consistency lemmas for the constraint patterns can be used to automate proofs in interactive analysis approaches. Fourth, normalizing the OCL formulas, e. g., based on the calculi presented in [7] or simplification rules [12,16,28], increases the number of UML/OCL specifications to which our pattern-based analysis approach is applicable. Fifth, if a constraint specification contains inconsistent constraints, the inconsistencies must be remedied [31]. It would be interesting to investigate whether constraint patterns can support remediation in an efficient and user-friendly way.

Acknowledgments

References

1. Jörg Ackermann and Klaus Turowski. A Library of OCL Specification Patterns to Simplify Behavioral Specification of Software Components. In *Proceedings of Conference on Advanced Information Systems Engineering.*, number 4001 in LNCS, pages 255–269, 2006.
2. Wolfgang Ahrendt, Thomas Baar, Bernhard Beckert, Richard Bubel, Martin Giese, Reiner Hähnle, Wolfram Menzel, Wojciech Mostowski, Andreas Roth, Steffen Schlager, and Peter H. Schmitt. The KeY Tool. *Software and System Modeling*, 4(1):32–54, 2005.
3. Bernhard K. Aichernig and Peter Gorm Larsenz. A Proof Obligation Generator for VDM-SL. In *FME '97: Industrial Applications and Strengthened Foundations of Formal Methods*, volume 1313 of *LNCS*, pages 338–357, 1997.

	Interactive Theorem Proving	Witness Creation	SAT and Model Checking	Pattern-Based Analysis
coverage	full UML/OCL	full UML/OCL	UML, subset of OCL	UML, subset of OCL
flexibility	high	high	low	medium
complexity	undecidable	undecidable	exponential	polynomial
full automation	no	no	yes	yes

Table 2 Comparison of consistency analysis approaches.

4. David Basin, Jürgen Doser, and Torsten Lodderstedt. Model Driven Security: from UML Models to Access Control Infrastructures. *ACM Transactions on Software Engineering and Methodology*, 15(1):39–91, January 2006.

5. Daniela Berardi, Diego Calvanese, and Giuseppe De Giacomo. Reasoning on UML Class Diagrams. *Artificial Intelligence*, 168(1):70–118, 2005.

6. Behzad Bordbar and Kyriakos Anastasakis. UML2Alloy: A Tool for Lightweight Modelling of Discrete Event Systems. In *Proceedings of IADIS International Conference in Applied Computing 2005*, pages 209–216, Algarve, Portugal, 2005.

7. Achim D. Brucker. *An Interactive Proof Environment for Object-oriented Specifications*. PhD Thesis, ETH Zurich, March 2007. ETH Dissertation No. 17097.

8. Achim D. Brucker, Jürgen Doser, and Burkhart Wolff. Semantic issues of OCL: Past, present, and future. *Electronic Communications of the EASST*, 5, 2006.

9. Achim D. Brucker and Burkhart Wolff. HOL-OCL – A Formal Proof Environment for UML/OCL. In José Fiadeiro and Paola Inverardi, editors, *Fundamental Approaches to Software Engineering (FASE08)*, number 4961 in Lecture Notes in Computer Science, pages 97–100. Springer-Verlag, Budapest, Hungary, 2008.

10. François Bry and Rainer Manthey. Checking Consistency of Database Constraints: a Logical Basis. In *VLDB '86: Proceedings of the 12th International Conference on Very Large Data Bases*, pages 13–20, San Francisco, CA, USA, 1986. Morgan Kaufmann Publishers Inc.

11. Jordi Cabot, Robert Clarisó, and Daniel Riera. UMLtoCSP: A Tool for the Formal Verification of UML/OCL Models Using Constraint Programming. In *ASE '07: Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering*, pages 547–548, New York, NY, USA, 2007. ACM.

12. Jordi Cabot and Ernest Teniente. Transformation Techniques for OCL Constraints. *Sci. Comput. Program.*, 68(3):179–195, 2007.

13. M. Cadoli, D. Calvanese, G. De Giacomo, and T. Mancini. Finite Model Reasoning on UML Class Diagrams Via Constraint Programming. *AI*IA 2007: Artificial Intelligence and Human-Oriented Computing*, pages 36–47, 2007.

14. Dan Chiorean, Mihai Paşca, Adrian Cârcu, Cristian Botiza, and Sorin Moldovan. Ensuring UML Models Consistency Using the OCL Environment. In *UML 2003 - Workshop: OCL 2.0 - Industry standard or scientific playground?*, 2003.

15. Edmund M. Clarke, Orna Grumberg, and Doron A. Peled. *Model Checking*. MIT Press, 2000.

16. Alexandre L. Correa and Cláudia Werner. Refactoring Object Constraint Language Specifications. *Software and System Modeling*, 6(2):113–138, 2007.

17. Dolors Costal, Cristina Gómez, Anna Queralt, Ruth Raventós, and Ernest Teniente. Facilitating the Definition of General Constraints in UML. In Oscar Nierstrasz, Jon Whittle, David Harel, and Gianna Reggio, editors, *MoDELS 2006*, number 4199 in LNCS, pages 260–274. Springer-Verlag, 2006.

18. Dolors Costal, Cristina Gómez, Anna Queralt, Ruth Raventós, and Ernest Teniente. Improving the Definition of General Constraints in UML. *Software and Systems Modeling*, 2008.

19. Flemming M. Damm, Bo Stig Hansen, and Hans Bruun. On Type Checking in VDM and Related Consistency Issues. In *4th International Symposium of VDM Europe on Formal Software Development - Volume I*, volume 551 of *LNCS*, pages 45–62, 1991.

20. Ádám Darvas and Peter Müller. Reasoning About Method Calls in Interface Specifications. *Journal of Object Technology*, 5:59–85, 2006.

21. James P. Davis and Ronald D. Bonnell. Propositional Logic Constraint Patterns and Their Use in UML-Based Conceptual Modeling and Analysis. *IEEE Transactions on Knowledge and Data Engineering*, 19(3):427–440, 2007.

22. Martin Davis, George Logemann, and Donald Loveland. A Machine Program for Theorem-proving. *Commun. ACM*, 5(7):394–397, 1962.

23. Dino Distefano, Joost-Pieter Katoen, and Arend Rensink. Towards Model Checking OCL. In *Proceedings of the ECOOP Workshop on Defining a Precise Semantics for UML*, 2000.

24. Matthew B. Dwyer, George S. Avrunin, and James C. Corbett. Property Specification Patterns for Finite-state Verification. In *FMSP '98: Proceedings of the second workshop on Formal methods in software practice*, pages 7–15, New York, NY, USA, 1998. ACM Press.

25. Wolfgang Emmerich, Anthony Finkelstein, and Christian Nentwich. Consistency Management with Repair Actions. In *Proceedings of the 25th International Conference on Software Engineering*, pages 455–464, Portland, Oregon, 2003. IEEE Computer Society.

26. Carles Farré, Ernest Teniente, and Toni Urpí. Checking Query Containment with the CQC Method. *Data & Knowledge Engineering*, 53(2):163–223, 2005.

27. Jean H. Gallier. *Logic for Computer Science: Foundations of Automatic Theorem Proving*. Harper & Row, 1986.

28. Martin Giese and Daniel Larsson. Simplifying Transformations of OCL Constraints. In Lionel C. Briand and Clay Williams, editors, *MoDELS*, volume 3713 of *Lecture Notes in Computer Science*, pages 309–323. Springer, 2005.

29. Martin Gogolla, Jørn Bohling, and Mark Richters. Validating UML and OCL Models in USE by Automatic Snapshot Generation. *Software and Systems Modeling*, 4(4):386–398, 2005.

30. Martin Gogolla and Mark Richters. Expressing UML Class Diagrams Properties with OCL. In *Object Modeling with the OCL, The Rationale behind the Object Constraint Language*, pages 85–114, London, UK, 2002. Springer-Verlag.

31. Sven Hartmann. Coping with Inconsistent Constraint Specifications. In Hideko S. Kunii, Sushil Jajodia, and Arne Sølvberg, editors, *ER*, volume 2224 of *Lecture Notes in Computer Science*, pages 241–255. Springer, 2001.

32. IBM. developerWorks. http://www.ibm.com/developerworks/, December 2007.

33. Daniel Jackson. Alloy: A Lightweight Object Modelling Notation. *ACM Transactions on Software Engineering and Methodology*, 11(2):256–290, 2002.

34. Daniel Jackson, Ian Schechter, and Ilya Shlyakhter. Alcoa: The Alloy Constraint Analyzer. *Proceedings of the International Conference on Software Engineering*, pages 730–733, 2000.

35. Viviane Jonckers, Tom Mens, Jocelyn Simmonds, and Ragnhild Van Der Straeten. Using Description Logic to Maintain Consistency between UML Models. In Perdita Stevens, Jon Whittle, and Grady Booch, editors, *UML*, volume 2863 of *Lecture Notes in Computer Science*, pages 326–340. Springer, 2003.

36. Cliff B. Jones. *Systematic Software Development using VDM*. Prentice Hall, 1990. ISBN 0-13-880733-7.

37. Günter Todt Jürgen-Michael Glubrecht, Arnold Oberschelp. *Klassenlogik*. Bibliographisches Institut, Mannheim/Wien/Zürich, 1983.

38. Ken Kaneiwa and Ken Satoh. Consistency Checking Algorithms for Restricted UML Class Diagrams. In Jürgen Dix and Stephen J. Hegner, editors, *FoIKS*, volume 3861 of *Lecture Notes in Computer Science*, pages 219–239. Springer, 2006.

39. Stuart Kent. Model Driven Engineering. In Michael J. Butler, Luigia Petre, and Kaisa Sere, editors, *IFM*, volume 2335 of *Lecture Notes in Computer Science*, pages 286–298. Springer, May 2002.

40. Anneke Kleppe and Jos Warmer. *The Object Constraint Language. Second Edition.* Addison-Wesley, 2003.

41. Jochen Küster. *Consistency Management of Object-Oriented Behavioral Models*. PhD Thesis, University of Paderborn, Jan. 2004.

42. Jochen M. Küster, Ksenia Ryndina, and Harald Gall. Generation of Business Process Models for Object Life Cycle Compliance. In *Proceedings of the 5th International Conference on Business Process Management (BPM)*, volume 4714 of *LNCS*, pages 165–181. Springer, 2007.

43. Marcel Kyas, Harald Fecher, Frank S. de Boer, Joost Jacob, Jozef Hooman, Mark van der Zwaag, Tamarah Arons, and Hillel Kugler. Formalizing UML Models and OCL Constraints in PVS. *Electronic Notes in Theoretical Computer Science*, 115:39–47, 2005.

44. Michael Leuschel and Michael J. Butler. ProB: An Automated Analysis Toolset for the B Method. *STTT*, 10(2):185–203, 2008.

45. Barbara H. Liskov and Jeannette M. Wing. A Behavioral Notion of Subtyping. *ACM Transactions on Programming Languages and Systems*, 16(6):1811–1841, 1994.

46. Azzam Maraee and Mira Balaban. Efficient Reasoning about Finite Satisfiability of UML Class Diagrams with Constrained Generalization Sets. In David H. Akehurst, Régis Vogel, and Richard F. Paige, editors, *ECMDA-FA*, volume 4530 of *Lecture Notes in Computer Science*, pages 17–31. Springer, 2007.

47. Elita Miliauskaitė and Lina Nemuraitė. Representation of Integrity Constraints in Conceptual Models. *Information Technology and Control*, 34(4):355–365, 2005.

48. Tobias Nipkow, Lawrence C. Paulson, and Markus Wenzel. *Isabelle/HOL - A Proof Assistant for Higher-Order Logic*. Number 2283 in Lecture notes in computer science. Springer-Verlag Berlin Heidelberg New York, 2002.

49. Object Management Group (OMG). UML 2.0 OCL Final Adopted Specification. http://www.omg.org/cgi-bin/apps/doc?ptc/03-10-14.pdf, 2003.

50. Object Management Group (OMG). Unified Modeling Language: Superstructure. Version 2.1. http://www.omg.org/cgi-bin/doc?ptc/2006-04-02, April 2006.

51. Sam Owre, S. Rajan, John M. Rushby, Natarajan Shankar, and M. Srivas. PVS: Combining Specification, Proof Checking, and Model Checking. *Computer-Aided Verification, CAV*, 96:411–414, 1996.

52. Anna Queralt and Ernest Teniente. Reasoning on UML Class Diagrams with OCL Constraints. In *Proceedings of the 25th International Conference on Conceptual Modeling (ER 2006)*, volume 4215 of *LNCS*, pages 497–512. Springer, 2006.

53. Anna Queralt and Ernest Teniente. Decidable Reasoning in UML Schemas with Constraints. In Zohra Bellahsene and Michel Léonard, editors, *CAiSE*, volume 5074 of *Lecture Notes in Computer Science*, pages 281–295. Springer, 2008.

54. Mehrdad Sabetzadeh, Shiva Nejati, Sotirios Liaskos, Steve Easterbrook, and Marsha Chechik. Consistency Checking of Conceptual Models via Model Merging. In *Proceedings of the 15th IEEE International Requirements Engineering Conference 2007*, 2007.

55. Colin F. Snook and Michael J. Butler. UML-B: Formal Modeling and Design Aided by UML. *ACM Trans. Softw. Eng. Methodol.*, 15(1):92–122, 2006.

56. Michael Wahler. *Model-Driven Software Development: Integrating Quality Assurance*, chapter A Pattern Approach to Increasing the Maturity Level of Class Models. Idea Group Inc., 2008.

57. Michael Wahler. *Using Patterns to Develop Consistent Design Constraints*. PhD thesis, ETH Zurich, Switzerland, February 2008. No. 17643.

58. Michael Wahler, Lee Ackerman, and Scott Schneider. Using the IBM Constraint Patterns and Consistency Analysis Extension. A Step by Step Guide. Online document. http://www.ibm.com/developerworks/edu/dw-r-conpatcon.html?S_TACT=105AGX15&S_CMP=EDU, May 2008.

59. Michael Wahler, Jana Koehler, and Achim D. Brucker. Model-Driven Constraint Engineering. *Electronic Communications of the EASST*, 5, 2006.