A Plug-In for Flexible and Incremental Consistency Management

Robert Wagner[1], Holger Giese[2], Ulrich A. Nickel[1]
Software Engineering Group
Department of Computer Science
University of Paderborn
Warburger Straße 100
D-33098 Paderborn, Germany
[wagner|hg|duke]@upb.de

Abstract

The problem of inconsistency detection and management is central to the development of large and complex software systems. Although sufficient support for model consistency is a crucial prerequisite for the successful and effective application of CASE tools, the current tool support is rather limited. In this paper we present a flexible and incremental consistency management realized in the Open Source UML CASE tool Fujaba. The consistency management is highly configurable and can be adapted individually to user, project or domain specific demands. For the specification of syntactical consistency rules we provide a built-in formalism based on graph grammars.

1. Introduction

Nowadays, the modeling of large object-oriented software systems incorporates many informal and semi-formal notations, with the Unified Modeling Language (UML) [19] being the most prominent example. The UML provides different diagrams describing the system under construction from different, partly overlapping view points, e.g. class diagrams for the static structure and state charts for the behavior of the system. This separation of concerns on the one hand reduces the complexity of the overall specification, but on the other hand the increasing number of used notations very often leads to a wide range of inconsistencies [9, 10]. For example, syntactical inconsistencies violating the well-formedness of models, behavior inconsistencies between different diagrams [7] or inconsistencies during refinement of diagrams [5]. Of course, the general problem of inconsistency is much broader and comprises a large number of disciplines. For a survey we refer to [17] and for a research agenda to [8].

Meanwhile, there are many commercial and Open Source UML modeling tools available. These tools promise increasing development productivity and a gain in quality of the system under development, e.g. by automated code generation from the design models. However, the built-in consistency management of many tools is not satisfactory.

One reason for this is, that tools often try to enforce consistency and do not allow temporal inconsistencies during development. In [1, 17], Nuseibeh et al. emphasize the importance of tolerating inconsistencies and propose a conceptual framework for inconsistency management which allows inconsistencies to be ignored, deferred, circumvented, ameliorated, or resolved.

Another reason is, that most tools with consistency checking support only consistency checks on user demand [18, 20]. In those tools, the whole specification is checked even if no changes were made. However, starting from a consistent state, inconsistencies are introduced by model changes. Hence, a smarter approach will monitor model changes and check only the fraction of the model which was modified. Thus, the effort spent on consistency checking will be

reduced.

Another problem concerns the flexibility of consistency rules being checked. In most CASE tools, the consistency checks being performed are rather static and predefined as they are hard coded into the tool [15]. Thus, new consistency rules neither can be added nor can existing consistency rules be adapted to special user, enterprise, project, target language, or domain specific demands. However, during large projects you will never obtain a complete set of rules covering all relevant inconsistencies. In fact, the set of consistency rules will be expanded and refined through the whole lifecycle of a project [17]. Thus, for a tool developer it becomes infeasible to identify all consistency rules in advance.

In this paper we present a plug-in for a flexible and incremental consistency management realized within the FUJABA TOOL SUITE [3]. FUJABA itself is an Open Source UML CASE tool project. It was started by the software engineering group at the University of Paderborn in fall 1997 and has a special focus on code generation from UML diagrams resulting in a visual programming language. Hence, consistency management was an important issue from the beginnings since consistent specifications are a required prerequisite for an error-free implementation.

The structure of the paper is as follows: In the next section we will present the architecture of our consistency management plug-in. We will discuss in more detail how a more appropriate consistency management should look like and how the previously discussed techniques are realized in our plug-in. Section 3 explains the requirements to be fulfilled when specifying a consistency rule for our plug-in. This is accompanied by an example of the built-in consistency rule specification based on graph-grammars. Section 4 gives an overview about different applications for our consistency management and outlines how domain specific consistency rules are employed. Finally, we conclude and outline future work.

2. Architecture

As outlined in the introduction, sufficient support for model consistency is a crucial prerequisite for the successful and effective application of CASE tools. The current support of this requirement is, however, rather limited. Either the consistency is enforced and therefore often hinders the systematic development of

appropriate solutions by rather tedious restrictions or consistency can only be checked on request and then usually results in a large number of more or less syntactical errors. What is instead required is a more flexible and incremental consistency management.

To understand how a more appropriate consistency management should look like, we first have to understand the different phases of (1) detecting a possible inconsistency, (2) checking that is indeed an inconsistency, and (3) resolving the inconsistency w.r.t. a given consistency rule. The general lifecycle of a model modification w.r.t. consistency is depicted in Figure 1.
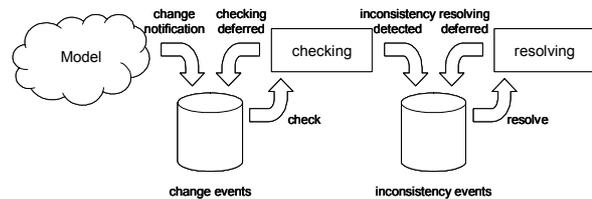
Figure 1. Consistency lifecycle of a model modification

To detect a possible inconsistency, most tools usually simply wait for the user command to check all rules. A better idea realized within the presented plug-in is to observe the model modifications and employ an on demand consistency management approach that reacts on different change events or specific user commands in a flexible manner. The incremental checking algorithms employed in the plug-in only study that fraction of the system which has been modified and thus often the effort to check consistency can be drastically reduced.

For each change event we check whether it results in a modification that is inconsistent. The change events are processed by analysis engines which usually simply report detected inconsistencies via related events. In more complex cases multiple analysis engines may cooperate via higher level change events which permit to decompose the detection of inconsistencies into a sequence of less complex steps. If possible, automated checking mechanisms are employed. If, however, only user inspection can decide whether we have a real inconsistency or not, the consistency management plug-in keeps track of the required manual inconsistency checks and informs the user.

For the final step of resolving an inconsistency, also automatic as well as manual handling is supported. We decide to not combine these two last phases, as a proper consistency management system must tolerate that a

detected inconsistency remains in the model until it is resolved in a later stage of the development when the required context information is available.

For the automated detection, checking and repair mechanisms, we provide a built in formalism which employs graph grammars. Additionally, external tools as for instance model checkers can be integrated for checking and repair. Therefore, different tools which transform the model into a specific semantic domain and then employ their special checking or repair algorithm can be integrated (see Section 4).

As the different employed techniques can show rather different run-time complexity, we permit to configure for each rule whether automatic checking and/or repair is executed automatically or on demand.

Starting from a consistent model, inconsistencies can only be introduced by model changes. In our tool, models are changed by executing user commands from predefined pull-down or pop-up menus. In general, a user command is not an atomic operation and usually consists of more than one modification to the meta-model instance. Thus, inconsistencies during the execution of a user command are not unusual and are often temporary. A consistency check after each single modification will produce unnecessary overhead. Moreover, the consistency management will inform and confuse the engineer with false positive inconsistency notifications.

To tackle this problem, the automatic consistency checking within our plug-in is not executed until a user command is completed. During the execution of a user command all references to the modified objects are stored in a set. Additionally this set is marked with the command's name responsible for the modification, i.e. the cause of the modification. This allows us to specify some consistency rules and/or repairing actions which will be not executed after this special command. Once the user command complete, the set is stored in the change event queue and is ready for being checked.

The consistency checking runs in the background and processes each change event, i.e. each set of modified elements, in a sequential manner. Nevertheless, the background processing of change events can result in problems due to the parallel user interactions. To overcome this problem the user interactions and the automatic checking are synchronized using a coordination strategy which gives user interactions a higher priority to not hinder the engineer. However, the consistency checking of a rule cannot be interrupted at any given point of time since the analysis results may become invalid after a user interaction. Therefore the checking mechanism is interrupted only after a consistency rule was completely checked and before a new rule is executed. To guarantee a short response time for user interactions, the execution time of consistency rules has to be short. Consistency rules with a long execution time will block up the CASE tool and should only be executed on demand or in parallel in the background.

If an inconsistency is detected, it can be resolved either automatically or manually. In the case of automatic resolution of inconsistencies, some consistency rules and repair actions may introduce new inconsistencies. Even worse, contradictory consistency rules and repair actions can result in non terminating chain of change events, checking activities, inconsistency events, repair activities and change events. To prevent such a cyclic execution and to detect repair actions which introduce inconsistencies, the algorithm was extended and works in two phases.

In the first phase each necessary repair activity is executed. During a repair, all elements modified by the appropriate repair action are stored and it is noticed which consistency rule caused the repair action to be executed.

In the second phase, the consistency rules are executed once again for the previously stored elements collected during the preceding repair. If an inconsistency is detected, it is checked whether this inconsistency already occurred during the preceding checking and repair phase w.r.t. the current consistency rule. If the inconsistency occurs for the first time this means, that a repair action is faulty. Otherwise, i.e. if the inconsistency was already repaired in the first phase, a contradictory rule introduced the inconsistency once again. In both cases a further repair action is not executed. Instead, the user is informed about the detected problems and has to resolve the conflict in the consistency rule(s) and repair action(s).

In Figure 2 the part of the consistency management system architecture relevant for the user is summarized. The rules are organized in catalogues and different categories, which permits to support domain or project specific consistency management system configurations as discussed in the following subsection. The system employs a given configuration and binds the rule catalogue and its analysis engines at run-time into the FUJABA CASE tool. These rules are then applied to the currently developed model with the beforehand outlined execution options.
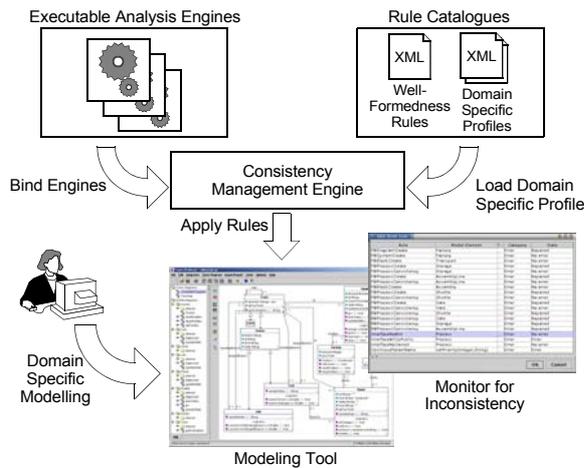
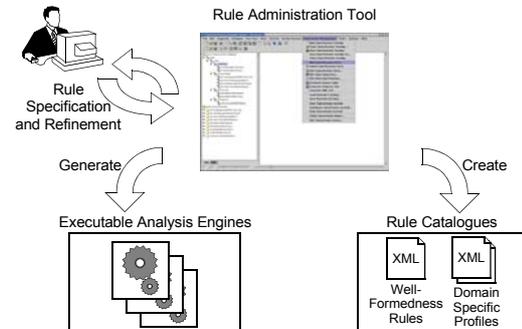**Figure 2.** Using the consistency management tool

For example, consistency rules and categories can be activated or deactivated by the user on demand. If a category is deactivated, the entries of the category are not considered and thus all the rules within a category are not checked. This way, the engineer can adjust the consistency checking to her/his current needs.

With another setting the user can specify whether or not she/he will be informed immediately about any detected inconsistencies. If the immediate notification option is selected, the system informs the user as soon as an inconsistency is detected by displaying an information dialogue. In most cases, especially for minor inconsistencies, the option will be deactivated in order to not hinder the engineer.

If an inconsistency is detected, it can be resolved either automatically or manually. This behavior can be controlled by the repair option. If no repair action is specified or the automatic repair is disabled, the developer is informed about the unresolved inconsistency and must resolve it manually. This way, the automatic resolution of inconsistencies can be also circumvented by the user if required.

The architecture relevant for the administration of the consistency management system is visualized in Figure 3. The plug-in permits to first develop the required consistency rules with a UML extension based on graph grammars as outlined in the following section. Additionally the rules can be organized in form of rule catalogues. Thus, catalogues for different domains, projects or even process phases can be developed. Such pre-defined catalogues can then be loaded on demand or at start-up of the plug-in to

configure the FUJABA CASE tool to the project specific needs of the developer and its current activity. The engineer may switch on demand between different profiles containing different consistency rules, e.g. one profile for the analysis and another for the design life-cycle.

Figure 3. Rule administration in the consistency management tool

3. Rule Specification

The Unified Modeling Language (UML) is defined by the abstract syntax of the underlying meta-model [19]. We will use a simplified fragment of this logical model as a running example.
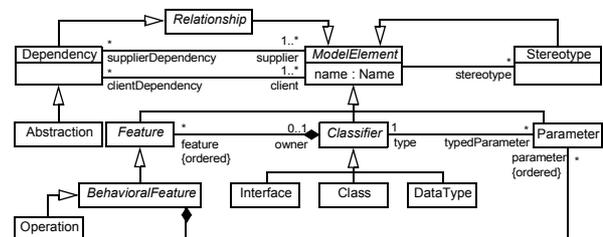

Figure 4. Simplified fragment of the UML meta-model

Figure 4 shows the simplified fragment of the meta-model concerning class diagrams. To simplify the meta-model fragment we have omitted some classes in the original inheritance hierarchy, e.g. the meta-classes *Stereotype* and *Classifier* inherit directly from the *ModelElement* meta-class and not - as in the original definition - from the *GeneralizableElement* meta-class. We have also omitted classes not needed for our example, e.g. *Association* and *StructuralFeature*.

Figure 5. Inconsistent UML model

In Figure 5, a sample class diagram is shown. The diagram contains the interface *Runable* and the class *Thread* which are connected by a realization dependency. Hence, the *Thread* class has to realize the operation *run()* from the interface *Runable*. Note that this does not hold for our example since the class *Thread* does not define an appropriate operation *run()*.
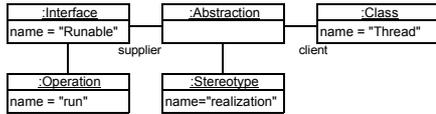
Figure 6. Meta-model instance

In Figure 6, our sample class diagram is represented as a meta-model instance using the simplified meta-model fragment from Figure 4. The object of type *Class* represents the *Thread* class. The instances of type *Interface* and *Operation* represent the *Runable* interface and its operation *run()*. The realization dependency is represented by an instance of *Abstraction* and an associated *Stereotype* instance named realization. The *Interface* and the *Thread* have links to the *Abstraction* object. The former acts in a supplier role whereas the latter acts in the client role.

Every instance of a meta-class has to conform to the abstract syntax specified by the meta-model. To obtain a syntactically correct model, additional well-formedness rules have to be specified. In the UML, well-formedness rules are notated as Object Constraint Language (OCL) expressions.

```
context Classifier inv:
  self.specification.allOperations->
    forAll(interOp|self.allOperations->
      exists(op|op.hasSameSignature(interOp)))
```

Figure 7. Well-formedness rule as OCL expression

A relevant well-formedness rule of the UML states that for each *Operation* in a specification realized by the *Classifier*, the *Classifier* must have a matching *Operation* [19]. The corresponding OCL expression is depicted in Figure 7. Note, that our example in Figure 5 violates this well-formedness rule because the realizing class *Thread* does not declare a counterpart for the specified interface operation *run()*.

```
context Classifier:
  specification : Set(Classifier);
  specification = self.clientDependency->
    select(d|
      d.oclIsKindOf(Abstraction)
      and d.stereotype.name = "realization"
      and d.supplier.oclIsKindOf(Classifier))
    .supplier.oclAsType(Classifier)


context Classifier:
  allOperations : Set(Operation);
  allOperations = self.allFeatures->
    select(f|f.oclIsKindOf(Operation))


context BehavioralFeature:
  hasSameSignature(b:BehavioralFeature):Boolean;
  hasSameSignature(b) =
    (self.name = b.name) and
    (self.parameter->size = b.parameter->size) and
    Sequence{1..(self.parameter->size)}->
      forAll( index:Integer|
        b.parameter->at(index).type =
          self.parameter->at(index).type and
      b.parameter->at(index).kind =
        self.parameter->at(index).kind)
```

Figure 8. Additional operations used in the OCL rule

Although the presented OCL expression seems to be quite small it has to be mentioned for clarity that some additional operations are needed. They are depicted in Figure 8. The operation *specification* yields the set of *Classifiers* that the current *Classifier* realizes. The operation *allOperations* results in a set containing all *Operations* of the *Classifier* itself and all its inherited *Operations*. The operation *hasSameSignature* checks if the argument has the same signature as the instance itself.

The main drawback of the OCL is the fact that it is not an operational language. Hence, the OCL lacks the ability to modify object structures which is a required prerequisite for automated resolving of inconsistencies.

The modification of object structures is a well known application for graph-grammars [21]. Hence, we have decided to use graph rewriting rules both to specify the properties to be checked and the resolution actions for detected inconsistencies. The idea of using graph-grammars for the visual specification of consistency rules and appropriate repair actions is not new [2, 6]. We rely on these concepts and show how they can be applied in a CASE tool (cf. [24]).

Let us return to our example and see how the well-formedness rule from Figure 7 can be expressed with a graph-grammar. The consistency rule depicted in Figure 9 contains the specification of a counter example, i.e. it specifies an inconsistent situation in the meta-model instance that is not allowed.
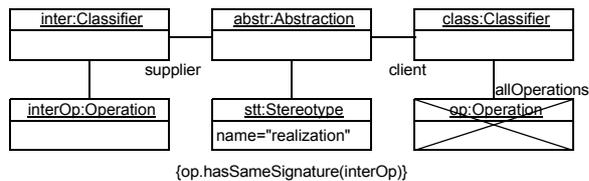
Figure 9. Inconsistency rule expressed as a graph-grammar

Compared to the object-structure of our concrete example in Figure 6 the specified rule is more general. It conforms to the OCL well-formedness rule since it describes the general situation between specification and realization classifiers of different kinds. Thus, this rule also holds for our example where the specification classifier is of type *Interface* and the realization classifier is of type *Class*.

In a graph-grammar based consistency check, a graph rewriting system will search for the pattern and as soon as a match has been found an inconsistency will be reported to the user.

The consistency checking is performed by applying the graph-grammar rule as follows: After a change event has occurred, the applicability of the rule is checked by looking for a match of the modified model element in the specified graph-grammar. If the element can be mapped to one of the graph-grammar nodes, the remaining structure is searched for. If there is no such match in the consistency rule, or if the remaining structure cannot be found, the consistency rule is not applicable for this element.

If a match for the whole structure is found, all negative application conditions (NAC) [12] have to be checked. They are specified by cross out nodes and express that some parts must not exist for a rule to be applied. The NAC node may be accompanied by an additional condition specifying the forbidden node in more detail.

In our example the additional condition *op.hasSameSignature(interOp)* express that only those *Operation* nodes have to be examined which have the same signature as the operation *interOp* defined in the specification classifier. If no matching operation *op* in the realizing classifier is found, i.e. no operation having the same signature as the operation *interOp* defined in the specification classifier, an inconsistency is found. Otherwise, either the rule was not applicable or there is already an operation *op* in the realizing classifier which realizes the operation *interOp* from the specification classifier.

The repair activity is only executed if automatic inconsistency resolving is activated. In our example the repair activity is simply performed by creating and adding the missing *Operation* instance *op* to the realization classifier and assigning it the signature from the operation *interOp* defined in the specification classifier. It is also specified using a graph grammar and is depicted in Figure 10.
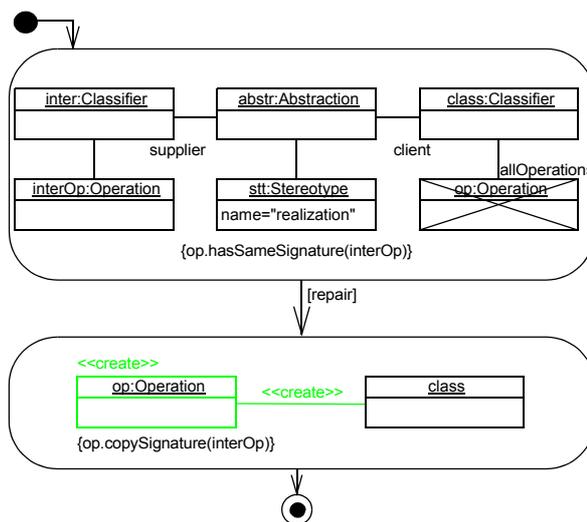
Figure 10. Repair action for automatic inconsistency resolution

Handling inconsistencies may be as simple as adding or deleting some information from the description. However, in some cases resolving inconsistencies immediately is not always possible or desired [1, 17]. If, for example, the user deletes a realization operation it will be not very clever to recreate the deleted operation just to ensure consistency. Often additional information is needed from the engineer. Therefore, the automated repair may remain unspecified.

In our opinion, visual expressions are more readable than their textual counterparts. Of course, it is rather a matter of taste and education whether a visual representation is more readable than a textual OCL expression or the other way round. Therefore, our consistency checking architecture is not bound to a particular rule specification technique. Thus, it is also possible to integrate a checking mechanism which employs OCL expressions (cf. [4]).

4. Applications

As already mentioned, we implemented the described consistency management within the FUJABA TOOL SUITE. Therefore, we were able to evaluate our results in practical use, namely in the ISILEIT project. It deals with the agent-oriented specification and modelling of distributed flexible production control systems [16]. The considered case study consists of a track based material flow system with autonomous shuttles. These shuttles supply the robots, which perform different productions tasks, autonomously with material.

Within the project we have integrated parts of the Specifications and Description Language (SDL) [14] and the UML resulting in an executable specification language. The integration mainly refers to the SDL-UML mapping defined in [13]. Of course, the SDL model and the corresponding UML model have to be consistent, according to these mapping rules. These consistency checks are performed by our consistency management system.

Another aspect of the case study is that we have to generate code for different target platforms. We use Programmable Logic Controllers (PLCs) [22, 23] to control the stations of the production line. Therefore, we have to take some restrictions into account. For example, when specifying the behavior of a PLC, we do not allow the instantiation of objects at runtime, because dynamic memory allocation is not possible for PLCs. Moreover, the statechart, which defines the reactive behavior of the PLC must not terminate. These two requirements are proved statically by applying consistency rules as described in section 3.

A second application of the presented consistency management of the FUJABA TOOL SUITE is currently developed within the Special Research Initiative 614 -- Self-optimizing Concepts and Structures in Mechanical Engineering -- University of Paderborn. A compositional model checking approach for real-time UML models of mechatronic systems [11] is supported by visualizing the validity of model checking results for given temporal logic constraints in the UML diagrams in a consistent manner.

At first, the consistency management is used to detect whether a model change might invalidate a temporal logic constraint that has been checked beforehand. If a model change might have affected the constraint, the status of the constraint is changed to „*unknown*". Then, the model checking is initiated and

is processed in background mode to check the constraints for the relevant fraction of the modified model. While in the usual case the scalability problems of model checking renders such an approach impractical, the compositional nature of the employed approach ensures, that only small models have to be checked when changing the model. When the background model checking process terminates, the analysis result is integrated into the model by setting the constraint as either „true" or „false" depending on the outcome of the process.

If in the meantime before the model checking process has terminated a related model element has been changed again, the model checking process is aborted. To avoid the overhead of frequent starts and aboard of model checking processes for a specific constraint, the initiation of background processes is delayed until no change activities which could effect the constraint has occurred in a reasonable time frame.

Another problem arises when integrating the model checking results into the current model. As the model checking process runs in parallel in the background, directly modifying the model from within this process or a related Java thread could result in inconsistent model updates due to concurrent access to the data. Thus, the consistency management offers also help to schedule the required model updates in such a manner that problems due to the concurrent access can be excluded.

5. Conclusion and Future Work

In this paper we described a flexible and incremental consistency management, which defines a three-phase consistency lifecycle. The phases are decoupled to allow a detection and resolution of inconsistencies at different points of time. Concerning these points of time, each rule can be configured, depending on its complexity and the needs of the developer. Such rules can be combined to rule catalogues to support the software for different application domains. Moreover, we described a graphical and operational specification language, based on graph grammars, which enables us to define consistency rules and repair actions.

Our future work focuses on a more convenient way to monitor the inconsistencies for the user. A major task is to annotate the inconsistencies directly in the diagrams to provide a visual feedback. Further on, our approach allows us to incorporate other tools to

perform more complex consistency checks. We hope to gather more experience concerning the usage of external tools within our system, which also affects a more intelligent monitoring of checking results.

6. References

[1] R. Balzer. Tolerating inconsistency. In *Proc. of the 13th International Conference on Software Engineering, Austin, Texas, USA*, pages 158–165. IEEE Computer Society Press, 1991.

[2] P. Bottoni, M. Koch, F. Parisi-Pressice, and G. Taenzer. Consistency checking and visualization of OCL constraints. In *UML 2000*, LNCS 1936. Springer Verlag, 2000.

[3] S. Burmester, H. Giese, J. Niere, M. Tichy, J. Wadsack, R. Wagner, L. Wendehals, and A. Zündorf. Tool integration at the meta-model level within the fujaba tool suite. In *Proc. of the Workshop on Tool-Integration in System Development (TIS), Helsinki, Finland, (ESEC / FSE 2003 Workshop 3)*, September 2003.

[4] B. Demuth, H. Hussmann, and S. Loecher. OCL as a specification language for buisness rules in data base applications. In M. Gogolla and C. Kobryn, editors, *6th International Conference on the Unified Modeling Language, Toronto, Canada*, LNCS 2185. Springer Verlag, October 2001.

[5] A. Egyed. Automatically validating model consistency during refinement. In *Technical Report, Center for Software Engineering, University of Southern California, Los Angeles*, October 2000.

[6] H. Ehrig and A. Tsiolakis. Consistency analysis of UML class and sequence diagrams using attributed graph grammars. In H. Ehrig and G. Taentzer, editors, *ETAPS 2000 Workshop on Graph Transformation Systems, Berlin, Germany*, 2000.

[7] G. Engels, J. Küster, L. Groenewegen, and R. Heckel. A methodology for specifying and analyzing consistency of object-oriented behavioral models. In V. Gruhn, editor, *Proc. of the 8th European Software Engineering Conference (ESEC)*, pages 186–195. ACM Press, September 2001.

[8] A. Finkelstein. A foolish consistency: Technical challenges in consistency management. In M. T. Ibrahim, J. Küng, and N. Revell, editors, *Proc. of the 11th International Conference on Database and Expert Systems Applications (DEXA'00), London, UK*, LNCS 1873, pages 1–5. Springer Verlag, September 2000.

[9] C. Ghezzi and B. Nuseibeh. Special issue on managing inconsistency in software development (1). *IEEE Transactions on Software Engineering*, 24(11):906–1001, November 1998.

[10] C. Ghezzi and B. Nuseibeh. Special issue on managing inconsistency in software development (2). *IEEE Transactions on Software Engineering*, 25(11):782–869, November 1999.

[11] H. Giese, M. Tichy, S. Burmester, W. Schäfer, and S. Flake. Towards the compositional verification of real-time uml designs. In *Proc. of the European Software Engineering Conference (ESEC), Helsinki, Finland*, September 2003.

[12] A. Habel, R. Heckel, and G. Taenzer. Graph grammars with negative application conditions. *Fundamenta Informaticae*, 26(3,4):287–313, 1996.

[13] International Telecommunication Union (ITU), Geneva. *ITU-T Recommendation Z.109: SDL Combined with UML*, November 1999.

[14] International Telecommunication Union (ITU), Geneva. *ITU-T Recommendation Z.100: Specification and Description Language (SDL)*, 2000.

[15] C. Nentwich, W. Emmerich, and A. Finkelstein. Flexible consistency checking. In *Research note, University College London, Dept. of Computer Science*, 2001.

[16] U. Nickel, W. Schäfer, and A. Zündorf. Integrative specification of distributed production control systems for flexible automated manufacturing. In M. Nagl and B. Westfechtel, editors, *DFG Workshop: Modelle, Werkzeuge und Infrastrukturen zur Untersttzung von Entwicklungsprozessen*, pages 179–195. Wiley-VCH Verlag GmbH and Co. KGaA, 2003.

[17] B. Nuseibeh, S. Easterbrook, and A. Russo. Leveraging inconsistency in software development. *IEEE Computer*, 33(4):24–29, April 2000.

[18] Object International. *Together Control Center, the Together case tool. Online at http://www.togethersoft.com*.

[19] OMG. *Unified Modeling Language Specification Version 1.5*. Object Management Group, 250 First Avenue, Needham, MA 02494, USA, September 2002.

[20] Rational. *Rose, the Rational Rose case tool. Online at http://www.rational.com*.

[21] G. Rozenberg, editor. *Handbook of Graph Grammars and Computing by Graph Transformation*, volume 1. World Scientific, Singapore, 1999.

[22] SIEMENS AG, Nürnberg. *SIMATIC S7-300 - The universal PLC (Product Brief)*, September 2001.

[23] SIEMENS AG, Nürnberg. *S7-300 Automation System, Hardware and Installation:CPU 31xC and CPU 31x*, June 2003.

[24] R. Wagner. Realisierung eines diagrammübergreifenden Konsistenzmanagement-Systems für UML-Spezifikationen. Master's thesis, University of Paderborn, Department of Mathematics and Computer Science, Paderborn, Germany, November 2001.