Consistency Checking of Conceptual Models via Model Merging

Mehrdad Sabetzadeh Shiva Nejati Sotirios Liaskos Steve Easterbrook Marsha Chechik
Department of Computer Science
University of Toronto, Toronto, ON, Canada.
{mehrdad,shiva,liaskos,sme,chechik}@cs.toronto.edu

Abstract

Requirements elicitation involves the construction of large sets of conceptual models. An important step in the analysis of these models is checking their consistency. Existing research largely focuses on checking consistency of individual models and of relationships between pairs of models. However, such strategy does not guarantee global consistency. In this paper, we propose a consistency checking approach that addresses this problem for homogeneous models. Given a set of models and a set of relationships between them, our approach works by first constructing a merged model and then verifying this model against the consistency constraints of interest. By keeping proper traceability information, consistency diagnostics obtained over the merge are projected back to the original models and their relationships. The paper also presents a set of reusable expressions for defining consistency constraints in conceptual modelling. We demonstrate the use of the developed expressions in the specification of consistency rules for class and ER diagrams, and i^* goal models.

1 Introduction

In model-based requirements elicitation, conceptual models are used to capture the needs and goals of the involved stakeholders. These models are often subject to certain sanity criteria, including well-formedness constraints imposed by the modelling notation being used as well as quality considerations for facilitating model understandability, maintenance, and evolution.

For example, in class diagrams, well-formedness constraints disallow cyclic inheritance. Cyclic dependencies, on the other hand, are not disallowed; but, their use may be discouraged, as a matter of good practice, to avoid coupling. An important problem in conceptual modelling is checking compliance with such constraints and generating proper feedback if a violation occurs. This problem is usually studied under the topic of *consistency checking* [10, 18]. Consistency is a very broad term and may have different interpretations in different contexts. In this paper, we concentrate on consistency from a *structural* perspective, using the term to refer to properties that models are expected to satisfy irrespective of their semantic content.

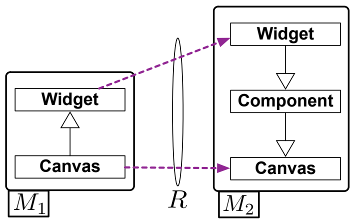


Fig. 1. Pairwise inconsistency.

Consistency checking is not restricted to isolated models. Conceptual modelling is a collaborative effort that may be distributed over different teams of people. These teams construct distinct but overlapping models representing different perspectives on the overall system. The overlaps between the models are typically described by *mappings* that equate corresponding model elements. A key aspect of consistency checking in distributed development is ensuring that mappings respect the constraints of interest. For example, consider the class diagrams M_1 and M_2, and their mapping R in Fig. 1. Although M_1 and M_2 are consistent individually, they are not pairwise consistent because the mapping defined between them gives rise to cyclic inheritance: Canvas is a descendant of Widget in M_1, while Widget is a descendant of Canvas in M_2. Hence, a mapping based on name equalities results in an inconsistency, indicating a modelling error or a terminological conflict between M_1 and M_2.

Existing techniques by and large focus on checking consistency of individual models and of individual mappings between model pairs (i.e. pairwise consistency). These techniques do not fully address the problem of consistency checking in distributed development because they do not provide a method for verifying *global* consistency properties [28, 18]. For example, Fig. 2 shows three models, M_1, M_2, M_3, related by three mappings, R_1, R_2, R_3. Each model and each mapping, when taken individually, satisfies the acyclic inheritance constraint. However, the system as a whole is inconsistent because the combination of R_1, R_2, and R_3 implies a loop in the inheritance chain.

This example underscores the need for a consistency

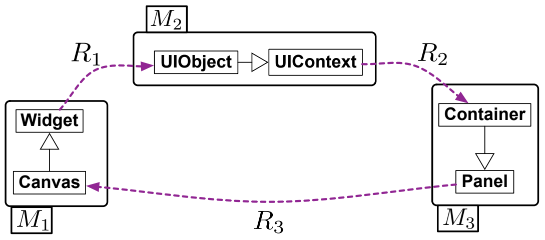


Fig. 2. Global inconsistency.

checking technique that can *simultaneously* use information from multiple models and mappings. Existing approaches such as xlinkit [17] work well for pairwise checking. However, since these approaches do not clearly separate consistency rules from model mappings, it becomes very difficult to generalize the rules beyond pairwise checking.

In this paper, we propose a consistency checking approach that addresses the above problem for homogeneous models, i.e. models described in the same notation. Our work is motivated by the observation that consistency checking of a set of (homogeneous) models can be done in a more general and succinct way if we first merge the models according to the mappings defined between them. The implementation of this approach requires a merge operator that is well-defined for *any* system of interrelated models even when they are *incomplete* or *inconsistent*. We developed such a merge operator in our earlier work [23] for graphical modelling notations. We use this operator as a basis for the consistency checking approach presented here.

For example, rather than trying to check the acyclic inheritance constraint on the configuration in Fig. 2, we construct a merge, shown in Fig. 3, and interpret the constraint over it. By keeping proper traceability information, we project the diagnostics obtained from consistency checking of the merge back to the originating models and mappings.

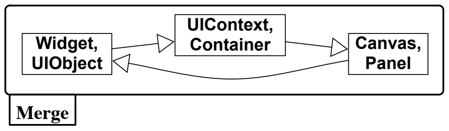


Fig. 3. Merge of the models in Fig. 2.

In addition to providing a solution for verifying global consistency properties, our approach has the advantage that it requires only a single consistency rule to be developed for each consistency constraint – the rule applies no matter how many models are involved and how they are interrelated with one another.

We provide an implementation of our approach within a logic-based constraint specification framework. To describe consistency rules, we use first-order relational calculus augmented with counting and transitive closure operations. This logic, on one hand, provides enough expressive power to characterize the main structural constraints of conceptual models, and yet is tractable enough to be applicable to large modelling problems. In our work, we use a highly optimized relational interpreter, CrocoPat [2], that implements a variant of this logic.

To simplify the specification of consistency rules, we develop a set of generic and reusable expressions capturing recurrent patterns across the constraints of different modelling notations. We demonstrate the usefulness of these expressions for describing constraints over class and Entity-Relationship (ER) diagrams, and i^* goal models [31].

This paper is organized as follows: Section 2 provides background information. Section 3 discusses our generic expressions for specifying consistency rules and illustrates their use. Section 4 presents our consistency checking approach for distributed conceptual models. Section 5 describes the implementation of our approach and reports on experimental results. Section 6 compares our work to related research. Section 7 summarizes the paper and outlines directions for future work.

2 Background

2.1 Graph-Based Modelling

The work described in this paper focuses on conceptual modelling formalisms with graphical notations. In this section, we introduce a general notion of graph adopted from algebraic approaches to graph-based modelling. This notion captures various formalisms including class and ER diagrams, goal models, state machines, and Petri Nets [22].

Definition (graph) *A* (directed) graph *is a tuple $G = (N, E, \mathsf{Src}_G, \mathsf{Tgt}_G)$ where N is a set of nodes, E is a set of edges, and $\mathsf{Src}_G, \mathsf{Tgt}_G : E \to N$ are functions respectively giving the source and the target of each edge.*

Every element i (i.e. node or edge) is assumed to have an implicit and immutable attribute, called uid, that uniquely identifies i across a modelling project. We write i.uid to refer to the value of i's uid. To capture properties such as label, stereotype, colour, order, etc., we attach to every element a set of $(\text{Property}, \text{value})$ tuples.

Graph-based models typically have typed elements. A standard way for assigning types is to parameterize models with a metamodel graph [4]. Assuming that the metamodel for the chosen modelling language is given by a graph $\mathcal{M}$, every model $G = (N, E, \mathsf{Src}_G, \mathsf{Tgt}_G)$ is equipped with a type function $t : G \to \mathcal{M}$, mapping each element of G to an element of $\mathcal{M}$. The type function respects the structure of G: If t maps an edge e of G to an edge u of $\mathcal{M}$, the endpoints of e are respectively mapped to those of u, as illustrated in Fig. 4. In this figure, $\mathcal{M}$ is the extends–implements fragment of the metamodel for Java class diagrams.

2.2 Relational Specification

We use the Relational Manipulation Language (RML) [2] for describing consistency rules. RML provides the expressive power of first-order logic with transitive closure

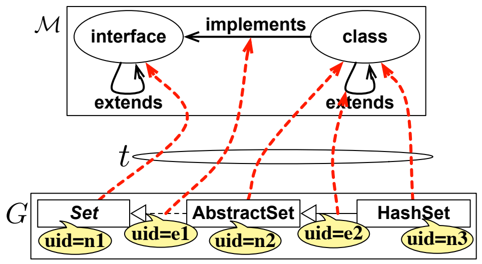


Fig. 4. Example of typed graphs.

stmt	::=	rel_var(*term*, ...) ; |
		rel_var(*term*, ...) := *rel_expr* ; |
		IF *rel_expr* { *stmt* ... } ELSE {*stmt* ... } |
		FOR var IN *rel_expr* {*stmt* ... } |
		PRINT *print_expr*;
rel_expr	::=	rel_var(*term*, ...) | *rel_expr* op_1 *rel_expr* |
		FA(var, *rel_expr*) | /* $\forall$ quantification */
		EX(var, *rel_expr*) | /* $\exists$ quantification */
		TC(*rel_expr*) | /* transitive closure */
		!*rel_expr* | /* negation */
		rel_expr op_2 *rel_expr* | *num_expr* op_2 *num_expr*
num_expr	::=	num_literal | *num_expr* op_3 *num_expr* |
		#(*rel_expr*) /* counting */
print_expr	::=	*rel_expr* | *term*
term	::=	var | str_literal | STRING(*num_expr*)

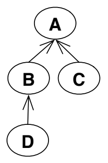
(1) op_1 can be one of: '|' (or), '&' (and), '–>' (implies).
(2) op_2 can be one of: '=', '!=', '<', '>'.
(3) op_3 can be one of: '+', '-', '*', '/'.

Fig. 5. Partial grammar of RML.

and counting operators. These operators are key to specifying rules involving reachability and multiplicity.

While our consistency checking framework is not tied to a particular constraint language, there are two factors that make RML particularly appealing: (1) RML's domain-independent and easy-to-use syntax and (2) its efficient interpreter, CrocoPat [2]. CrocoPat encodes relational predicates as Binary Decision Diagrams (BDDs) [3] which are compact data structures for representing and manipulating relations. The use of BDDs makes CrocoPat highly scalable in terms of both time and memory.

RML has an imperative style of execution and runs programs statement by statement. A partial and slightly simplified grammar of the language is shown in Fig. 5. The complete grammar can be found in [1]. For example, consider the graph shown at right, and assume that the relation E(x,y) denotes "there is an edge from x to y". To check if there exists a node without any outgoing edges, we use the existential and universal quantifiers, EX and FA in Fig. 5, to define the expression EX(x, FA(y, !E(x, y))). This expression holds over our example graph, witnessed by node A. For another example, suppose we want to count the num-



Algorithm. GraphToRml
Input: Graph $G = (N, E, \mathsf{Src}_G, \mathsf{Tgt}_G)$, and type function $t : G \to \mathcal{M}$.
Output: A set of RML statements.

```
1:  for every node and edge $i$ in $G$ :
2:    if $i$ is a node:
3:      output Node($i$.uid)
4:    else :    /* $i$ is an edge */
5:      output Edge($i$.uid)
6:      output Src($i$.uid, $\mathsf{Src}_G(i)$.uid)
7:      output Tgt($i$.uid, $\mathsf{Tgt}_G(i)$.uid)
      /* Translate type information */
8:    output Type($i$.uid, $t(i)$)
      /* Translate properties */
9:    for every property $(\mathrm{Property}_k, \mathrm{val}_k)$ of $i$ :
10:     if $\mathrm{val}_k$ is boolean :
11:       if $\mathrm{val}_k = true$ : output $\mathsf{Property}_k$($i$.uid)
12:     else : output $\mathsf{Property}_k$($i$.uid, $\mathrm{val}_k$)
```

Fig. 6. GraphToRml algorithm.

```
Node("n1");                  Node("n3");                 Type("e1", "implements");
Type("n1", "interface");     Type("n3", "class");        Edge("e2");
Label("n1", "Set");          Label("n3", "HashSet");     Src("e2", "n3");
Node("n2");                  Edge("e1");                 Tgt("e2", "n2");
Type("n2", "class");         Src("e1", "n2");            Type("e2", "extends");
Label("n2", "AbstractSet");  Tgt("e1", "n1");
```

Fig. 7. RML encoding of the model in Fig. 4.

ber of predecessors of A. The expression #(E(x, "A")), for a free variable x, returns the number of all assignments ℓ to x for which E(ℓ, "A") holds. Hence, the expression evaluates to 2. The following program prints to the standard output all these satisfying assignments, i.e. B and C:

```
FOR n IN E(x, "A") {
    PRINT n;
}
```

Note that the relation after IN must have one free variable, here x. For a last example, suppose we want to define a relation Reachable(x, y) that holds iff "there is a path from x to y". This is done by the following statement:

```
Reachable(x, y) := TC(E(x, y));
```

where TC denotes the transitive closure operator. The semantics of TC for a relation E with two free variables is described recursively as follows: $\mathsf{TC}(E(x,y)) \equiv E(x,y) \vee \exists z.E(x,z) \wedge \mathsf{TC}(E(z,y))$.

2.3 Translating Graphs to Relations

To evaluate relational expressions over a model, we translate it into a set of predicates. Fig. 6 provides an algorithm, GraphToRml, for translating a graphical model, as defined in Section 2.1, to RML statements. Fig. 7 shows the result of translation for the class diagram in Fig. 4.

3 Generic Consistency Checking Expressions

In this section, we provide a set of generic expressions for specifying structural consistency constraints in conceptual modelling. These expressions capture a number of re-

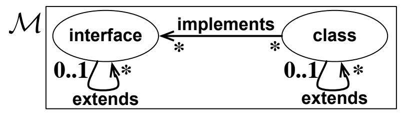


Fig. 8. Example of multiplicity annotations.

curring patterns that we observed in the consistency constraints of class diagrams, e.g. [26, 9], goal models [12, 15], and database schemata [24].

Table 1 shows a list of representative consistency constraints for the studied notations. Some of these constraints (e.g., **C1**, **C4**, and **C6**) capture fundamental well-formedness criteria, while others (e.g., **C2**, **C5**, and **G1**) describe desirable model qualities. The table provides an implementation of the constraints in RML with occurrences of our generic expressions bolded and underlined. In the remainder of this section, we discuss these expressions under the following three headings:

- ***Compatibility expressions***, used for ensuring compatibility of the type of an edge with the types of its endpoints.
- ***Multiplicity expressions***, used for defining a minimum and a maximum number for edges of a given type incident to a node.
- ***Reachability expressions***, used for checking existence of paths of edges of a given type between two nodes.

Compatibility expressions. Compatibility constraints constitute the most primitive class of well-formedness criteria for conceptual models. In Section 2.1, we described a generic mechanism for capturing these constraints through a structure-preserving type function t (e.g. see Fig. 4). The mechanism can be easily expressed in logical terms. To verify that the source and the target of an edge are respectively of types β and γ, we define Compatible$_{\beta,\gamma}$(x) that holds for all edges x satisfying the compatibility constraint:

```
Compatible$_{\beta,\gamma}$(x) := FA(n, FA(m, Src(x, n) & Tgt(x, m) –>
                                    Type(n, $\beta$) & Type(m, $\gamma$)));
```

Since compatibility constraints are essential to the integrity of models and mappings, they are typically enforced at design time, and respected by subsequent manipulations such as model merging. There are numerous instances of compatibility constraints in the notations we studied. Since these constraints are very similar, we show only one, namely, **C1**, in Table 1.

Multiplicity expressions. Multiplicity constraints are often specified using annotations over the metamodel graph. Fig. 8 shows the metamodel of Fig. 4 annotated with multiplicity constraints. For example, consider the extends self-loop incident to the class node. According to the annotations of this edge, a class can extend at most one (i.e. 0..1) class, but each class can be extended by several (i.e. $*$) classes.

Given a multiplicity-annotated metamodel $\mathcal{M}$, we produce a set of logical expressions for validating conformance of an instance model to the multiplicity constraints prescribed by $\mathcal{M}$. Each edge in $\mathcal{M}$ gives rise to two multiplicity expressions – one for each endpoint. We consider multiplicity annotations of four kinds: "k", "$k..l$", "$k..*$", and "$*$", where k and l are constants. An RML program for checking a multiplicity annotation "$k..l$" attached to the source side of a metamodel edge α is as follows:

```
SourceMultiplicity$^{kl}_{\alpha}$(x):= FALSE(x);
FOR n IN Node(v) { /* v is a dummy free variable */
        IF ( $k \leq$ #(Src(e, n) & Type(e, $\alpha$)) $\leq l$) {
            SourceMultiplicity$^{kl}_{\alpha}$(x):= (x=n) | SourceMultiplicity$^{kl}_{\alpha}$(x);
        }
}
```

The program initializes SourceMultiplicity$^{kl}_{\alpha}$(x) to an empty unary relation, FALSE(x). If a node n respects the multiplicity constraint, it gets added to the relation SourceMultiplicity$^{kl}_{\alpha}$(x). The multiplicity expression SourceMultiplicity$^{\square}_{\alpha}$(x), where $\square$ is "k" or "$k..*$", can be implemented similarly[1]. Implementation of a multiplicity constraint attached to the target side of a metamodel edge is done by replacing Src(e, n) with Tgt(e,n) in the above program. In Table 1, constraints **C3**, **C6**, **C8**, and **G2** use multiplicity expressions.

Another useful expression similar to multiplicity expressions is for detecting parallel edges of a given type. This is implemented by ParallelEdges$_{\alpha}$(x, y) shown below:

```
ParallelEdges$_{\alpha}$(x, y) := FALSE(x, y);
FOR n IN Node(v) { /* v is a dummy free variable */
    FOR m IN Node(w) { /* w is a dummy free variable */
        IF (#(Src(e, n) & Tgt(e, m), & Type(e, $\alpha$)) > 1) {
            ParallelEdges$_{\alpha}$(x, y):= ((x=n) & (y=m)) | ParallelEdges$_{\alpha}$(x, y);
        }
    }
}
```

This program initializes ParallelEdges$_{\alpha}$(x, y) to an empty binary relation, FALSE(x, y), and then, adds to it all pairs (n, m) of nodes between which there are parallel edges. An example constraint using ParallelEdges$_{\alpha}$(x, y) is **G4** in Table 1.

Reachability expressions. Several consistency constraints involve finding nodes that are reachable or unreachable via edges of a certain type. For example, in goal modelling, we may want to ensure that all goals are reachable via goal decomposition edges. In UML class diagrams, we may want to check that all descendants (via subclassing edges) of a given class have a certain property. For an edge of type α, we define a relation Reachable$_{\alpha}$(x, y) that holds iff a path from x to y made up of α-edges exists:

```
E$_{\alpha}$(x, y) := EX(e, Src(e, x) & Tgt(e, y) & Type(e, $\alpha$));
Reachable$_{\alpha}$(x, y):= TC(E$_{\alpha}$(x, y));
```

For example, in the class diagram of Fig. 10, Reachable$_{\text{extends}}$("E","A") holds, indicating that E reaches A via extends edges.

A special case of reachability analysis is cycle detection. Cycles of edges of certain types can be indicative of a mod-

[1]No expression is needed for the "$*$" multiplicity annotation.

Lang.	Textual constraint	RML constraint
Class/ER Diagrams	**C1.** An implements edge relates a class to an interface [26]	**C1**() := FA(e, Type(e, ”implements”) –> **Compatible**$_{\text{class,interface}}$(e));
	C2. Every abstract class has a concrete implementation [9]	**C2**() :=FA(c1, Type(c1, ”class”) & Abstract(c1) –> EX(c2, (Concrete(c2) & **Reachable**$_{\text{extends}}$(c2, c1))));
	C3. A class does not extend more than one class [9]	**C3**() := FA(c, Type(c, ”class”) –> **SourceMultiplicity**$^{01}_{\text{extends}}$(c));
	C4. Inheritance is acyclic [26]	**C4**():= FA(c, Type(c, ”class”) –> !**OnCycle**$_{\text{extends}}$(c));
	C5. All classes are reachable from a root class [9]	**C5**() := FA(c1, Type(c1, ”class”) –> EX(c2, Type(c2, ”class”) & IsRoot(c2) & **Reachable**$_{\text{extends}}$(c1, c2)));
	C6. Final classes do not have subclasses [9]	**C6**() := FA(c, Type(c, ”class”) & Final(c) –> **TargetMultiplicity**$^{0}_{\text{extends}}$(c));
	C7. Inheritance is redundancy-free [9]	**C7**() := FA(c1, Type(c1, ”class”) & FA(c2, Type(c2, ”class”) –> !**RedundantPaths**$_{\text{extends}}$(c1, c2)));
	C8. Every entity has a unique key [24]	**C8**() := FA(e, Type(e, ”entity”) –> **SourceMultiplicity**$^{1}_{\text{key\_link}}$(e));
i Goal Models*	**G1.** Goal dependencies are acyclic [12]	**G1**() := FA(g, Type(g, ”goal”) –> !**OnCycle**$_{\text{depends}}$(g));
	G2. A resource does not have multiple dependers [12]	**G2**() := FA(r, Type(r, ”resource”) –> **SourceMultiplicity**$^{01}_{\text{depends}}$(r));
	G3. Each loop in a goal decomposition tree includes at least one OR-decomposition [15]	**G3**() := FA(g1, Type(g1, ”goal”) –> !**OnCycle**$_{\text{decomposes}}$(g1) | EX(g2, Type(g2, ”goal”) & ORNode(g2) & **ReachVia**$_{\text{decomposes}}$(g1, g2, g1)));
	G4. Parallel contribution links do not exist [15]	**G4**() := FA(g1, Type(g1, ”goal”) & FA(g2, Type(g2, ”goal”) –> !**ParallelEdges**$_{\text{contributes}}$(g1, g2)));
	G5. Goal fulfillment cannot precede subgoal fulfillment [15]	**G5**() := FA(g1, Type(g1, ”goal”) & FA(g2, Type(g2, ”goal”) & **Reachable**$_{\text{decomposes}}$(g1, g2) –> !E$_{\text{precedes}}$(g2, g1)));

Table 1. Examples of well-formedness and quality constraints in different notations.

elling problem. In class diagrams, for example, inheritance can never be cyclic. For an (edge) type α, the following RML statement creates a relation OnCycle$_\alpha$(x) that holds for all nodes x residing on a cycle of α-edges:

```
OnCycleα(x) := Reachableα(x, x);
```

Reachability is also used for detecting path redundancies. For example, if we have three classes A, B, C, such that C extends B and B extends A, it would be redundant to have an extends edge from C to A because this is already implied by the path C $\rightarrow$ B $\rightarrow$ A. Existence of multiple paths of edges of the same type between two nodes can be captured by RedundantPaths$_\alpha$(x, y) defined as follows:

```
ReachViaα(x, z, y) := (Reachableα(x, z) | (z = x)) &
                      Reachableα(z, y) ;
DistinctPathEndsα(x, y) := EX(v, EX(z, ReachViaα(x, v, y) &
                      ReachViaα(x, z, y) & Eα(z, y) &
                      Eα(v, y) & !(z = v)));
RedundantPathsα(x, y) := EX(z, (ReachViaα(x, z, y) | (y = z)) &
                      DistinctPathEndsα(x, z)) | ParallelEdgesα(x, y);
```

ReachVia$_\alpha$(x, z, y) holds iff there is a path (of length ≥ 1) from x to y passing through z. DistinctPathEnds$_\alpha$(x, y) holds iff there are paths from x to y whose final edges (to y) are different. And, RedundantPaths$_\alpha$(x, y) holds iff there are distinct paths or parallel edges from x to y.

As evidenced by Table 1, expressions involving variants of reachability are very common. In particular, **C2**, **C5**, and **G5** require checking reachability in its general form; **C4**, **G1**, and **G3** require checking cyclicity; and **C7** requires checking redundancy.

4 Consistency Checking of Distributed Models

In Section 3, we developed a generic platform for expressing consistency constraints over individual models. We now show how this generalizes to distributed models.

A common approach for extending consistency checks to distributed models is to write consistency rules for the mappings between models (e.g. [7, 17]). For example, if we have a mapping R that equates elements of two models M_1 and M_2, we may wish to check that the mapping does not introduce cycles. This can be achieved by checking that each model individually is acyclic, and writing a new rule to check the mapping:

```
MCycleα(x,y) := R(x, y) & EX(z, EX(t, R(z, t) &
                Reachableα(x, z) & Reachableα(t, y)));
```

If we apply this rule to M_1 and M_2, the relation MCycle$_\alpha$ will hold for all pairs (x,y) of mapped elements that give rise to a cycle across the two models[2].

This approach is cumbersome for several reasons. First, it requires many new consistency rules: each existing consistency constraint (for a single model) may need to be rewritten to take account of each type of mapping that can hold between models. It also introduces an undesirable coupling between consistency rules and model mappings. Consistency rules will refer to the possible mappings between models, and model mappings must be checked for their impact on the consistency rules.

[2]Formally, MCycle$_\alpha$ is applied to the *disjoint union* of M_1 and M_2.

Second, this approach does not easily generalize beyond pairwise checking. This is because global consistency rules must consider the interactions between different mappings in the system. For example, a global rule to check for cyclic inheritance in a set of models such as those of Fig. 2 would need to refer to all of the mappings between the models in the scope of a single rule. This makes the specification of global consistency rules very complex.

Below, we present an alternative approach that does not suffer from these problems. Given a set of models and a set of mappings between them, we first construct a merged model using the merge operator developed in our earlier work [23]. We then check the consistency of this merged model and project the resulting diagnostics back to the original models and mappings using the traceability information generated during merge. This approach exploits the fact that our merge operator works even when the models to be merged are incomplete and/or inconsistent: the merge is well-defined for *any* set of models and mappings. Further, it is fully automatic; hence, users need not understand the merge process to use the results of consistency checking.

4.1 Model Merging

We first briefly review our merge operator. For more information, see [23]. The operator hinges on three abstractions: *models*, *mappings*, and *interconnection diagrams*. Each model is described as a graph, and each mapping as a binary relation over two models equating their corresponding elements. Mappings preserve type information, i.e. they do not equate elements that have different types. Further, they preserve structure, i.e. if a mapping R maps an edge e to an edge e', it must also map the source and target of e to the source and target of e', respectively.

The third abstraction, the interconnection diagram, captures a set of models and a set of known or hypothesized mappings between them. Two examples of interconnection diagrams are in Figs. 1 and 2. A more complex example with four models, $M_1, \ldots, M_4$, and four mappings, $R_1, \ldots, R_4$, is shown in Fig. 9. For convenience, we used a consistent vocabulary for naming the elements of $M_1, \ldots, M_4$, hence defining $R_1, \ldots, R_4$ based on name equalities. In general, models may not have a common vocabulary, and mappings are not necessarily based on vocabulary similarities [23].

The input to the merge algorithm is an interconnection diagram $D = \langle M_1, \ldots, M_i, R_1, \ldots, R_j \rangle$. The algorithm works by unifying elements in $M_1, \ldots, M_i$ that fall into the same equivalence group induced by $R_1, \ldots, R_j$. As an example, we have delineated by thin dashed lines one of the several equivalence groups in Fig. 9. Note that each unmapped element in the input models falls into a distinct equivalence group of its own. The merged model has exactly one element corresponding to each equivalence group.

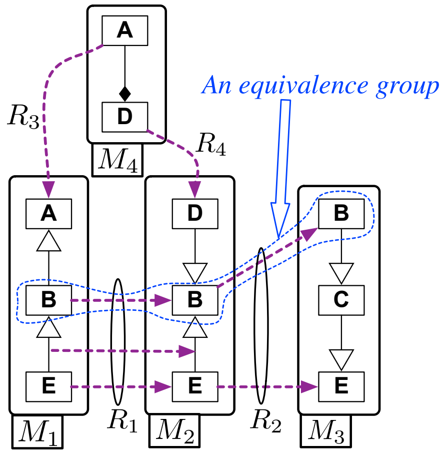


Fig. 9. Example interconnection diagram.

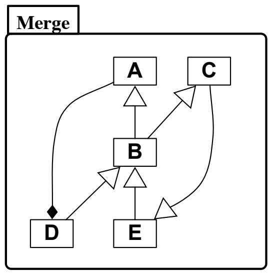


Fig. 10. Merge of the diagram in Fig. 9.

Since mappings denote equality of mapped element pairs and hence are symmetric, the directionality of mappings is ignored in the computation of equivalence groups.

The set of property tuples of every element i in the merged model is computed by taking the union of the property tuples of the elements represented by i, noting that i does not inherit the uids of these elements, but instead has a new uid of its own. Fig. 10 shows the resulting merge for the interconnection diagram in Fig. 9.

To support traceability back to the originating sources, for each element of the merged model, we store a link to the equivalence group inducing it. For example, for class B in the merge, we store the link "B@$M_1 \xrightarrow{R_1}$B@$M_2 \xrightarrow{R_2}$B@M_3", indicating that this class is the result of unifying all classes named B in models M_1, M_2, and M_3. Further, the link shows the mappings involved in the unification, namely, R_1 and R_2. In practice, traceability links are based on the uids of model elements, rather than their names. However, we ignore this technicality in our presentation. Fig. 11 shows the traceability information stored for the nodes of the merged model in Fig. 10. Similar traceability information is stored for the edges (not shown).

Two inconsistencies can be identified in the merge shown in Fig. 10: (1) B has two parents; (2) B, C, E form a cycle. These inconsistencies, as we discuss in Section 4.2, can be projected back to the original models and mappings using the traceability information produced during merge.

A:	A@$M_1 \xleftarrow{R_3}$ A@M_4
B:	B@$M_1 \xrightarrow{R_1}$ B@$M_2 \xrightarrow{R_2}$ B@M_3
C:	C@M_3
D:	D@$M_2 \xleftarrow{R_4}$ D@M_4
E:	E@$M_1 \xrightarrow{R_1}$ E@$M_2 \xrightarrow{R_2}$ E@M_3

Fig. 11. Traceability information for nodes.

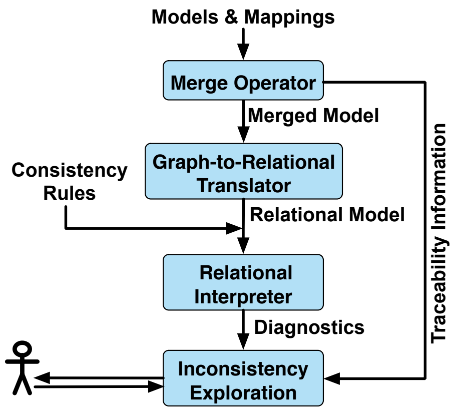


Fig. 12. Overview of consistency checking.

4.2 Consistency Checking via Merge

An overview of our consistency checking approach is shown in Fig. 12. Given a set of models and a set of mappings between them, we begin by constructing a merged model as described in Section 4.1. This model is translated into a set of relational predicates using the algorithm in Fig. 6. The result, along with the consistency rules of interest, is sent to a relational interpreter for consistency checking and producing diagnostics for any inconsistencies found. Users can then explore these diagnostics and project them back to the source models and mappings by utilizing the traceability data produced during the merge operation.

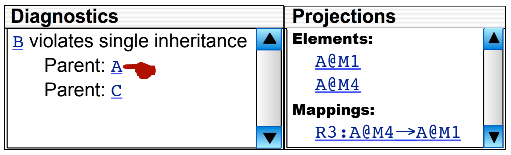
To obtain useful diagnostics from the relational interpreter, we instrument each consistency constraint with appropriate messages. For example, to report the classes that violate single inheritance, we instrument SourceMultiplicity$^{01}_{\text{extends}}$(x) (see Section 3) as follows:

```
FOR n IN !SourceMultiplicity$^{01}_{\text{extends}}$(x) {
        PRINT n, " violates single inheritance", ENDL;
        FOR e IN (Src(y, n) & Type(y, "extends")) {
              Parent(x) := Tgt(e, x);
              PRINT ["       Parent: "] Parent(x);
        }
}
```

The above instrumentation code, when executed over the merged model of Fig. 10, generates the following:

```
B violates single inheritance
        Parent:  A
        Parent:  C
```

This transcript, in addition to identifying B as an inconsistent element, provides *context* information about the inconsistency. Hence, we can immediately know that it is the relationship of B with A and C that causes B to fail the check. The nature of context information varies depending on the constraint in question. For example, to describe the context in which B violates acyclic inheritance, we need to include the path B$\rightarrow$C$\rightarrow$E$\rightarrow$B in the diagnostics (see Fig. 10). It is this varying nature of context information that necessitates custom instrumentation code for different constraints.



B violates single inheritance
Parent: A
Parent: C

A@M1
A@M4

R3:A@M4→A@M1

Fig. 13. Linking diagnostics with traceability data.

The instrumentation code for a consistency constraint can be easily enhanced to generate *hyperlinked* diagnostics. This makes it possible to link the diagnostics generated over a merged model to the source models and mappings using the traceability data provided by the merge operator. For example, Fig. 13 depicts a hyperlinked version of the diagnostics discussed earlier. Clicking on a link in the Diagnostics window retrieves the traceability data for the selected element and displays it in the Projections window. Fig. 13 shows the result of clicking on class A. The projections are also in a hyperlinked format, allowing users to navigate to the source models and mappings.

Note that the projections include *all* available information about the origins of the selected element. This information may not be minimal, i.e. not all models and mappings appearing in the projections are necessarily responsible for the occurrence of the inconsistency in question. For example, model M_4 and mappings R_3, R_4 do not play a role in the violation of single inheritance – the violation would occur even if we removed M_4, R_3, and R_4 from the interconnection diagram in Fig. 9. However, as shown in Fig. 13, M_4 and R_3 appear in the projections for A.

For a simple interconnection diagram like the one in Fig. 9, it may be reasonable to repeat the merge with different subsets of $M_1, \ldots, M_4$ and $R_1, \ldots, R_4$, and identify the minimal configuration that can produce a particular violation. This is, however, exponential in the number of models and mappings in the interconnection diagram, and hence not scalable to systems with many interrelated models. Scalable solutions may be found for certain constraints and certain patterns of interconnection diagrams, but we have not explored this in our work yet.

On the other hand, filtering out information that is seemingly irrelevant to the occurrence of an inconsistency may not be always desirable. For example, although model M_4 and mapping R_3 are not responsible for the violation of single inheritance, they may still be involved in a resolution of the problem. For example, a possible resolution is to create a new mapping between M_4 and M_3 and unify A and C. Deducing this resolution needs the knowledge that R_3 unifies

		# elements (nodes + edges)			
		500	1,000	5,000	10,000
Consistency Rule	Dangling Edges	< 1 sec	< 1 sec	5 sec	10 sec
	Parallel Edges	< 1 sec	1 sec	15 sec	57 sec
	Type Violations	< 1 sec	< 1 sec	4 sec	11 sec
	Multiple Inheritance	< 1 sec	< 1 sec	7 sec	24 sec
	Cyclic Inheritance	1 sec	3 sec	1 min 6 sec	4 min 48 sec

Table 2. Consistency checking running times.

A in M_4 and A in M_1, and filtering out M_4 and R_3 from the diagnostics would effectively eliminate this alternative.

5 Preliminary Evaluation

In this section, we discuss tool support and provide initial evidence for the usefulness of our approach.

Tool Support. We have implemented our approach as an extension to our prototype tool, TReMer [27], which provides an integrated environment for model construction, mapping, and merging. TReMer currently supports state machines, ER diagrams, and simple UML domain models. In the future, we plan to extend TReMer to support other notations, such as goal and class diagrams. Our consistency checking extension allows users to check a system of interrelated models against a given set of consistency constraints using the process shown in Fig. 12. Consistency constraints are verified by the CrocoPat relational interpreter [2]. Our tool is available for download from [27].

Computational Scalability. To validate computational scalability, we need to ensure that both model merging and consistency checking are scalable. The complexity of our merge algorithm is linear in the size of the input models and in the number of the input mappings [23]. This is dominated by the complexity of consistency checking for most interesting consistency constraints. We used CrocoPat for checking constraints of varying complexity over UML domain models with 500 to 10,000 elements. These were structurally realistic models assembled from smaller real-world models. We introduced inconsistencies of various kinds into these models so that about 10% of the elements in each model appeared in the results of inconsistency analysis. Table 2 shows the running times for a number of checks on a Linux PC with a 2.8 GHz Pentium CPU and 1 GB of memory. The reported times include finding the inconsistencies and generating proper diagnostics for them. The results indicate that the method is scalable to handle realistically large requirements modelling problems.

Case Study. We motivated our work by two main improvements it brings to distributed development: (1) Eliminating the need to have separate rules for checking consistency of models and consistency of mappings; and (2) Generalization from pairwise consistency checking to global consistency checking where the interactions between different mappings in the system are also considered.

Writing consistency rules is typically a laborious task; hence, the first improvement increases productivity by requiring the development of just a single rule for each consistency constraint. To evaluate the practical utility of the second improvement, we conducted an exploratory study aimed at investigating how global consistency checking could facilitate the analysis of relationships between distributed models. We based our study on models developed by students as an assignment in a recent offering of a senior undergraduate course on object-oriented analysis and design. To ensure privacy, these models were anonymized by a third party prior to our study.

The assignment had the students write a UML domain model for a hospital based on a short and intentionally ambiguous textual description. We studied five models developed independently by five individual students. These models were roughly equal in size, each with 60 to 70 elements; however, there were remarkable discrepancies in the way the models were structured. Other studies suggest that such discrepancies are quite common when models are developed independently [25, 8].

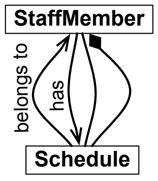
The ultimate goals of our study were: (1) to construct a coherent set of mappings to express the overlaps between the studied models; and (2) to systematically explore how these models differed from one another. To achieve these, we began by hypothesizing a set of initial mappings between the models. We then employed global consistency checking as a way to discover anomalies in these mappings, and later to investigate the differences between the models with respect to the mappings between them. Below, we briefly describe our findings and highlight the advantages of global consistency checking over pairwise consistency checking. Full details of our study are available from [27].

First, we automatically constructed a merge based on the initial mappings defined between the models. Consistency checking of this merge revealed several potential anomalies. In particular, the merge had 3 sets of identically-named concepts and 8 sets of parallel links (edges). All of these anomalies were due to the unstated overlaps between the models, which manifested themselves as duplicate elements in the merge. Note that, although not observed in our study, the anomalies could have had other causes. For example, some identically-named concepts could have been homonyms, and some parallel links could in fact have been necessary to distinguish between the different link roles.

The generated inconsistency diagnostics along with the traceability data stored for the merge allowed us to quickly identify the origins of duplicate elements and unify them by defining new correspondences. If we wanted to do this by pairwise checking of the five source models, we would have needed to check $(5 * 4)/2 = 10$ individual mappings between model pairs. Constructing a merge and checking global consistency made it possible to perform this task in a single shot. Further, for global consistency checking, we did not have to state all mappings between model pairs, because merge automatically results in transitive mappings. For example, having a mapping that equates Doctor in model M_1 and Physician in M_2, and a mapping that equates Physician in M_2 and MD in M_3, automatically maps Doctor to MD.

After refining the initial mappings with the newly discovered correspondences, we concentrated on analyzing structural discrepancies between the source models. These discrepancies were primarily due to the use of competing alternatives for capturing the relationships between the concepts in the domain. For example, to relate the StaffMember and Schedule concepts, one could choose among several alternatives, e.g. (1) an unlabelled undirected association, (2) a labelled directed association either saying "StaffMember *has a* Schedule" or "Schedule *belongs to* StaffMember", and (3) a composition link expressing a containment relation between StaffMember and Schedule.

Model merging provided a convenient way to bring together and visualize the alternatives used in different models for relating concept pairs. As an example, the figure on the right shows the relevant fragment of the merge for the StaffMember–Schedule pair. To detect and enumerate alternative choices for relating concept pairs, we developed a variant of the parallel edges rule (see Section 3), which ignored link types and directionality. Checking the merged model against this rule yielded 19 groups of links. Each group captured the set of alternatives proposed in different models for relating two specific concepts. The groups referenced a total of 70 links in the source models. Being able to simultaneously view *all* proposed alternatives for relating two concepts is crucial for resolving conflicts and building consensus between the source models. Pairwise checking would have allowed us to deal with only two alternatives at a time.



Finally, we re-examined the source models to apply the knowledge gained from our analysis, by deleting incorrect elements from these models. At this step, global consistency checking provided us with quick feedback on the impact of decisions made over one model on other models. For example, if we decided to delete a concept C from some model, we could automatically check all other models which had a concept C' equated with C, to verify that no (non-deleted) links were incident to C' – such links would become dangling if we propagated the deletion of C to C'. Employing pairwise checking for performing such sanity checks after a change can be costly or even ineffective when more complex sanity criteria are involved.

In summary, constructing merged models and checking global consistency allowed us to do various types of analysis that would be either expensive or impossible to do by pairwise checking. The traceability information generated during the merge operation made it possible to project inconsistencies back to the originating models and mappings, and take steps to resolve them. Since our merge process is fully automatic, we did not incur overhead costs for generalizing from pairwise to global consistency checking.

6 Related Work

Generic constraint expressions. Developing generic expressions for describing correctness properties of models is not a new idea. For example, [5] provides templates for capturing temporal properties of systems. However, these templates are specifically for behavioural models, and are inapplicable to non-behavioural ones such as class diagrams. The closest work to ours is that of [30], which describes a set of constraint patterns for UML models. However, this work lacks generality and only considers the family of UML notations. In contrast, our work applies to a wider class of notations including those for goal and ER models.

Consistency checking. Inconsistency management is a well-studied topic in view-based requirements engineering [10]. In this domain, the term "inconsistency" usually refers to a situation where a *pair* of models do not obey a relationship that should hold between them [19]. This definition is restrictive in that it makes inconsistency a *pairwise* notion; however, it has the advantage of being easily applicable to *heterogeneous* models. Our work, in contrast, treats inconsistency as a *global* notion, but its scope is currently limited to *homogeneous* models only. More research needs to be done to determine if this limitation can be removed.

Early approaches to consistency checking of views use standard first-order logic for writing consistency rules [10, 7]. The expressiveness of these approaches is limited because first-order logic cannot capture reachability [16] and lacks convenient means for describing multiplicities.

Recent work on consistency checking of views addresses these limitations by using more expressive logics. For example, xlinkit [17] employs XPath –a query language for XML– augmented with a transitive closure operator for describing consistency rules. Similarly, [21] explores the use of theorem-proving and object-oriented programming to provide a rich platform for constraint specification. These approaches indeed offer sufficient expressive power to cover a wide range of consistency constraints; however, they do not address the key problem tackled in our work, which is consistency checking of *arbitrary* and *unknown* configurations of models and mappings.

The idea of consistency checking via merge was first explored in prior work by Easterbrook and Chechik [6]. The work uses temporal logic model checking to reason about behavioural properties of state machine models when they are merged. This earlier work is not applicable to structural models, because temporal logic cannot capture important structural constraints such as multiplicities [16]. Other consistency checking approaches based on temporal logic suffer from similar limitations if applied to structural models.

Several approaches to consistency checking of requirements work by consolidating different stakeholders' descriptions into a unified knowledge base and checking its overall consistency. For example, [11] translates textual requirements into a knowledge base of logical statements and finds inconsistencies by applying theorem proving and model checking. These approaches can reason about global consistency; however, to build a unified knowledge base, they assume that stakeholders have already agreed on a unified vocabulary for expressing their requirements. We do not make this assumption in our work and use explicit mappings to capture the relationships between different stakeholders' vocabularies. This makes it possible to hypothesize alternative relationships between these vocabularies and explore how each alternative affects global consistency.

There is a large body of research specifically dealing with consistency in UML. For UML models, consistency rules are usually described in the Object Constraint Language (OCL) [20]. Several tools exist for checking UML models against OCL expressions, e.g. the Dresden OCL toolkit [13]. Despite their merits, these tools are not suited to conceptual modelling because of OCL's strong orientation toward implementation [29].

We have considered using Alloy [14] instead of RML for expressing constraints. While Alloy provides the full expressive power of first-order logic with transitive closure and counting, we chose RML primarily for two reasons: (1) its leaner syntax; and (2) its imperative semantics (as opposed to Alloy's declarative semantics). The latter factor was particularly important in our work because it provided the flexibility to use RML as a scripting language for generating inconsistency diagnostics.

7 Conclusions and Future Work

We presented a tool-supported approach for consistency checking of distributed models. Our approach enables checking *inter*-model properties of a set of models via checking *intra*-model properties of their merge. To simplify the specification of consistency rules, we developed a set of generic expressions for characterizing recurrent patterns in structural constraints of conceptual models.

Our current work applies to homogeneous models only. Extending this to heterogeneous models presents a challenge because merge cannot be defined at a notational level. In future work, we plan to address this limitation by developing ways to merge models at a logical level. Another area for future work is to further study the practical utility of our approach by conducting user trials and observing user interactions with our tool for inconsistency exploration.

Acknowledgments. We thank Dirk Beyer, Juergen Dingel, Zinovy Diskin, Paul Gries, Renée Miller, and the anonymous RE 2007 reviewers for their insightful comments. Financial support for this work was provided by Bell Canada (through the Bell University Labs), OGS, and NSERC.

References

[1] D. Beyer and A. Noack. CrocoPat 2.1 introduction and reference manual. Technical report, U. Berkeley, 2004.

[2] D. Beyer, A. Noack, and C. Lewerentz. Efficient relational calculation for software analysis. *IEEE TSE*, 31(2):137–149, 2005.

[3] R. Bryant. Graph-based algorithms for boolean function manipulation. *Trans. on Computers*, 8:677–691, 1986.

[4] A. Corradini, U. Montanari, and F. Rossi. Graph processes. *Fundamenta Informaticae*, 26(3–4):241–265, 1996.

[5] M. Dwyer, G. Avrunin, and J. Corbett. Patterns in property specifications for finite-state verification. In *ICSE*, pages 411–420, 1999.

[6] S. Easterbrook and M. Chechik. A framework for multi-valued reasoning over inconsistent viewpoints. In *ICSE*, pages 749–750, 2001.

[7] S. Easterbrook and B. Nuseibeh. Using viewpoints for inconsistency management. *Soft. Eng. J.*, 11(1):31–43, 1996.

[8] S. Easterbrook, E. Yu, J. Aranda, Y. Fan, J. Horkoff, M. Leica, and R. Qadir. Do viewpoints lead to better conceptual models? An exploratory case study. In *RE*, pages 199–208, 2005.

[9] A. Egyed. *Heterogeneous View Integration and its Automation*. PhD thesis, USC, 1999.

[10] A. Finkelstein, D. Gabbay, A. Hunter, J. Kramer, and B. Nuseibeh. Inconsistency handling in multiperspective specifications. *IEEE TSE*, 20(8):569–578, 1994.

[11] V. Gervasi and D. Zowghi. Reasoning about inconsistencies in natural language requirements. *ACM TOSEM*, 14(3):277–330, 2005.

[12] J. Horkoff. Using i^* models for evaluation. Master's thesis, U. Toronto, 2006.

[13] H. Hussmann, B. Demuth, and F. Finger. Modular architecture for a toolset supporting OCL. *Sci. Comput. Program.*, 44(1), 2002.

[14] D. Jackson. *Software Abstractions Logic, Language, and Analysis*. The MIT Press, 2006.

[15] S. Liaskos. Quality criteria for variability modelling. Technical Report CSRG-549, U. Toronto, 2007.

[16] L. Libkin. *Elements of Finite Model Theory*. Springer-Verlag, 2004.

[17] C. Nentwich, W. Emmerich, A. Finkelstein, and E. Ellmer. Flexible consistency checking. *ACM TOSEM*, 12(1):28–63, 2003.

[18] B. Nuseibeh, S. Easterbrook, and A. Russo. Making inconsistency respectable in software development. *J. of Sys. and Soft.*, 56(11), 2001.

[19] B. Nuseibeh, J. Kramer, and A. Finkelstein. A framework for expressing the relationships between multiple views in requirements specification. *IEEE TSE*, 20(10):760–773, 1994.

[20] Object Management Group. OCL 1.4 Specification.

[21] R. Paige, P. Brooke, and J. Ostroff. Metamodel-based model conformance and multi-view consistency checking. *ACM TOSEM*, 2007.

[22] G. Rozenberg, editor. *Handbook of graph grammars and computing by graph transformation (Vol. 1): Foundations*. World Scientific, 1997.

[23] M. Sabetzadeh and S. Easterbrook. View merging in the presence of incompleteness and inconsistency. *RE J.*, 11(3):174–193, 2006.

[24] S. Spaccapietra and C. Parent. View integration: A step forward in solving structural conflicts. *Knowl. and Data Eng.*, 6(2):258–274, 1994.

[25] D. Svetinovic, D. Berry, and M. Godfrey. Concept identification in object-oriented domain analysis: Why some students just don't get it. In *RE*, pages 189–198, 2005.

[26] The Unified Modeling Language. `http://www.rational.com/uml/`.

[27] TReMer+: A tool for merging and consistency checking of distributed models. `http://www.cs.toronto.edu/~mehrdad/tremer/`.

[28] A. van Lamsweerde, R. Darimont, and E. Letier. Managing conflicts in goal-driven requirements engineering. *IEEE TSE*, 24(11):908–926, 1998.

[29] M. Vaziri and D. Jackson. Some shortcomings of OCL, the object constraint language of UML. Technical report, MIT, 1999.

[30] M. Wahler, J. Koehler, and A. Brucker. Model-driven constraint engineering. In *MoDELS Wrkshp. on OCL for (Meta-)Models in Multiple Application Domains*, pages 111–125, 2006.

[31] E. Yu. Towards modeling and reasoning support for early-phase requirements engineering. In *RE*, pages 226–235, 1997.