Towards 2D Traceability in a Platform for Contract Aware Visual Transformations with Tolerated Inconsistencies

Pieter Van Gorp
Department of Mathematics and Computer Science
University of Antwerp, Belgium

Frank Altheide
Database and Information Systems
University of Paderborn, Germany

Dirk Janssens
Department of Mathematics and Computer Science
University of Antwerp, Belgium

Abstract

Today's model-driven engineering tools focus on the automatic transformation of software models and lack essential support for interacting with developers. This paper presents some lessons learned from building a standard compliant platform for the visual development of interactive *consistency maintenance software. Based on an established requirements engineering case study, the paper illustrates the need for developer interaction and the controlled tolerance of inconsistencies. This motivates the role of traceability links in two dimensions: links between application models allow one to maintain consistency incrementally and tolerate inconsistencies in a controlled manner. In the other dimension, links between transformation models enable the refinement of declarative descriptions of consistency contracts into constructive transformations. Such transformations can be generated automatically from the contracts but tend to be optimized subtly by a transformation expert.*

1 Introduction

The specification of a software system, be it embedded control software or a large information system, is not monolithic. It consists of contributions from different disciplines (requirements engineering, security, enterprise architecture, ...) and models the system under development from different perspectives and at different levels of abstraction. The parts can overlap and contain redundant information, in any case they are interdependent.

If the software system that results from this specification is to be free of errors then the parts of the specification need to be *consistent*. Consistency according to the IEEE "is the degree of uniformity, standardization, and freedom from contradiction among the documents or parts of a system or component" [1]. This paper uses this definition in the narrow sense by focussing on software models that need to be kept free of contradiction.

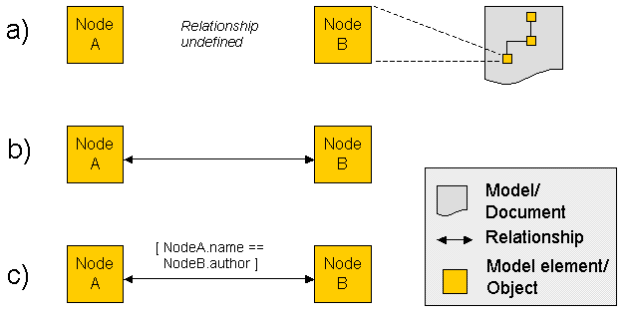

Figure 1. Levels of Traceability

Software models can be treated as graphs whose nodes represent model elements. Such nodes have attributes like *name*, *author*, *visibility*, etc. Edges represent links between the model elements. In an object oriented model, such edges represent *associations*, *method calls*, *type declarations*, etc. We will treat text documents as models as well: the nodes in this case are *sections*, *chapters*, etc. The set of relationships between the various models that are part of the specification can be perceived as an *interconnection graph* that joins the otherwise independent models.

As Figure 1 illustrates, the relationships can differ in expressiveness: from very generic "is related to" relationships to precise relationships upon which formal constraints have been defined. Note that the IEEE defines *traceability* as the "the degree to which a relationship can be established between two or more products of the development process, especially products having a predecessor-successor or master-

subordinate relationship to one another" [1]. The relationships in this definition are only established in the mind of a software developer. Therefore, this paper defines *traceability links* as the software artifacts that make these relations explicit in interconnection graphs called *traceability models*. The union of a traceability model and the models it connects is called an *integrated model*.

This paper will illustrate the role of traceability models in tools for interactive consistency maintenance. Consistency can be achieved manually by reviews and inspections but also automatically by tools that evaluate constraints and restore consistency where constraints are not satisfied. In some cases, inconsistencies should be tolerated. This paper will show how constraints can be relaxed in a controlled manner.

Maintaining consistency over the course of the development is important but also work-intensive and time-consuming, therefore automation is desirable. The degree of automation that can be achieved however is highly dependent on the development environment, on existing processes and tools. There is a trade-off between costs and the degree of automation the developers obtain. In this paper we want to discuss the spectrum of consistency maintenance approaches and present three existing approaches specifically:

- *ToolNet* as a representative tool for *manually* maintaining the consistency of models in commercial off-the-shelf (COTS) tools,
- *CAViT* as a *general purpose* framework for *automating* model transformations based on MOF, OCL and UML based graph rewriting, and
- *Triple Graph Grammars (TGGs)* as a *dedicated* formalism for *fully automatic* consistency maintenance.

We will portray the work we carried out to combine these three approaches in the *ICONS* framework for supporting *interactive* consistency maintenance. ICONS is designed for interoperability with MOF compliant COTS tools such as MagicDraw while hiding irrelevant API details from transformation writers.

The remainder of this paper is structured as follows: section 2 presents some models from a case study to motivate the need for consistency maintenance techniques. Section 3 discusses the solution space. Section 3.1 introduces the reader to the functionality and implementation challenges of tools like *ToolNet*. Section 3.2 presents the other side of the tool spectrum by discussing the *CAViT* framework. Section 3.3 motivates the contributions of this paper. It illustrates the need for *interaction*, *controlled inconsistencies* and *fine-grained consistency constraints* in tools like *ICONS*. Section 3.4 presents triple graph grammars as a formalism that allows one to model a large class of fully automatic transformations concisely yet lacks support for modeling developer interaction and controlled inconsistencies. Section 4 therefore presents the new two-dimensional traceability technique for combining the conciseness of TGGs with the flexibility of *CAViT*. This section forms the main theoretical contribution of this paper. Obviously, the paper concludes by summarizing the lessons learned.

2 Case Study

To give a realistic idea of models that need to be kept consistent, this section presents some heterogeneous models of a sample application. The application is based on the *Meeting Scheduler* problem statement that was proposed by Van Lamsweerde et al. [2] as a benchmark for requirements elicitation and software specification techniques. The problem statement of the benchmark was published deliberately imprecise and incomplete [3]. The first part of the problem statement reads as follows:

> *Meetings are typically arranged in the following way. A meeting initiator asks all potential meeting attendees for the following information based on their personal agenda:*
>
> - *a set of dates on which they cannot attend the meeting (hereafter referred as exclusion set);*
> - *a set of dates on which they would prefer the meeting to take place (hereafter referred as preference set).*
>
> *A meeting date is defined by a pair (calendar date, time period). The exclusion and preference sets are contained in some time interval prescribed by the meeting initiator (hereafter referred as date range).*

The actual goal of the system is to propose an *optimal* meeting location and date and streamline the communication among the participants of a meeting [2]. The requirements specification distinguishes between different conflict types and describes ways of resolving them.

Section 2.1 presents a conceptual model that formalizes the concepts, the associations and their multiplicities from the problem domain. Section 2.2 presents a robustness model for the "confirm meeting" use case scenario. The fragments in this paper should merely illustrate some realistic dependencies between models in different languages and should not be regarded as a complete or stable specification of a meeting scheduler. Instead, the sample models are meant to make the constraints in sections 3.1.2, 3.2.2 and 3.3.2 more concrete.

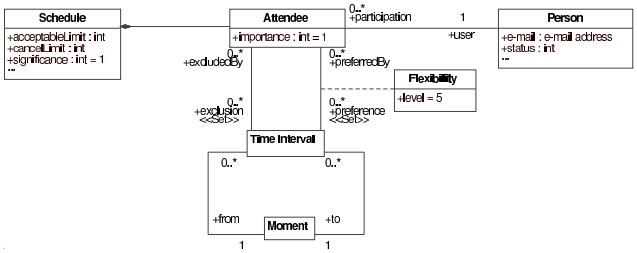

Figure 2. Conceptual Model.

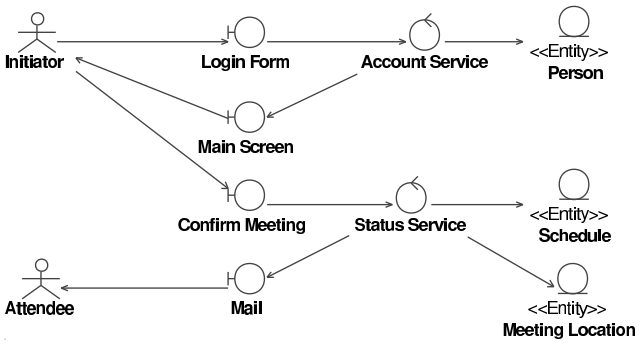

Figure 3. Robustness Model.

2.1 Conceptual Model

Figure 2 shows a conceptual model (CM) of a Meeting Scheduler application, specified in UML syntax. At the conceptual level, analysts are free to use constructs such as association classes, views, and other language features. Such features may not be supported directly by the implementation language but they allow one to represent the problem domain in a way as close as possible to one's perception of reality.

A complete conceptual model contains all relevant nouns and verbs from a problem domain as classes and operations. We could establish some traceability in this paragraph by describing in natural language how the words from Figure 2 relate to those from the requirements specification presented above. However, this would not be an adequate basis for automating the consistency maintenance between these two models. Moreover, developers may want some tool support for navigating from one model element to all its related elements in another model.

2.2 Robustness Model

The model-view-controller (MVC) pattern has been found beneficial for system evolvability [4]. Therefore, Rosenberg and Scott propose to move from analysis to design by creating *robustness model*s [5]. User interface screens are represented by *boundary objects* (or *interfaces*), persistent classes from the conceptual model are represented by *entities* while application behavior is encapsulated by *control objects* (or *services*). A set of architectural rules (like "only services are allowed to access entities") assists developers to create an evolvable design. Figure 3 shows a robustness model (RM [5]) of the application under study. Note that the entity Schedule corresponds to the class Schedule from Figure 2.

The model describes the way a meeting can be confirmed by an initiator. After logging in, the initiator is directed to the main user interface screen of the application. There, he can select a particular meeting and click the "confirm meeting" button. This action triggers an update of the meeting status and a booking of the meeting location. Finally, the success of the use case is confirmed by sending a mail to all meeting attendees.

3 Maintaining Consistency

This section presents an overview of existing consistency maintenance approaches to provide insight in their power and limitations. It illustrates how some of these limitations can be overcome by *combining* the strengths of the presented approaches.

3.1 Manual Consistency Maintenance

This section illustrates how tools like *ToolNet* can assist developers in the manual construction of traceability links.

3.1.1 User Experience

A manual traceability tool like *ToolNet* allows one to create and navigate links between model elements. Figure 4 shows how a user may request all links related to the word "meeting" in the requirements document. A popup shows that this word is related to a model element with name "Meeting" in the application model. This model is stored in a UML tool and contains both the conceptual and the robustness model.

Developers can also perform completeness checks on the set of traceability links. They may for example check whether every class in the conceptual model traces to a word in the requirements document. The underlying rationale of such a check is that artificial concepts in a conceptual model should be avoided whenever possible.

3.1.2 Implementation Challenges

ToolNet is a traceability tool for models residing in COTS development tools. Basili and Boehm [6] define COTS

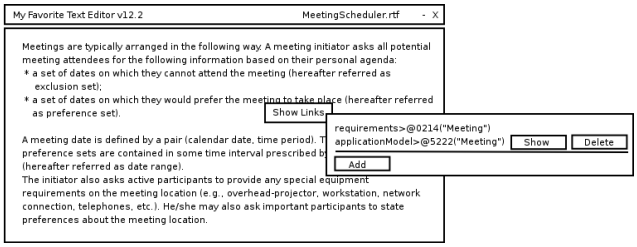

Figure 4. Sketch from *ToolNet* operating on the requirements document.

components as software with the following three characteristics: the buyer has no access to the source code, the vendor controls its development and it has a nontrivial user base. Unfortunately, the features of a component may not match the buyer's requirements completely and it usually supports only a part of the overall development process.

This poses a number of integration challenges. First of all, extension and adaptation of the component is limited by the mechanisms provided by the vendor. Secondly, the components are unaware of each other. Thirdly, a vendor is not bound to adhere to standards but focuses on implementing a set of features to gain a competitive edge over his competitors. Lastly, the buyer has only limited influence on the development of the component. The visions of a specific buyer and the vendor may diverge, forcing this buyer to exchange one component for another.

Still one needs a common language to reason about model elements and traceability links. *ToolNet*'s metamodel of such a language originally only supported directed and undirected binary links. The following OCL code formalizes the constraint described in section 3.1.1 on a metamodel for undirected n-ary traceability links:

```
let requirementsWordForEachCmClass(): Boolean=
 allClassesFromModel(cm)->forAll(cmClass|
  nodesOfElement(cmClass).link->select(
   oclIsKindOf(Class2Word)
  ).node->select(role="word")->size>0
 )
```

Note that this constraint does not need the actual content of the text processor's repository. This illustrates that some useful constraints can be enforced without knowing the precise metamodels of the integrated tools. *ToolNet* exploits this observation by implementing the minimalistic metamodel as the basis for a bus-like integration of COTS tools such as DOORS, Word, Matlab, StateFlow and MagicDraw. By exposing JMI interfaces for the traceability models, their repository can be integrated seamlessly with more expressive repositories such as the one containing the conceptual and the robustness model elements. The OCL constraint can thus be executed on the federated repository of the integrated model.

3.2 Automatic Consistency Maintenance

In model-driven engineering, stepwise refinement is supported by means of *automatic* model transformations to increase developer's productivity and to *enforce* consistency. *CAViT* is a MOF compliant framework that extends the principles of design by contract for the implementation of model transformations. It allows architects to formalize the relationship between transformation rules and the consistency constraints they maintain in an object-oriented manner. *CAViT* is a "general purpose" framework since it can be applied for both horizontal and vertical transformations and does not pose limitations on the number of input or output models of a transformation [7].

Transformation rules are implemented as methods of a class that holds references to the input and output models. The behavior of the methods is modeled as story diagrams. Story diagrams are the core of the UML based controlled graph transformation language and are supported by tools such as Fujaba, MoTMoT and MOFLON. Consistency constraints are implemented as "transformation contracts". Such a contract is defined by a pair of constraints: the *postcondition* describes the effect of a transformation method on the set of models provided that the *precondition* is satisfied.

CAViT extends traditional design by contract by requiring that each postcondition corresponds to an *invariant* of the transformation class in which it is contained *[8]*. By implementing all desired consistency constraints as such invariants and postconditions, each instance of a transformation class can maintain the consistency of its contained models. More specifically, each time an invariant is violated for a transformation object, the *CAViT* engine can call the transformation method whose postcondition corresponds to that invariant and whose precondition holds at that point in time.

3.2.1 User Experience

Suppose that a developer has completed a conceptual model of the meeting scheduler by doing a noun/verb analysis on the requirements specification. Now he wishes to optimize the robustness of the application by building a design model that respects the model-view-controller principles. Therefore, he wants to construct a robustness model based on the main use cases. The developer wants to use his conceptual model as a basis for the persistent entities of this robustness model. However, he does not want to clutter the conceptual model itself with the upcoming design details. Therefore, he decides to copy all classes from the conceptual model to the robustness model and mark them as entities. Ideally, the construction of this initial robustness model from a conceptual model is supported by the development tool. With

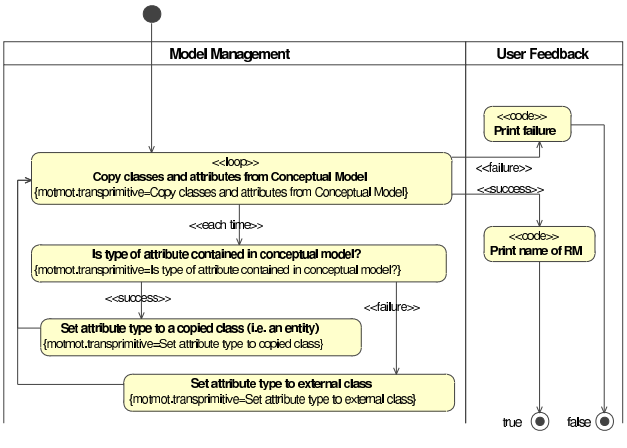

Figure 5. Control flow of the *cmClasses2rmEntities* method.

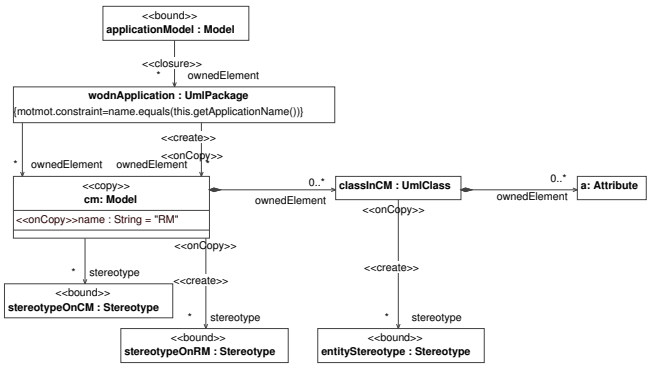

Figure 6. Graph transformation rule displaying what elements are copied from the conceptual to the robustness model.

CAViT, software architects can build a library of transformations that developers can apply in a black-box manner. The following section illustrates how a conceptual model can be transformed automatically to a minimalistic robustness model. Application developers would launch this transformation by one click on a button.

3.2.2 Implementation Challenges

CAViT is built for integration with COTS modeling tools. As with *ToolNet*, a particular challenge is that the repositories of these tools cannot be changed. However, by following suitable guidelines in the design of a transformation model, a transformation repository can be wrapped seamlessly around its application model repositories. More specifically, associations between a transformation class and a metaclass from the application models should be specified unidirectionally.

In summary, a software architect needs to provide three artifacts: (1) a MOF compliant definition of a transformation class, (2) a declarative consistency contract stating when a transformation method can establish what kind of consistency constraint and (3) a constructive body satisfying that contract.

Figure 5 and 6 show models of the body of the *cmClasses2rmEntities* method contained by the *CMconsistentRM* class. This method automatically constructs an initial robustness model from a conceptual model, as discussed in the previous section (3.2.1). The method starts by copying all classes and attributes from the conceptual model. For each attribute, it checks whether its type is contained in the conceptual model. If so, it sets the type of the copied attribute to a copied class. Otherwise, it sets the type of the copied attribute to an "external class". A class is said to be external when it is not defined in the conceptual model. A library class for "String" may be external. Once the copying process is finished, the name of the robustness model is printed. Since such robustness model is the result of the copying, this indicates a successful execution of the transformation method. Finally, *true* is returned as a formal indication of success. If something goes wrong – for example: the conceptual model cannot be found in the application model of *CMconsistentRM* – the transformation prints out an indication of failure and *false* is returned.

Figure 6 models the behavior of the first state of the story diagram by showing what elements are copied from the conceptual model to the robustness model. The rule applies the story diagram syntax for copying subgraphs [9]. It specifies that the root of the copied subgraph is the conceptual model itself. That model is matched by looking for an element whose type is *Model* and that is connected to the "Conceptual Model" stereotype. Moreover, the rule specifies that this *cm* node should be owned by a UML package whose name is equal to the *applicationName* attribute of *CMconsistentRM* and that is recursively owned by the input UML model. As soon as *cm* is matched, the subgraph represented by its classes and their attributes with the latter's types is copied. Implicitly, the transformation engine creates a traceability link between each element and its copy. Explicitly again, the rule fragment on Figure 6 connects each class in the subgraph copy to the "Entity" stereotype. Additionally, a new link is created from the package containing the application's models to the node that is copied from *cm*. This node is also connected to the "Robustness Model" stereotype and its name is changed to "RM". With other words, the generated robustness model will be added to a model container, it is decorated with the proper stereotype and it gets a new name. A detailed explanation of each node, link and stereotype on Figure 6 is given in [9] which

introduces the copy operator by means of an example similar to the one used in this paper.

The following constraint illustrates how the postcondition of the *cmClasses2rmEntities* method can be expressed in OCL. It expresses that all classes from the conceptual model should correspond to entities in the robustness model.

```
let CMconsistentRMcontract(): Boolean=
 conceptualmodelTracesToRobustnessmodel() and
 allClassesFromModel(cm)->forAll(cc: Classifier |
  allClassesFromModel(rm)->exists(rc: Classifier |
   this.traces->exists(t2 |
    t2.node->exists(cNode | cNode.content=cc)
    and t2.node->includes(rNode | rNode.content=rc)
   ) and
   cc.name=rc.name and
   rc.hasStereotype("entity") and
   cc.attributes()->forAll(ca|
    rc.attributes()->exists(ra|
     ca.name=ra.name and
     ca.type.name=ra.type.name and (
      ca.type<>ra.type or (
       -- take care of potentially used built-in types
       not allClassesFromModel(cm)->includes(ra.type)
      )
     )
    )
   )
  )
 )
```

The constraint is complex in that it asserts existential and qualitative properties about a variety of concepts (*model*s, *classes* and *attribute*s). This is acceptable for automatically establishing a global model property in one particular constraint violation scenario (or one particular *precondition*). However, it is too coarse grained to be associated with a set of small transformations that maintain parts of the constraint in an interactive manner. Therefore, it is decomposed in the following section.

3.3 Interactive Consistency Maintenance

The two previous sections might create the impression that all inconsistencies should be corrected automatically. However a number of authors have argued that inconsistencies should be tolerated under certain circumstances and that overly strict consistency maintenance can even be "foolish" [10]. Nuseibeh et.al. [11] even argue that each inconsistency must be treated differently. This is reflected in their proposed consistency maintenance process consisting of the following phases: (1) monitor for inconsistencies, (2) diagnose inconsistencies by means of locating, identifying and classifying, and finally (3) handle inconsistencies. The proposed actions are to ignore, tolerate or resolve an inconsistency. Tolerating an inconsistency can mean to defer, circumvent or ameliorate it.

3.3.1 User Experience

ICONS supports the consistency maintenance process by combining the facilities presented in the previous two sections. *CAViT* is used to monitor consistency constraints that were formalized in OCL. Diagnosis is supported both automatically and semi-automatically. More specifically, a model transformation can automatically assess a particular consistency violation and repair it without developer intervention. However, as an added value to *CAViT*, *ICONS* also enables transformations to present a set of violating model elements to a developer. Developers can then transform these elements manually or provide parameters to the *CAViT* backend. They can also specify that a particular inconsistency should be ignored by manipulating the *ignoreConstraints* property of a traceability link.

A transformation has multiple constraints, each of which can be *satisfied*, *violated* or *ignored*. This state is visualized by means of a traffic-light analogy. Each constraint can use an arbitrary number of traceability links that may be used by other constraints as well. The state of individual traceability links can be computed on-demand. More specifically, developers can request to highlight all model elements that cause a particular inconsistency. Based on the *ToolNet* infrastructure, *ICONS* then allows developers to query for all links related to those elements.

3.3.2 Implementation Challenges

As pointed out before, the transformation metamodel needs to enable developers to ignore inconsistencies. Secondly, *ICONS* can only offer an interactive user experience when constraints are checked in a fine-grained manner. Synchronization between fine-grained constraints is realized by means of different classes of traceability links. More specifically, Figure 7 illustrates that one subclass of *Link* is provided for each set of related model element types.

The following OCL fragments illustrate how a *part* of the consistency of classes and entities is checked by two *fine-grained* constraints:

```
let eachClassTracesToAnEntity(): Boolean=
 conceptualmodelTracesToRobustnessmodel() and
 allClassesFromModel(cm)->forAll(cmc|
  allClassesFromModel(rm)->exists(rmc|
   this.traceabilityLinks->select(
    oclIsKindOf(Class2Entity)
   )->exists(l|
    l.node->contains(cmc) and l.node->contains(rmc)
   )
  )
 )
```

```
let classEntity_name_match(): Boolean=
 traceabilityLinks->select(oclIsKindOf(Class2Entity))
 ->forAll(l|
  l.ignoreConstraints or l.node->forAll(n1,n2|
   n1.content.name=n2.content.name
  )
 )
```

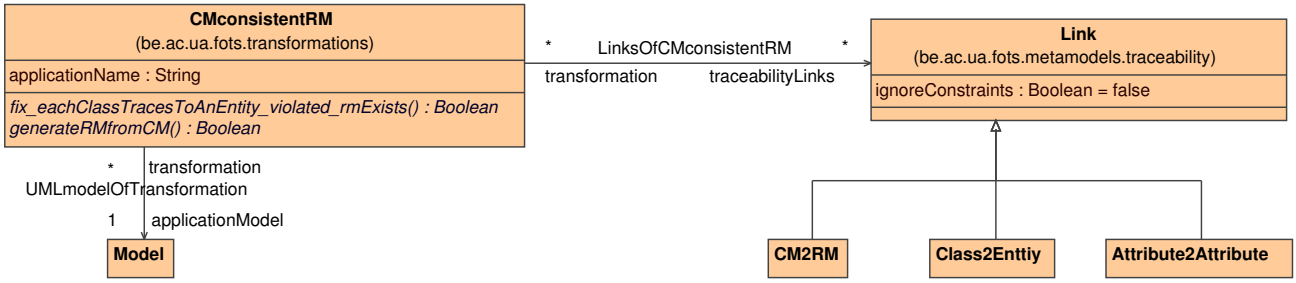

Figure 7. MOF instance representing the transformation model that defines the structure of the *CMconsistentRM* transformation and its traceability links.

Figure 8 models how the the *CAViT* backend of *ICONS* contains callbacks for interacting with developers. The transformation iteratively selects those classes that are related to entities with a different name and for which the traceability link is not allowed to be ignored. For each pair of class and entity, the system prompts for user interaction. More specifically, the user should indicate whether the inconsistency should be solved automatically, whether it should be ignored, or whether it should be solved manually. In the automatic case, the user should indicate whether or not the class name has precedence over the entity name. In the manual case, the system highlights the conflicting pair in the model editor.

The behavior of most states is modeled as story patterns (primitive graph rewriting rules). Others contain a callback to one of two MOF compliant user interaction methods: *setFocus* or *chooseAlternative*. A detailed discussion thereof is left out due to space restrictions but can be found in an extended version of this paper [12].

One can observe that there is a strong similarity between the consistency constraints between nodes that represent (1) the conceptual model and the robustness model, (2) classes and entities, and (3) attributes of classes and entities. More specifically, for each such pair at least the names of the nodes should be the same. When implementing the constraints and the transformations for each pair by OCL and story diagram models as shown above, the maintenance of these artifacts becomes problematic if they are not related by traceability links of some form. On the other hand, treating all cases by parametrized constraint and transformation methods may become either too complex (due to all kinds of conditionals) or too simplistic (when all inconsistencies are resolved in exactly the same manner). To make the latter more concrete, the next section shows how the consistency of our example elements could be maintained fully automatically.

3.4 Triple Graph Grammars

This section illustrates how triple graph grammars can be used to maintain the consistency between all related model elements in a fully automatic manner. It will be shown that the formalism supports the modeling of a large class of inconsistency resolution strategies in a compact manner. However, we will also make explicit that it was not designed for modeling interaction with developers.

3.4.1 Concept

Triple graph grammars are an extension to pair graph grammars aimed at concisely specifying how two metamodel-based languages should be mapped while maintaining relations between model elements in a third language. Consequently, triple graph grammars operate on models consisting of three related submodels: a source model, a target model and a correspondence model. A TGG rule distinguishes between these models by marking nodes with a $\ll left \gg$, $\ll right \gg$ or $\ll map \gg$ flag. By creating such marked nodes in a TGG rule one can concisely specify how a recurring set of consistency violations between the source and target models should be corrected. More specifically, one TGG rule corresponds to six primitive graph rewriting rules [13]: three rules for adapting changes to the source model and three for adapting changes to the the target model. To indicate the direction of the change propagation, the former three are called the left-to-right or "forward" rules while the latter are called right-to-left or "backward". Both groups of rules consist of a *creation rule*, a *deletion rule* and a *consistency rule*. The former makes sure that when an element is found in one model, a corresponding element exists in the other model. The second rule makes sure that when an element is deleted from one model, its corresponding element is deleted from the other model. The latter rule makes sure that when attribute updates on an element in one model trigger a violation of a consistency

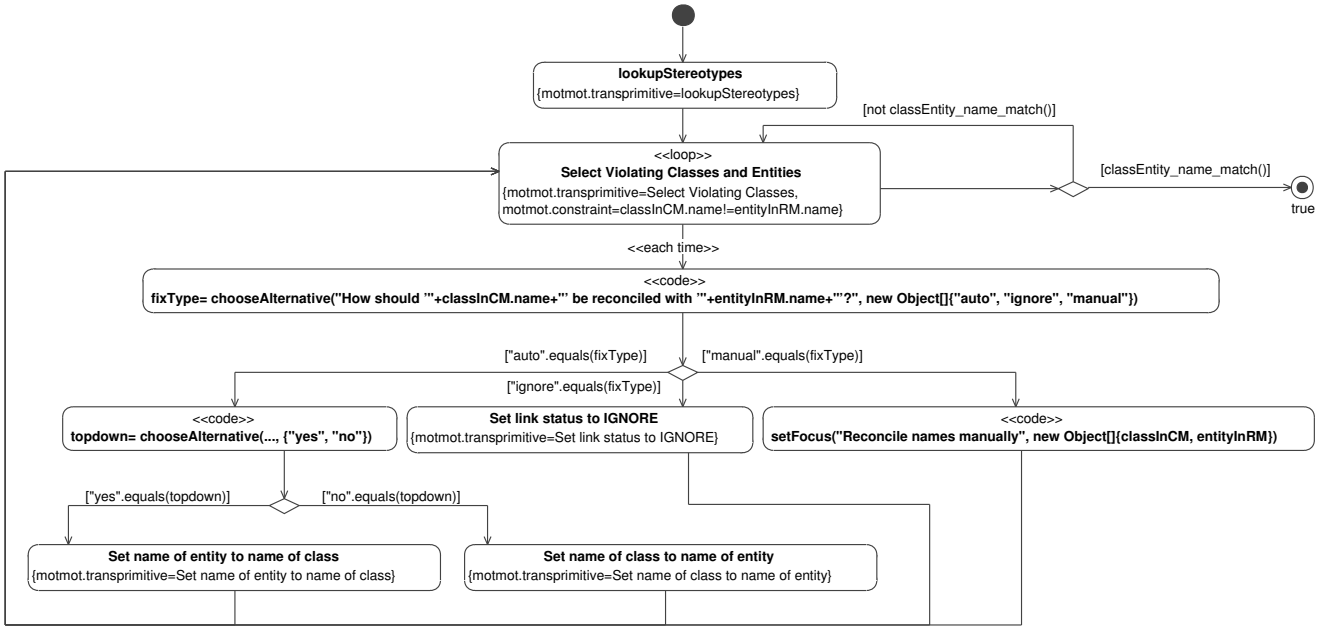

Figure 8. Story diagram modeling the reconciliation of a class when its entity has a different name.

constraint related to an element in the other model, that the element in the other model is updated.

3.4.2 Application to Example

Figure 9 shows a TGG rule for keeping a class in the conceptual model consistent with an entity in the robustness model. Note that this rule is dependent on the rule that keeps the nodes representing the conceptual and robustness models consistent with one another. To keep the TGG rule simple, *Link* is supposed to have a direct to-many association to the *ModelElement* metaclass of the UML metamodel (instead of having an intermediate *Node* metaclass like in the more realistic OCL fragments shown before). *ModelElement* is the superclass of *Model*, *UmlClass* and all other model elements.

The rule considers the conceptual model as the source (or *left*) model and the robustness model as the target (or *right*) model. The rule matches all *cm* nodes connected to an *rm* node by means of an *m2m* link of type *CM2RM*. The *cm* node represents an element of type *Model* within the host graph (a UML model that contains conceptual models, robustness models and metadata expressed as profiles). By requiring that *cm* is connected to *stereotypeOnCM*, the rule makes sure that *cm* corresponds to a conceptual model. Similarly, *rm* represents a robustness model. The left *ownedElement* association end is connected to *classInCM*, a node representing a class in the conceptual model. In the target model, *entityInRM* should represent an entity. This

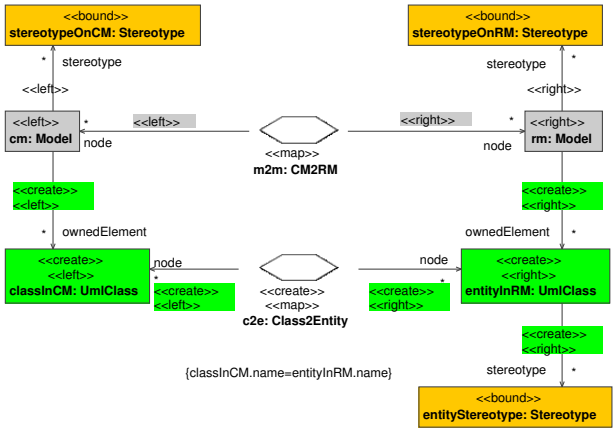

Figure 9. TGG Rule for classes and entities. As on Figure 7, *CM2RM* is supposed to subclass *Link*.

is realized by asserting that *entityInRM* is of type *UmlClass* and that it is connected to the *entityStereotype* node. Note that the nodes of type *Stereotype* are assumed to be bound in previous rewrite rules. Once a match against this structure has been found, the rule asserts that the name of the class in the conceptual model should always be the same as the name of the entity in the robustness model.

As stated, the behavioral semantics of TGG rules is defined by six primitive (called "operational") rewrite rules: three forward and three backward rules. Figures 10 to 12 present the forward creation, forward deletion and forward consistency rules derived from the example TGG rule shown on Figure 9. The forward creation rule creates an entity represented by node *entityInRM* and a traceability link (node *c2e*) for each class (node *classInCM*) in the conceptual model (node *cm*) that is not linked (node *c2e_nac*) to an entity (node *entityInRM_nac*) in the robustness model (node *rm*) yet. The forward deletion rule deletes all entities represented by node *entityInRM* and their traceability link (node *c2e*) from the robustness model (node *rm*) if there doesn't exist a class (node *classInCM*) on the other end of the traceability link. The forward consistency rule is triggered when a class is linked to an entity with a different name. The rule restores the consistency between such violating elements by overwriting the name of the entity with the name of the class.

The backward creation, deletion and consistency rules propagate changes in the other direction but using an equivalent match and update approach. They are omitted from this paper due to space considerations.

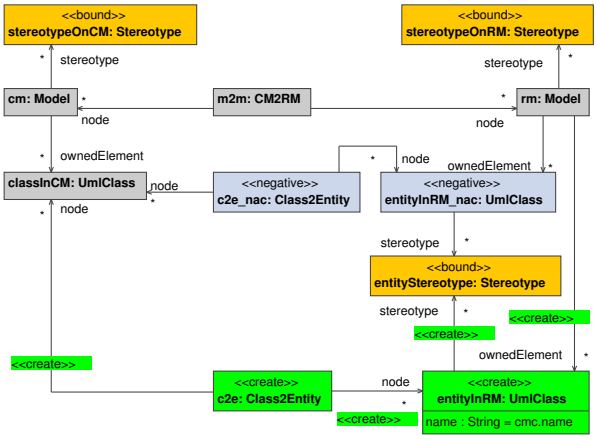

Figure 10. Forward Creation Rule for classes and entities. Nodes marked as $\ll negative \gg$ should *not* occur in the host graph in order to trigger a rewrite rule.

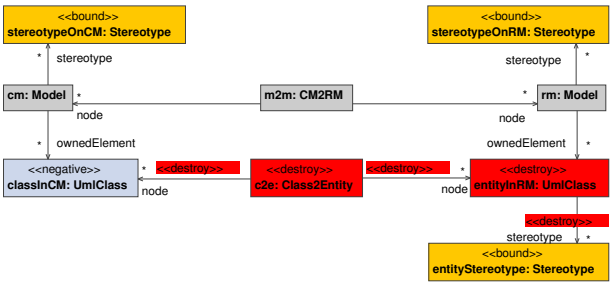

Figure 11. Forward Deletion Rule for classes and entities.

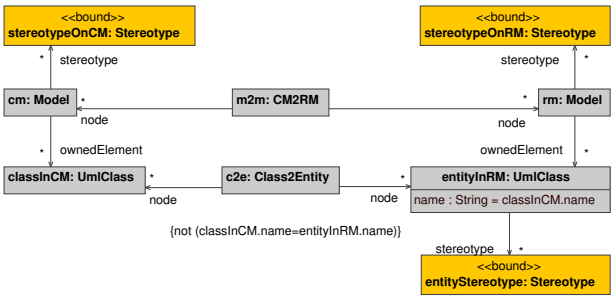

Figure 12. Forward Consistency Rule for classes and entities.

3.5 Discussion

This section makes a brief comparison of the power and limitations of *CAViT*, *ICONS* and triple graph grammars.

CAViT allows one to model consistency contracts in the side-effect-free Object Constraint Language. The behavior of the methods that maintain those contracts is modeled by controlled graph rewriting rules, which consist of a set of primitive graph rewriting rules and a control flow diagram. When using the triple graph grammar formalism, six primitive rewriting rules are generated automatically from one triple rule by a higher-order transformation. This generative aspect makes triple graph grammars obviously a more productive formalism when considering the constraint violation scenarios covered by the six primitives.

However, one cannot always assume that the six generated rules cover the intentions of a transformation writer exactly. Instead, subtle modifications may be required, developer interaction may need to be added and some automatic transformation rules may need to be replaced by manual transformations. In fact, recent work from the inventors of triple graph grammars confirms that a fully automatic or

batch application of transformation rules is not always desirable [14]. Moreover, a TGG based consistency maintenance tool developed in the context of the ISILEIT project supports various user interaction scenario's on the implementation level [13]. This illustrates once more that developer interaction is essential to make consistency maintenance tools usable in practice.

Therefore, we propose to apply the *ICONS* principles to the generated rewrite rules by embedding them in a control flow diagram. This allows one to express that certain rules should only be applied under particular conditions. Moreover, *ICONS* methods like *chooseAlternative* and *setFocus* can be used to model what information is gathered from developers and what model elements are highlighted when developers are to restore a consistency constraint manually.

When adapting the transformations generated from TGG rules as such, one is actually applying model refinement to transformation models. To manage the complexity of the increasing number of related transformation models, one can maintain traceability links between a high-level TGG rule and its derived primitive rules. The following section discusses that in the overall set of models, one can then identify two dimensions of traceability.

4 Two-Dimensional Traceability

This section describes an architecture for modeling tools that allow one to maintain traceability links in two dimensions: in the first dimension, links allow one to reason about relations between application models while in the second dimension, links allow one to reason about relations between transformations that maintain the application models consistent.

4.1 Application to Example

This section presents how a software architect specialized in development processes based on conceptual and robustness modeling may want to adapt the rewrite rules generated from the TGG rule that models the relationship between classes and entities from a high level of abstraction.

The architect decides that there should be cascading deletion behavior from classes to entities. Therefore, he allows the forward deletion rule shown on Figure 11 to be active at all times. However, the architect decides that when an entity is removed from a robustness model, the corresponding class should not be automatically removed from the conceptual model. Instead, the modeling tool should allow the developer to choose between two alternatives: either the class is removed from the conceptual model, or the entity is regenerated from its class. Therefore, the architect constructs an activity diagram expressing this behavior. The activity diagram contains:

- a state calling the *chooseAlternative* method to query the developer for his intentions,
- two story patterns: the former is based on the backward deletion rule derived from the TGG rule shown on Figure 9 while the latter is based on the forward creation rule shown on Figure 10.

The architect can structure his transformation models such that the TGG rule, its derived primitive rules and the manually created activity diagram are grouped together. He thereby applies some kind of ad-hoc traceability technique corresponding to the first case (a) of Figure 1. However, there are several reasons for creating a more explicit kind of traceability relationship between the related transformation models:

- Scalability: there may not always be an optimal package structure that reflects how large sets of transformation models are related to one another.
- Expressiveness: a dedicated traceability tool such as *ToolNet* allows one to create traceability links across tools based on different metamodels.
- Queryability: as illustrated in section 3.1.2, even simple traceability models allow one to define useful queries for checking consistency constraints. A traceability model is also a structured basis for ad-hoc queries like "retrieve all activity diagrams making use of the forward creation rule derived from TGG rule X".
- Low Cost: an appropriate architecture allows one to apply a traceability tool designed with application models (dimension 1) in mind in the context of the management of links between transformation models (dimension 2) as well. The following section presents such an architecture.

4.2 Reference Architecture

This section presents a reference architecture supporting traceability in the two discussed dimensions. The architecture is based on a minimalistic prototype we are constructing to demonstrate the proposed solution to the case study in practice.

A key design characteristic of this prototype is the application of UML profiles as a light-weight metamodeling approach. UML profiles allow one to customize editors based on the standard UML metamodel to the look-and-feel of another visual language. We have designed the prototype as such since today's heavy-weight metamodel-independent editors [15] still either require too much investment in configuring the concrete syntax or do not comply to the MDA standards compatible with *CAViT*.

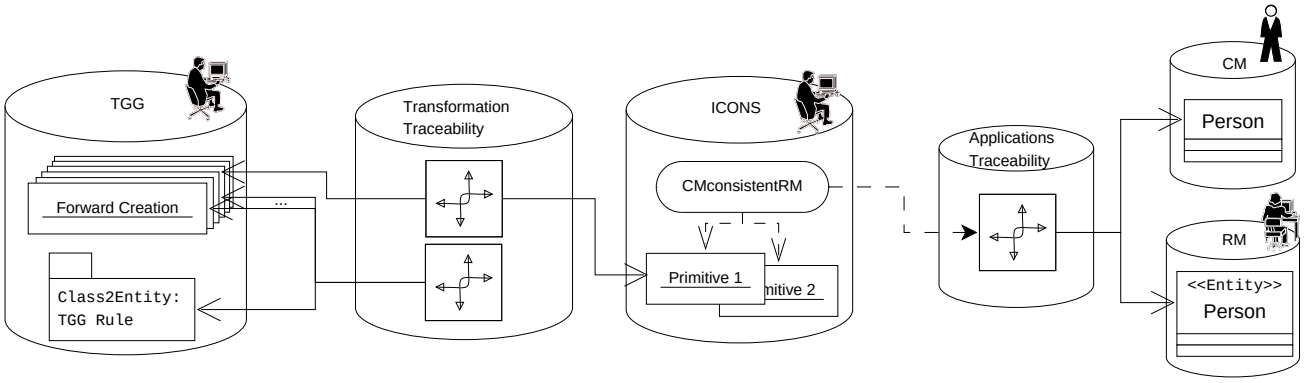

Figure 13. Reference architecture of a consistency maintenance tool-chain using 2D traceability.

This design decision implies that the *CM*, *RM*, *ICONS* and *TGG* repositories physically correspond to only one MOF repository, being the JMI compliant repository of the *MagicDraw* UML tool. Thanks to that, *ToolNet* only needed to be extended with a JMI based link repository, a repository communication component (called a *Tool Link* [16]) for JMI and a user interface plugin for *MagicDraw*. The disadvantage is that the higher order transformation for normalizing triple graph grammars to operational rules needs to be implemented again in a UML profile and MOF compliant manner. When Fujaba would be supported by *ToolNet*, we could reuse an existing Fujaba plugin for triple graph grammars instead [17].

The prototype relies on the *ToolNet* widgets for visualizing traceability links in the two dimensions. Industrial validation of *ToolNet* indicated that end-users prefer table-based widgets [16] over a graph-based tool like Fujaba's dynamic object browsing (*DOBS* [18]). However, since the application of *DOBS* in a debugging context seems promising [19], we would like to learn to what extent a graph-based visualization of transformations and their traceability links can improve the development experience of model transformation writers. Therefore, we are constructing a JMI based storage layer for DOBS which will allow one to browse and update the abstract syntax graphs of all the discussed models (*conceptual*, *robustness*, *transformation, traceability*, ...) in a uniform manner.

Links from the *application dimension* are managed by the repository shown at the top part of Figure 13, between the repositories containing the conceptual model (CM) and the robustness model (RM). Links from the *transformation dimension* are managed by the repository shown on the bottom left part. The link shown at the bottom of this repository is used to relate a triple graph grammar rule with its six derived operational rules. The link above relates such an operational rule to a primitive rewrite rule embedded in an activity diagram from the ICONS repository.

5 Conclusion

Software models are expressed in a variety of languages with a variety of tools. It is widely accepted that the management of inconsistencies that can arise within and between such models requires a means to describe consistency constraints, detect violations and correct the models accordingly.

Unfortunately, today's model-driven engineering tools focus on the fully automatic transformation of software models and lack essential support for interacting with developers. This paper presented some lessons learned from building *ICONS*, a MOF compliant platform for the visual development of *interactive* consistency maintenance software. By modeling transformations in OCL and story diagrams, we identified the need for a more abstract formalism. While formalisms such as triple graph grammars already allowed one to express a general consistency constraint in a concise and visual manner, the derived tools encoded developer interaction in their low-level implementation language.

This motivated the integration of triple graph grammars with *ICONS* which results in the use of traceability links in two dimensions. The new (second) dimension consists of links supporting the stepwise refinement of abstract transformation models into operational transformation models. Essentially, this illustrates that model-driven engineering techniques can be used for managing the complexity arising *within* tools supporting model-driven engineering.

Acknowledgements

This work has been sponsored by the European research training network "Syntactic and Semantic Integration of Visual Modeling Techniques (SegraVis)" through the University of Antwerp.

References

[1] IEEE, editor. *IEEE Standard Computer Dictionary: A Compilation of IEEE Standard Computer Glossaries*. Institute of Electrical and Electronics Engineers, 1990.

[2] A. v. Lamsweerde and R. Darimont and P. Massonet. The Meeting Scheduler System - Problem Statement. Technical report, Université Catholique de Louvain - Département d'Ingénierie Informatique, B-1348 Louvain-la-Neuve (Belgium), 1992.

[3] M.S. Feather, S. Fickas, A. Finkelstein, and A. van Lamsweerde. Requirements and specification exemplars. *Automated Software Engineering*, 4(4):419–438, 1997.

[4] Takako Nakatani, Tetsuo Tamai, Atsushi Tomoeda, and Harumi Matsuda. Towards Constructing a Class Evolution Model. In *Fourth Asia-Pacific Software Engineering and International Computer Science Conference*, page 131, Clear Water Bay, Hong Kong, December 1997. IEEE Computer Society.

[5] Doug Rosenberg and Kendall Scott. *Use case driven object modeling with UML: a practical approach*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1999.

[6] Victor R. Basili and Barry Boehm. Cots-based systems top 10 list. *Computer*, 34(5):91–93, 2001.

[7] Tom Mens and Pieter Van Gorp. A taxonomy of model transformation. In *Proc. Int'l Workshop on Graph and Model Transformation*, 2005.

[8] Pieter Van Gorp and Dirk Janssens. CAViT: a consistency maintenance framework based on visual model transformation and transformation contracts. In J. Cordy, R. Lämmel, and A. Winter, editors, *Transformation Techniques in Software Engineering*, number 05161 in Dagstuhl Seminar Proceedings. Internationales Begegnungs- und Forschungszentrum für Informatik (IBFI), Schloss Dagstuhl, Germany, 2005.

[9] Pieter Van Gorp, Hans Schippers, and Dirk Janssens. Copying Subgraphs within Model Repositories. In Roberto Bruni and Dániel Varró, editors, *Fifth International Workshop on Graph Transformation and Visual Modeling Techniques*, Electronic Notes in Theoretical Computer Science, pages 127–139, Vienna, Austria, 1 April 2006. Elsevier.

[10] Anthony Finkelstein. A foolish consistency: Technical challenges in consistency management. In *Proceedings of the 11th International Workshop on Database and Expert Systems Applications*, 2000.

[11] Bashar Nuseibeh, Steve Easterbrook, and Alessandra Russo. Leveraging inconsistency in software development. *Computer*, 33(4):pp. 24–29, 2000.

[12] Frank Altheide, Pieter Van Gorp, and Dirk Janssens. ICONS: an interactive consistency maintenance platform. Technical report, Universiteit Antwerpen, Department of Mathematics and Computer Science, 2020 Antwerpen, Belgium, 2006.

[13] Sven Burmester, Holger Giese, Jörg Niere, Matthias Tichy, Jörg P. Wadsack, Robert Wagner, Lothar Wendehals, and Albert Zündorf. Tool integration at the meta-model level: the Fujaba approach. *International Journal on Software Tools for Technology Transfer*, 6(3):203–218, August 2004.

[14] A. Königs. Model Transformation with Triple Graph Grammars. In *Model Transformations in Practice Satellite Workshop of MODELS 2005, Montego Bay, Jamaica*, 2005.

[15] Karsten Ehrig, Claudia Ermel, Stefan Hänsgen, and Gabriele Taentzer. Generation of visual editors as eclipse plug-ins. In *ASE '05: Proceedings of the 20th IEEE/ACM international Conference on Automated software engineering*, pages 134–143, New York, NY, USA, 2005. ACM Press.

[16] F. Altheide, H. Dörr, and A. Schürr. Requirements to a Framework for sustainable Integration of System Development Tools. In H. Stoewer and L. Garnier, editors, *Proc. of the 3rd European Systems Engineering Conference (EuSEC)*, pages 53–57, Toulouse, 2002. AFIS PC Chairs.

[17] Lars Grunske, Leif Geiger, and Michael Lawley. A Graphical Specification of Model Transformations with Triple Graph Grammars. In *First European Conference Model Driven Architecture - Foundations and Applications*, number 3748 in Lecture Notes in Computer Science, pages 284–298. Springer, 7 November 2005.

[18] Leif Geiger, Christian Schneider, and Albert Zündorf. Statechart Modeling with Fujaba. In *Proc. 2nd International Workshop on Graph-Based Tools*, Rome, Italy, October 2004. Satelite event of ICGT.

[19] Leif Geiger. Design Level Debugging mit Fujaba. In *Informatiktage*, Bad Schussenried, Germany, 2002. der Gesellschaft für Informatik.