Consistency Problems in UML-Based Software Development

Zbigniew Huzar[1], Ludwik Kuzniarz[2], Gianna Reggio[3], and Jean Louis Sourrouille[4]

[1] Department of Computer Science, Worcław University of Technology, Worcław, Poland
[2] School of Engineering, Blekinge Institute of Technology, Ronneby, Sweden
[3] DISI, Università di Genova, Genova, Italy
[4] Department of Information Technology and Computer Engineering, INSA, Lyon, France

Abstract. This survey of the workshop series *Consistency Problems in UML-based Software Development* aims to help readers to find the guidelines of the papers. First, general considerations about consistency and related problems are discussed. Next, the approaches proposed in the workshop papers to handle the problems are categorized and summarized. The last section includes extended abstracts of the papers from the current workshop.

1 Why Consistency?

The introduction of the first workshop could have been the same for the series: *The Unified Modeling Language (UML) has become an industrially accepted standard for object-oriented modeling of large, complex systems as well as a basis for software development methodologies. During the development process, artifacts representing different aspects of the system are produced. The artifacts should be properly related to each other in order to form a consistent description of the developed system. The problems concerning and related to consistency between diagrams and models produced within the UML-based development process are presented and discussed within the scope of the workshop. In particular, two kinds of problems concerning consistency are addressed – those related to consistency between diagrams within a given model and named as an intra-consistency problem and those concerning consistency between different models and named as an inter-consistency problem. The papers selected and included in the workshop materials are intended to present a spectrum of problems that occur when consistency is concerned, starting from a general perspective and methodology for systematic checking of consistency, through possible ways of extending UML to enable consistency checking and checking consistency through model transformations, followed by examples of practical realization of the checking in practice and possible tools support, ending with formalization of the notions of consistency.*

The number of submissions and participants shows the importance of the issue. Each workshop proposed to focus on particular topics: *consistency definition and* *verification* (I), *examples of inconsistencies* (II), and *dependency relationship* (III). However, the papers tackle problems in all areas related to consistency, from several points of view and using various approaches.

1.1 Intra-model Consistency

Consistency problems do not seem to arise in many notations such as programming languages. So, a preliminary question is: Where are the UML consistency problems coming from?

When using the UML during the development process, many artifacts representing different aspects of the system are produced, and these artifacts should be properly related to each other in order to form a consistent description of the developed system. There are two main reasons for having many different UML artifacts describing the same system:

- multiview nature of UML models: at some level of abstraction a system is described as a collection of views dealing with different, possibly overlapping, aspects,
- the system is developed throughout different phases and iterations, with each one producing a new, more refined description of the system.

Another source of inconsistency is the imprecise semantics of the UML. A UML expression (i.e., a set of model elements) may have multiple interpretations, among which some are inconsistent. Why is the UML semantics not precise? It was the wish aim of the UML authors not to supply a precise definition of the UML to broaden the area in which the UML applies. An advantage is that such imprecise UML models can be implemented in many ways. The counterpart is that we do not know if there is one possible implementation of a UML model. This issue is called intra-model or horizontal consistency. For instance, intra-consistency is expected between model elements representing the static and dynamic views of the modeled domain.

1.2 Inter-model Consistency

Furthermore, consistency problems arise in the UML because there is no definition of relationships between models preserving consistency such as the refinement relationship. A UML–based software development is a modeling process. From the requirements to the code, the software development process produces more and more detailed models. A model is a collection of UML model elements that represent a system at a given level of abstraction. At each level, the produced model should be consistent with the models at the upper, more abstract levels. This issue is called inter-model or vertical consistency. For example, a design model should be inter-consistent with an analysis model.

1.3 Main Issues Related to Consistency Addressed in Contributions

The papers presented and discussed during the workshops deal with the following important issues: definition of consistency, relationships between consistency and development process, approaches to check consistency, and checking tools. The positions are briefly summarized below. Regarding tools, the two main approaches are:

- to check directly that the UML model has the required properties (expressed by OCL or other means), using standard tools when available, and
- to translate the model into a formal language such as B or production rules, and then to perform checks using companion tools of the target language.

2 A Survey of the Workshop Contributions

2.1 Consistency Definitions

Rule-Based Definitions

The semantics of the UML includes constraints that induce restrictions on the use of notations in order to ensure that model interpretations are licit [26]. To avoid inconsistencies and to make the semantics precise, most papers propose adding constraints or well-formedness rules such as the UML ones [4,7,8,9,11,19,21,25, 26,27,30]. A model is inconsistent when it violates the added constraints, i.e., when there is no licit interpretation [27]. In [20], a class diagram is consistent if there is at least one instantiation possible that satisfies all the diagram constraints. UML artifacts form a hierarchy and all the components of an artifact should be intra and inter consistent for the artifact to be consistent [11]. Some papers only deal with model properties that do not ensure the entire model consistency: the behavior should be deadlock free [24], sequence diagrams should be consistent with statecharts [3,15], etc.

[2] presents an approach to define which UML models are intra-consistent following an algebraic approach, that is distinguishing in a UML model a "signature" which defines the model vocabulary, which is then used to check the well-definedness of the other parts in quite a modular way.

Refinement

Furthermore, constraints are added to enforce the inter-model consistency, i.e., to define the refinement relationship. Applying the ODP consistency approach [6], a set of specifications (models) is consistent if there exists a specification that is the refinement of each of the specifications in the set with respect to a refinement relationship. In [21], consistency constraints include conformance to standard, good practice and stakeholders´ specific constraints. [13] presents a general framework for defining refinement relationships between UML models, trying to distinguish between abstraction refinement and semantics refinement, where only the first may be automatically checked.

Translational Definitions

Adding constraints can be seen as a declarative approach. In a translational approach, a model is consistent if its translation into a formal language (such as B or Object-Z) satisfies some good properties [6,24,23,20,3]. This approach does not enforce the entire UML model consistency, for instance in [20] only class diagrams, object diagrams and statecharts are taken into account, while [24] only deals with behavior. Quite different, but also based on transformations, graphical consistency conditions specify the situations that must not occur [16]. [22] introduces a formal language OOL, and proposes transforming a subset of UML models into OOL specifications. The well-defined consistency and a refinement calculus of OOL are then used to check the corresponding UML models.

Constraint Completeness

A further question is to write the entire list of constraints: examples of classification are given (between pairs of diagrams in [8], by abstraction levels in [26]), but it is likely that no complete list exists. Assuming that syntactic rules are expressed formally and semantic rules use natural language, if all the constraints cannot be expressed by syntactic rules, consistency cannot be checked automatically [21,27].

Role of Dependency in Defining Consistency

[29] presents a UML profile allowing one to express dependency relationships among model elements characterized by behavioral properties, such as *call/update/access* preservation, to help establish correct refinement among models. These relationships are formally defined using Description Logic. Similarly, [17] sketches another profile for expressing different kinds of dependency, precisely implicit and explicit usage among model elements.

2.2 Consistency and Development Process

Refinement

During the development process, model consistency should be preserved through refinement: Object-Z and CSP provide refinement concepts for checking the translations of UML models in [24], while in [16] model transformations are expressed using graphs. Another approach proposes defining a profile with transformation rules using the UML extension mechanisms [25].

Development Methodology

Moreover, models should be consistent with the development methodology or process (e.g., USDP: Unified Software Development Process in [11], COMET in [8], general process in [19]). In [11] a three-layer framework is adapted to the development process, while [6] uses the ODP principle of viewpoints (i.e., partial specifications) to check UML consistency. Good practice rules and specific development rules should also be added [21] or followed [18], preferably in a UML profile [25,27]. [5], instead, considers the consistency problem in the component-based development process *KobrA*. [14] considers the problem of the consistency among the artifacts produced following the USDP and proposes a UML profile expressing such artifacts and defining rules expressed with OCL to enforce consistency; such rules are then checked using any standard OCL tool. [2] proposes a UML based development method which requires models to be produced with a precise structure, and equipped with guidelines helping to detect the most probable inconsistencies.

Incompleteness
Several authors underline that the under-specification of the UML induces incompleteness [26,19], while models should be complete for consistency checks. Rules can be checked on existing models, and examples of results given in [19] show that inconsistencies are related to the development practices of the designers.

Domain Specific Cases
[10] presents the consistency aspects of the MERODE method for developing information systems; the method is based on the formal language CSP and proposes the use of views of three different kinds, with two having a UML-like syntax. [1] treats consistencies due to a too rigid application of design patterns; to avoid these they propose presenting patterns using an extension of the UML 2.0 collaboration template, which allows to constrain the parameters and to perform some actions at the instantiation time, such as deletion of model elements.

2.3 Consistency Checking

Most papers deal with either intra-model consistency [4,6,7,8,9,11,15,20,21,23,24, 26], some with inter-model consistency [11,16,25], others deal with both, such as [23], which translates models into B that supplies a refinement relationship. Obviously, a tool is required to check consistency, not only to automatically check constraints but also to help users to find and correct errors. Depending on the approach, declarative or translational, tools are faced with different problems. Examples of checks applied to models are given in [14,15,18].

Constraint Checking
Each tool is associated with a suitable representation for the constraints. The most direct way to express constraints is the OCL (Object Constraint Language). The checking tool is standard and could be embedded in the modeling tools, as in [14]. The OCL used to express the rules is enriched with a transitive closure operator and temporal operators in [4], and with actions in [8]. [26,21] use production rules that add reasoning capabilities to constraints, and unlike OCL which is side-effect free, allow actions such as corrections or tips. The graph rewriting rules in [30] describe the resolution actions for detected inconsistencies. Based on rules in XML, the *xlinkit* framework allows checking consistency of models mapped to XML using the XMI [9]. The graphical conditions in [16] are kinds of patterns, and checking constraints comes down to matching graphs in the UML model. [29] describes a tool, RACOoN, for checking consistency conditions expressed in Description Logic by combining a UML tool, an XML translator and a logical reasoning tool.

Model Translation

Model translation into a formal language is very appealing since checking tools already exist. Only the notions common to UML and the target language can be translated, and the inter-consistency definition depends on its refinement relationship. In [6] a detailed discussion of the translational approaches illustrated with LOTOS and Object-Z is given. In [24], static aspects are translated into Object-Z while behavioral ones are translated into CSP: only deadlocks and interface properties are checked. The B specification and the UML model are handled in parallel in [7], but the question of how to automate the translation of UML/OCL models into B is not answered. UML models are decorated with additional expressions to allow the translation into B in [20], but for temporal properties another approach is proposed. LTS and traces are used in [3] to check behavior properties. [5] proposes to reduce the consistency issues into deadlock detection problems to be checked using the SPIN tools. On the other hand, [10] describes a tool, MERMAID, which monitors the constructions of the models required by the MERODE method, helping to ensure their consistency (also if some post-mortem checks are implemented). Simulation approaches do not give proofs but they increase the confidence in the model, e.g., trace validation in [15]. Translation to description logic is suggested in [28] to maintain consistency, together with the use of an accompanying tool to prove the feasibility of the approach. Consistency checking based on the consistency rules expressed as graphs rewriting rules and their implementation in the UML CASE tool is presented in [30].

3 Extended Abstracts

On Understanding of Refinement Relationship

Bogumiła Hnatkowska, Zbigniew Huzar, Lech Tuzinkiewicz

The software development process is both iterative and incremental in nature. Modeling constitutes an important step of this process; its key artifacts are described as models, i.e. abstract representations of the entities being modeled. There are many relationships between models. The «refine» relationship is an interesting one as it reflects the evolution of artifacts within the software development process. The relationship is not precisely defined in the UML standard. Its informal definition relates to other, not well-defined notions: “perspective”, “abstraction level”, and “semantic level”. The paper proposes definitions of these notions in the UML terms. Refinements defined in the paper are based on the change of abstraction levels and on the change of semantic levels. The first kind of refinement is independent of the interpretation of models while the second kind depends on model interpretation. Therefore two models’ categories were introduced: non-interpretable and interpretable, based on the formal definition of abstract and semantic levels. The elaborated definitions may be used for describing different step-wise model transformations.

Consistency and Refinement of UML Models

Zhiming Liu, He Jifeng, Xiaoshan Li, Yifeng Chen

In UML-based software development, artifacts created in the development process are modeled and analyzed from static and dynamic views using different kinds of UML notations. Under the multiple views of UML, the developers can decompose a software design into smaller parts of manageable scales. A development process starts from a system requirement model consisting of a class diagram, a family of sequence diagrams, and a family of state diagrams. Such a model can be established through horizontal requirement incrementally by adding information and incorporating use cases one by one. A development process also cycles through a number of steps of vertical refinement from the requirement model into a system design model. Therefore, the horizontal and vertical consistency are the inevitable challenging issues, which arise from such a multi-view and multi-notational approach.

In this paper, we use a formal object-oriented specification language (OOL) to formalize and combine UML models. With OOL, a specification of an object system is a combination of its class declarations, method declarations and specifications of method bodies. Different sub-models of a system model are formalized as different parts in an OOL specification. The consistency of the different sub-models is defined as the well-formedness of the corresponding OOL specification. With the formalization, we develop a set of refinement laws of UML models to capture the essential nature, principles, nature and patterns of object-oriented design. We can apply the refinement calculus of OOL specifications to treat refinement of system models in UML. With the support of the incremental and iterative features of object-orientation and the Rational Unified Process (RUP), the refinement process will preserve the consistency and correctness of the system.

UML 2.0 Model Consistency – The Rule of Explicit and Implicit Usage Dependencies

Shiri Kremer-Davidson, Yael Shaham-Gafni

The notion of dependency is modeled in UML using the Dependency relation. The UML specification intentionally defines the Dependency concept vaguely in order to serve as a "catch all" relation, describing any relationship that is not a generalization or association. The specification further defines several subtypes of Dependency: Abstraction, Realization, Substitution, and Usage, which have a stronger semantic meaning. For all of these modeling constructs the UML specification does not describe any relation to the behavioral aspects or to model elements representing runtime entities.

In this paper we investigate the runtime implications for the usage dependency. We define the notions of explicit dependencies: dependencies that are explicitly created by the modeler as part of the static aspect of a UML model, and implicit usage dependencies: usages that can be inferred form the behavioral portions of a UML model. Based on these notions we propose a definition for the semantics of the usage dependency and a corresponding consistency notion. We propose an implementation of such semantics and consistency through a UML Profile. We provide an example to illuminate our ideas and describe several scenarios in which having knowledge of the explicit and implicit dependency information and the consistency between them is beneficial.

Consistency Checking of USDP Models

Bogumila Hnatkowska, Anita Walkowiak

The aim of the paper is to propose a method for checking consistency of UML models. Because the content of UML models strongly depends on used methodology it was assumed that models that are basic outcomes of USDP process are considered. Our aim was to improve the USDP process with some mechanisms validating prepared models against some known rules. The rules belong to two categories:

– well-formed rules, defined in UML standard document,
– new well-formed rules resulting from applying USDP for software development.

In the paper three USDP models are refined and formalized, i.e. *Context Model*, *Use Case Model*, and *Analysis Model*. The models are defined in terms of a new language called *Robust Software Development UML* (*RSD_UML*). *RSD_UML* is a part of *Robust Software Development Profile* (*RSDP*). The profile introduces new stereotypes basing on standard UML elements. *RSD_UML* language is defined similarly to UML standard. Its syntax and static semantics are defined formally by OCL expressions, while its dynamic semantics is defined informally, in natural language. It was observed that most of the intra-consistency rules relate to the way of proper construction of models. For example, the rules state that collaboration at given semantic level (e.g. analysis) should represent a behavioral element from the previous model (e.g. requirements). Example models written in XMI were prepared in two different CASE tools, i.e. Rational Rose, and Poseidon for UML. OCL Evaluator was used for models verification against inter and intra-consistency rules.

Formalizing Behaviour Preserving Dependencies in UML

Ragnhild Van Der Straeten

In the context of Model-Driven Development (MDD), models are primary assets that embody a consistent view on the system under study. On the one hand, during model driven software development, software models can evolve into a new version.

Model refactorings are a particular kind of model evolution which preserve the behaviour as specified by the model. On the other hand, within the software development life-cycle, models can gradually be refined resulting in a full-fledged implementation. At every refinement step, this refinement process adds more concrete details to the model. In general, refinements preserve certain correctness issues, e.g. program refinements imply the preservation of program correctness. The behaviour preserving properties identified in this paper for model refactorings can also be used in the context of refinements. These properties express that certain parts of the specified behaviour have to be preserved. In the context of model refinements, these behaviour preserving properties can be interpreted as correctness properties between a certain model and its refined version. In the rest of the paper, we refer to these properties as behaviour preserving properties. The goal of this paper is threefold. First of all, definitions of behaviour preserving properties are given in the context of UML models. During the development process, we also want to indicate between which UML elements or models certain properties are valid. In UML the dependency relationship is used to describe relationships between models and their elements.

However, it lacks a precise definition. Thus, the second aim of the paper is to extend the UML metamodel with specialized dependency relationships expressing the preservation properties. Thirdly, these dependency relationships are formalized using a logic approach. This allows the automatic checking of these relationships between UML models and elements. This is illustrated by a simple but nevertheless representative example.

Behavioral Consistency Checking for Component-Based Software Development Using the KobrA Approach
Yunja Choi, Christian Bunse
The KobrA method is a structured approach for component-based system development, providing a natural way of identifying and refining system components by separating the external view (interface or contract) from its internal view (detailed functionalities and their realization). The method is designed to reduce system complexity by separating concerns and facilitates software reuse, thus, saving time and effort for software development.

Nevertheless, understanding the overall interactions and relations of many components in a KobrA model often goes beyond human capability, mainly due to its way of specifying different aspects of a component in various UML diagrams. For example, statecharts are used to specify the abstract level component behavior and activity/sequence/collaboration diagrams are used to specify detailed internal component behavior. While this approach facilitates a systematic, iterative specification-refinement paradigm, it can also produce unexpected inconsistencies among these different diagrams as well as among the different levels of refinement. A systematic consistency checking mechanism is a must to ensure the basic quality of a system.

In this paper, we aim at providing an overall consistency checking mechanism integrated into the development process of KobrA, named *consistency checking using environment modeling*. We first define generic consistency requirements in the KobrA approach, with an emphasis on the behavioral consistency between different levels of specifications and realizations. The consistency requirements are then reinterpreted as consistency between a set of state transition systems describing the system behavior (*reactive* systems) and a sequence of stimuli describing the system environment (*action* systems). Two behavioral models are considered consistent if the reactive system accepts every stimulus generated by the action system. In this way, we transform various consistency issues into a deadlock detection problem that can be automated. We demonstrate the automated consistency checking using the model checker SPIN on a hypothetical elevator system.

Implementing Consistency Management Techniques for Conceptual Modeling
Raf Haesen, Monique Snoeck
Most software development methodologies justify the use of multiple independent models to represent all aspects at the different stages in the development process. This can make the resulting information system inconsistent at different levels: inconsistencies can arise between different views of a single system, between

documents at different development life cycle stages, or in a single document. The use of a single model and different views to that model can avoid this problem: all views have to be built according to well-formedness rules for that view and consistency between the related views must be checked. In this way it is possible to obtain a model that reaches a feasible level of validity and improved completeness. Validity means that all statements made by the model are correct and relevant to the problem, whereas completeness means that the model contains all the statements about the domain. This paper presents different techniques to maintain consistency of one view and the use of the same techniques to enforce and check consistency between the views. First we discuss the three strategies of consistency management: consistency by analysis, consistency by monitoring and consistency by construction. Finally we present a concrete implementation of these rules in a modeling tool, based on the object-oriented domain modeling method MERODE.

Improving Pattern Support in UML CASE Tool
Samir Ammour, Xavier Blanc, Mikal Ziane, Philippe Desfray
In this paper we improve the UML2.0 Collaboration Templates mechanism to better support patterns in UML CASE tools. In our research and prototyping activities, we have identified that two important problems lead to severe limitations: Collaboration Templates are not versatile enough to support design patterns correctly. First, they constrain their parameters inappropriately. Second, the instantiation of UML Collaboration Templates does not allow us to modify or to suppress model elements, which is sometimes necessary. Both problems make it difficult to maintain the UML models' consistency when applying design patterns. Collaboration Templates may lead to inconsistencies in models. We thus propose to explicitly constrain Collaboration Template parameters using pattern constraints and to allow the suppression or modification of model elements using pattern actions. Pattern constraints are OCL expressions to control which elements can be bound to the template parameters to preserve the consistency of models. Pattern actions are written in an action language such as Action Semantics or an extension of OCL. They are used to modify and delete model elements to remove inconsistencies when applying design patterns. We have prototyped this approach in the Objecteering UML CASE tool. Both these improvements proved quite useful in several applications, and will be included in a future version of the Objecteering CASE tool.

References

I. L. Kuzniarz, G. Reggio, J.L. Sourrouille, Z. Huzar, *Workshop on Consistency Problems in UML-based Software Development I*, UML 2002, Blekinge Institute of Technology, Research Report 2002:06. Available at `http://www.ipd.bth.se/uml2002/`.

II. L. Kuzniarz, G. Reggio, J.L. Sourrouille, Z. Huzar, M. Staron, *Workshop on Consistency Problems in UML-based Software Development II*, UML 2003, Blekinge Institute of Technology. Research Report 2003:06. Available at `http://www.ipd.bth.se/consistencyUML/UML2003workshop.asp`.

III. Z. Huzar, L. Kuzniarz, G. Reggio, J.L. Sourrouille, *Workshop on Consistency Problems in UML-based Software Development III*, UML 2004. Available at `http://uml04.ci.pwr.wroc.pl/`.

1. S. Ammour, X. Blanc, M. Ziane and P. Desfray, *Improving Pattern Support in UML CASE Tools,* in III
2. E. Astesiano and G. Reggio, *An Algebraic Proposal for Handling UML Consistency,* in II
3. P. Bhaduri and R. Venkatesh, *Formal Consistency of Models in Multi-View Modelling,* in I
4. J.-P. Bodeveix, T. Millan, C. Percebois, C. Le Camus, P. Bazex and L. Feraud, *Extending OCL for verifying UML models consistency,* in I
5. Y. Choi and C. Bunse, *Behavioral Consistency Checking for Component-based Software Development Using the KobrA Approach,* in III
6. J. Derrick, D. Akehurst and E. Boiten, *A framework for UML consistency,* in I
7. G. Génova, J. Llorens and J. M. Fuentes, *The Baseless Links Problem,* in II
8. H. Gomaa and D. Wijesekera, *Consistency in Multiple-View UML Models: A Case Study,* in II
9. C. Gryce, A. Finkelstein and C. Nentwitch, *Lightweight Checking for UML Based Software Development,* in I
10. R. Haesen and M. Snoeck, *Implementing Consistency Management Techniques for Conceptual Modeling,* in III
11. B. Hnatkowska, Z. Huzar, L Kurniarz and L. Tuzinkiewicz, *A systematic approach to consistency within UML based software development process,* in I
12. B. Hnatkowska, Z. Huzar, L Kuzniarz and L. Tuzinkiewicz, *Refinement relationship between collaborations,* in II
13. B. Hnatkowska, Z. Huzar and L. Tuzinkiewicz, *On Understanding of Refinement Relationship,* in III
14. B. Hnatkowska and A. Walkowiak, *Consistency Checking of USDP Models,* in III
15. T. Huining Feng and H. Vangheluwe, *Case Study: Consistency Problems in a UML Model of a Chat Room,* in II
16. J. Hendrik Kausmann, R. Heckel and S. Sauer, *Extended Model Relations with Graphical Consistency Conditions,* in I
17. S. Kremer-Davidson and Y. Shaham-Gafni, *UML 2.0 Model Consistency – the Rule of Explicit and Implicit Usage Dependencies,* in III
18. L. Kuzniarz and M. Staron, *Inconsistencies in Student Designs,* in II
19. C. Lange, M.R.V. Chaudron, J. Muskens, L.J. Somers and H.M. Dortmans, *An Empirical Investigation in Quantifying Inconsistency and Incompleteness of UML Designs,* in II
20. K. Lano, D. Clark and K. Androutsopoulos, *Formalising Inter-model Consistency for the UML,* in I
21. W. Qian Liu, S. Easterbrook and J. Mylopoulos, *Rule Based detection of Inconsistency in UML Models,* in I
22. Z. Liu, H. Jifeng, X. Li and Y. Chen*, Consistency and Refinement of UML Models,* in III
23. R. Marcano and N. Levy, *Using B formal specifications for analysis and verification of UML/OCL models,* in I
24. H. Rasch and H. Werheim, *Consistency between UML Classes and Associated State Machines,* in I
25. W. Shen, Y. Lu and W. Liong Low, *Extending the UML Metamodel to Support Software Refinement,* in II

26. J.-L. Sourrouille and G. Caplat, *Checking UML Model Consistency,* in I
27. J.-L. Sourrouille and G. Caplat, *A Pragmatic View on Consistency Checking of UML Models,* in II
28. R. Van Der Straeten, T. Mens and J. Simmonds, *Maintaining Consistency between UML Models Using Description Logic,* in II
29. R. Van Der Straeten, *Formalizing Behaviour Preserving Dependencies in UML,* in III
30. R. Wagner, H. Giese and U. A. Nickel, *A Plug-In for Flexible and Incremental Consistency Management,* in II