JOHANNES KEPLER
UNIVERSITÄT LINZ | JKU

TNF

Technisch-Naturwissenschaftliche
Fakultät

Automated Consistency Management Framework for the Model Based Software Development

DISSERTATION

zur Erlangung des akademischen Grades

Doktor

im Doktoratsstudium der

Technischen Wissenschaften

Eingereicht von:
Alexander Reder

Angefertigt am:
Institute für Systems Engineering und Automation

Beurteilung:
Univ.-Prof. Dr. Alexander Egyed, MSc (Betreuung)
Assoz. Univ.-Prof. Mag. Dr. Wieland Schwinger, MSc

Linz, Jänner, 2013

Eidesstattliche Erklärung

Ich erkläre an Eides statt, dass ich die vorliegende Dissertation selbstständig und ohne fremde Hilfe verfasst, andere als die angegebenen Quellen und Hilfsmittel nicht benutzt bzw. die wörtlich oder sinngemäß entnommenen Stellen als solche kenntlich gemacht habe.

Die vorliegende Dissertation ist mit dem elektronisch übermittelten Textdokument identisch.

Linz, 16.Jänner, 2013

Alexander Reder

Abstract

The development of a software product is a multistage process and we know the later an error is discovered the more costly it is to repair it. Erroneous implementation of requirements are often found very late because the early stages of the development process lack efficient support for detecting and avoiding errors. Integrated development environments used to implement software products provide support for detecting errors, but the detection is limited to the programming languages. These environments typically are not able to check if code really implements the costumer requirements correct, i. e., if the program is semantically correct. Perhaps, this can be done in the design stage, however, the error detection of design modeling tools are often limited to specific languages. Furthermore, many approaches are batch-based where the error detection (the detection of inconsistencies between requirements and their realization) is done occasionally, usually after it has been triggered manually. This time consuming process has the downside this it may provide error feedback late and in potentially large quantities depending on the time that has passed. Apart from that, detecting inconsistencies is worthless if it is not known what to do with this information. Eventually inconsistencies need to be resolved and there are limited alternatives on how to do so. Currently, no solution exists that combines the detection of inconsistencies with their eventual resolutions, including their side effects, in an scalable, efficient, and incremental technology. This thesis represents an approach for model-based software development that supports the user during the entire life cycle of an inconsistency: from the detection to resolution. As a proof of concept, the approach is implemented as an Eclipse plug-in for the IBM Rational Software Architect, the UML modeling language and the OCL constraint language. While the approach is generic and not limited to these tools and languages, the tool implementation was used for the evaluation of applicability,correctness, and scalability. It shows that the approach provides the functionality to define arbitrary constraints, supports instant detection of inconsistencies and proposes solutions how to resolve them as well as calculating the side effects of this solution.

Kurzfassung

Die Entwicklung eines Software Produktes ist ein mehrstufiger Prozess und es ist bekannt, dass je später ein Fehler gefunden wird, umso kostspieliger es ist diesen zu beseitigen. Fehlerhaft umgesetzte Anforderungen werden oft sehr spät entdeckt, da in den ersten Phasen des Entwicklungsprozesses kaum effektive Methoden zur Erkennung und Vermeidung von Fehlern existieren. Werkzeuge, die zur Implementierung verwendet werden, besitzen zwar effektive Methoden zur Erkennung von Fehlern, allerdings sind diese Methoden beschränkt auf die verwendete Programmiersprache. Diese Werkzeuge sind in der Regel nicht im Stande zu überprüfen, ob die Kundenanforderungen korrekt umgesetzt wurden, d. h. ob das Programm semantisch korrekt ist. Unter Umständen kann das in der Design Phase erfolgen, jedoch die Fehlererkennung der Modellierungswerkzeuge beschränkt sich meist auf spezifische Modellierungssprachen. Darüber hinaus sind viele der verwendeten Ansätze auf sogenannter Stapel Auswertung aufgebaut, wo die Fehlererkennung (die Erkennung von Inkonsistenzen zwischen Anforderungen und deren Realisierung) sehr sporadisch passiert, gewöhnlich erst dann, wenn sie von Hand angestoßen wird. Dieser zeitaufwendige Prozess hat denn Nachteil, dass es späte Rückmeldungen über Fehler gibt und sehr viel Information auf einmal liefert, abhängig von der Zeit, die seit der letzten Überprüfung vergangen ist. Abgesehen davon ist die Erkennung von Inkonsistenzen wertlos, wenn man nicht weiß, was mit dieser Information anzufangen ist. Irgendwann müssen diese Inkonsistenzen aufgelöst werden und es existiert eine beschränkte Anzahl an Alternativen dafür. Aktuell existiert keine Lösung, die die Erkennung von Inkonsistenzen, deren Auflösung und die Berechnung derer Effekte, auf eine skalierbare, effiziente und inkrementeller Basis unterstützt. Diese Arbeit stellt einen Ansatz für die modellbasierte Softwareentwicklung dar, der den Benutzer während des gesamten Lebenszyklus einer Inkonsistenz unterstützt: von der Erkennung bis zu dessen Auflösung. Für eine Machbarkeitsstudie wurde dieser Ansatz als Eclipse basiertes Plug-In für den IBM Rational Software Architect, mit UML als Modellierungssprache und OCL als Sprache für die Regeln, realisiert. Während dieser Ansatz generisch ist und nicht auf eine Modellierungssprache oder Regelsprache limitiert ist, evaluiert die Werkzeug Implementierung die Anwendbarkeit, Korrektheit und die Skalierbarkeit. Es zeigt, dass der Ansatz die Funktionen zur Erstellung beliebiger Regeln, der sofortigen Erkennung von Inkonsistenzen und dem Vorschlagen von Lösungen als auch der Berechnung deren Effekte im gesamten Modell, unterstützt.

Danksagungen

Besonderen Dank möchte ich meinem Betreuer Alexander Egyed sagen, der es mir ermöglicht hat, in Universitärem Umfeld diese Arbeit anzufertigen als auch für die wertvolle Unterstützung die er mir in den letzten Jahren zukommen lies. Darüber hinaus möchte ich auch noch meinem Zweitbetreuer Wieland Schwinger dank sagen, für die investierte Zeit und die wertvollen Anregungen.

Auch möchte ich allen danken, die mich im Laufe meines Studiums und auch während meiner Arbeit laufend unterstützt und begleitet haben. Auch möchte ich dem FWF für die finanzielle Unterstützung danken, welche mir die Präsenz auf einigen Konferenzen ermöglichte.

Einen ganz besonderen Dank möchte ich auch noch meiner Familie ausdrücken, die es mir überhaupt erst ermöglicht hat, diesen Bildungsweg einzuschlagen und mir auch in schwierigen Zeiten immer zur Seite gestanden ist. Des Weiteren auch noch einen ganz besonderen Dank an meine Partnerin, die mir als gutes Beispiel voran gegangen ist und mir immer wieder die notwendige Motivation gegeben hat, dieses Werk hier zu vollenden.

Contents

List of Figures

List of Tables

Glossary

AMOR Adaptable Model Versioning

ARL Abstract Rule Language

CAMUS Compute All Minimal Unsatisfiable Subsets

CiM Consistency in Models

CLP Constraint Logic Programming

CNF Conjunctive Normal Form

CQC Checking Query Containment

CSP Constraint Satisfaction Problem

CVS Concurrent Versions System

DOPLER Decision-Oriented Product Line Engineering for effective Reuse

EMF Eclipse Modeling Framework

ER Entity Relationship model

HUMUS High-level Union of Minimal Unsatisfiable Sets

IDE Integrated Development Environment

MCS Minimal Correcting Set

MDT Modeling Development Tools

MOF Meta Object Facility

MUS Minimal Unsatisfiable Set

OCL Object Constraint Language

PLE Product Line Engineering

RSA (IBM) Rational Software Architect

s-DAG suggestion-Directed Acyclic Graphs

SAT Satisfiability Problem

SMT Satisfiability Modulo Theories

SPL Software Product Line

SVN Subversion

UML Unified Modeling Language

XMI XML Metadata Interchange

XML Extensible Markup Language

Chapter 1

Introduction

"To err is human, but to really foul things up requires a computer."
Paul R. Ehrlich, American Scientist, 1932

When using a software product, malfunctions and incorrectly/incomplete implemented requirements cause problematic failures that are costly to localize and expensive to repair. To reduce the danger of such situations, the stakeholder requirements and the quality of the software product should be validated in each step of the software development process. In this thesis we present a framework that enables the validation of arbitrary model constraints during design modeling. The stakeholder requirements as well as well formedness rules from the design language build the set of constraints.

1.1 Starting Point

The development of a software product is a vital process with the goal to achieve the requirements that are specified by the customer. However, during the development process one or more developers try to realize the customers needs. Unfortunately, requirements can be interpreted in many ways, especially at the beginning at the development process and the delivered product does not conform to the product expected by the customer (Figure 1.1).

In the software development process different tools are used that focus on different aspects of the software development, often from the perspective of different stakeholders, such as customers, analysts, designers, implementer, tester, or maintainer. Software development combines these different perspectives in a multistage process. At the beginning is the elicitation of requirements from the customer and at the end the complete product – that should meet the customer needs – is shipped to the customer. In between a team of developers realize the customer requirements. So, the phases of the software development are the elicitation of requirements, designing and implementing the system, shipping it to the customer, and its verification and maintenance.

In the traditional software development the three essential models are the Water Fall model by Royce [14, 97], Rook's V-Model [96], and Boehm's Spiral Model [16]. While

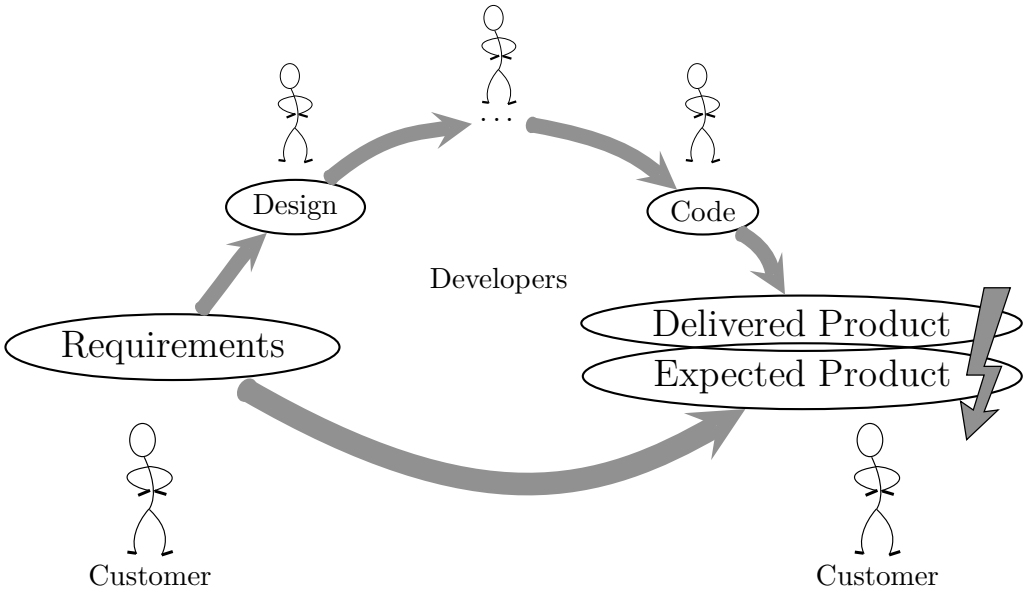


FIGURE 1.1: From the Customer Requirements to the Delivered Software Product

the water fall and V-model are highly structured, the spiral model is more agile, but all include similar stages and during the years tools have been developed to support the participants in the software system development process in realizing their goals. On each stage of this process the needs of the costumer, the team of developers and designers must be considered as well as the domain specific needs, like the model and implementation languages, the environmental constraints, etc.

Aside from the traditional software development methods, more and more agile methods [3] have become popular, like extreme programming [10] or Scrum [99]. These methods are distinguished by a faster incremental and iterative development process. All processes have in common the desire to specify the software quickly and earlier – and they all desire to analyze the correctness and consistency early on.

The different tools used during the software development are rarely interrelated, i. e., a tool that is made for the elicitation of requirements [17] is not connected to the tool that is made for planning or designing the software product. Thus, it is hardly possible to validate if the designed product conforms the requirements, because the tools are able to validate the design against the used domain specifications from the underlying design language, if any. However, Boehm [15] describes the economics of software engineering and the costs to repair errors in the different stages of the software development process, beginning at the elicitation of the requirements until the installation at the costumer. Figure 1.2 shows the relative cost of fixing an error caused by an inconsistency between the requirements and their implementation in the single development stages. This shows how important it is to detect and resolve errors as early as possible during the development process.

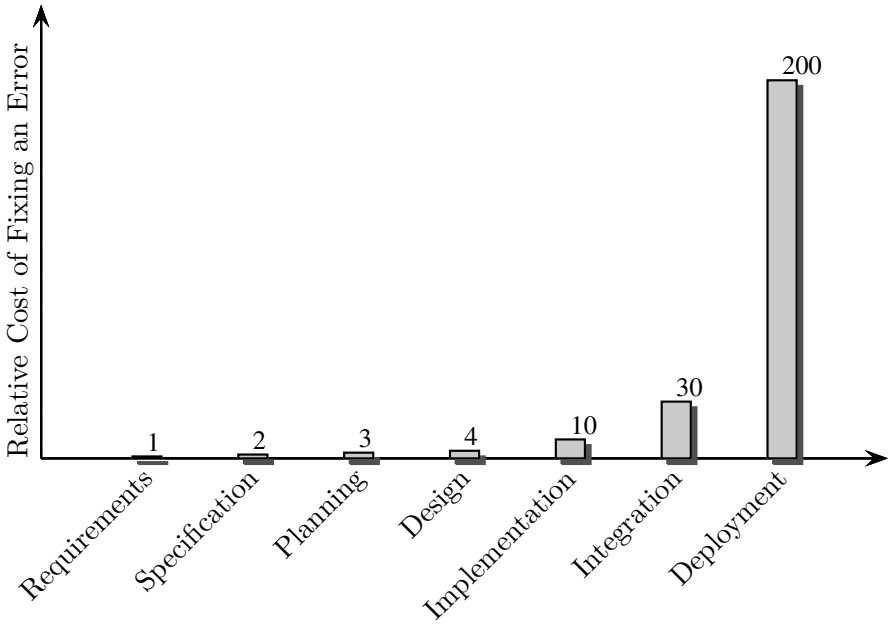


FIGURE 1.2: Relative Cost of Fixing an Error [15]

While the stakeholder needs/requirements are better understood later in the development process, errors are more costly to repair — an exponential growth as Figure 1.2 shows. If, for example, an error is detected in the planning or design phase and not at the implementation phase, the costs can be reduced by up to 66%. Even more drastic is the situation if the error is detected in the deployment phase, where the costs are more than 50 times higher than in the planning or design phase. Unfortunately, the tools used for the implementation rarely provide an adequate support in detecting design errors, especially when it comes to requirements others than the ones from the used design language — the stakeholder requirements. Validating the design and implementation specific needs is almost never integrated in the development tools, especially in tools to implement the software product. *Integrated Development Environments* (IDE) such as Eclipse [38] or NetBeans [2] are well appointed to validate syntactically and best practice constraints for common used programming languages such as Java or C/C++ and can also be customized by the implementer. However, as these tools are used relatively late in the development process, they provide few support in detecting errors regarding stakeholder requirements — inconsistencies between the specifications and their realizations. Hence, there is need for appropriate tool support in validating customized constraints early in the development process — the planning and design phase. This thesis focuses primarily on these phases but it will be shown also that the proposed approach can be applied on various domains and phases of the software development process.

Over the last decade much research has been going on in this area that addresses the consistency checking during model based software development. It started with batch

based approaches [46] that evaluate the consistency of software models periodically. This is a time consuming process and may lead to overwhelming amount of information for designers, if the validation is done in long intervals. However, these approaches are still state-of-the-art, because of they are easy to implement and are able to define hierarchical constraints. However, they are rarely used due to their inconvenient character. In the last years more and more approaches are developed that have incremental characteristics [39, 54, 57, 73, 95], i. e., inconsistencies are detected directly from a change made in the design model. Unfortunately, these approaches are often specialized to single modeling and design languages. Furthermore, the performance depends on the complexity of the constraints, which is expected but given that these approaches typically validate all constraints, this is expensive, i. e., they perform well on small constraints but not on complex, hierarchically structured constraints as shown in [92].

Detecting inconsistencies is but one facet of managing the consistency of software systems. When inconsistencies are detected they must be resolved or at least, in the spirit of tolerating inconsistencies [7], the consequences of an inconsistency on the software product should be visualized. To resolve an inconsistency, information must be known on how the inconsistency can be resolved. This is a much more complicated process than detecting inconsistencies, because it must be known what parts need changing and how these parts must be changed to resolve the inconsistency. To determine both, what and how, is important and difficult: Important because it must support the modeler in deciding where to resolve an inconsistency and difficult because in certain situations no concrete value for how to resolve an inconsistency can be determined.

In the last years the scientific community on consistency management is concentrating on the resolution of detected inconsistencies. A distinction is made between approaches that detect single inconsistencies and combinations thereof [40, 41, 74, 95, 110] or approaches that resolve all inconsistencies at once [71]. Both have their advantages and drawbacks that depend on the situation where they are used. Batch based solutions are able to generate a complete consistent model, if one exists. These approaches are based on Prolog or SAT solvers [56, 65, 67] and the generation of complete models take some time. Unfortunately, these approaches generate complete consistent models only — this requires a set of constraints that do not contradict each other, a prerequisite that might not necessarily hold for arbitrary definable constraints as mentioned before.

On the other hand, approaches that consider the inconsistencies in isolation provide a restricted view of the solutions for an inconsistency where the designer has to decide what solution must be applied. A challenge that these approaches struggle with is to detect the effects on other constraints, i. e., the side effects that the changes have on other constraints [90]. Without these side effects the resolution of an inconsistency can be very hard. Fortunately, some approaches are able to detect the side effects of the inconsistency resolutions (e. g., [41]). The inconsistencies are determined by executing a proposed resolution on the model an track the effects. This is time consuming as there might exist many alternatives that must be checked and each alternative acts like a normal model change.

The time it takes to detect inconsistencies, generate resolutions and determine the overall effects of the resolution is very important for incremental approaches. Miller [72] analyzed the time that is acceptable for the user to wait during interactive work. The user4 expects a response of a few 100ms after making a modification in the model. An occasional multisecond response time is likely to be acceptable too. Even though, if an approach is used that validates a modification in a few milliseconds, the testing of an inconsistency resolution easily exceeds this timings [92]. For example, if the validation of a constraint takes about 10ms, 10 constraints are affected that cause 10 inconsistencies with 10 potential resolutions. The total time it takes to provide all the information, the inconsistency, the resolutions and their side effects, will take 10s. This time is far away from the upper limit of acceptance and earlier publications ([39, 41]) show that these assumptions are realistic and for larger models this number will increase. The number of constraints, inconsistencies, resolutions, and side effects cannot be reduced. Therefore, only a reduction of the re-validation time can improve this, because a re-validation is needed to determine the side effects of an inconsistency resolution.

1.2 Motivation

The multistage process of developing a software product comes with its own meta models for each phase and their predefined constraints as well as people that participate in these phases [19]. Nearly all phases of the software development process define artifacts which are often captured in model form and thus have to deal with errors and inconsistencies that arise regarding predefined constraints (constraints derived from the used meta model and not from the user requirements) but solutions differ in the expressiveness and usefulness. Especially in the early phases of the development process the support for detecting and managing inconsistencies is rather low which leads to increasing costs of a software product as was discussed earlier.

This thesis is in line with capabilities in most modern IDEs (e. g., Eclipse [38], NetBeans [2]) that provide instant feedback on errors (i. e., currently in form of syntax errors) as well as suggestions for completing/correcting errors (i. e., currently in form of code completion). However, these tools are used in a late stage where errors may already have cause the most harm (Figure 1.2) and the error detection and completion is limited to simple, often syntactic language constructs. The need of convenient support to validate if a software product conforms the requirements of the stakeholders is given in the earlier phases. As mentioned above, this would help to reduce the overall development and maintenance costs of a software product and will also increases the costumer satisfaction.

Some of the tools used provide excellent support in detecting inconsistencies and solutions to resolve them, especially during the implementation phase. Integrated development environments (IDE), for example, are such tools where the software developer gets informed about syntax and design errors that violated the rules or design guidelines of a specific programming language (inconsistencies between the language specification and its usage).

1.3 Vision

Our vision is a conceptual framework that can be used and adapted to be used in nearly each phase of the software development process where arbitrary constraints can be defined. The models can be syntactically correct but they may be inconsistent regarding the stakeholder requirements which may lead to errors in the implementation. In Figure 1.3 a typical workflow of developing a model with our envisioned framework is shown. The model in the center provides elements and their characterizing properties. For the proposed approach to be applicable, the tools that will be used must provide a mechanism to access the elements and their properties. The starting point of the circle is the designer. The dark arrows are the information flow from and to the designer and the light arrows show the automated information flow in the envisioned framework.

The designer adds and deletes elements or to modify properties of an element to develop a model. The vision is an automated inconsistency detection and resolution mechanism, based on user definable constraints. The source of the constraints can be a) domain specific constraints (e. g., from the modeling language) and b) from project requirements (e. g., stakeholders or environment) that can be expressed on the meta model level.

To detect inconsistencies, concrete elements from the model must be determined on which the constraints will be validated. If the validation detects an inconsistency in the model, all the elements that are involved in the inconsistencies are determined. Based on a white box analyzes of the constraint validations, repairs are generated. Furthermore, the effects (side effects) of the calculated repairs on other constraint validations must be calculated (e. g., if new inconsistencies might exists or other inconsistencies can be resolved too). The repairs should now guide the designer in resolving the inconsistencies in the model and the designer has to decide which repair or part of a repair must be applied on the model and when the changes will be applied.

As can be seen in Figure 1.3, changes in the model are made by the designer only, but the framework supports the designer with the prepared statements of how to change the model, to resolve an inconsistency and fulfill the set of defined constraints. This automated workflow is triggered each time when a change is made in the model, i. e., the guidance is triggered after a change is made by the designer.

The vision of the proposed system should help to improve the quality of a software product while reducing the time and costs of developing and maintaining it. In the following sections the research questions, the final goal and the steps to reach this vision are explained as well as the expected difficulties to deal with.

1.4 Research Questions

RQ 1: How can models and constraints be generalized to become applicable for managing the consistency in a broad applications scope?

RQ 2: Is it possible to generate appropriate and directly executable repair alternatives for inconsistencies without the loss of the generalizable applicability?

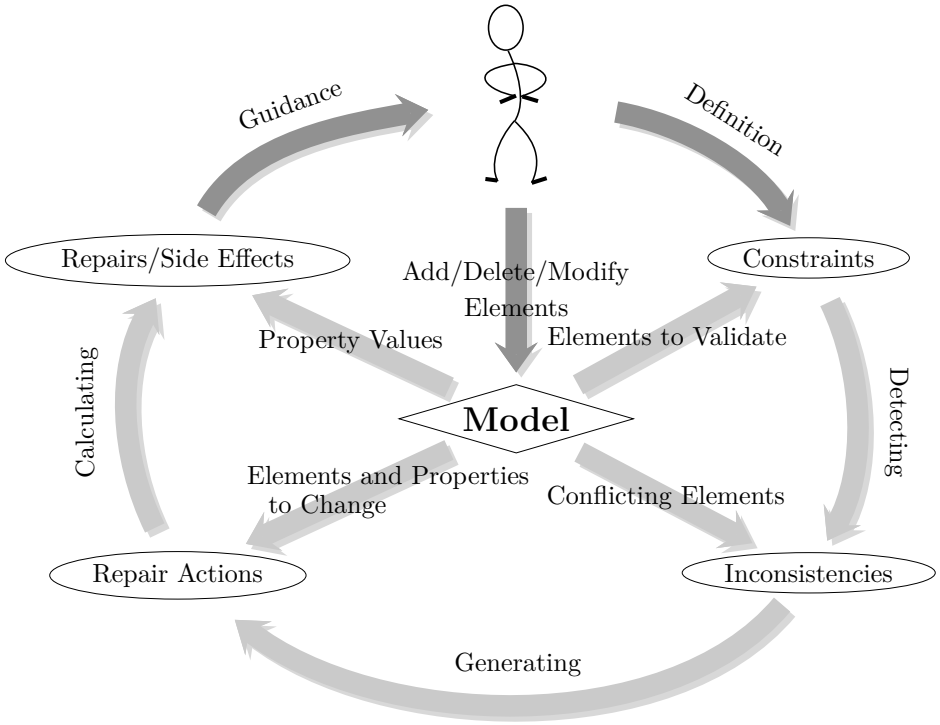


FIGURE 1.3: Model Development Workflow

RQ 3: Can the proposed solution be implemented and is it commonly usable in terms of performance (response time, memory usage) to enable interactive work?

The questions are reflected in the following section. In this section we show a plan of intermediate goals to reach the proposed final goal of this thesis and to answer these question. It also reflects the development history of this thesis.

1.5 Final Goal

The final goal of this work is an consistency management framework that is able to automatically

1. process user definable constraints on a model of elements and their characterizing properties,
2. detect inconsistencies based on the set of constraints,
3. generate solutions to resolve the detected inconsistencies,
4. prepare the solutions to be executable by a designer, and

5. visualize the inconsistencies and their solutions in a software tool that is used in the software product development as well as the evaluation of the approach.

The proposed solution should be generalizable and thus be applicable on any domain language that consist of elements and properties that characterize them (e. g., UML [81], Entity Relationship Models [25], ...), and constraint language that must be convertible to first order predicate logic [8] (e. g., OCL [80], Alloy [56], ...).

To evaluate the proposed approach and to answer the research questions, a prototype is implemented for the UML and OCL in the IBM Rational Software Architect^{TM}(RSA). The implementation tracks the changes made by the user and calculates new inconsistencies and their solutions immediately when they occur and present them to the user in the model and in separated views. This implementation should make the modeling of software products as comfortable as the software developers are used to from IDEs.

The first part of the evaluation covers the answering of research question 1, if the proposed approach is generic applicable on a broad scope of applications. This evaluation is done informally based on ongoing projects that use this technology or at least parts of it. The projects use meta models and constraint languages different from UML and OCL.

In the second part of the evaluation research question 2 will be answered. It will be shown that the proposed approach is working correct, i. e., if all inconsistencies are detected correctly (no inconsistency will be overseen or no inconsistency will be detected that is no inconsistency) and if the generated repairs are correct (they are able to resolve the inconsistency). The evaluation is done using our prototype implementation and comparing our reasoner to the built-in OCL environment of the Modeling Development Tools (MDT) for Eclipse. This will be empirical evaluated on a set of small (approx. 100 model elements) to medium sized (approx. 67.000 model element) UML models and a set of 20 UML well formedness rules (for the UML meta model) expressed in OCL. UML well formedness rules are used because they can be applied on all UML models used for the evaluation. Furthermore, a semi formal discussion about the reasoning and repair generation algorithm to guarantee that the proposed results are correct.

The third part of the evaluation will answer research question 3 and will show that this approach is scalable regarding the model size, i. e., the response time is short enough to not disturb the user in his interactive work [72] and the memory consumption allows the work with larger models. As the detection of inconsistencies and the generation of repairs to resolve the inconsistencies is done immediately after a user changed the model, the calculations and visualizations must be done in less than 0.1s [72]. As this highly depends on the used tool and underlying hardware, this will be a snapshot of the actual state-of-the-art. This will be done by an empirical evaluation on our set of UML models and UML well formedness rules.

At the end of the evaluations, the limitations encountered during the development process of this thesis and the evaluations are discussed.

1.6 Intermediate Goals

To achieve the final goal five intermediate goals must be reached. The first four intermediate goals have to be reached sequentially, i. e., a preceding goal is the fundamental of the succeeding one. The fifth is realized in parallel to the other four goals to achieve the answer of the research questions, to ensure the practical realization of the proposed approach. In an early stage of this thesis in [88] a first overview about this work was given and this initial overview has been refined for this thesis.

1.6.1 Building an Overview of Existing Work

Before starting developing a completely new approach it must be evaluated what is the actual stat-of-the-art in this domain and where are the limitations of the existing work. Based on the existing work an appropriate point to start must be found, i. e., it must be evaluated if some or a combination of the existing work is able to act as a fundamental for this vision and what partial problems are resolved and which are unresolved. This exhaustive study of literature can be found in the illustration and background in Chapter 2 as well as in the related work in Chapter 3.

1.6.2 Approach to Detect Inconsistencies

The second goal is to find an appropriate approach to detect inconsistencies and that is able to define user defined constraints. The approach must be incremental (the detection of inconsistencies must be done immediately when the occur) and scale on large models. The used constraint language must be commonly used and well documented. The writing of constraints must be intuitive as the system design process itself. The realization of this goal can be found in Section 5.2 and Section 5.3, where an existing incremental approach to detect inconsistencies [39] is generalized to be applicable to different modeling languages [108] and able to process custom user defined constraints [51].

1.6.3 Generating Solutions to Repair the Detected Inconsistencies

The third step is to generate solutions to repair the detected inconsistencies. In this phase repair actions are generated that repair single inconsistencies in the model. The repair actions consist of the action that must be taken (add, remove, or modify) and the model element as well as the property that is affected by this action [90]. How the repair actions are generated can be found in Section 5.4 and Section 5.5.

1.6.4 Preparation of the Repair Actions to Resolve the Inconsistencies

For the repair actions to be directly executable in the model, concrete values for the model element property to change must be known. If both, the model element property and the value that must be assigned to it, is know, the repair action becomes concrete.

Furthermore, the repair actions for the single inconsistencies must be combined to generate a complete repair for an inconsistent model. In this goal a fundamental point must be considered, that the repair actions in the repair must not contradict each other. How this goal is realized can be found in Section 5.5.

1.6.5 Realization in a Software Tool

To show the feasibility of the proposed approach and to evaluate it, it is necessary to realize the approach in a software tool that is used in the software development process [89]. While the first four goals are realized sequentially and build on one another, the implementation of the proposed approach is done in parallel to the first four goals. Therefore, the approach is implemented in parallel because of to verify the feasibility of the approach. The realization of the tool implementation as plug-in for the IBM Rational Software Architect (RSA) is shown in Chapter 6. As mentioned, the tool implementation is used to evaluate the usability based on quantitative [92] and qualitative criteria [90, 91], introduced in Section 1.5, is shown in Chapter 7.

1.7 Structure of the Thesis

In Chapter 2 the basic terms used in the thesis are defined as well as an example to illustrate the addressed problem. The existing work and work that is related to this work is presented in Chapter 3. Chapter 4 shows the basic principles used in the proposed approach shown in Chapter 5. In Chapter 6 a prototype implementation of the approach is shown that is used for the evaluations shown in Chapter 7. In Chapter 7 a discussion about the limitations of the proposed approach is also given. Chapter 8 concludes the thesis and gives an outlook about actual ongoing work that uses the technology presented in this work.

1.8 Summary

In this chapter an overview about the starting point, motivation and the main vision of this thesis was shown. Furthermore, the main research questions were presented and the plan of how these questions will be answered.

Chapter 2

Illustration, Background and Definitions

> *"Problems are not stop signs, they are guidelines."*
> Robert H. Schuller, American Clergyman, 1926

To be more concrete what this thesis is about the basic terms and the main problems we are tackling are introduced in this chapter based on a running example that guides us through the thesis. The introduced example consists of different views on an UML model and is used to demonstrate how arbitrary constraints can be validated on this model to detect inconsistencies and to resolve them.

2.1 Introductory Example

Most approaches that are comparable to the one presented in this thesis are made for the model based software development [29, 39, 73, 95, 104, 110]. The example we use is an UML [85] model containing a class diagram (Figure 2.1), state machine diagram for one class (Figure 2.2), and a sequence diagram (Figure 2.3). In this example we show three different views on one model. Moreover, it will be shown how UML and OCL can be generalized to a more general and abstract form of representation. The diagrams of the model are free of inconsistencies regarding UML but later, when we introduce constraints (Section 2.3), inconsistencies will be introduced too.

2.1.1 Notation used in this Thesis

Before we introduce a concrete example, the notation used in this thesis is introduced.

The `Typewriter` font is used for elements and properties of a system as well as for code that is used in the text (in separate listings normal roman font with syntax highlighting is used), e. g., the element `Display` has the operation `stop`.

The *Slanted* font is used for element type description, e. g., the element `Display` is of the UML type *Class*.

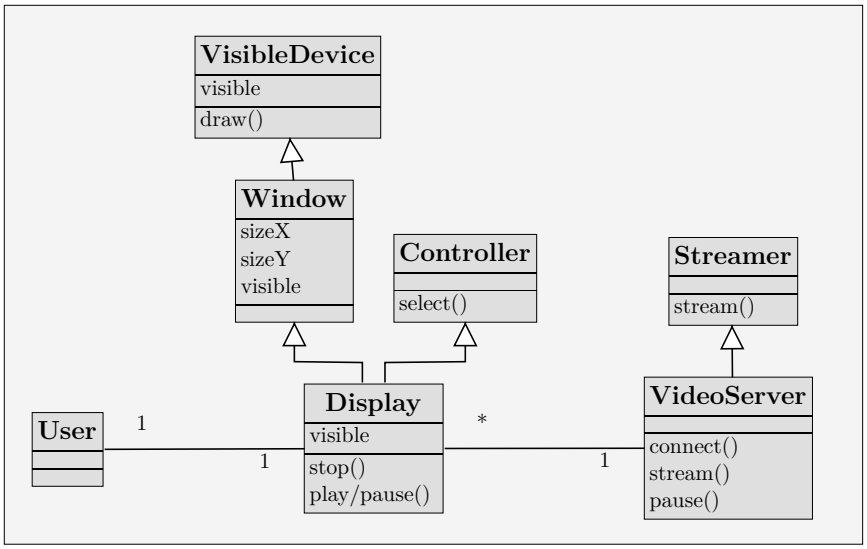


FIGURE 2.1: UML Class Diagram of a Video on Demand System

Values of properties are written in 'single quotes', e. g., the `name` property value of the `Display` is 'Display'.

Roman letters (M, e, p,...) are used for elements that refer to the element. Uppercase letters are for sets and lower case letters are for single value elements.

Greek letters (α, γ, σ,...) are used for elements that refer to constraints.

Tuples are written in angle brackets, e. g., $\langle t, \gamma \rangle$.

Set of values are written in curly brackets, e. g., $M = \{a, b, c\}$.

2.1.2 Structural Diagrams

Structural diagrams describe the architecture of a software model. In Figure 2.1 a typical class diagram is shown that is used as the running example. The basic meta model is the UML [81] from which *Class*es containing *Attribute*s and *Operation*s are used. The example shows an simplified excerpt of a video on demand system. The model consists of a class that uses the system (`User`), a class that allows to control the system and the showing of the video (`Display`, `Window`, `Controller`, and `VisibleDevice`), and finally a *Class* that provides the video data (`VideoServer`, `Streamer`).

The `User` class is an abstraction of the operator of this system. It is representative for any operator, whatever if the operator is human or another system. It does not contain any *Attribute*s or *Operation*s.

The `Display` class contains both, a controller and visualization function. This *Class* contains an *Attribute* `visible` that indicates, if the `Display` is visible to the user. The

`Display` also contains *Operation*s to control a video (`stop` and `play/pause`). Hence, this class extends the `Controller` class and the `Window` class. The `Controller` class provides an *Operation* to `select` something (this class is an abstract specification of a controller that does not exactly specifies what it controls). The `Window` provides *Attributes* for the size of the window (`sizeX` and `sizeY`) as well an *Attribute* that indicates if the window is `visible`. The `VisibleDevice` class is a super class for all classes that are able to represent some content to a user, such as the `Window` and `Display` classes. It also provides an *Attribute* `visible` that indicates if the content is visible. Please note that UML allows multiple inheritance, the redefinition of *Attribute*s and the overwriting of *Operation*s. A possible restriction can come from the used implementation language which is not known. The only restriction that can be made are using constraints (will be discussed in Section 2.3).

The `VideoServer` class provides and streams the video data. It contains the *Operation*s `connect` to connect to the video server, `stream` to stream the video data to a `Display`, and `pause` to pause the video stream. The `stream` operation is overridden from the super class `Streamer` which provides the main streaming capabilities.

A `User` is connected to one (1) `Display` and a `Display` can be controlled and watched by one (1) `User`. A `Display` is connected to one (1) `VideoServer` but a `VideoServer` can provide the data to any number of `Display`s (*).

Each element of the diagram is of a specific type from the UML meta model. For example, the *Class* `Display` is of the UML type *Class*. Furthermore, the elements do have properties that are specified in the meta model. A UML class, for example, has the property `name`, i. e., the name property of the UML class `Display` is 'Display'. Additional, the values of properties can be instances of meta model elements, i. e., the properties have the type of an UML meta model element. For example, an *Attribute* of a *Class* is a property of the UML class and is of the UML type *Attribute* which itself has properties, like a name (e. g., 'visible'). The abstraction to elements and their properties is valid for all elements from the UML meta model.

2.1.3 Behavior Diagrams

Aside the static view of structural diagrams, behavior diagrams are used to describe the dynamic aspects of a software system. To illustrate that the diagrams are interrelated in Figure 2.2 a state machine diagram for the *Class* `VideoServer` in `active` state is shown. The initial state after a display connects to a streamer is `stopped`. The state of this class changes form `stopped` to `playing` with the *Transition* `stream` and changes back with the *Transition* `wait`.

Additional to a state machine diagram, Figure 2.3 shows a sequence diagram. This sequence diagram shows how a video will be selected and started. First, the `User` has to `select` a video on the `VideoServer`. When the `User` selects a video, the `Display` connects to the `VideoServer` and the `VideoServer` changes into the state `stopped`. After that, the `User` can start the video by sending the `play` message to the `Display`. The `Display` itself sends the `stream` message to the `VideoServer`. The `VideoServer` changes its state from `stopped` to `playing` and starts streaming (`stream`) the video data

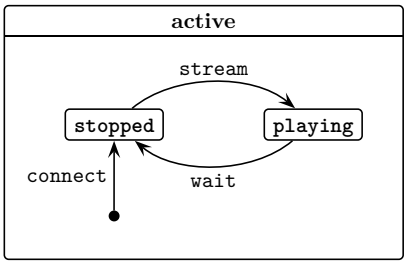


FIGURE 2.2: State Machine Diagram for the *Class* 'VideoServer'

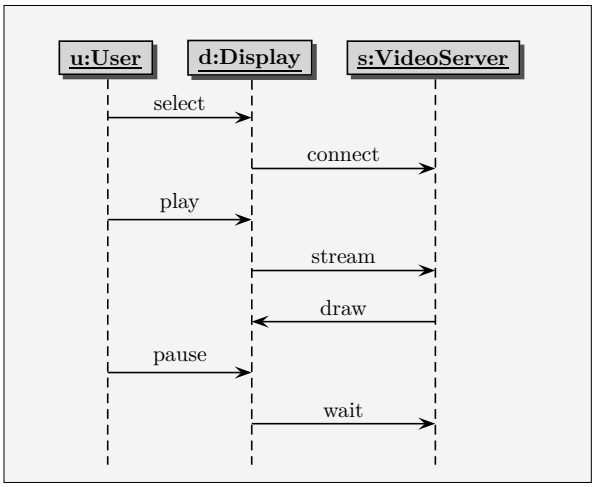


FIGURE 2.3: UML Sequence Diagram for Selecting and Starting a Video

to the `Display`. The video stream can be paused by sending a `pause` message to the `Display` that themselves sends a `pause` message to the `VideoServer` which changes its state from `playing` to `stopped`.

The shown diagrams are a small excerpt of the capabilities of the UML but these diagrams are commonly used and are able to represent static and dynamic aspects of a software system. Later we will see how these and all the other diagram types can be abstracted to only a few characteristic elements and properties.

2.2 Domain Language Abstraction

The Meta Object Facility [79] (MOF) introduced by the Object Management Group (OMG) describes the different abstraction levels for the model-driven engineering. In Figure 2.4 the different levels are shown, beginning at the most abstract level M3 and ending at the concrete software product (the source code) at level M0. In our example we used UML as a modeling language. The UML model is located at level M1 and the

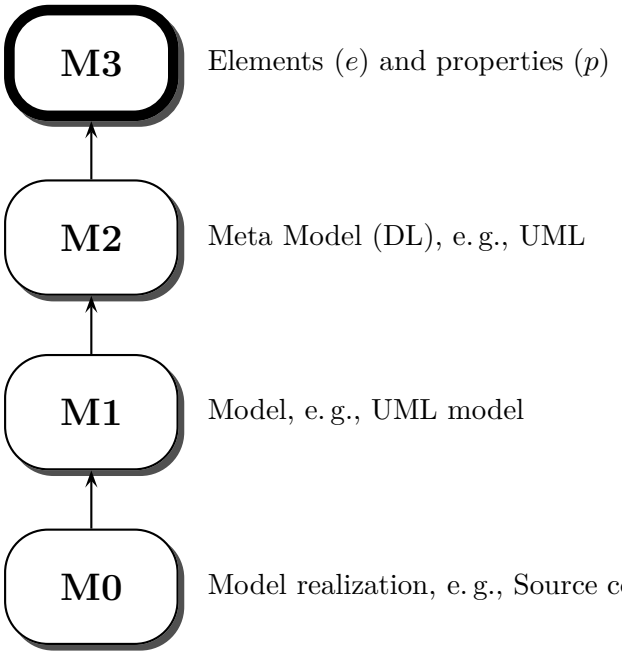


FIGURE 2.4: Meta Object Facility Levels [79]

UML meta model at level M2. UML in its standard has UML as its meta model. The level M3 is the level where the used design language can be abstracted to become general applicable, i. e., to be used for other model languages than the UML.

UML is a general purpose language that provides a lot of semantics in each of its diagram types. However, for the definition of a meta model for our consistency management these semantics can be ignored as they can be expressed as constraints described in Section 2.3. In the following we define the main artifacts of how a meta model must be build up.

Definition 1. *A **Model** (M) consists of **elements** (e), where the elements can have **properties** (p). The property types can be any simple types like Boolean, Integer, Float, String,... or elements themselves. The elements of a model are instances ($\pitchfork$) of a specific **type** (t) defined by the **design language** (DL).*

$$\begin{aligned} M &:= \bigcup e \\ t &\in DL \\ e &\pitchfork t \\ e.p &\rightarrow M \cup \text{‘any value‘} \pitchfork t \end{aligned}$$

2.3 Constraints

In the last sections it was described of what parts a model consists of but it was ignored how semantic can be expressed in such models. Constraints can be used to define semantics in a model and its diagrams as well as to restrict model constructs (like the redefinition of *Attribute*s in a class diagram). The constraints can come from the used model language (e. g., UML), the used implementation languages (e. g., Java does not allow multiple inheritance whereas it is allowed in C) or the stakeholder requirements (the basic functionality of the software, e. g., how a video stream can be controlled by the customer). While model language dependent constraints are well structured and are easy to formalize, the stakeholder requirements are often expressed in natural language. These requirements must be formalized in some way and in the field of requirements engineering [19] approaches exists that are able to do so. However, this formalization comes with the problem that this process is error prone and that there might exist contradictions in the formalized set of constraints which must be detected too. Another important issue that we have to deal with in this thesis.

In our example we use the Object Constraint Language (OCL) [80], a well formalized constraint language that is based on first-order logic [8]. For illustration purposes we use constraints describing UML well-formedness (i. e., UML allows the violation but for better understanding of a model these constraints should be met) because they are applicable on all UML models. Constraint 1 shows a constraint defined in OCL that validates if a *Message* in a sequence diagram is defined as an *Operation* in the receiving *Class* from the class diagram. This represents a typical constraint provided by the UML meta model.

Constraint 1 A *Message* must be defined as an *Operation* in the receiving *Class*

```
1 context Message:
2 self.receiveEvent.covered-> forAll(l:Lifeline|
3     l.represents.type.ownedOperation->exists(o:Operation|
4         self.name=o.name))
```

Line 1 defines the context of the constraint. The context is the UML type *Message*, i. e., this constraint must hold for all elements in the model of this type — the context element is an element of the used meta model, e. g., the UML. Line 2 to Line 4 specify the condition that must hold for the context element. The details of how a condition is build up is described in Section 2.4.

Definition 2. *A **constraint** (C) specifies a property and condition that must hold in the model. It consists of the **context** (t) and a **condition** (γ). The condition validates to a Boolean ($\mathcal{B}$) value where 'false' indicates a violation (inconsistency).*

$$\begin{aligned} C &:= \langle t, \gamma \rangle \\ \gamma &: e \mapsto \mathcal{B} | e \Uparrow t \\ \gamma(e) = false &\Rightarrow inconsistent \end{aligned}$$

As can be seen, this constraint applies to *Message*s from a sequence diagram. Therefore, this constraint must be validated for each *Message* in a sequence diagram shown in Figure 2.3. The constraint is violated in Figure 2.3 by the *Message*s `play` (γ_1(`play`) $\mapsto$ *inconsistent*), `pause` (γ_1(`pause`) $\mapsto$ *inconsistent*) and `wait` (γ_1(`wait`) $\mapsto$ *inconsistent*), because no corresponding *Operation*s exist in the *Class* `Display` and `VideoServer`, nor in its super classes `Controller`, `Window`, `VisibleDevice` or `Streamer`.

The last constraint shows that a constraint validation can cover more than one type of diagram in the model. The next constraint applies to class diagrams only. Constraint 2 validates if an *Attribute* of a *Class* is not defined in a super class. The context of this constraint is the UML type *Class*.

Constraint 2 An *Attribute* must not be defined in a parent *Class*

```
5  context Class:
6  let attrNames:Bag(String)=self.attribute->collect(p:Property|p.name) in
7  self.allParents()->forAll(c:Classifier|
8      c.attribute->forall(p:Property|
9          not attrNames->exists(s:String|
10             s=p.name)))
```

This constraint is validated for each *Class* in the class diagram. The one validation on the *Class* `Display` and `Window` will fail because the *Attribute* `visible` is defined in at least one of their super classes.

Constraint 3 State *Transition* must be defined as an *Operation* in the Owner's *Class*

```
11 context Transition:
12     self.owner.stateMachine<>null implies
13         let classifier:BehavioredClassifier=self.owner.stateMachine.
               context in
14             classifier.oclIsTypeOf(Class) implies
15                 classifier.ownedOperation->exists(o:Operation|o.name=self
                       .name)
```

Constraints can have interrelation (i. e., their validation overlap in some parts of the model). Therefore we introduce Constraint 3 that validates if for a given *Transition* in a state machine diagram an *Operation* exists in the owner's *Class*. This constraint causes an inconsistency also, because no *Operation* `wait` exists in the *Class* `Streamer`. As can be seen, this constraint accesses the same *Operation*s as Constraint 1. Thus, there can be effects on the validations of the constraints if one of the *Operation*s will be changed. This is important when an inconsistency must be resolved because resolving one inconsistency will cause new inconsistencies or resolve more than one due to these overlaps.

2.4 Constraint Structure

First order logic is commonly used to formally express constraints for many different applications. The constraints are expressed in first order logic. Such a constraint itself is an expression in first order logic that contains then again of zero to any number of sub expressions, i. e., a constraint is a recursive build up construct of expressions, starting at a root expression without a parent and ends at leaf expressions without sub expressions.

Definition 3. *A constraint condition consist of a set of hierarchical ordered Boolean and non-Boolean* ***expressions*** *where the Boolean expressions (ϵ_x) consists of an* ***operation*** *(o), a set of 0 to $*$* ***arguments*** *(α), a* ***validation result*** *(σ) and an* ***expected result*** *(ς). The arguments of a Boolean expression are expressions itself and they are tree based ordered, i. e., each expression has exactly one parent (ρ) and is in a set of arguments of an other expression except the root expression (ϵ_0). The root expression has 'true' as expected result ('true' is equivalent to consistent). A property call expression (ϵ_p) accesses a model element property. Property call expressions do not have arguments and an expected result. Constant expressions (ϵ_c) consist of a parent and validation result only.*

$$
\begin{aligned}
\epsilon_{x>0 \wedge x \neq p} &:= \langle o, \alpha, \rho, \sigma \in \mathcal{B}, \varsigma \in \mathcal{B} \rangle \\
\epsilon_0 &:= \langle o, \alpha, \sigma \in \mathcal{B} \rangle \\
\epsilon_p &:= \langle p, e, \rho, \sigma \in M \cup \text{`any value'} \rangle \\
\epsilon_c &:= \langle \rho, \sigma \in \text{`any value'} \rangle \\
\forall i > 0 \exists j \geq 0 \wedge j \neq p &: \quad \epsilon_i \in \epsilon_j.\alpha \wedge \epsilon_i.\rho = \epsilon_j
\end{aligned}
$$

2.4.1 Basic Constraint Elements

Table 2.1 shows the common used operations of OCL. The first column shows the formal notation for the first order logic operations and in informal description of operations that are custom to OCL, whereas the second column the OCL notation shows. Column three and four specify the types of the arguments and the last column the result type of the operation types. The operation with its arguments build an expression.

The first seven rows of Table 2.1 show the common operations of first order logic with their arguments. The negation ($\neg$) inverts the result of its argument, i. e., if the arg_1 validates to 'true' the expression validates to 'false' and vice versa. The conjunction ($\wedge$) validates to 'true' only if both arguments validate to 'true' and the disjunction ($\vee$) validates to 'true' if at least one argument validates to 'true'. An implication ($\Rightarrow$) validates to 'true' if the first argument validates to 'false' or both arguments validate to 'true', otherwise the implication validates to 'false'. An equality relation ($=$) compares two arguments and validates to 'true' only if both sides are equal. The quantifiers first argument is a collection of elements of any type, i. e., the source of the expression. The second argument is a condition that is validated using one or a combination of elements from the source. An universal quantifier ($\forall$) validates to 'true' only if the second argument holds (i. e., the second argument validates to 'true') for all elements

operation type	OCL	argument types		result type
		arg_1	arg_2	
$\neg arg_1$	`not` arg_1	Boolean	-	Boolean
$arg_1 \wedge arg_2$	arg_1 `and` arg_2	Boolean	Boolean	Boolean
$arg_1 \vee arg_2$	arg_1 `or` arg_2	Boolean	Boolean	Boolean
$arg_1 \Rightarrow arg_2$	arg_1 `implies` arg_2	Boolean	Boolean	Boolean
$arg_1 = arg_2$	arg_1 `=` arg_2	any	any	Boolean
$\forall a \in arg_1 : arg_2$	arg_1`->forAll(`$a : arg_2$`)`	Collection	Boolean	Boolean
$\exists a \in arg_1 : arg_2$	arg_1`->exists(`$a : arg_2$`)`	Collection	Boolean	Boolean
constant		-	-	any
property call	arg_1`.`arg_2	element	property	any
collect	arg_1`->collect(`$a : arg_2$`)`	collection	property	collection
select	arg_1`->select(`$a : arg_2$`)`	collection	Boolean	collection
variable	`let` arg_1 `in` arg_2	any	any	any

TABLE 2.1: Commonly used Operation provided by OCL

from the source whereas the existential quantifier ($\exists$) validates to 'true' if at least one validation of the second argument holds.

The *constant* operation allows the use of constants in constraints. A constant is, for example, the value 'true' or 'false'. A constant can be of any type, Boolean, Float, Integer, String,.... The *property call* expression allows the access to properties of elements. To access the `name` property of an *Operation* (e. g., Constraint 1, Line 4) the first argument is the element (*Operation* `o`) and the second the property (`name`) that is accessed. The result of the expression's validation can be of any type.

The *collect* and *select* expressions are taken over from the OCL. The *collect* expression is a property call (arg_2) on a set of elements (the source arg_1), i. e., it collects the values of a property from a set of elements and the result is a set of the same cardinality as the set of elements of the source: $|Source| = |Result|$. In contrast, the *select* expression is a filter expression where the result contains only these elements from the source (arg_1) where the condition specified in arg_2 holds. Hence, the result set is a subset of the source set: $Source \supseteq Result$.

The last expression, the *variable*, is also inspired by the OCL. In OCL the `let` expression has as its first argument (arg_1) the declaration of a variable that can be used in the expression defined in the `in` clause (arg_2).

2.4.2 Concrete Constraint Validation

Constraint 1 starts with a property call expression (Line 2). The first called property is `receiveEvent` on the context element instance (a *Message*). The context element is represented in the variable `self`. For illustration we use a concrete validation of this constraint on the *Message* `play`, hence, the variable `self` is instantiated with the *Message*

`play`. The `receiveEvent` property is an UML element of type *MessageOccurenceSpecification* (it is an unnamed element). On the returned *MessageOccurenceSpecification* the property `covered` is called which returns the *Lifeline*s the *Message* points to. In our case only one *Lifeline* is in the set of returned *Lifeline*s — the *Lifeline* `d`.

Next, over the returned *Lifeline*s will be iterated. This is done with an universal quantifier (`...->forAll(l:Lifeline|...`). The source (the element to iterate over) is the just now mentioned property call. In the universal quantifier the condition is defined that must hold for the source elements (Line 3). The condition itself starts with a property call expression. First, the property `represents` is called on a *Lifeline* of the source set, represented as variable `l`. The property `type` called on the returned element from the `represents` property call on *Lifeline* `l` returns the corresponding class of the *Lifeline* — the *Class* `Display`. On this *Class* the property `ownedOperation` is called which returns a set of all containing *Operation*s of this *Class*.

These *Operation*s are the source for the existential quantifier. An existential quantifier is equally build up like the universal quantifier. The condition of the existential quantifier is an equals relation (Line 4). This relation compares the name of the context element (`self.name`) with the name of an *Operation* (`o.name`) from the source of the existential quantifier. As the *Class* `Display` does not contain an *Operation* named 'play', the existential quantifier validates to 'false', therefore the validation of the universal quantifier validates to 'false' also, and hence the validation of this constraint fails → an inconsistency has been detected.

Constraint 2 shows some special expression of OCL. In Line 6 of Constraint 2 a variable is declared that holds all the names of the context element attributes. The `collect` operation from OCL is applied on a set of elements where the specified property is called on each element and the result of this *Operation* is a set of values from the property calls. In this constraint the names of the class' attributes are collected. The `allParents` (Line 7) property is a recursive call of the UML properties `generalization` and `general` of the context class and all its super classes. The result of this property call is a set containing all super classes. Line 8 to Line 10 contain constructs similar to the one used in the first constraint, only the properties may differ.

2.5 Incremental Consistency Checking

Up to now we are able to detect inconsistencies, but to detect them each constraint must be validated, i. e., if something will be modified in the model, all constraints must be re-validated. Validating the consistency in this way is a time consuming process, especially applied to large models. However, this is not practicable for interactive usage, hence we have to reduce the effort for re-validation. Egyed presented in his work [39] an incremental approach for consistency checking — the ModelAnalyzer approach. This approach collects, while validating the constraint, all the accessed model element properties. A constraint will be re-validated only if the model change affects one of the collected model element properties. The collected model elements are the *scope* of a constraint validation.

Definition 4. *The **Scope** (S) of a constraint validation is the set of all element properties (e.p) of the constraint validation. One element of the scope is called **Scope Element**. These elements are directly derived from the constraint validation.*

$$\forall e.p \in S | e.p \text{ is accessed during the constraint validation}$$

Definition 4 does not imply that the scope is minimal, i. e., it is not guaranteed that all element properties have an effect on the validation result of the constraint. However, it guarantees that all element properties are in the scope that do have an effect on the validation result, i. e., it is complete. This property is valid even if short circuit validation is used. Later we will see that this is not necessarily true for the detection of all scope elements that cause an inconsistency.

Theorem 1. *A constraint that has been validated on a context element $e_c | C := \langle e_c, \gamma \rangle$ must be re-validated only, if at least one element property that has been accessed during the initial validation $e.p \in S(\gamma(e_c))$ has been changed.*

$$\gamma(e_c) \neq \gamma(e_c)' \quad \Rightarrow \quad \exists e.p \in S(\gamma(e_c)) | (e.p = r \rightarrow e.p = r' \wedge r \neq r')$$

Proof. To prove Theorem 1 we use proof by contradiction. We assume that their exists an element property that is not in the scope of the constraint validation and a change of this element property affects the constraint validation.

$$\exists e.p \notin S(\gamma(e_c)) | (e.p = r \rightarrow e.p = r' \wedge r \neq r' \wedge \gamma(e_c) \neq \gamma(e_c)')$$

If such an element property exists, it must have had an influence on the initial validation of the constraint and as such due to Definition 4 it must be included in the scope of the constraint validation which is in contradiction to our assumption, hence Theorem 1 must be correct. □

Constraint 1, for example, accesses the properties `receiveEvent` and `name` from the *Message* to which the constraint is applied. Additional the property `covered` of the element returned by the `receiveEvent` (an element of UML type *MessageOccurence-Specification*) property is called, the `represents` property of the *Lifeline* returned by the `covered` property, the `type` of the *Lifeline* (UML type *Class*), the `ownedOperation` property of the *Lifeline*s' type, and finally the `name` property of the *Class*' operations. The quantifier iterate over all elements from the source (the value of a model element property), e. g., *Lifeline*('d').`represents` is one element of the set from *MessageOccurenceSpecification*().`covered`.

When we consider the validation of this constraint on the *Message* `play`, the scope of this validation is:

1. *Message*('play').`receiveEvent` → *MessageOccurenceSpecification* with no name
2. *MessageOccurenceSpecification*().`covered` → {'*d*'} set of *Lifeline*s
3. *Lifeline*('d').`represents` → *Property*('d')

4. *Property*(‘d‘).`type` → *Class*(‘Display‘)
5. *Class*(‘Display‘).`ownedOperation` → {‘*play/pause*‘, ‘*stop*‘, ‘*select*‘, ‘*draw*‘} set of *Operation*s
6. *Message*(‘play‘).`name` → *String*(‘play‘)
7. *Operation*(‘play/pause‘).`name` → *String*(‘play/pause‘)
8. *Operation*(‘stop‘).`name` → *String*(‘stop‘)
9. *Operation*(‘select‘).`name` → *String*(‘select‘)
10. *Operation*(‘draw‘).`name` → *String*(‘draw‘)

In total ten model element properties were accessed during the constraint validation, i. e., only these properties contributed to the validation result of this constraint. The name of the *Message* `play` will be accessed more than once, i. e., each time it is compared to one of the *Operation*s name. The sequence of how the *Operation*s were accessed is non deterministic, because an unordered set of *Operation*s is returned by the `ownedOperation` property call. This constraint is re-validated only, if one of these properties is changed. If, for example, the name of the *Message* ‘play‘ changes, the complete constraint will be re-validated. But if, for example, the *Message* ‘select‘ changes, this particular constraints needs no re-validation.

To achieve a better performance of the re-validation, a challenge of this work is to reduce the scope of a constraint validation to those elements that really affect the validation result. If, for instance, in our example the property `name` of *Message* `play` will be changed to ‘play/pause‘, existing approaches on consistency management would re-validate the complete constraint.

The expressions represent the atomic structure (in the sense of validation) of a constraint. State-of-the-art approaches in incremental consistency management [39, 73] have to re-validate the entire constraint if a model modification affects it. A key challenge in this work is to achieve an optimized re-validation of affected constraints. Optimized for performance (low re-validation time) and low memory usage. Often performance gain is achieved with an increased memory usage (e. g., RETE algoritm [50]) and in this thesis the focus is on the re-validation of the atomic structures, the expressions, and simultaneously keeping the memory usage as low as possible. In [112] Chang et al. presented an promising approach for an optimized re-validation algorithm for first order logic for the consistency management in pervasive systems and in this work a similar approach will be used.

2.6 Understanding an Inconsistency — What Caused an Inconsistency

Up to now the work is very similar to the work by Egyed [39] which is the basis for this thesis. The scope built by this approach is not minimal in the sense that it contains

scope elements that are not immediately contributing to an inconsistency, i. e., the values of the scope elements can be changed to any other value and the validation result of the constraint would not change. However, while some scope elements do not contribute to an inconsistency, some scope element might be missing, if, for example, the validation stops at the first occurrence of an constraint validation (e. g., short circuit validation in C [49], Java, ...).

As short circuit validation can improve the performance of a validation and prevents the validation from errors (e. g., programming language Java: `if (x != null && x.name = 'foo') {...}`), in the context of consistency checking some scope elements can not be detected due to short circuit validation. To illustrate this issue, we take the constraint expressed in Constraint 2. At first we will list all the scope elements accessed during a complete validation of the constraint on the *Class* 'Display'. The scope elements are in the sequence when they are first accessed:

1. *Class*('Display').`attribute` → {'*visible*'} set of *Properties*
2. *Property*('Display.visible').`name` → *String*('visible')
3. *Class*('Display').`generalization` → *Generalization* with no name
4. *Generalization*().`general` → *Classifier*('Window')
5. *Classifier*('Window').`generalization` → *Generalization* with no name
6. *Generalization*().`general` → *Classifier*('VisibleDevice')
7. *Classifier*('VisisbleDevice').`generalization` → {} empty set
8. *Classifier*('Window').`attribute` → {'*sizeX*','*sizeY*','*visible*'} set of *Properties*
9. *Property*('Window.sizeX').`name` → *String*('sizeX')
10. *Property*('Window.sizeY').`name` → *String*('sizeY')
11. *Property*('Window.visible').`name` → *String*('visible')
12. *Classifier*('VisibleDevice').`atrribute` → {'*visible*'} set of *Properties*
13. *Property*('VisibleDevice.visisble').`name` → *String*('visible')

In total twelve scope elements are accessed during a complete validation. Using short circuit validation, i. e., the validation stops when the first violation of the constraint is detected, the validation stops at the scope element *Property*('Window.visible').`name` (Item 11), i. e., the last two scope elements will not be accessed and the second violation (the `visible` property of the `VisibleDevice` class violates the constraint too) will also not be detected. However, some of the scope elements in the list do not have an influence on the inconsistency. Scope element *Classifier*('VisibleDevice').`generalization` (Item 7), *Property*('Window.sizeX').`name` (Item 9) and *Property*('Window.sizeY').`name`

(Item 10) do not violate the constraint, but these scope elements are accessed during the validation, even though short circuit validation is used. Moreover, some of the elements that violate the constraint too, might not be detected with short circuit validation. Hence, the cause of an inconsistency is a subset of the scope plus some elements that might not be detected due to an earlier detection of an element that already violated the validation [91].

Definition 5. *The **Cause** (ζ) of an inconsistency is the part of the constraint validation that immediately caused the inconsistency. The cause consist of a **cause of expressions** ($\zeta_\epsilon(\gamma)$) and of a **cause of scope elements** ($\zeta_{e.p}(\gamma)$).*

$$\zeta_\epsilon(\gamma) := \bigcup \epsilon_x \in \gamma | \epsilon_x.\sigma \neq \epsilon_x.\varsigma$$
$$\zeta_{e.p}(\gamma) := \bigcup \epsilon_p.e.p \in \gamma | \epsilon_p.\rho.\sigma \neq \epsilon_p.\rho.\varsigma$$

The filtering of the elements that immediately caused an inconsistency is an important pre-requisite for generating repairs, because it filters all those scope elements that need no modification to resolve an inconsistency. The ModelAnalyzer approach is able to generate repairs based on the scope of the constraint validation [40, 89] only. However, this approach is conservative in that sense that it considers each scope element as the potential faulty one and as such repairs are generated that would modify a model element property that need no change. After generation, these unnecessary repairs can be only removed through time consuming testing.

2.7 Repairing an Inconsistency

While it is important to allow and tolerate inconsistencies [7] during the software development process, they must be repaired eventually. However, to repair an inconsistency two things are important to know: First, what needs repairing, i. e., the location, and second, how it can be repaired, i. e., how the value of the location must be changed to resolve an inconsistency. Nentwich et al. [74] introduced abstract and concrete repairs, where abstract repairs represent the locations where to repair and concrete repairs are the locations and how to change the value to resolve an inconsistency. Additional to how to repair an inconsistency it should be known what are the effects (side effects) of a repair action on the model besides the resolution of the addressed inconsistency.

2.7.1 Repairs

A pre-requisite for abstract repairs is to detect the element properties (scope elements) that caused the inconsistency which is described in [88] and in the last section. However, how the elements must be changed to resolve an inconsistency is not specified by the cause. Furthermore, to resolve an inconsistency it could be possible that all elements in the cause must be modified, certain combinations of elements, or it could be enough to change one element of the cause. The only thing we know from the cause is, that a change of a certain combination of elements of the cause is able to resolve the inconsistency.

From the cause we get the scope element(s) that have to be changed somehow to resolve an inconsistency, but information how to change them is still missing. If no appropriate value can be determined, we denote a repair action as *abstract*. A concrete value transforms an abstract repair action into a *concrete* repair action. A special case of a concrete repair action is a *conditional concrete* repair action. This type of repair action gives a hint for a concrete value, for example, if it is known that a model element property must not have a particular value, or if the value of a model element property must be larger than a specific value. For an abstract and a conditional concrete repair action the input from the designer is needed. In this thesis, if not defined otherwise, we talk about concrete repair actions.

Definition 6. *A **repair action** (r) is a change in the model that resolves an inconsistency by itself, or in combination with other repair actions. It consists of a **change type** (t_r), a **scope element** (e.p), and a value (v) that must be applied to the scope element. The type of a change can be **add** which adds an element to the scope element, **delete** which deletes en element from the scope element, or **modify** which changes a scope element to a specific value. An **abstract** repair action ($\overline{r}$) is an action where no concrete value can be calculated (?). The value for a **conditional concrete** repair action is attached with a condition that specifies a range (e. g., $v >$ 'x' or $v \neq$ 'x').*

$$
\begin{aligned}
r &:= \langle t_r, e.p, v\rangle | v \neq ? \\
\overline{r} &:= \langle t_r, e.p, v\rangle | v = ? \\
t_r &\in \{add(+), delete(-), modify(\times)\} \\
e.p &\in \zeta_{e.p}(\gamma)
\end{aligned}
$$

Considering our example class- and sequence diagram from Figures 2.1 and 2.3, the inconsistency from the validation of the constraint condition γ_1(`play`) can be repaired by renaming the *Message* `play`, by the removal of this *Message*, by the renaming of one of the existing *Operation*s of the `VideoServer` or the `Streamer`, or by the addition of a new *Operation* to one of these *Class*es.

As mentioned before, a single repair action can be sufficient to resolve an inconsistency. However, some inconsistencies do have alternatives ($\bullet$) to resolve them and the alternatives may be composed of several repair actions ($+$) (i. e., more that one scope element must be modified) to resolve the inconsistency. A single alternative of actions needed to resolve an inconsistency is a *repair* (R).

Definition 7. *A **repair** (R) is a combination of repair actions that must be executed on the model to resolve an inconsistency. A repair that contains at least one abstract repair action is an **abstract** repair $\overline{R}$.*

$$
\begin{aligned}
R(\gamma_i) &:= +r | \gamma_i \rightarrow \gamma_c \\
R \rightarrow \overline{R} &\Leftrightarrow \exists r \in R | r \rightarrow \overline{r} \\
\gamma_i &\ \ldots\ \text{violated (inconsistent) constraint condition} \\
\gamma_c &\ \ldots\ \text{satisfied (consistent) constraint condition}
\end{aligned}
$$

A repair must be complete in the sense that all actions which are needed to resolve an inconsistency are included and it must be minimal in the sense that no action is included in the repair which does not contribute to the resolution of an inconsistency. In Chapter 7 we will proof that the proposed approach fulfills these properties.

To repair the inconsistency of $\gamma_1(\texttt{play})$ we decide to rename the *Operation* `select` of the *Class* `Controller` to `play`:

$$\begin{aligned} r_1 &:= \langle \times, \mathit{Operation}(\text{`}\mathit{select}\text{`}).\texttt{name}, \text{`}\mathit{play}\text{`} \rangle \\ R(\gamma_1(\texttt{play})) &:= \{r_1\} \end{aligned}$$

The fact that this modification might cause other inconsistencies will be discussed in the following section.

2.7.2 Side Effects

Repair actions change element properties in a model and, hence, repair actions can have other effects than the resolution of one inconsistency. This effects can be positive, if another inconsistency than the one to repair becomes consistent, or negative, if a consistent constraint validation becomes inconsistent.

Definition 8. *The influence of a repair action onto other constraint validations than the one to repair is a* ***side effect*** *(S). It belongs to a single repair action and consists of validated constraint conditions that are affected by this repair action. A side effect type (t_s) can be* ***positive*** *if the repair action resolves the other constraint validation or* ***negative*** *if it causes a new inconsistency. If the repair action is an abstract repair action, the side effect type is set to* ***unknown****.*

$$\begin{aligned} S(r(\gamma_i)) &:= \langle \gamma_x, t_s \rangle \\ t_s &\in \{\mathit{positve}(p), \mathit{negative}(n), \mathit{unknown}(u)\} \end{aligned}$$

In [76] it was evaluated how common overlapping scope elements are, i. e., how many scope elements are accessed by at least two different inconsistent constraint validations. For this thesis this evaluation was extended using 29 UML models on the set of 19 constraints (Appendix A). Figure 2.5 shows that in average 80% of the inconsistent constraint validations have at least one scope element in common, hence it can be assumed that side effects are very likely and must be considered for resolving inconsistencies. In addition to the effects on other constraint validations, those commonalities can be used to determine concrete values for repair actions.

A side effect is not necessarily bound to a constraint different to the one that should be repaired, but it can also affect parts of the constraint that is repaired. If, for example, a scope element will be modified by a repair action but is accessed more than once during the constraint validation it might be possible that the modification resolves the violated part of the constraint but causes an inconsistency in another part of the constraint. There can be two reasons for that: 1) the constraint contains a contradiction, i. e., two conditions must be fulfilled by constraint that contradict each other (e. g., a element

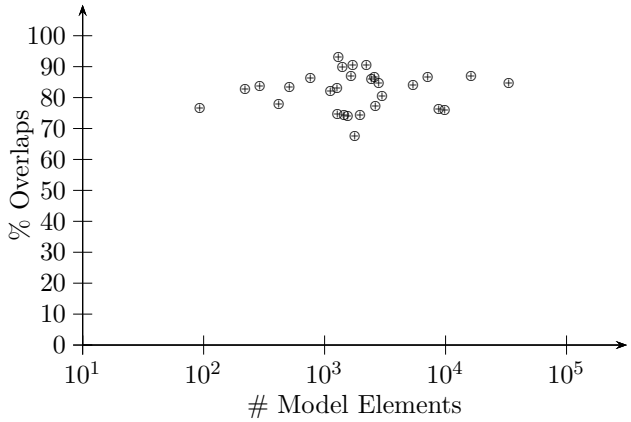


FIGURE 2.5: Overlapping Scope Elements

property must be named 'x' and the same model property must be named 'y'), or 2) the value of the repair action is not the best choice for the repair action. Therefore, the side effects will be used to detect constraints that cannot be satisfied by any state of the model and to derive values for repair actions, both, valid (values that can be assigned to a element property without causing a new inconsistency) and invalid values (values that must not be assigned to a element property because it will cause a new inconsistency).

We consider the repair for the inconsistency caused by the constraint validation $\gamma_1(\texttt{play})$. The repair $R(\gamma_1(\texttt{play}))$ affects the scope element *Operation*('select').`name` which will be accessed by the constraint validation $\gamma_1(\texttt{select}) \mapsto \textit{consistent}$ also which is actually not violated. But if $R(\gamma_1(\texttt{play}))$ will be applied, the validation of $\gamma_1(\texttt{select})$ will fail because the *Operation* `select` does not exists anymore in the *Class* `Display` nor in its super *Class* `Controller`, thus we encounter the side effect:

$$S_1(r_1) \quad := \quad \langle \gamma_1(\texttt{select}), n \rangle$$

2.8 Keeping the Performance in Mind

Performance is import in this work because the proposed solution works interactive, i. e., new information about the consistency of the model will be generated immediately after a change to the model. To achieve this our work is based on an incremental approach that needs a one-time validation of the complete model and set of constraints but new validations are done on changes to the model, i. e., the amount of re-validation is limited to the set of constraints that are affected by a change [92].

The gain of performance often comes with an increased usage of memory as it is the case with reasoning algorithms that are based on the RETE algorithm [50]. These algorithms keep the complete data structures in memory and based on pattern matching the re-validation of the affected parts of a validation is triggered. However, keeping the

complete data structures comes with the limitation of the system size and the number of constraints.

SAT based reasoning provides all the functionality needed for the proposed solution, but it comes with the need of encoding the system as well as the constraints into CNF (Conjunctive Normal Form). Recent approaches use SAT based reasoning for that purpose, all struggle with this encoding into CNF, which is time consuming and rarely incremental in contrast to the reasoning process .

2.9 Summary

In this section we introduced a simple example of an UML model containing a class diagram and a sequence diagram as well as a state machine diagram for a class. It was shown how the consistency of the model can be validated based on two example constraint expressed in OCL. Furthermore, it was defined how the cause of an inconsistency can be determined, how the inconsistencies can be repaired and what are the key challenges that this work has to deal with.

Chapter 3

Related Work

"Seek not to follow in the footsteps of men of old; seek what they sought."
Matsuo Basho, Japanese Poet, 1644-1694

In this chapter we give an overview about the work that exist in the domain of consistency checking and the processes that a consistency management framework consists of. We illustrate in more detail the underlying techniques and the previous and ongoing research in this domain.

3.1 Consistency Management

Inconsistencies are differences between the specification of a system and its realization. This thesis focuses mainly only managing consistency in software models but the concepts introduced are applicable to a broader spectrum than model-based software development.

Spanoudakis and Zisman [101] give an overview about the domain of consistency management. They define an inconsistency as a consistency rule that is is not satisfied by software models and differentiate five kind of consistency rules: well-formedness rules, description rules, application domain rules, development compatibility rules and development process compliance rules. Furthermore, they describe a process for managing inconsistencies. The main activities in this process are the detection of overlaps in the software models, the detection of inconsistencies based on a set of consistency rules, the diagnosis of inconsistencies, handling of inconsistencies, the tracking of inconsistencies and the specification and application of an consistency management policy. The diagnosis of an inconsistency is a major activity, because only if an inconsistency is thoroughly understood, actions can be defined to resolve an inconsistency and calculate all the effects on the model that the resolution of an inconsistency has.

Finkelstein et al. [48] defined (in)consistency management as the process to support the stakeholders goals in the software development process. Nuseibeh et al. [78] identified that inconsistencies improve the understanding of requirements in a team of developers. Furthermore, that it is desirable to tolerate inconsistencies, i. e., that there is no need

to resolve them immediately. In this work we detect inconsistencies and calculate the repairs immediately, but we leave it to the user how to deal with them.

Software models are not only used to understand the costumer needs but also as basis for the implementation. Lange et al. [62] made an empirical assessment about the completeness of UML [43] design model and the degree of how the model can be interpreted and misinterpreted. They observed that with an increase of completeness of the model the rate of misinterpretation decreases. Later on Nugroho et al. [77] analyzed the relation between the level of detail in UML models and the defect density in the source code. This study indicates that the more detailed UML models are, the less defects are in the source code. From this it follows that an early detection of defects, errors and inconsistencies is essential for the quality of a software product.

An efficient consistency management is not only required between costumer requirements and the software model or between the software model and the source code, but also during the design process of the model. In the design phase a model language is used (e. g., UML) that has well-formedness rules that must be enforced. Furthermore, on larger projects usually more than one person or team is working on different parts of the project that overlap and different versions of a project must be maintained. This comes with the problem that the different branches must be merged, which leads to inconsistencies. The AMOR project [5] is a versioning system for software models. Line based versioning system like SVN [84] or CVS [86] are not useable for managing and merging different versions of a software model because software models do have graph based representations and are attached with semantics that conventional versioning systems cannot deal with. Their approach also provides the support for collaborative work [22]. In contrast, this thesis does not focus on the problem merging different versions of models but on providing efficient support in resolving inconsistencies fast caused by merge conflicts.

3.2 Formalizing Requirements

To validate if a model is in a consistent state, a set of rules (specification) must be defined that the model must conform to, i. e., a consistent model conforms to its specification. Such a rule is a constraint that must be fulfilled by at least one state of the realized model. However, the origin of these constraints is varying. First of all, there are constraints from the used modeling language. In the case of an IDE such constraints are defined for the programming language, like c [58], c++ [105], java [6], c# [55] and so on. Formalizing constraints for such a formalized language is much easier than for requirements that come from the needs of a software model [70].

As it is our goal to evaluate the consistency of well formalized requirements, as well as requirements that come from semi to non formalized sources, a convenient way must be found to transform any kind of requirement into a formalized shape. However, this problem has been recognized decades ago [47] and supporting tools have been developed to improve the requirements elicitation process. Boehm et al. [18, 19] present an approach that provides a collaborative technique to negotiate about the requirements of software

systems. The tools improved the requirements elicitation process, but the problem of formalizing requirements still exist and Boehm et al. presented an hybrid approach to formalize informal user requirements [60]. Their work is based on experiences of informal stakeholder requirements from more than 100 projects.

Cimatti et al. [28] formalize requirements into object models and temporal constraints to validate if the set of requirements is free of contradictions and if they are compatible with the basic scenarios. Their work is based on Satisfiable Modulo Theory (SMT) solvers and is implemented in an extended version of the NuSMV [27] model checker. The formalization of requirements is a pre-requisite, so that the approach developed in this thesis can show its full potential.

3.3 Detecting Inconsistencies

Traditional approaches on detecting inconsistencies in software models transform the model into some intermediate representation that is validated against a set of constraints. Using an intermediate representation has many advantages as there exist well known and established techniques (e. g., SAT Solver [12, 33]) for reasoning.

Biehl and Loew [11] present an approach that uses transformations between high-level artifacts (model elements) and low-level artifacts (source code) to validate the consistency in model-based development. Their approach addresses the domain of aspect oriented programming and they compare the model with the source code to check for consistency. The model and the source code are transformed into one graph and fact extraction is used for the validation. Winkelmann et al. [109], for example, presents an approach that translates meta models and OCL [80] constraints into graphs so that the constraints can be checked during the instances generation process.

Malgouryres et al. [68] present and approach that enables the validation of UML models using Constraint Logic Programming (CLP). CLP has as input facts and rules representing the model and the constraints. The output of a CLP program are conjunctions of constraints that satisfy the goal of the CLP program. Hence, if the output is empty then an inconsistency has been detected. Queralt [87] transform both UML diagrams (class- and sequence diagrams exclusively) and OCL consistency rules into a logical representation to verify the consistency of the model. The Checking Query Containment (CQC) [44] method is used to check the satisfiability of the UML diagrams and the constraints.

In the domain of Software Product Lines (SPLs) Czarnecki and Pietrozek [29] use OCL to define well-formedness rules for the verifications of feature-based model templates which are analyzed by SAT solvers. The ability to translate constraints requires detailed understanding of the constraint semantics which is very relevant in this work. Campbell et al. [24] make use of the SPIN model checker to evaluate a range of consistency problems within and across UML diagrams for embedded systems. The UML diagrams are transformed into formalization rules for Promela, so that it can be used with SPIN. Their approach checks structural and behavior aspects of the UML diagrams.

Straeten et al. [103, 104] explore the use of description logic to detect inconsistencies between class, sequence and state machine diagrams. They differentiate inconsistencies between the different diagram types and inconsistencies that occur during the evolution of the diagrams. The diagrams and the constraints expressed in OCL are transformed into a logic and abstract syntax representation to be used with the tools Loom [66] and RACER [53].

Zisman and Kozlenkov [113] present a goal-driven knowledge base approach to manage the consistency of UML models and diagrams. UML specifications, expressed in XMI (XML Metadata Interchange), are transferred into a knowledge base and with the help of patterns and axioms consistency rules are expressed. Cheng et al. [26] introduce VisualSpecs which uses transformations to substitute the imprecision of OMT (a language similar to UML) with algebraic specifications. Conflicting specifications are then interpreted as inconsistencies. ViewPoints [37], an approach developed by Easterbrook et al., is another classical approach to consistency checking. It addresses the problem of inconsistencies and relations between different views of a software model and that the specifications for the model can evolve during the design process. An aspect that this thesis addresses also by allowing to define arbitrary constraints.

Fortunately, the complete transformation to an intermediate representation is not a pre-requisite for consistency checking. Indeed, it is possible to write constraints that directly compare design models rather than transforming them first [45, 52, 73, 95]. For example, xLinkIt [73] evaluates the consistency of "documents". They use XML as representation of the documents and XPath as well as XLink to validate the consistency and to express the inconsistencies in the documents. Such documents could be anything including UML design models that are often represented in XMI documents. Constraints are expressed in a uniform manner and xLinkIt is capable of checking the consistency of models incrementally. Their approach is type triggered, i. e., a consistency rule is validated if the type of the changed element matches the type for which the consistency rule is written for. During the validation process a set of links are generated. The links can be consistent links, if the consistency rule is not violated or inconsistent links, if the consistency rule is violated. These links are used for the re-validation of the consistency rules. Type triggered approaches are able to reduce the re-validation effort compared to batch-based approaches. However, they are not applicable to large scale models or documents as the evaluation data shows. It requires between 5 and 24 seconds for evaluating changes and the tool is thus not able to keep up with an designer's rate of model changes.

ArgoUML [94, 95] was probably the first UML design tool to implement incremental design checking but it required annotated consistency rules. Their annotations were lightweight but so where their computational benefits. The approach by Reiss [93] is in principle similar to xLinkIt. Rather than defining consistency rules on XML documents, Reiss defines consistency rules as SQL queries that are evaluated on a database which may hold a diverse set of artifacts. Reiss' use of a database makes his approach certainly more incremental. However, the incremental updates in his study suggest non-instant performance (with 30 seconds to 3 minute build times).

Blanc et al. [13] achieve near instant performance thanks to the re-writing of constraints for each relevant model change. This requires the designer to re-factor consistency rules to understand the impact of model changes. If done correctly, this leads to good performance. However, since writing these annotations may cause errors, they are no longer able to guarantee the correctness of the incremental consistency checking process.

In [112] Xu et al. present an incremental approach to detect inconsistencies based on pattern matching. Each rule expressed in first order logic is converted into a consistency computation tree where the operations like *and*, *or*, *exists*, *forall*, ... build internal nodes. Each of the operations has one or more branches (depends on the operation's arity) that are terminated by specific patterns. These patterns are used to trigger the re-validation of the rules. This is very similar to our approach, but they use this technology exclusively to improve the performance of the re-validation of consistency rules. We use this technology to get additional information from violated rules, i. e., what exactly caused the violation and how to resolve this inconsistency. Furthermore, we calculate effects that are beyond the resolution of one particular inconsistency and optimize the scope (pattern) to reduce the re-validation effort and time.

Blanc et al. [13] introduce methodological inconsistencies that constrain the order of operations that should be taken into account during the validation of the constraints. A specification for a software product normally consists of a set of rules that must be satisfied and Easterbrook and Nuseibeh [36] point out that an inconsistent specification cannot be satisfied. To detect inconsistencies Cabot et al. [23] developed an event triggered approach that is applicable to UML and OCL, and generates a set of actions that would violate the defined consistency rules. A rule is considered as violated if one of these action is executed on the model. During the validation, the consistency rule is modified in a way that the best context for the re-validation is found.

This thesis is based on Egyed's work on incremental consistency checking [39, 51], where the defined constraints are treated as black boxes (black box validation). The constraints can be defined during run time of the consistency checker, so that the constraints can be defined for the use of UML and for other domain specific languages [59, 100]. It does not need any annotations or modifications of existing languages to validate an UML model against a set of arbitrary constraints. The internal structure of the constraints is invisible to the user and can be defined in any language. The validation is triggered by a context element and the re-validation is based on a scope of model elements that are accessed during the validation of the constraint.

In this thesis we are concentrating on analyzing constraints (i. e., white box validation) that are expressed in first-order predicate logic. Due to white box validation the scope as well as the elements that need repairing can be filtered and it can be determined how they can be repaired more precisely compared to black box validation. Since decades predicate logic is interpreted as programing language and is used in the field of artificial intelligence and program analysis and optimization [61, 106]. We ingest some of these early ideas and include them to achieve an approach that is able to detect the

main cause of an inconsistency based on the structure of a first-order predicate logic expression.

3.4 Resolving Inconsistencies

Once an inconsistency has been detected it must be resolved eventually. In the spirit of tolerating inconsistencies [7] the moment when an inconsistency is detected does not necessarily need to be the same where it should be resolved. Hence, inconsistencies and their solutions should be tracked and must be kept up-to-date to the actual design progress until they will be resolved.

Inconsistencies might occur in every document and software models are expressed as documents. So in the field of document management systems (DMS), Scheffczyk et al. [98] present an approach that checks the consistency of interrelated documents that are processed by a team of authors. They use suggestion-Directed Acyclic Graphs (s-DAGs) [1] as representation of the documents and the constraints. The repairs for the documents are derived from the s-DAG representation and not from the documents. To eliminate unnecessary repairs, heuristics are used. As this might be useful in the field of DMS but this is not useful for model-based software development, because each software project has different requirements and so no generic heuristics can be derived that are applicable for all software projects. Furthermore, the generation process of repairs is independent of the inconsistency detection process and is a multilevel process that results in increasing time consumption, whereas our approach is able to detect and generate repairs on demand as soon as the model is changed.

The approach by Almeida et al. [4] considers the model and the last changes made as input to resolve inconsistencies. They use inconsistency rules and generator functions that are computed for every change that has been spotted a possible cause and a set of resolutions for the detected inconsistency. Van Der Straeten et al. [102, 103], use a knowledge base expressed in description logic as well as the query and rule language nRQL to generate repairs for inconsistent models. The inconsistencies are detected by nRQL queries, where the variables of these queries are bound to model elements. The resolutions are represented as nRQL rules that consist of statements that add or remove data from the model to resolve the inconsistency. This approach considers all inconsistencies at one time and generates a set of repair actions that transform the model from an inconsistent state to a consistent one — if a solution exists. As this approach must transform the model and the inconsistency rules into description logic, it has no incremental characteristic, i. e., the operation is similar to batch based approaches which are very time consuming.

Mens et al. [71] propose an approach that uses graph transformations to detect and resolve inconsistencies. The transformation rules are composed of a left-hand side that defines elements that must be absent or present to activate the rule. The right-hand side specifies how the graph must be transformed. The inconsistencies are resolved by transforming the graph structure of the conflict nodes that are added to the left-hand side when an inconsistency is detected.

Briand et al. [21] identifies specific change propagation rules for all types of changes and computes change actions for UML models that are relevant for our work in generating repair actions for UML. However, problematic is that there is no guarantee of correctness or completeness associated with these rules.

Based on the xLinkIt approach by [73], in [74] a repair framework for inconsistent distributed documents is presented. It uses a static analysis of the constraint structure to determine the model elements that must be changed and in some cases also the values how the model elements must be changed. To distinguish between what elements must be changed and how they must be changed, they introduced the concept of *abstract* repair actions (*where* to change, i. e., the model element) and *concrete* repair actions (*how* to change, i. e., the values that must be applied to the model element). However, their approach is conservative and may suggest incorrect repair actions because they do not consider the run time behavior of the constraint's validation.

Dam [30, 31, 32] analyzed and developed an approach in his thesis how OCL constraints can be violated or resolved respectively, based on the internal structure of the constraints. He also distinguishes five different actions that can be taken in the model to achieve a violation or resolution. Abstract repair plans are generated at compile time, i. e., the set of OCL constraints is statically defined in the tool, and these abstract actions are instantiated if the constraint is violated by the model. The repair plans that resolve the inconsistency are ranked and provided to the designer who decides which plan is executed. The repair plans itself can also be modified or executed partially. This approach is designed exclusively for OCL and a proof is given that this approach is correct and complete regarding the single OCL operations. Furthermore, this approach, in contrast to [74], is able to consider all inconsistencies at one time. A major difference to our approach is that they generate abstract actions in advance, at compile time, where we do not generate abstract actions in their sense. But our repair actions are generated based on a completely instantiated consistency rule, i. e., our repair action generation process starts after an inconsistency has been detected. We navigate the violated constructs from top to the model elements at the bottom of the validated constraint to generate the repairs. The logical constructs of the constraints define the repair actions that must be taken to resolve an inconsistency. As we know the model elements that must be changed, we are also able to detect side effects of the repair actions and we are able to deal with more than one inconsistency.

Xiong et al. [110] presents an approach that combines the detection of errors and provides actions to repair them on UML models. They use their own language for the definition of the consistency relations (constraints). This language, called *Beanbag*, has an OCL-like syntax and provides a fixing semantic for elements that are changed. However, when writing consistency relations, the designer also has to specify how this relation has to be fixed when it is violated — a manual and error prone activity which may lead to incorrect repair actions also.

Based on the previous approach on consistency checking [82], Egyed et al. [40, 41] present how to repair inconsistencies in models and how the generated choices are evaluated. However, this approach is overly conservative and generates repairs for all model

elements accessed by the validation of a constraint while often only a subset thereof causes the inconsistency.

As these approach considers the validation of a constraint as black box, i. e., they do not consider the run-time behavior of a constraint, the quality of the generated repairs is not optimal, i. e., some repairs are missing or some proposed repairs are no repairs. Nöhrer et al. introduced HUMUS [75] (High-level Union of Minimal Unsatisfiable Sets) to detect the cause of SAT [33] (Boolean Satisfiability Problem) problems in the domain of Product Line Engineering (PLE). The result of SAT solvers normally is SAT (if a solution exists) or UNSAT (if no solution exists). Their approach is able to generate all the elements that cause an UNSAT state of a software product line. As the proposed approach in this thesis does not use SAT solver for reasoning (problem of transforming the model and the constraints into CNF), the proposed approach has to deal with the same problem of missing elements (due to, for example, short circuit validation) that must be repaired to resolve an inconsistency.

In addition to the detection of elements that might be overseen, if the concrete validation of a constraints is not considered, the run-time information of a constraint validation can be used to generate concrete repair values for a repair. Malik et al. [69] present an approach to repair data structures during run time of a program. This approach combines software testing and debugging with data structure repairs. They use the data structure repair tool *Juzi* [42]. *Juzi* is a tool that tries to repair Java data structures during run-time, based on the constraints defined in a `repOk` method. This method is like a constraint definition in our approach. If the execution of this method fails, the tool tries to repair the accessed data structures by setting the last accessed field to (1) null, (2) field that have already been visited during `repOk`'s validation, and (3) one node that has not yet been visited [42]. Only these actions are reported as a tuple in the form of $\langle o, f, o' \rangle$ that will repair the data structure, i. e., where the `repOk` method succeeds. The concrete actions generated by *Juzi* are abstracted such that they can be applied to the running java code. The abstraction prioritizes expressions that start with a local variable declared by the method [69]. To validate the repaired methods they use Korat [20], a tool that uses JML [63] and JUnit [9] tests to generate counter examples from existing test cases and specifications.

Most approaches are able to generate single repair values for the repairs of an inconsistency but they fail if ranges of values are given in constraints or some values are excluded because an infinite set of solution then might exists. Xiong et al. [111] present an approach for software product lines that is able to deal with such constraints that specify a specific range of allowed values. This topic is also very import for this thesis because the proposed approach is able to generate concrete repair actions for violations of such constraint constructs.

3.5 Summary

In this chapter an overview was given about work that is directly relevant to this work as well as work that already uses technologies in other domains than the model-based software development that are very useful for the proposed approach.

Chapter 4

Basic Principles

"Any sufficiently advanced technology is indistinguishable from magic."
Arthur C. Clark, English physicist & science fiction author, 1917

In this chapter the basic principles of the proposed approach are presented. It will be shown how the constraint management can be generalized to be applicable on an arbitrary domain language and constraint language. The basic principle of this approach is the break-down of the constraints into their smallest/atomic units — the expressions. Each type of expression needs a special treatment to achieve the best performance and to determine the repairs for detected inconsistencies. Hence, for the basic principles of the proposed approach we roughly distinguish between Boolean expressions and an expression to access model element properties. Moreover, the concept of expected and validated results will be introduced.

4.1 Concept of Expected and Validated Results

In the definition of the expression presented in Section 2.4, we defined for Boolean expressions an expected result (ς) and validated result (σ). This follows the constraints that define aspects that are expected to be realized in a model. If the constraints are satisfied (i. e., the defines aspect is realized in the model), the validation result of the constraint is the expectation — the expected result. A constraint is always a Boolean condition where the expected result is 'true'. Therefore, a constraint validation indicates an inconsistency if it validates to 'false' — the expected result differs the validated result.

This principle we take over for all Boolean expression in a constraint. Therefore we can distinguish which parts of a constraint validation are not violated (the expected result equals its validated result) and which parts of a constraint validation are violated (the expected result differs is validated result). From this information we conclude what needs repairing, if the constraint validation fails (i. e., an inconsistency has been detected).

	a	b	$\sigma(\gamma)$	$S(\gamma)$	$\zeta_{e.p}(\gamma)$	$R(\gamma)$
1	$false$	$false$	$false$	$\{a\}$ or $\{b\}$	$\{a, b\}$	$R(a) + R(b)$
2	$true$	$false$	$false$	$\{b\}$	$\{b\}$	$R(b)$
3	$false$	$true$	$false$	$\{a\}$	$\{a\}$	$R(a)$
4	$true$	$true$	$true$	$\{a, b\}$	$\emptyset$	$\emptyset$

TABLE 4.1: Validations of a Conjunction $\gamma := a \wedge b$

4.2 Boolean Expressions

As Boolean expressions we define all expression types that have a Boolean result. This includes conjunctions, disjunctions, negations, universal and existential quantifiers, as well as the equality relations.

4.2.1 Conjunctions

For the illustration we use a simple conjunction as condition for the fictitious constraint $\gamma := a \wedge b$. Table 4.1 shows the possible validation results of this condition. The first column shows the number of the possibility, the second and the third column the values for a and b, and the fourth column the validation result $\sigma(\gamma)$. The expected result $\varsigma(\gamma)$ as well as the expected results for a and b are 'true' ('true' indicates a consistent validation). In the fifth column the scope for the re-validation is shown ($S(\gamma)$) and in the sixth column the cause of scope elements ($\zeta_{e.p}$) of the inconsistency (a validation to 'false'). The cause of expressions (ζ_e) includes all expressions of a constraint validation, where the expected result differs the validated result. Therefore, only the cause of scope elements is listed in the table. The last column shows which parts of the expression must be repaired ($R(\gamma)$) so that the constraint validation becomes consistent. a and b in this example are placeholders for other expressions that probably access model element properties. For simplicity, in the scope ($S(\gamma)$), cause ($\zeta_{e.p}(\gamma)$) and the repairs ($R(\gamma)$), the expressions a and b are representative for expressions that access model elements, i. e., $R(a)$ means to repair the accessed model element property accessed by a.

In the first line a and b are 'false' and therefore the validation result is 'false', i. e., inconsistent. For the re-validation of this expression it is enough to keep a or b, because if we, for example, keep a and after a modification a becomes 'true' we have to re-validate b anyhow to get the new validation result (we do not consider shortcut validation). On the other hand, if b becomes 'true' the validation result can only change if a also becomes 'true'. So, it does not matter that the modification of b does not trigger a re-validation. Hence, it will be enough to keep one of those two expressions in the scope. However, the cause for this inconsistency includes a and b because both caused the inconsistency. Therefore, the repair for this inconsistency includes the repairs for a and for b. This is a combination of repairs denoted by a '+'. To repair a and b more than one action might be needed. This depends on the expressions behind a and b explained later in Chapter 5.

The second validation is still inconsistent, only a validates to 'true'. Now, the scope of this validation must be b, because only if b changes the validation result of this expression can change. The validation of a can be dismissed, but when b changes a must be re-validated. The cause of this validation is reduced to b only and therefore repairs have to be generated for b only. The third validation where a is 'false' and b 'true' is similar except a and b must be exchanged.

The fourth validation result is 'true' (consistent) because both, a and b are 'true'. The scope of this validation must include a and b, because a change of one or both of them will affect the validation result of the validation. The cause and the repairs are empty because no inconsistency has been detected.

4.2.2 Negation

The negation is the most important operation because it allows the transformation of, for example, a conjunction into a disjunction or implication as well as from an universal quantifier to an existential quantifier. The basic operation of a negation is the inversion of the validation result and the expected result of the argument is also inverted. Moreover, it affects the generation of concrete values for the repair actions. To illustrate the effects of a negation we use this operation in combination with other expressions shown in the next sections.

4.2.3 Negated Conjunctions

For illustration we negate the example conjunction: $\gamma := \neg(a \wedge b)$. The expected result for γ is still 'true' but for the conjunction the expected result now is 'false'. Table 4.2 shows the validations of the negated conjunction. As can be seen, the scope of the validation remains the same as for the conjunction, i. e., the negation of a conjunction does not influence the scope of the expression. Different is the situation for the cause and the repairs.

An inconsistency occurred only for the fourth validation (the one that was consistent for the non negated conjunction) and the cause includes a and b. To repair this inconsistency, it is now sufficient to repair a or b (denoted by a '$\bullet$'), i. e., a negation makes a combination of repairs to an alternative of repairs. Therefore, for each possible validation result of a conjunction a separate cause and set of repairs can be generated regarding the expected result of the conjunction and the validation results of the arguments.

4.2.4 Disjunctions

When we invert a and b ($\gamma := \neg(\neg a \wedge \neg b)$), we get a *disjunction* $\gamma := a \vee b$ (DeMorgan's Law[1]):

$$\neg(\neg a \wedge \neg b) \quad \equiv \quad a \vee b$$

[1] Augustus DeMorgan, 1806-1871, British mathematician and logician

	a	b	$\sigma(\gamma)$	$S(\gamma)$	$\zeta_{e.p}(\gamma)$	$R(\gamma)$
1	$false$	$false$	$true$	$\{a\}$ or $\{b\}$	$\emptyset$	$\emptyset$
2	$true$	$false$	$true$	$\{b\}$	$\emptyset$	$\emptyset$
3	$false$	$true$	$true$	$\{a\}$	$\emptyset$	$\emptyset$
4	$true$	$true$	$false$	$\{a, b\}$	$\{a, b\}$	$R(a) \bullet R(b)$

TABLE 4.2: Validations of a Negated Conjunction $\gamma := \neg(a \wedge b)$

	a	b	$\sigma(\gamma)$	$S(\gamma)$	$\zeta_{e.p}(\gamma)$	$R(\gamma)$
1	$false$	$false$	$false$	$\{a, b\}$	$\{a, b\}$	$R(a) \bullet R(b)$
2	$true$	$false$	$true$	$\{a\}$	$\emptyset$	$\emptyset$
3	$false$	$true$	$true$	$\{b\}$	$\emptyset$	$\emptyset$
4	$true$	$true$	$true$	$\{a\}$ or $\{b\}$	$\emptyset$	$\emptyset$

TABLE 4.3: Validations of a Disjunction $\gamma := a \vee b$

Table 4.3 shows the validations of a disjunction. In contrast to the conjunction, in the first case where both arguments are 'false', the scope elements from a and b must be kept in the scope because if either one of them changes the overall validation result of this expression will change. But for the fourth validation, where both arguments are 'true', it is sufficient to keep only the scope elements of one of the arguments in the scope because to change the overall validation result of a disjunction both of them must change their result. In the other two cases the scope elements from the argument that validates to 'true' must be kept in the scope, because if this argument changes then the second must be validated to determine if the expression will change its validation result. The other argument that validates to 'false' can change its result without any effect on the overall validation result of the disjunction.

A disjunction validates to 'false' only (i. e., inconsistent) if both arguments validate to 'false'. Hence, the cause for this inconsistency is a and b. However, to resolve the inconsistency it is sufficient to repair only one of the two arguments. Hence we get two alternatives to resolve the inconsistency, repair a or repair b.

4.2.5 IMPLICATIONS

Another important expression in first order logic is the *implication* $\gamma := a \Rightarrow b$. An implication can be expressed as disjunction or conjunction:

$$\begin{aligned} a \Rightarrow b &\equiv \neg a \vee b \\ a \Rightarrow b &\equiv \neg(a \wedge \neg b) \end{aligned}$$

Table 4.4 shows the validation results for an implication where the expected result is 'true'. An implication validates to 'false' only if a is 'true' and b is 'false', otherwise the

	a	b	$\sigma(\gamma)$	$S(\gamma)$	$\zeta_{e.p}(\gamma)$	$R(\gamma)$
1	$false$	$false$	$true$	$\{a\}$	$\emptyset$	$\emptyset$
2	$true$	$false$	$false$	$\{a, b\}$	$\{a, b\}$	$R(a) \bullet R(b)$
3	$false$	$true$	$true$	$\{a\}$	$\emptyset$	$\emptyset$
4	$true$	$true$	$true$	$\{b\}$	$\emptyset$	$\emptyset$

TABLE 4.4: Validations of an Implication $\gamma := a \Rightarrow b$

validation result is 'true'. The scope for the first validation, where both arguments are 'false', must be a because only then, if a changes, the result of b can have an effect on the overall validation result of the expression (as can be seen in validation three). The second validation is the one that is inconsistent and the scope for this validation contains both arguments because a change of one of them can change the overall validation result. As a consequence, both arguments cause the inconsistency. To resolve this inconsistency it is sufficient to repair a or to repair b. The scope of the fourth validation, where both arguments are 'true', contains b only because only if b becomes 'false' the validation result of the expression changes. However, a must be validated to get the new validation result and the new scope/cause/repairs. If a becomes 'false', the overall validation result does not change, hence there is no need to keep a in the scope.

4.2.6 Universal Quantifiers

To illustrate an universal quantifier we take the condition $\gamma := \forall a \in A : a$. Table 4.5 shows the validations of an universal quantifier on a set of two elements where the condition must hold. In contrast to the conjunction where the second and the third column showed the arguments of the expression, the second column now shows the source set of the quantifier. For simplicity the set of elements are Boolean values and the condition is simply the element of the source set and to illustrate the basic principle of quantifier the number of variables (number of elements from the source) for the quantifier condition is limited to one, but the basic principle is applicable to conditions with more than one variable.

The first validation fails because all elements in the source are 'false'. The scope that must be maintained for this validation includes the source of the elements (usually the source is determined from the model, i. e., it is a property call). Additionally to the source, one validation of the quantifier's conditions must be kept in the scope, because an addition or deletion of an element might have an influence of the quantifier's validation result, if, for example, all elements are removed where the condition validates to 'false'. To keep one validation that validates to 'false' is sufficient because all validations (this includes the one that is kept in the scope) must be 'true'. The change of the one element leads to further re-validations which lead to a new scope (another failed validation will be kept in the scope if there are more) and probably to a new validation result (if all other validations are 'true') of the overall quantifier.

	$A = \{a_1, a_2\}$	$\sigma(\gamma)$	$S(\gamma)$	$\zeta_{e.p}(\gamma)$	$R(\gamma)$
1	$\{false, false\}$	$false$	$\{A, a_1\}$ or $\{A, a_2\}$	$\{A, a_1, a_2\}$	$\langle -, A, a_1\rangle + \langle -, A, a_2\rangle$ • $R(a_1) + R(a_2)$
2	$\{true, false\}$	$false$	$\{A, a_2\}$	$\{A, a_2\}$	$\langle -, A, a_2\rangle$ • $R(a_2)$
3	$\{false, true\}$	$false$	$\{A, a_1\}$	$\{A, a_1\}$	$\langle -, A, a_1\rangle$ • $R(a_1)$
4	$\{true, true\}$	$true$	$\{A, a_1, a_2\}$	$\emptyset$	$\emptyset$
5	$\emptyset$	$true$	$\{A\}$	$\emptyset$	$\emptyset$

TABLE 4.5: Validations of an Universal Quantifier $\gamma := \forall a \in A : a$

In contrast to the scope, the cause of that inconsistency includes the source and all the failed validations of the quantifier‘s condition, because each failed validation violates the condition and cause the inconsistency. To repair this inconsistency several alternatives exist. Removing (denoted by a ‘−‘) these elements from the source that violated the quantifiers condition, repairing the failed validations of the quantifiers condition, or a combination of the first two alternatives, i. e., deleting some elements and repairing the remaining failed validations.

While the first line of Table 4.5 shows a validation where all elements of the source violate the quantifier‘s condition, line two and three show validations where only one element of the source violates the condition. In these cases it is sufficient to keep the source and the elements from the violated condition in the scope and in the cause. To repair these inconsistencies two alternatives exist: 1) deleting the element that violates the quantifiers conditions or 2) repairing the validation of the quantifier‘s condition that fails.

Line four and five show validations that do not cause an inconsistency. In these cases the source must be kept in the scope because an addition of an element to the source might lead to a violation of the quantifier‘s condition. If the source is not empty (line four) then all the validations of the quantifier‘s condition must be kept in the scope because each change to one validation can change (i. e., the quantifier becomes inconsistent) the overall validation result of the quantifier.

4.2.7 Existential Quantifiers

To illustrate an existential quantifier we start with the negation of the universal quantifier $\gamma := \neg\forall a \in A : a$. Table Table 4.6 shows how the negated universal quantifier is validated on a set of two elements, the required scope for re-validation, the cause of an inconsistency and the parts that must be repaired. The expected result for the complete expression is ‘true‘ but ‘false‘ for the universal quantifier itself. As can been seen,

	$A = \{a_1, a_2\}$	$\sigma(\gamma)$	$S(\gamma)$	$\zeta_{e.p}(\gamma)$	$R(\gamma)$
1	$\{false, false\}$	$true$	$\{A, a_1\}$ or $\{A, a_2\}$	$\emptyset$	$\emptyset$
2	$\{true, false\}$	$true$	$\{A, a_2\}$	$\emptyset$	$\emptyset$
3	$\{false, true\}$	$true$	$\{A, a_1\}$	$\emptyset$	$\emptyset$
4	$\{true, true\}$	$false$	$\{A, a_1, a_2\}$	$\{A, a_1, a_2\}$	$\{\langle +, A, a_3\rangle\}\bullet$ $R(a_1) \bullet R(a_2)$
5	$\emptyset$	$false$	$\{A\}$	$\{A\}$	$\{\langle +, A, a_3\rangle\}$

TABLE 4.6: Validations of a negated Universal Quantifier $\gamma := \neg\forall a \in A : a$

the scope remains the same as for the universal quantifier. But the validation now is inconsistent for the cases that were consistent in the non negated quantifier. For the case where both elements are 'true', the cause is the source of the quantifier and all elements in the source. But to repair this inconsistency it is enough to add an element (denoted by a '+') to the source (i. e., a deletion becomes an addition due to a negation of the expression) that validates to 'false'. An alternative is also to repair one of the validations (a combination becomes an alternative due to the negation). A combination of additions (deletions for the non negated quantifier) and repairing some of the validations does not make necessarily sense in that case. If the set is empty, the cause is the source of the quantifier only, i. e., only adding an element to the source of the quantifier that validates to 'false' can resolve this inconsistency.

When we negated the condition of the universal quantifier, we get the existential quantifier:

$$\begin{aligned} \neg\forall a \in A : \neg a &\equiv \exists a \in A : a \\ \neg\exists a \in A : \neg a &\equiv \forall a \in A : a \end{aligned}$$

This can be proven by DeMorgan's law, as an universal quantifier can be seen as a cascading conjunction and an existential quantifier as a cascading disjunction for which DeMorgan's law is valid. Table 4.7 shows the validations of the existential quantifier. The only validations where this expression is 'false' are when the condition of this quantifier validates to 'false' for all elements from the source or if the source set is empty (validation one and five).

4.2.8 Equality Relations

The equality relation has, like the conjunction, two arguments that will be compared against each other and validates to a Boolean result, 'true' if both arguments are equal and 'false' otherwise. The arguments of this relation can be constants or values from element properties. This expression type provides besides the source of a quantifier, concrete model element properties for the scope, cause and the repairs. Moreover, it provides concrete data how a model element property can be modified to resolve an inconsistency. The equality relation itself has an expected result but its arguments do

	$A = \{a_1, a_2\}$	$\sigma(\gamma)$	$S(\gamma)$	$\zeta_{e.p}(\gamma)$	$R(\gamma)$
1	$\{false, false\}$	$false$	$\{A, a_1, a_2\}$	$\{A, a_1, a_2\}$	$\{\langle +, A, a_3\rangle\}\bullet$ $R(a_1) \bullet R(a_2)$
2	$\{true, false\}$	$true$	$\{A, a_1\}$	$\emptyset$	$\emptyset$
3	$\{false, true\}$	$true$	$\{A, a_2\}$	$\emptyset$	$\emptyset$
4	$\{true, true\}$	$true$	$\{A, a_1\}$ or $\{A, a_2\}$	$\emptyset$	$\emptyset$
5	$\emptyset$	$false$	$\{A\}$	$\{A\}$	$\{\langle +, A, a_3\rangle\}$

TABLE 4.7: Validations of an Existential Quantifier $\gamma := \exists a \in A : a$

	a	b	$\sigma(\gamma)$	$S(\gamma)$	$\zeta_{e.p}(\gamma)$	$R(\gamma)$
1	‘x‘	‘x‘	$true$	$\{a, b\}$	$\emptyset$	$\emptyset$
2	‘x‘	‘y‘	$false$	$\{a, b\}$	$\{a, b\}$	$\{\langle \times, a, \text{‘}y\text{‘}\rangle\} \bullet \{\langle \times, b, \text{‘}x\text{‘}\rangle\}$
3	‘x‘	`const`(‘x‘)	$true$	$\{a\}$	$\emptyset$	$\emptyset$
4	‘x‘	`const`(‘y‘)	$false$	$\{a\}$	$\{a\}$	$\{\times, a, \text{‘}y\text{‘}\}$

TABLE 4.8: Validations of an Equality Relation $\gamma := a = b$

not have one (cf. Section 2.4). Considering a single equality relation $\gamma := a = b$, the expected result ‘true‘ is to be considered consistent.

Table 4.8 shows the validation of an equality relation. The first two rows consider the comparison of two elements that are modifiable, i. e., the values can be changed (e. g., properties of elements). In contrast, the bottom two lines consider the comparison of on modifiable element and one constant value that cannot be changed. A constant value is considered as a part of the expression only (i. e., can not be in the cause of model element properties, but in the cause of expressions) and cannot be changed by a modification of the model.

The scope and the cause (in the case of an inconsistency) of the equality relation contains all model element properties of the expressions, regardless of the actual validation result of the expressions. To repair an equality relation, there exist two alternatives for the case of two modifiable elements: the first alternative modifies (denoted by a ‘×‘) the value from a to the value of b and the second alternative modifies the value from b to the value of a. If only one modifiable element exists in the relation, then the value of the modifiable element can be modified to the constant value.

4.2.9 Inequality Relations

To generate concrete repairs it is not enough to know the logical relationships, but also to know values that must be assigned to locations in the system. The equality relation is such an operation that provides information about what values can be assigned to specific properties of elements in a system. But what if the equality relation is negated? An inequality ($\neg(a = b) \equiv (a \neq b)$) relation expresses that an element property must

	a	b	$\sigma(\gamma)$	$S(\gamma)$	$\zeta_{e.p}(\gamma)$	$R(\gamma)$
1	‘x‘	‘x‘	$false$	$\{a, b\}$	$\{a, b\}$	$\{\langle \times, a, ? \setminus \text{‘}x\text{‘} \rangle\} \bullet \{\langle \times, b, ? \setminus \text{‘}x\text{‘} \rangle\}$
2	‘x‘	‘y‘	$true$	$\{a, b\}$	$\emptyset$	$\emptyset$
3	‘x‘	`const`(‘x‘)	$false$	$\{a\}$	$\{a\}$	$\{\times, a, ? \setminus \text{‘}x\text{‘}\}$
4	‘x‘	`const`(‘y‘)	$true$	$\{a\}$	$\emptyset$	$\emptyset$

TABLE 4.9: Validations of an Inequality Relation $\gamma := a \neq b$

not have a specific value but it does not provide information what value can/should be applied to this property.

In Table 4.9 the validations of an inequality relation are shown. In contrast to the equality relation, it is expected that the two arguments are not equal (if the expected result for the equality relation would be ‘false‘). The second and fourth validation are consistent. The scopes for these validations are the same as for the consistent validation of the equality relations (the property calls of the relation's arguments).

The first validation show the comparison of two property calls that have the same value and the third validation shows the validation of one property call an one constant argument. The scope and the cause for the first validation include both property calls and for the third validation only the one property call that is in the scope and cause, respectively. To repair the inconsistency one of the two arguments must be changed but, in contrast to the equality relation where single concrete values could be determined for the repairs, no concrete value can be generated. Theoretically an infinite number of possible values exists (each text except ‘x‘) that can be chosen to resolve this inconsistency, hence no concrete value can be determined (indicated by $? \setminus$ ‘x‘ — causes a conditional concrete repair action).

To overcome this circumstance two solutions exists: 1) request user input for an appropriate value or 2) try to find a value from other constraint validations. In Chapter 5 we will look into the second possibility to determine concrete values for such repairs.

4.3 Property Call Expressions

Property call expressions provide data that come from the model element properties. These expressions are the basis for the locations where repair actions must be applied as well as for the determination of how to repair an inconsistency — the concrete values. During the validation of a property call expression a scope element is generated and the value of the element property is the validation result of this expression. This expression does not have an expected result and does not have a scope nor a cause. But the scope element of that expression is *part* of a scope or cause. Furthermore, a scope element represented by one property call expression can be part of more property call expressions in the same constraint or in other constraints, i. e., a model element property can be accessed more than once by the set of constraints. In the illustrations for the logical

elements the elements a, b as well as the set A acted as placeholders for expressions that potentially can be property call expressions.

In addition, a property call expression can be a chain of property call expressions, i. e., the result of the first property call $(e_1.p_1 \mapsto e_2)$ is the source model element for the second property call $(e_2.p_2)$.

$$(e_2.p_2) \circ (e_1.p_1) \quad \equiv \quad e_1.p_1.p_2$$

It follows that a property call expression can provide more than one scope element for the scope and the cause. Moreover, this also has consequences for the repairs. The scope elements provided by the chain of property calls are used for the repairs determined from the parental Boolean expression, e. g., an universal quantifier generates an *add* repair for the scope element representing the source $(e_1.p_1.p_2)$ of the quantifier, hence, in our example this would be $\langle +, e_1.p_1.p_2, `true` \rangle$. For all scope elements provided by the earlier property call expressions an abstract modify repair will be generated $(\langle \times, e_1.p_1, ? \rangle)$.

4.4 Summary

In this chapter the basic principles our approach is based on are shown. The concept of the expected and validated result for the smallest units of a constraint — the expressions — is introduced together with the concept is used to calculate the scope of a constraint validation and the cause of an inconsistency as well as the repairs for those basic expressions.

Chapter 5

CiM Approach

"If you can dream it, you can do it."
Walt Disney, American Cartoonist, 1901-1966

In the last chapter the basic principles of our approach are shown based on single expressions. However, a constraint does not consists of a single expression, but of a set of interrelated expressions, i. e., one expression is the argument of another. How the interrelated expressions are used for the consistency management the CiM (Consistency in Models) approach is introduced in this chapter. The working, i. e., detecting and resolving inconsistencies based on an arbitrary set of constraints are illustrated on the example model and constraints introduced in Chapter 2.

5.1 Overview

It is presumed an arbitrary set of constraints and a model on which the constraints will validated. The working of the CiM approach is a four-stage process shown in Figure 5.1.

Stage one is the validation of the constraints to detect inconsistencies. The validation will be logged, i. e., for each expression the expected and validated results are stored, as well as the relationships of the expressions. Stage two calculates the scope to make the constraint validation ready for a re-validation, if one of the elements in the scope changes. Stage three is entered only, if an inconsistency has been detected, and the cause of that inconsistency is determined. Stage four calculates out of the cause the repairs. To resolve an inconsistency a set of one to many single repair actions are needed. This set of repair actions build a repair. Each of the repair actions can have side effects on other constraint validations that will be calculated too. The calculated results, the repairs and their side effects, will be presented to the user who decides how to deal with the inconsistencies, i. e., if and what repair should be applied on the model.

Figure 5.2 briefly recaps the example from Chapter 2 and shows an excerpt of the example UML diagrams and the constraint that will be validated on them. This example will be used to illustrate the detailed working of the approach shown in the next sections.

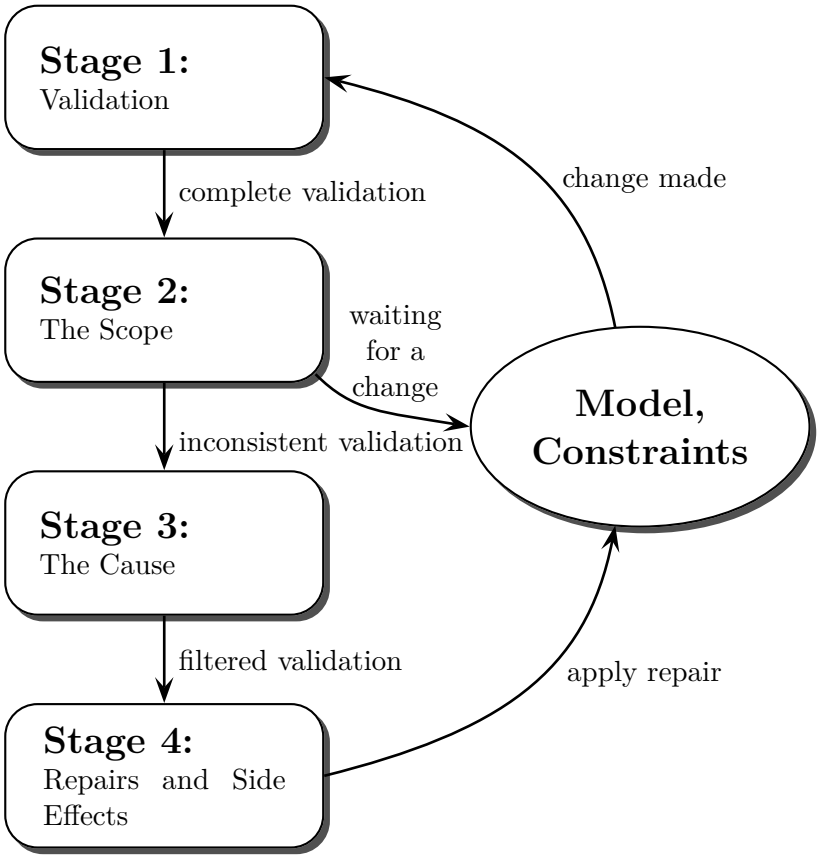
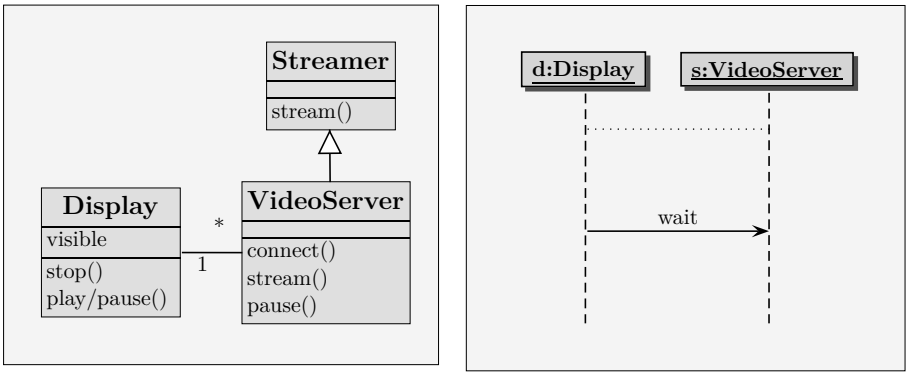


FIGURE 5.1: Working of the CIM Approach

5.2 Stage 1: Validation

To illustrate the working we introduce the basic data structure our approach is based on — the *validation tree* [90]. A validation tree represents the validation of a constraints, i. e., it consists of the expressions from the constraint condition with their validation results. In Algorithm 1 is shown how the validation tree is built. The build up process starts at the root expression of the constraint (Line 1 to Line 3). The recursive build up process of the expression starts with the root expression (Line 11). First a node for the expression will be created (Line 12). If the expression is not a property call expression (Line 13) then it will be checked if there exists a parent expression (Line 14). If a parent expression exists then it will be checked if the parent is a negation (Line 15). If the parent expression is a negation then the expected result of the expression `e` will be set to the inverted expected result of its parent. Otherwise the expected result will be set to the same expected result as of the parent. In the case where no parent exists (the root expression), the expected result will be set to 'true'.




```
context Message:
self.receiveEvent.covered-> forAll(l:Lifeline|
  l.represents.type.ownedOperation->exists(o:Operation|
    self.name=o.name))
```

FIGURE 5.2: Excerpt of the example Constraint, UML Class and Sequence Diagram

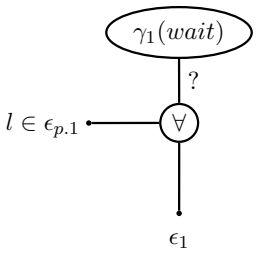


FIGURE 5.3: Validation Tree for $\gamma_1(wait)$ — Step 1

After that, for all arguments of the expression `e` recursively the validation tree will be built (Line 23). Then a branch between the node for the expression `e` and the new created node for the argument expression will be created (Line 25). When the validation tree has been build completely, the validation process starts by applying the operation of the expression `e` on all the arguments of the expression `e` in the recursive climb up.

To illustrate the build up of the validation tree we start with a simplified constraint $\gamma_1 := \forall l \in \epsilon_{p.1} : \epsilon_1$ validated on the *Message* `wait`. Figure 5.3 shows the first step until the first recursive call of the build validation tree algorithm.

Step 1: The first expression validated is the root expression ϵ_0 which is equivalent the Constraint 1 validated on the *Message* `wait`. The root expression has as its arguments the property call expression for the *Lifelines* that the *Message* is sent to ($\epsilon_{p.1}$) and a condition (ϵ_1) that must hold for the elements of $\epsilon_{p.1}$ that are assigned to the variable l. Later in the validation tree we indicate the assigned values to a variable using an arrow, e. g., $l \mapsto$'s', which means that the value 's' is assigned to the variable l. The validated

Algorithm 1 Building the Validation Tree

```
1  BuildValidationTree(Constraint c)
2      Expression e = RootExpression of c
3      BuildValidationTree(e)
4      if e.validationResult = true
5          c->consistent
6      else if e.validationResult = false
7          c->inconsistent
8      endif
9  end /*BuildValidaitionTree(Constraint)*/
10
11 BuildValidationTree(Expression e)
12     CreateNode n for e
13     if e is not PropertyCallExpression then
14         if e.hasParent then
15             if e.parent is Negation then
16                 e.expectedResult = not e.parent.expectedResult
17             else
18                 a.expectedResult = e.parnet.expectedResult
19             endif
20         else
21             e.expectedResult = true
22         endif
23         forall Expression a in e.arguments
24             BuildValidationTree(a)
25             CreateBranch from n to the node of a
26         endforall
27     endif
28     e.validationResult = Apply e.operation on e.arguments
29 end /*BuildValidaitionTree(Expression)*/
```

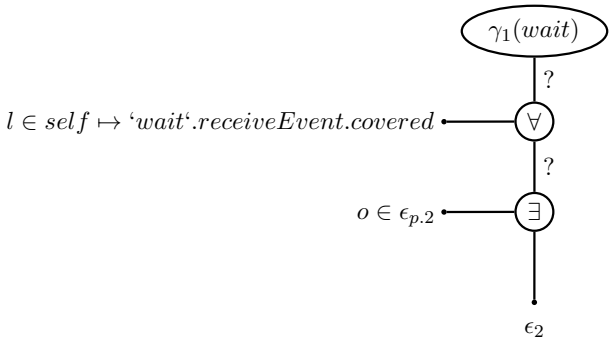


FIGURE 5.4: Validation Tree for $\gamma_1(wait)$ — Step 2

result is at the moment of the build up process unknown (indicated as '?'). The property call expression $\epsilon_{p.1}$ consists of two cascading property calls ($\epsilon_{p.1} \circ \epsilon_{p.1.1}$).

$$
\begin{aligned}
\epsilon_0 &\equiv \gamma_1(wait) \\
\epsilon_0 &:= \langle \forall, \{\epsilon_{p.1}, \epsilon_1\}, ? \rangle \\
\epsilon_{p.1} &:= \langle covered, \epsilon_{p.1.1}.\sigma, \epsilon_0, \{\text{'}s\text{'}\} \rangle \\
\epsilon_{p.1.1} &:= \langle receiveEvent, \text{'}wait\text{'}, \epsilon_{p.1}, MessageOccurenceSpecification \rangle \\
\epsilon_1 &:= \langle \exists, \{\epsilon_{p.2}, \epsilon_2\}, \epsilon_0, ?, \text{'}true\text{'} \rangle
\end{aligned}
$$

Step 2: The condition (ϵ_1) of a quantifier must be validated for the elements from the source ($\epsilon_{p.1}$). As this property contains only one *Lifeline*, only one expression of the quantifiers condition will be created. Figure 5.4 shows the expressions that are newly created in the next step in the build up process of the validation tree. The newly created expression is an existential quantifier that has as its source ($\epsilon_{p.2}$) the *Operation*s of the *Class* that represents the type of the *Lifeline* assigned to the variable `l` from the universal quantifier, the parent expression of the existential quantifier. The property call $\epsilon_{p.2}$ is a cascading property call too, consisting of three cascading property calls: $\epsilon_{p.2} \circ \epsilon_{p.2.1} \circ \epsilon_{p.2.2}$. The result are the three *Operations* of the *Class* `VideoServer`. The condition of the existential quantifier is the equality relation (ϵ_2) that compares if the name of the *Message* `wait` is equal the name of an *Operation* of the class. The result of the existential quantifier, its parent, the universal quantifier and consequentially the validation results of the expressions are still unknown until now.

$$
\begin{aligned}
\epsilon_{p.2} &:= \langle ownedOperation, \epsilon_{p.2.1}.\sigma, \epsilon_1, \{\text{'}stream\text{'}, \text{'}connect\text{'}, \text{'}pause\text{'}\} \rangle \\
\epsilon_{p.2.1} &:= \langle type, \epsilon_{p.2.2}.\sigma, \epsilon_{p.2}, \text{'}VideoServer\text{'} \rangle \\
\epsilon_{p.2.2} &:= \langle represents, l \mapsto \text{'}s\text{'}, \epsilon_{p.2.1}, Property \rangle \\
\epsilon_2 &:= \langle =, \{\epsilon_{p.3.1}, \epsilon_{p.3.2}\}, \epsilon_1, ?, \text{'}true\text{'} \rangle
\end{aligned}
$$

Step 3: The expressions for the condition (ϵ_2) of the existential quantifier will be built. This expression must be validated three times (ϵ_{2-1} to ϵ_{2-3}), one time for

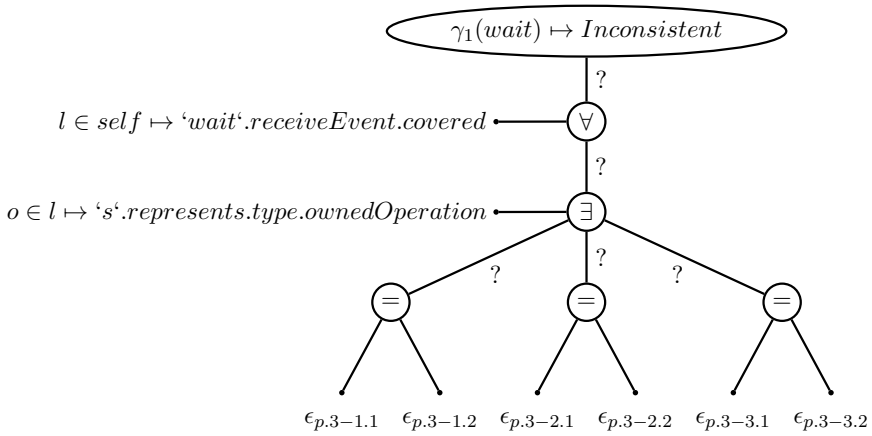


FIGURE 5.5: Validation Tree for $\gamma_1(wait)$ — Step 3

each *Operation* of the *Class* `VideoServer`. The three validations are represented as the three lower branches of the existential quantifier, represented as the equality nodes (=). Figure 5.5 shows the created equality relation expressions beneath the existential quantifier. The branches of the equality relations are property calls. On the left-hand side the name property of the *Message* and on the right-hand side the name property of the *Operation*s. The validation results of the equality relations and the other Boolean expressions are still unknown.

$$
\begin{aligned}
\epsilon_{p.3-1.1} &:= \langle name, self \mapsto \text{‘}wait\text{‘}, \epsilon_{2-1}, \text{‘}wait\text{‘} \rangle \\
\epsilon_{p.3-1.2} &:= \langle name, o \mapsto \text{‘}stream\text{‘}, \epsilon_{2-1}, \text{‘}stream\text{‘} \rangle \\
\epsilon_{p.3-2.1} &:= \langle name, self \mapsto \text{‘}wait\text{‘}, \epsilon_{2-2}, \text{‘}wait\text{‘} \rangle \\
\epsilon_{p.3-2.2} &:= \langle name, o \mapsto \text{‘}connect\text{‘}, \epsilon_{2-2}, \text{‘}connect\text{‘} \rangle \\
\epsilon_{p.3-3.1} &:= \langle name, self \mapsto \text{‘}wait\text{‘}, \epsilon_{2-3}, \text{‘}wait\text{‘} \rangle \\
\epsilon_{p.3-3.2} &:= \langle name, o \mapsto \text{‘}pause\text{‘}, \epsilon_{2-3}, \text{‘}pause\text{‘} \rangle
\end{aligned}
$$

Step 4: The validation tree will be validated, i. e., the result of constraint validation will be calculated. Until now, only the validation results for these expressions are known that build the leaves of the validation tree, i. e., the property calls of the quantifiers. The branches of the equality relation are also leaves and underneath these property calls no expressions to generate are left, i. e., the validation tree is build complete and the validation starts.

First the property calls for the *Message* name and the *Operation* ($\epsilon_{p.3-x.x}$) names are validated. These results are compared in the equality relations (ϵ_{2-x}) and a validation result is calculated (in our case all validation results are ‘false‘). Figure 5.6 shows the complete validation tree including the validation results of the Boolean expressions. The

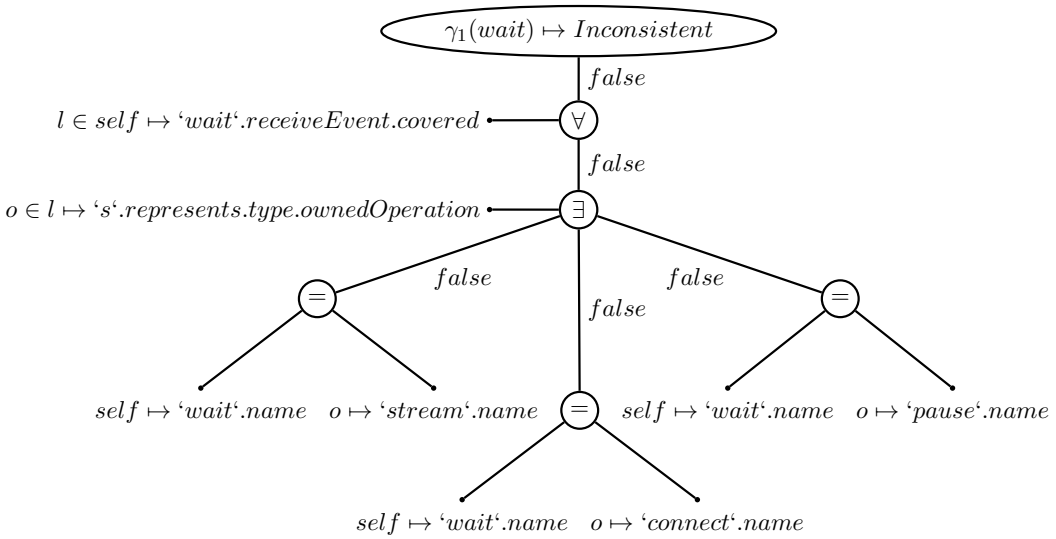


FIGURE 5.6: Complete Validation Tree for $\gamma_1(wait)$

validation result of an expression is propagated to its parent and this parent calculates its own validation result that is propagated to its parent again. This propagation process stops when the root expression has been reached and the validation result of the root expression indicates whether the constraint is violated (‘false‘↦ Inconsistent) or not (‘true‘↦ Consistent). What we can see here is that the build-up of a validation tree is top-down whereas the validation is bottom-up, starting as soon as a leaf (property call expression) has been reached.

5.3 Stage 2: The Scope

The scope (S) of a constraint validation are the model element properties (scope elements) that are accessed during the constraint validation and do have an influence on the validation result. The scope is needed to trigger a re-validation of the constraint or parts thereof.

5.3.1 Calculating The Scope

Constraints consist of expressions which have arguments. As shown in Chapter 4, what arguments influence the validation result depends on the validation results of the arguments only. Table 5.1 summarizes the scopes for Boolean expressions. The Boolean values are shortened to ‘f‘ for ‘false‘ and ‘t‘ for ‘true‘. Please note that the negation does not influence the scope. In the following we summarize the Boolean expression from the validation tree in Figure 5.6. The Boolean arguments of the expressions are replaced with their validation results. Expression ϵ_1 has instead of ϵ_2 the validation results of each validated branch (ϵ_{2-1} to ϵ_{2-3}) in its arguments. In the brackets next to the validation

a	b	A	S					
			$a \wedge b$	$a \vee b$	$a \Rightarrow b$	$a = b$	$\forall x \in A : x$	$\exists x \in A : x$
f	f	$\{a, b\}$	$\{a\}$ or $\{b\}$	$\{a, b\}$	$\{a\}$	$\{a, b\}$	$\{A, a\}$ or $\{A, b\}$	$\{A, a, b\}$
t	f	$\{a, b\}$	$\{b\}$	$\{a\}$	$\{a, b\}$	$\{a, b\}$	$\{A, b\}$	$\{A, a\}$
f	t	$\{a, b\}$	$\{a\}$	$\{b\}$	$\{a\}$	$\{a, b\}$	$\{A, a\}$	$\{A, b\}$
t	t	$\{a, b\}$	$\{a, b\}$	$\{a\}$ or $\{b\}$	$\{b\}$	$\{a, b\}$	$\{A, a, b\}$	$\{A, a\}$ or $\{A, b\}$
		$\emptyset$					$\{A\}$	$\{A\}$

TABLE 5.1: Scope for the Boolean Expression Types and Argument Results

result the expressions are given that provide this result. The only expressions that are not replaced with their validation results are the property call expression. These are the expressions that will provide the scope elements for the scope.

$$
\begin{aligned}
\epsilon_0 &:= \langle \forall, \{\epsilon_{p.1}, `false`(\epsilon_1)\}, `false` \rangle \\
\epsilon_1 &:= \langle \exists, \{\epsilon_{p.2}, `false`(\epsilon_{2-1}), `false`(\epsilon_{2-2}), `false`(\epsilon_{2-3})\}, \epsilon_0, `false`, `true` \rangle \\
\epsilon_{2-1} &:= \langle =, \{\epsilon_{p.3-1.1}, \epsilon_{p.3-1.2}\}, \epsilon_1, `false`, `true` \rangle \\
\epsilon_{2-2} &:= \langle =, \{\epsilon_{p.3-2.1}, \epsilon_{p.3-2.2}\}, \epsilon_1, `false`, `true` \rangle \\
\epsilon_{2-3} &:= \langle =, \{\epsilon_{p.3-3.1}, \epsilon_{p.3-3.2}\}, \epsilon_1, `false`, `true` \rangle
\end{aligned}
$$

The scope of a validation is calculated after the validation tree has been generated. In Algorithm 2 the additional calculations (Line 34) to filter the validation tree is shown. The `FilterArguments` algorithm filters the validation tree, i. e., the validation tree will be changed. All arguments will be iterated and based on the validation result of the expression `e` and the validation result of the argument `a`, the branch between those two expression will be deleted and the nodes for the arguments are removed. To check whether the branch and the node must be deleted is done with a lookup in Table 5.1. The leaves of the remaining validation tree represents the scope of the validation.

Algorithm 2 and Table 5.1 is applied onto the expressions to calculate the scope starting at the root expression ϵ_0. This is an universal quantifier that has only one argument that validates to 'false' which comes from ϵ_1. For an universal quantifier the source and the only one argument that validates to 'false' must be added to the scope in that case, i. e., the scope elements from $\epsilon_{p.1}$ and $\epsilon_{p.1.1}$ as well as the scope elements beneath ϵ_1.

$$S(\gamma_1(wait)) := \{self \mapsto `wait`.receiveEvent,$$

Algorithm 2 Building the Scope

```
30      /* Building the Validaition Tree*/
31      FilterArguments(e)
32  end /*BuildValidaitionTree(Constraint)*/
33
34  FilterArguments(e)
35      forall Expression a in e.arguments
36          Based on e.validationResult and a.validation result /* Table
                lookup */
37              delete branch between node of e and node of a
38              delete node a
39      endforall
40  end /* FilterArguments*/
```

$$\begin{aligned} & self \mapsto \text{`}wait\text{`}.receiveEvent.covered\} \\ & \cup S(\epsilon_1) \end{aligned}$$

ϵ_1 is an existential quantifier where each of the three arguments validates to 'false'. In this case we have to add the scope elements from the source of the existential quantifier to the scope ($\epsilon_{p.2}$, $\epsilon_{p.2.1}$ and $\epsilon_{p.2.2}$) and the scope elements from all the arguments that validates to 'false'.

$$\begin{aligned} S(\gamma_1(wait)) \quad := \quad & \{self \mapsto \text{`}wait\text{`}.receiveEvent, \\ & self \mapsto \text{`}wait\text{`}.receiveEvent.covered, \\ & l \mapsto \text{`}s\text{`}.represents, \\ & l \mapsto \text{`}s\text{`}.represents.type, \\ & l \mapsto \text{`}s\text{`}.represents.type.ownedOperation\} \\ & \cup S(\epsilon_{2-1}) \cup S(\epsilon_{2-2}) \cup S(\epsilon_{2-3}) \end{aligned}$$

The three arguments that validate to 'false' are the equality relations ϵ_{2-1} to ϵ_{2-3}. As these are equality relations, the scope elements from both arguments must be added to the scope. $\epsilon_{3-1.1}$ accesses the name property of the *Message* `wait`. This scope element is the same as for the expressions $\epsilon_{3-2.1}$ and $\epsilon_{3-3.1}$. The other scope elements that are accessed by the equality relations are different for the three equality relations and for the three *Operation*s from the *Class* `VideoServer`. The resulting scope for this validation is as follows:

$$\begin{aligned} S(\gamma_1(wait)) \quad := \quad & \{self \mapsto \text{`}wait\text{`}.receiveEvent, \\ & self \mapsto \text{`}wait\text{`}.receiveEvent.covered, \\ & l \mapsto \text{`}s\text{`}.represents, \\ & l \mapsto \text{`}s\text{`}.represents.type, \end{aligned}$$

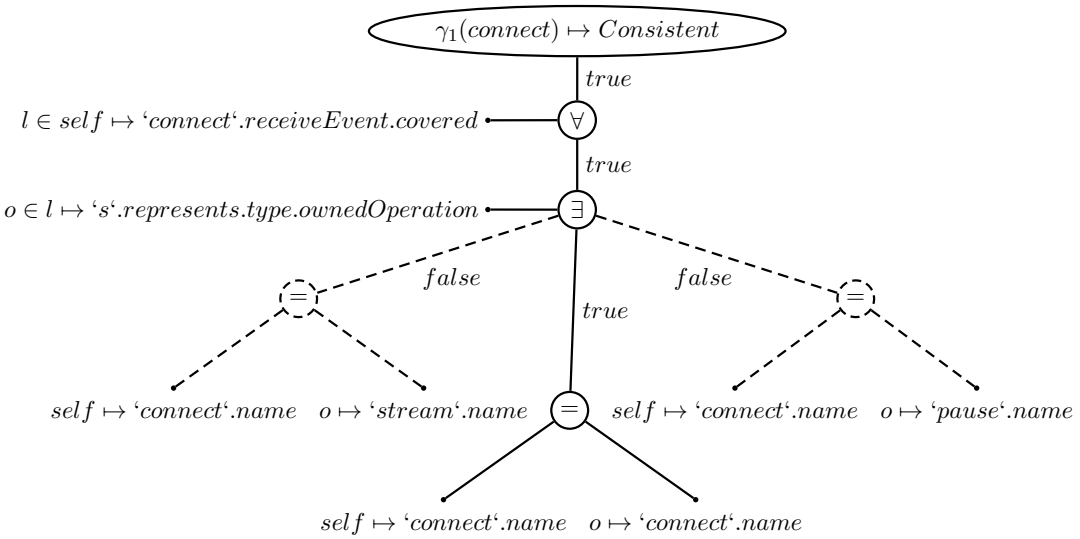


FIGURE 5.7: Validation Tree for $\gamma_1(connect)$

$l \mapsto$ ‘s‘$.represents.type.ownedOperation$,
$self \mapsto$ ‘$wait$‘$.name$,
$o \mapsto$ ‘$stream$‘$.name$,
$o \mapsto$ ‘$connect$‘$.name$,
$o \mapsto$ ‘$pause$‘$.name\}$

In total seven scope elements are in the scope of this validation and therefore, if one of these properties changes, a re-validation of the constraint is needed. The size of the scope is not always the same for the same constraint and may differ for different validations.

Another validation of Constraint 1 on the *Message* `connect` is shown in Figure 5.7. This validation does not violate the constraint (Consistent), but when we consider the complete validation nearly the same model element properties are accessed except the *Message*'s properties. But in contrast to the validation on the *Message* `wait`, one argument of the existential quantifier validates to ‘true‘ and therefore, the complete validation is also ‘true‘. From Table 5.1 we know that the scope must be calculated for those arguments of an existential quantifier that validates to ‘true‘, so the resulting scope contains two scope elements less, the two *Operation* names for `stream` and `pause`. The branches (arguments) where there is no need to consider them for the scope calculation are indicated as dashed lines in Figure 5.7. The savings do not lead to a reduced re-validation effort only, but also reduce the used memory. Why and how is discussed in the next section.

5.3.2 Triggering a Re-Validation

Using a scope to reduce the re-validation effort is not new, because it is already used in Egyed's ModelAnalyzer approach [39]. However, this approach was not able to filter out scope elements that do not have an influence on the validation result and a change of one scope element leads to a complete re-validation of the constraint. In this thesis we follow a different approach to re-validate a constraint and obtain better results regarding the re-validation for constraints after a change occurred in the model.

In the last section we showed how a validation tree is build and how the scope is calculated out of the validation tree. The validation tree is kept in memory and only these parts of a validation tree will be re-validated that are affected by a model change (similar to the RETE algorithm [50]). Lower re-validation times of RETE based algorithm result in a higher consumption of memory which might become a problem on larger models. However, the filtering of the scope elements, shown in the last section, leads also to a minimal validation tree and therefore to a significant reduction of the used memory, which is shown in the evaluations in Chapter 7.

Algorithm 3 Constraint Re-Validation

```
1  ElementPropertyChanged(ScopeElement e.p)
2      forall PropertyCallExpression e for e.p
3          ReValidate(e)
4      endforall
5  end /*ElementPropertyChanged*/
6
7  ReValidate(e)
8      oldResult = e.validationResult
9      e.validationResult = Apply e.operation on e.arguments /* Includes re-
           validation of all remaining arguments based on a table lookup */
10     if oldResult <> e.validationResult then
11         if e has parent then
12             ReValidate(e.parent)
13         else
14             Constraint of e changes
15         endif
16     endif
17 end /*ReValidate*/
```

Algorithm 3 is used for the re-validation of a constraint. For a changed model element property (a scope element) all property call expressions will be determined that call this scope element (Line 2). Then for all the determined expressions the `ReValidate` algorithm will be called (Line 7). At the beginning the validation result of the last validation will be stored followed by the re-validation of the expression, i. e., the operation of the expression is applied on the arguments of the expression that might have changed. Some of the arguments might need a complete new validation of a validation sub-tree to get a new validation result (due the scope generation / filtering). This is determined from Table 5.1. The old validation result is compared to the new validation result (Line 10)

and if they differ and the expression has a parent, a re-validation of the parent will be triggered until the top of the validation is reached. If the root expression (has no parent) is reached and the validation result has been changed, the complete constraint validation changes its result, i. e., an inconsistent validation becomes consistent and vice versa.

Changing the *Operation* 'pause' in the *Class* `VideoServer` requires a re-validation of the example constraint validation shown in the validation tree shown in Figure 5.6. As we already mentioned in the initial validation of the validation tree, the build up of the validation tree is top-down but the validation is done bottom-up. This is the same for the re-validation of a constraint. As we know the changed scope element ($\epsilon_{p.3-3.2} := \langle name, o \mapsto \text{`}pause\text{`}, \epsilon_{2-3}, \text{`}pause\text{`} \rangle$) we also know where to start with the re-validation. Figure 5.8 shows the validation tree for this change. The thick lines show the validation path bottom up, starting at the property call for the *Operation* name. The dashed lines are the branches that can be removed after the re-validation and the thin gray branches remain untouched during the re-validation.

The result of the scope element was 'pause' but after the change in the model the result now is 'wait'. Each change of a validation result of an expression will be propagated to its parent, the equality relation ϵ_{2-3}. This expression re-validates its second argument, the name property of the *Message* `wait`. This equality relation now validates to 'true', i. e., it changes its validation result and the new result is propagated to its parent, the existential quantifier ϵ_1. As an existential quantifier needs only one argument to validate to 'true', this expression changes its validation result which is propagated to its parent, the universal quantifier ϵ_0. As this is the only argument of the universal quantifier, this expression changes its validation result also to 'true' and therefore, the constraint is not violated any more. In addition, the scope for this constraint validation can be reduced, i. e., the two property calls on the other two *Operation*s can be removed from the scope.

As a second change in the model we consider an undo, i. e., we change the name of the *Operation* 'wait' back to the name 'pause'. Figure 5.9 shows the re-validations for that change in the validation tree. The re-validation starts at the same location as for the first change. The equality relation changes its result from 'true' to 'false' and propagates the result to the parent, the existential quantifier. As an existential quantifier needs at least one validation of its condition to validate to 'true', the source of the existential quantifier must be re-validated. The re-validation is needed even when no elements are added or removed from the source since then (this would trigger a re-validation) because only Boolean results are cached in the approach to reduce the memory consumption, hence, property calls must be re-validated in such cases. This is the same as for the equality relation where only one of the two arguments has been changed but the second one has to be re-validated as it is a property call expression.

After that the elements that are not validated until now must be validated using the quantifiers condition. All validations fail, hence the existential quantifier changes its validation result from 'true' to 'false' and this result will be propagated to its parent, the universal quantifier. As all elements of the universal quantifiers source have be validated, this expression changes its validation result from 'true' to 'false' also and the constraint validation is now inconsistent. The validation tree now is equal the initial validation tree.

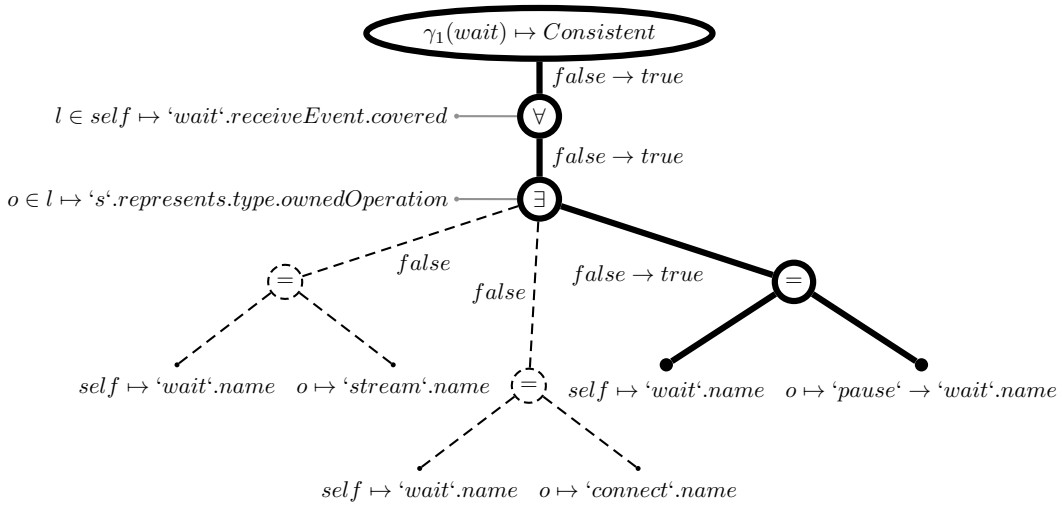


FIGURE 5.8: Changing the Name of *Operation* ‘pause‘ to ‘wait‘ in the Validation Tree for $\gamma_1(wait)$

Even in this case the one property call of the universal quantifier needs no re-validation, i. e., the re-validation effort was reduced.

To show a change that leads to prematurely termination of the re-validation we will consider the change of the *Message* named ‘wait‘ to ‘stop‘ in the sequence diagram. The scope element that represents the name property of the *Message* `wait` occurs three times in the validation tree — the left-hand side of the equality relations. This triggers three re-validations in the validation tree but the bottom up propagation stops at the equality relations because they do not change their validation result. Compared to a complete constraint re-validation two property calls and two quantifier expressions need no re-validation (see Figure 5.10).

The last change also shows, that a single change in the model can trigger multiple re-validations in one validation tree but also on other validation trees. If, for example, an *Operation* will be added to the *Class* `VideoServer` (the property *‘s‘.represents.type.ownedOperation*), this will trigger the re-validation of $\gamma_1(stop)$ and $\gamma_1(connect)$ and the *Operation* will be named ‘stop‘. For $\gamma_1(wait)$ this will create a new branch underneath the existential quantifier that validates to ‘true‘. So, all other branches underneath the existential quantifier can be removed, the new validation result will be propagated upwards, and the inconsistent has been resolved.

In the case of $\gamma_1(connect)$ the re-validation stops immediately because a new *Operation* in the *Class* `VideoServer` has no influence on the validation result of that constraint validation.

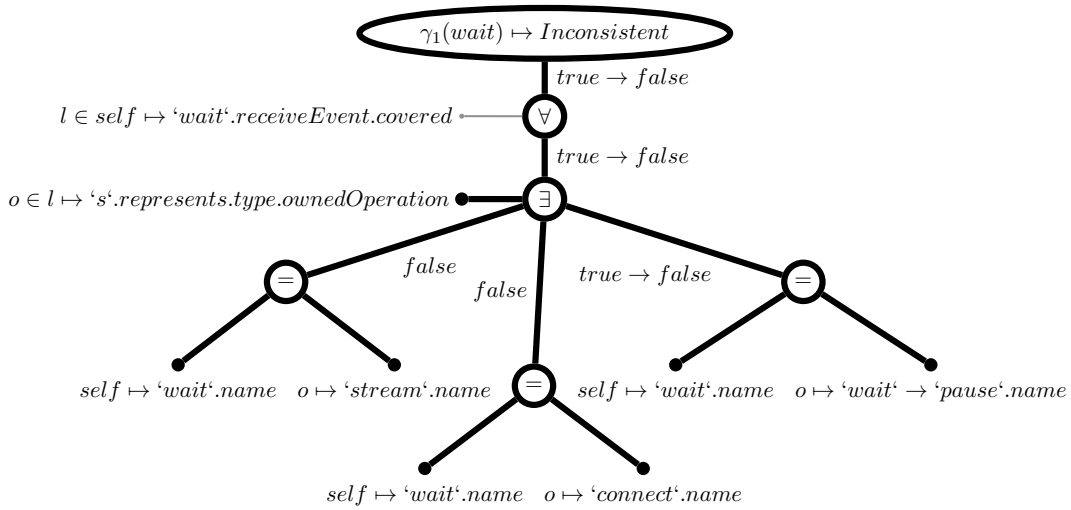


FIGURE 5.9: Changing the Name of *Operation* 'wait' to 'pause' in the Validation Tree for $\gamma_1(wait)$

a	b	A	$\zeta_{e.p}$					
			$a \wedge b$	$a \vee b$	$a \Rightarrow b$	$a = b$	$\forall x \in A : x$	$\exists x \in A : x$
f	f	$\{a, b\}$	$\{a, b\}$	$\{a, b\}$			$\{A, a, b\}$	$\{A, a, b\}$
t	f	$\{a, b\}$	$\{b\}$		$\{a, b\}$	$\{a, b\}$	$\{A, b\}$	
f	t	$\{a, b\}$	$\{a\}$			$\{a, b\}$	$\{A, a\}$	
		$\emptyset$						$\{A\}$

TABLE 5.2: Cause of Scope Elements for the Boolean Expression Types

5.4 Stage 3: The Cause

In [91] the calculation of the cause for an inconsistency will be published. The cause of an inconsistency consists of two parts: 1) the cause scope elements ($\zeta_{e.p}$) that violate a constraint 2) the cause expressions (ζ_ϵ) that are violated. In Chapter 4 we already showed how the cause of scope elements is determined for different Boolean expression and in Table 5.2 this is summarized shortly. To show how this principle is applied on a real example we use the validation tree (Figure 5.6) from Section 5.2.

To calculate the cause we need the expected result of the constraint and expressions, respectively. In Figure 5.11 the same validation tree is shown but instead of showing the validation result, now the validation result and the expected results are shown ($f \mapsto false$, $t \mapsto true$). Algorithm 4 is used to calculate the cause of an inconsistency. The process to calculate the cause starts at the root expression of the validation tree (Line 1) only if the validation tree detects an inconsistency (Line 2).

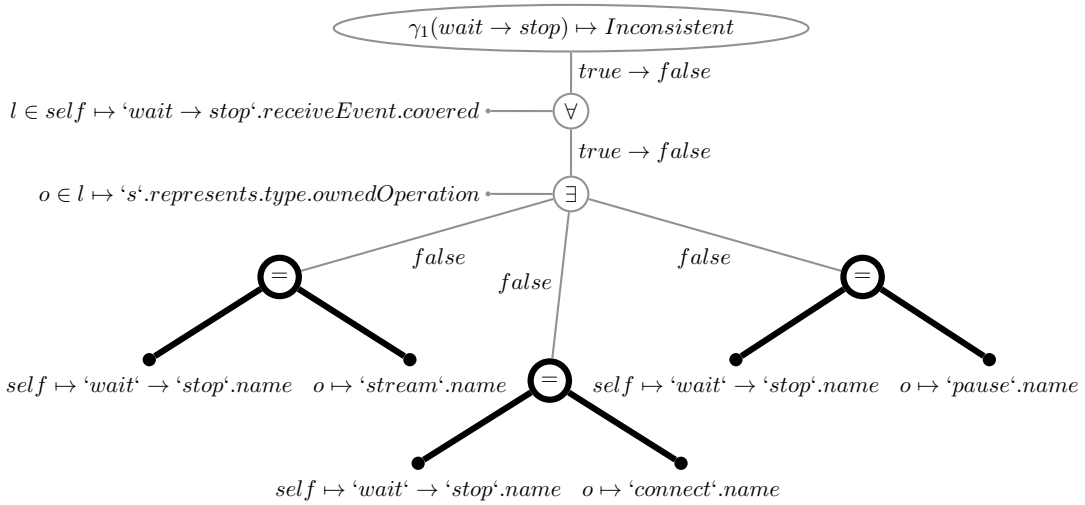


FIGURE 5.10: Changing the Name of *Message* ‘wait‘ to ‘stop‘ in the Validation Tree for $\gamma_1(wait \rightarrow stop)$

The main calculation of the cause starts at the root expression (ϵ_0) of the validation tree (Line 7). If the expression `e` is a property call expression then the scope element it refers to will be added to the cause of scope elements (Line 8). Otherwise it will be checked if the expression's validation result is not equal to its expected result and if so, the expression will be added to the cause of expressions and the `CalculateCause(Expression e)` algorithm will be recursively called for each argument of the expression `e` (Line 10).

ϵ_0 in our example is an universal quantifier and if the validated result is not equal to the expected result the scope element(s) from its source and all scope elements from the validation of the quantifiers condition must be included in the cause of scope elements ($\zeta_{e.p}$). Furthermore, the expression itself must be included in the cause of expressions (ζ_ϵ).

$$
\begin{aligned}
\zeta_{e.p}(\gamma_1(wait)) &:= \{self \mapsto \text{‘}wait\text{‘}.receiveEvent, \\
&\quad self \mapsto \text{‘}wait\text{‘}.receiveEvent.covered\} \\
&\quad \cup \zeta_{e.p}(\epsilon_1) \\
\zeta_\epsilon(\gamma_1(wait)) &:= \{\epsilon_0\} \cup \zeta_\epsilon(\epsilon_1)
\end{aligned}
$$

The next step is the cause calculation for the argument of the universal quantifier, the existential quantifier expression ϵ_1. The validated and expected result of that expression differ, hence the property calls from the quantifier's source must be added to the cause of scope elements and the scope elements from all the arguments. The expression ϵ_1 will be added to the cause of expressions too.

Algorithm 4 Calculating the Cause of an Inconsistency

```
1  CalculateCause(ValidationTree t)
2      if t->inconsistent then
3          CalculateCause(RootExpression of t)
4      endif
5  end /*CalculateCause(ValidationTree)*/
6
7  CalculateCause(Expression e)
8      if e isPropertyCallExpression then
9          addToCauseOfScopeElements(e.p)
10     elseif e.validationResult <> e.expectedResult then
11         addToCauseOfExpression(e)
12         forall Expression a in e.arguments
13             CalculateCause(a)
14         endforall
15     endif
16 \end /*CalculateCause(Expression)*/
```

$$
\begin{aligned}
\zeta_{e.p}(\gamma_1(wait)) \quad := \quad & \{self \mapsto \text{`}wait\text{`}.receiveEvent, \\
& self \mapsto \text{`}wait\text{`}.receiveEvent.covered \\
& l \mapsto \text{`}s\text{`}.represents, \\
& l \mapsto \text{`}s\text{`}.represents.type, \\
& l \mapsto \text{`}s\text{`}.represents.type.ownedOperation\} \\
& \cup \zeta_{e.p}(\epsilon_{2-1}) \cup \zeta_{e.p}(\epsilon_{2-2}) \cup \zeta_{e.p}(\epsilon_{2-3}) \\
\zeta_{\epsilon}(\gamma_1(wait)) \quad := \quad & \{\epsilon_0, \epsilon_1\} \cup \zeta_{\epsilon}(\epsilon_{2-1}) \cup \zeta_{\epsilon}(\epsilon_{2-2}) \cup \zeta_{\epsilon}(\epsilon_{2-3})
\end{aligned}
$$

Next we calculate the cause for the arguments of the existential quantifier expression — the three equality relations ϵ_{2-1} to ϵ_{2-3}. All three of them are violated (validated and expected result are not equal), hence all the scope elements underneath these expressions must be added to the cause of scope elements as well as all the three equality relation expression must be added to the cause of expressions. As there are no more children left in the validation tree, the cause calculation has been finished and the final cause for the inconsistency is as follows:

$$
\begin{aligned}
\zeta_{e.p}(\gamma_1(wait)) \quad := \quad & \{self \mapsto \text{`}wait\text{`}.receiveEvent, \\
& self \mapsto \text{`}wait\text{`}.receiveEvent.covered \\
& l \mapsto \text{`}s\text{`}.represents, \\
& l \mapsto \text{`}s\text{`}.represents.type, \\
& l \mapsto \text{`}s\text{`}.represents.type.ownedOperation, \\
& self \mapsto \text{`}wait\text{`}.name,
\end{aligned}
$$

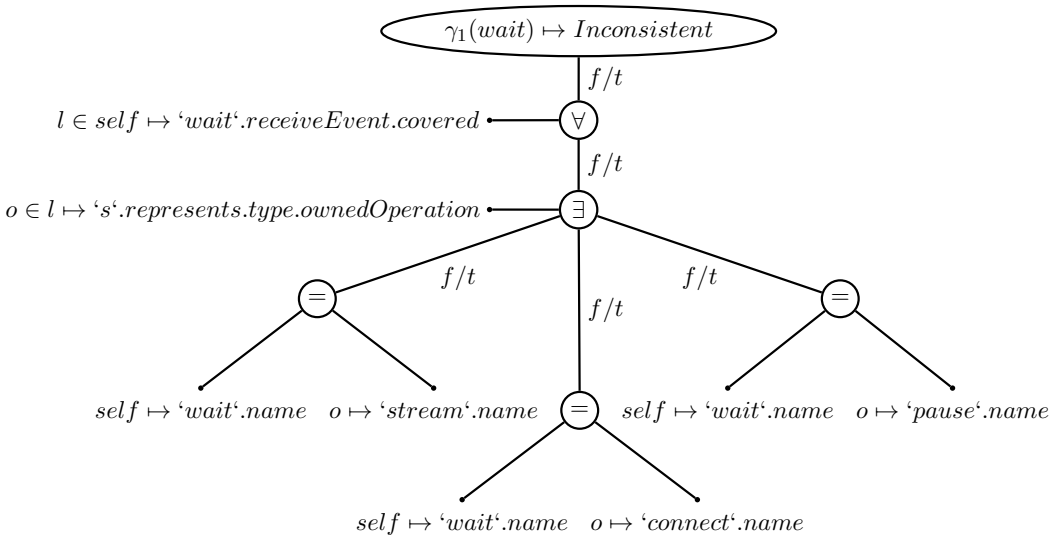


FIGURE 5.11: Validation Tree for $\gamma_1(wait)$ with Expected and Validated Results

$$
\begin{aligned}
& o \mapsto \text{‘}stream\text{‘}.name, \\
& o \mapsto \text{‘}connect\text{‘}.name, \\
& o \mapsto \text{‘}pause\text{‘}.name\} \\
& \cup \zeta_{e.p}(\epsilon_{2-1}) \cup \zeta_{e.p}(\epsilon_{2-2}) \cup \zeta_{e.p}(\epsilon_{2-3}) \\
\zeta_\epsilon(\gamma_1(wait)) \quad := \quad & \{\epsilon_0, \epsilon_1, \epsilon_{2-1}, \epsilon_{2-2}, \epsilon_{2-3}\}
\end{aligned}
$$

The cause of scope elements in this example is equal to the scope of the constraint validation but this is not necessarily true for each inconsistency. For that purpose we take a look at the inconsistency caused by Constraint 2 on the *Attribute* `visible` in the *Class* `Display`.

The validation tree shown in Figure 5.12 is a slightly simplified version of the complete validation tree of Constraint 2 on the *Class* `Display`. The simplification affects the `allParents` property. This property is a recursive call of the properties `generalization` and `general` on the classes and super classes and due to the increasing complexity this recursive properties calls are abstracted in the `allParents` call.

The dashed lines show the parts of the validation tree that can be dismissed due to the scope reduction. However, based on the scope reduction one of the two remaining branches that validate to ‘false‘ of the universal quantifier (that iterates over the parent class’ attributes) would be non deterministically removed from the validation tree (indicated as the dotted lines). To get the complete cause of that inconsistency, it is necessary to consider both branches of that existential quantifier that validate to ‘false‘ because to repair only one of them is not enough to entirely resolve that inconsistency (this is shown in Table 5.2). In this example we see that the scope and the cause are not necessarily the same for all validations.

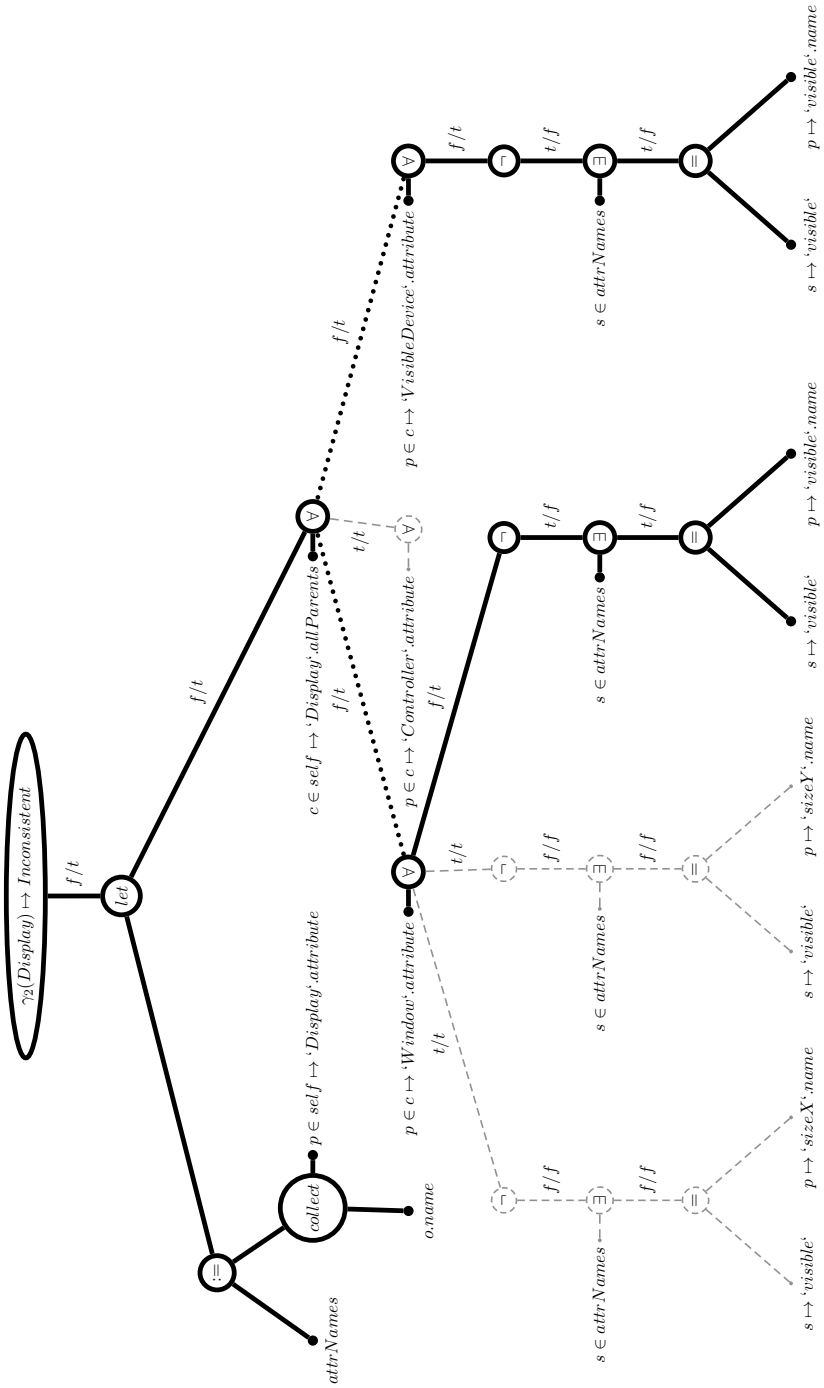


FIGURE 5.12: Validation Tree for $\gamma_2(Display)$

The thick lines in the validation tree now show the cause of that inconsistency (the dotted lines included). As can be seen, all branches of the validation tree, where the validated and expected result are not equal, are used to determine the scope elements that cause the inconsistency and all the expressions that are involved in the inconsistency. The *let* expression on top of the validation tree (the root expression) has only one argument that validates to a Boolean result. However, if this validation result is not equal to the expected result then all the scope elements must be included in the cause of scope elements in the case of an inconsistency. Therefore, both branches (arguments) must be used to calculate the cause.

The cause of an inconsistency serves primarily as information to the designer to visualize a) the model element properties that are involved in an inconsistency, and b) the expressions of a constraint validation that are violated. This information can help the designer to identify errors in the model and in the constraint definition itself. Furthermore, the validation tree representing the cause, i. e., the part of the validation tree then is the cause and will be used to generate the repairs discussed in the next section.

5.5 Stage 4: Repairs and Side Effects

The cause of an inconsistency consists of only the model element properties (scope elements) that are involved in the inconsistency and changing these properties can resolve an inconsistency. However, nothing is known about how many of these properties must be changed and how they must be changed — the only thing we know until now is that at least one of the model element properties must be changed to resolve the inconsistency. Table 5.3 summarizes which alternatives to resolve an inconsistency exist and how the repairs must be combined for different Boolean expressions. The basic structure for generating the repair alternatives is the validation tree and the outcome is a *repair tree*.

5.5.1 Repair Tree

To illustrate how a repair tree is generated out of the validation tree we use the inconsistent validation tree illustrated in Figure 5.11 (published in [90]). The process for generating the repair tree starts at the root expression of the validation tree, the universal quantifier.

Step 1: An inconsistency that is caused by an universal quantifier can be resolved by removing the elements (in our example only the *Lifeline* `s` is in the source) from the source or repairing the failed validations of the quantifier's condition. Figure 5.13 shows

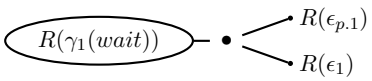


Figure 5.13: Repair Tree Generation for $\gamma_1(wait)$ — Step 1

a	b	A	R					
			$a \wedge b$	$a \vee b$	$a \Rightarrow b$	$a = b$	$\forall x \in A : x$	$\exists x \in A : x$
f	f	$\{a, b\}$	$R(a)$ $+$ $R(b)$	$R(a)$ $\bullet$ $R(b)$			$\langle -, A, a \rangle$ $+$ $\langle -, A, b \rangle$ $\bullet$ $R(a) + R(b)$	$\langle +, A, ? \mapsto \text{`}t\text{`} \rangle$ $\bullet$ $R(a)$ $\bullet$ $R(b)$
t	f	$\{a, b\}$	$R(b)$		$R(a)$ $\bullet$ $R(b)$	$\langle \times, a, \text{`}f\text{`} \rangle$ $\bullet$ $\langle \times, b, \text{`}t\text{`} \rangle$	$\langle -, A, b \rangle$ $\bullet$ $R(b)$	
f	t	$\{a, b\}$	$R(a)$			$\langle \times, a, \text{`}t\text{`} \rangle$ $\bullet$ $\langle \times, b, \text{`}f\text{`} \rangle$	$\langle -, A, a \rangle$ $\bullet$ $R(a)$	
		$\emptyset$						$\langle +, A, ? \mapsto \text{`}t\text{`} \rangle$

TABLE 5.3: Repairing Inconsistencies

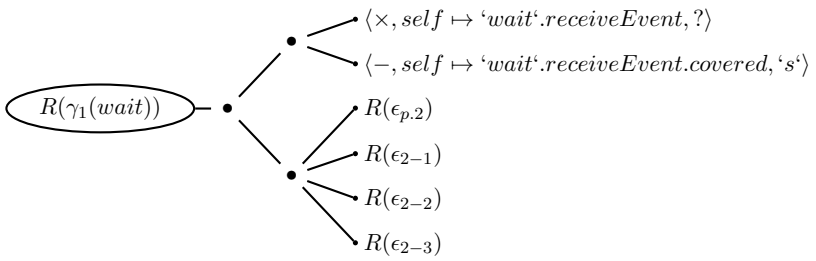


FIGURE 5.14: Repair Tree Generation for $\gamma_1(wait)$ — Step 2

the repair tree for this first step in the repair tree generation process. The '•' denotes the alternative, the upper branch is the removal of the element from the source ($\epsilon_{p.1}$) and the lower branch is representative for the failed validation of the quantifier's condition (ϵ_1).

Step 2: The repair actions for $\epsilon_{p.1}$ and ϵ_1 are created, shown in Figure 5.14. $\epsilon_{p.1}$ is a chain of property calls (cf. Section 4.3) where for the last property call in the chain ($self \mapsto \text{`}wait\text{`}.represents.covered$) a repair action, for the source of the universal quantifier, is created and in addition an abstract modify repair action for the first property call in the chain ($self \mapsto \text{`}wait\text{`}.represents$). The repair for ϵ_1 is replaced by the repairs for the existential quantifier, the alternatives to repair the source ($\epsilon_{p.2}$) or repairing the failed condition validations (ϵ_{2-1} to ϵ_{2-3}).

Step 3: The repair actions for $\epsilon_{p.2}$ and ϵ_{2-1} to ϵ_{2-3} are generated (shown in Figure 5.15). The property call expression for the source of the existential quantifier consists of a chain of three property calls. In contrast to the universal quantifier no concrete value could be generated for the element that must be added to the source. But considering

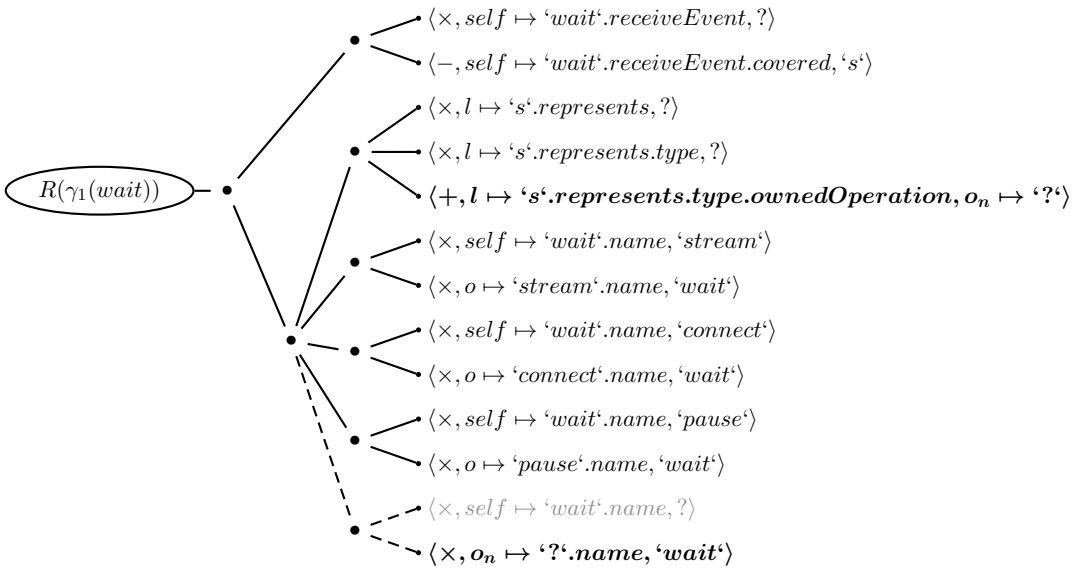


FIGURE 5.15: Repair Tree Generation for $\gamma_1(wait)$ — Step 3

the expected type for the source (is known from the property call expression) and the build up of the condition, a concrete value can be generated by simply adding an element of the expected type (an *Operation* in our case) and validating this element on the quantifiers condition. This additional validation is shown in Figure 5.15 as the branch drawn as dashed line. The initial value for the added element property is random (or not given as in the example), hence, the condition will be probably violated (it is not very likely that the correct name can be guessed, but if so, this value can be taken). If it is violated, repairs can be generated in the same way as for the normal validations of the condition and from this repairs a concrete value can be determined. This we can see in the three repair alternatives generated for the condition validations of the existential quantifier. Each alternative consists of either renaming the *Message* name to the *Operation* name or renaming the *Operation* name to the name of the *Message*. In the case of missing elements from the existential quantifier, a repair must be taken that modifies the randomly generated element, the *Operation*.

Step 4: The final step in the repair tree generation process is the inserting of the values that are determined from temporary repairs (the one for the element that must be added), the removal of the temporary repairs and a flattening of the repair tree (the cascading alternatives and/or combinations) to reduce the depth. Figure 5.16 shows the complete repair tree for the inconsistency detected in $\gamma_1(wait)$. This repair tree consists of eleven alternatives to resolve the inconsistency, one repair that removes a model element, one that adds a model element and nine that modify a model element property. From the eleven modify repairs, three are abstract (grayed in the repair tree), i. e., they cannot be executed directly in the model.

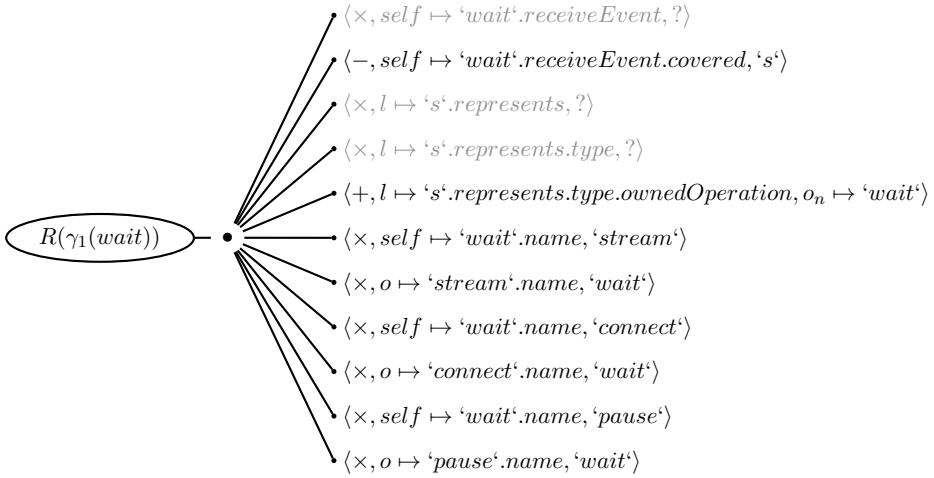


FIGURE 5.16: Complete Repair Tree for $\gamma_1(wait)$

The repair tree for the inconsistency detected in $\gamma_1(wait)$ consists of alternatives only. A more complex repair tree is shown for the inconsistency detected in $\gamma_2(Display)$ in Figure 5.17. The repair tree for this inconsistency consisting of alternatives, combinations and conditional concrete repair actions.

The repair tree starts with two alternatives. These alternatives come from the universal quantifier. The first alternative is to remove both (the combination) *Class*es from the `allParents` property of the context *Class* `Display`. The second alternative is a combination also, but with more alternatives to combine the second alternative. The alternatives of the first branch of the combinations contains four alternatives, the removal of the `visible` *Attribute* from the *Class* `Window`, the removal of the `visible` *Attribute* from the *Class* `Display` (indirectly derived from the `let` expression via the variable `attrNames`), the modification of the name property of the `visible` *Attribute* from the *Class* `Display`, and the modification of the name property of the `visible` *Attribute* from the *Class* `Window`. The two modify repair actions (gray) are conditional concrete repair actions, where only the name that is not allowed for the attribute names is known. The second branch of the combination is nearly the same as the first branch except that the alternatives are for the *Class* `VisibleDevice` instead of the *Class* `Window`.

The repair tree in Figure 5.17 consists of alternative and combined repair actions, i. e., the alternative repairs, that can be applied in the model, consist of combinations of repair actions. To get a flatten tree that consists of alternatives only, the combination must be applied on its children (similar to multiplying in mathematical arithmetic). Figure 5.18 shows a simple example how a combination of two branches of alternative repair actions can be combined.

The repair tree consists of a combination where the two branches that must be combined containing two alternatives. So that the final repair tree contains alternatives on the top node only, all the possible permutation must be generated, i. e., each alterna-

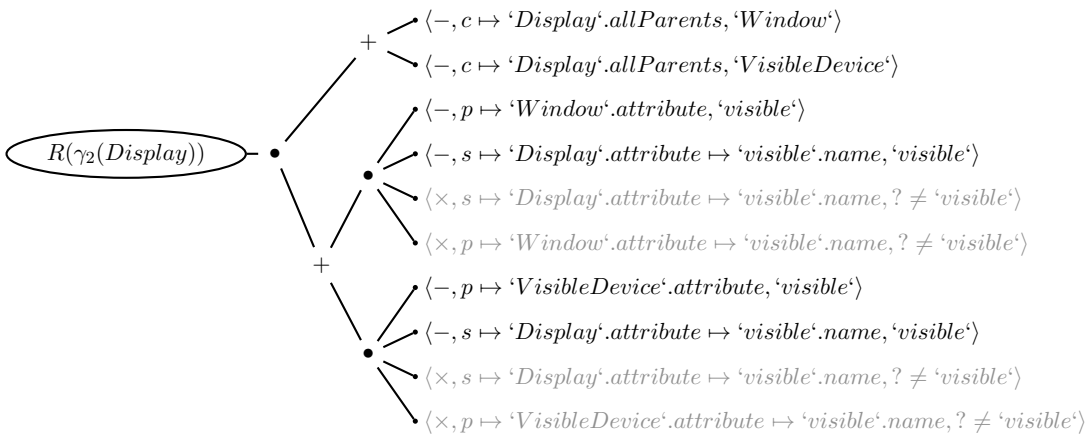


FIGURE 5.17: Repair Tree for $\gamma_2(Display)$

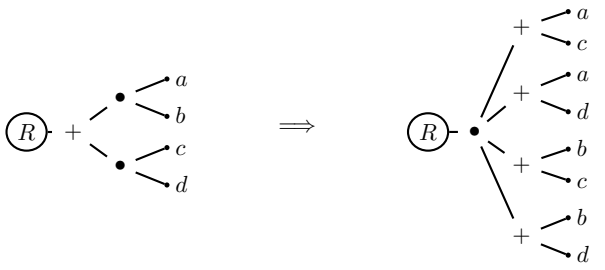


FIGURE 5.18: Combining Alternative Repair Action

tive from the first combination branch must be combined with the alternatives from the second combination branch. The result is a repair tree that has only one alternative node, but as alternatives a combination of single repair actions than can be applied in the model.

Figure 5.19 shows an excerpt of the flattened repair tree for the inconsistency detected by $\gamma_2(Display)$, due the large number of 17 alternatives. Some of the alternatives can be filtered out, if two combined actions change the same scope element (the one with the dashed lines between the '. . . ', they would both change the same scope element, actually in the same way). The flattened repair tree shows one of the major problems resolving inconsistencies — the exponential explosion of possible repair alternatives. Therefore, the flattened repair tree is used for the generation of applicable repairs only. But for presenting the possible solutions to the designer, the preferable style is the original repair tree (Figure 5.17) as it is more clearly arranged.

Using the knowledge of the used design and modeling language can increase the number of solutions how to resolve an inconsistency. Unfortunately, this would limit the general applicability of that approach and therefore it is not intended to provide such a mechanism in the general approach (will be left open for specific implementations).

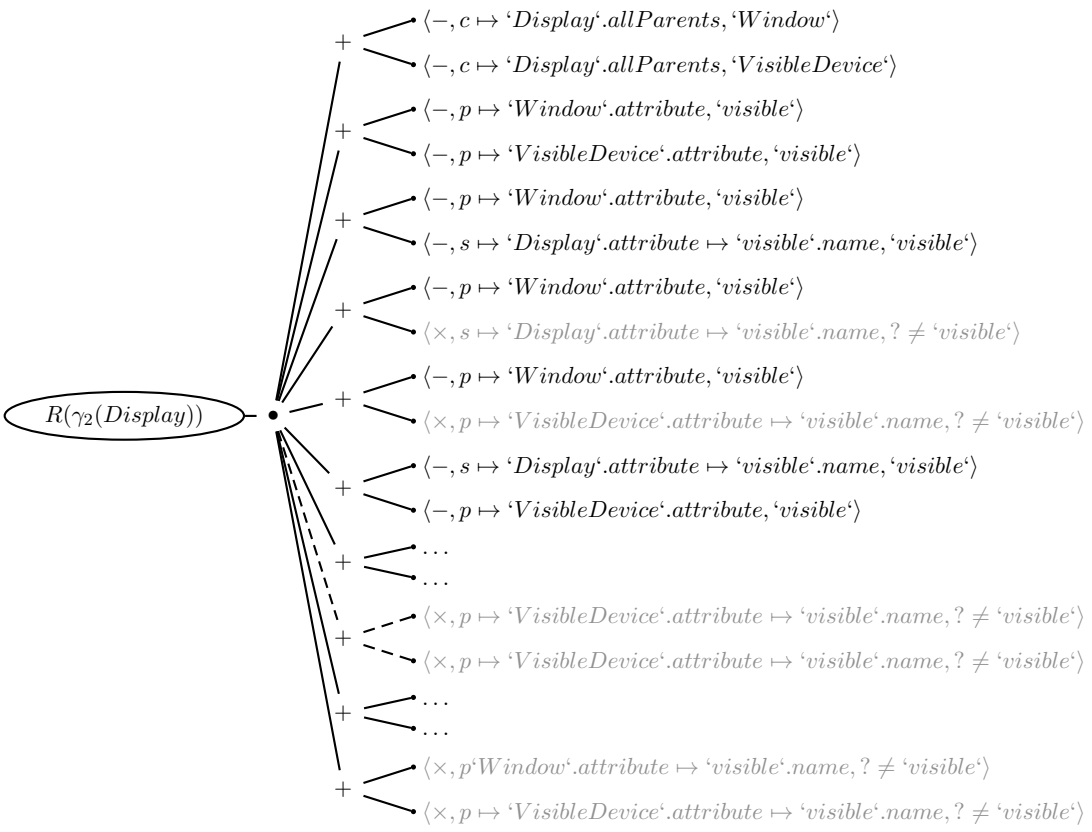


FIGURE 5.19: Flattened Repair Tree for $\gamma_2(Display)$

5.5.2 SIDE EFFECTS

Repairs, or more explicitly, the repair actions of a repair modify properties of model elements — the scope elements. As already seen before, a scope element can be accessed more than once during a constraint validation or in other constraint validations than the one that is violated. From that follows that a repair action (or combinations of repair actions) might have other effects than the resolution of an inconsistency — the side effects. Furthermore, side effects are also used to determine contradicting or missing values for repair actions.

In Figure 5.20 an abstraction of the scopes of the constraint validations is shown. Some of the scope elements overlap, i. e., they are accessed by more than one constraint validation. In this figure three validations are shown (val_1 - val_3) and each validation accesses properties (p_1 - p_{12}) of model elements. Each of the properties provides a value ($p_x \mapsto v$). The overlapping scope elements cause side effects if they are changed.

To illustrate how side effects are determined we use the validation trees for $\gamma_1(wait)$ and $\gamma_1(connect)$ as well as $\gamma_3(wait)$. First, we consider a side effect caused by repairing $\gamma_1(wait)$. One suggestion to resolve this inconsistency, proposed in the repair tree in

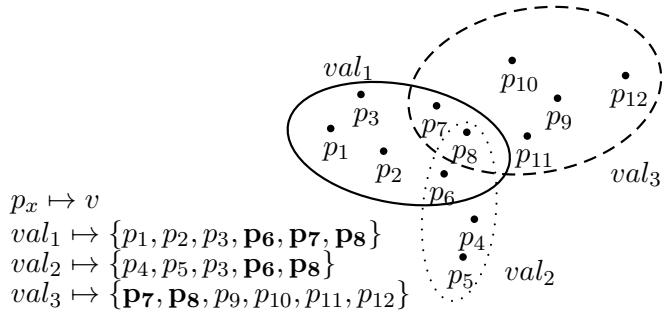


FIGURE 5.20: Overlaps that Cause Side Effects in Constraint Validations

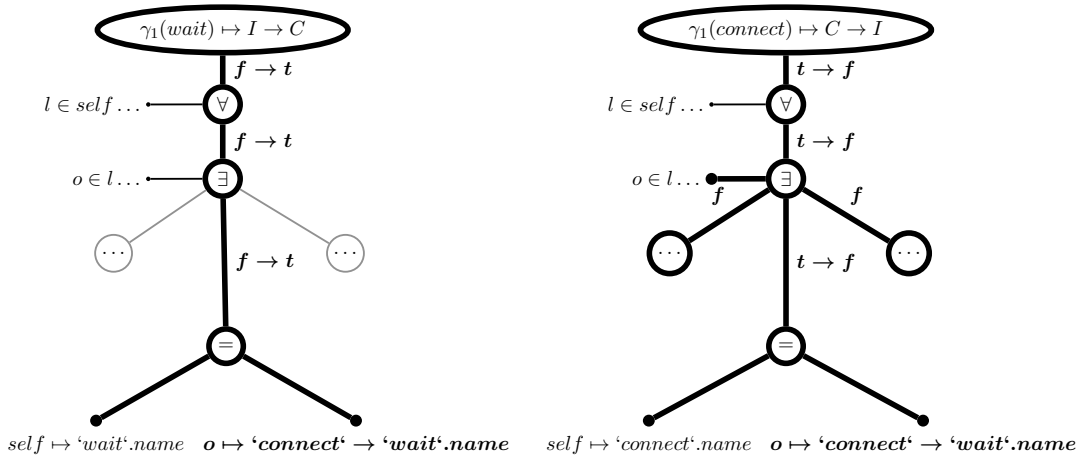


FIGURE 5.21: Negative Side Effect in Validation Tree $\gamma_1(connect)$

Figure 5.16, would be the renaming of *Operation* `connect` to the name 'wait' ($r_1 := \langle \times, o \mapsto$ '$connect$'.$name$, '$wait$'$\rangle$).

Figure 5.21 shows the two validation trees for $\gamma_1(wait)$ and $\gamma_1(connect)$ with the effects caused by the proposed repair. Only an excerpt of the complete validation trees that are affected by the applied repair are shown. The thick lines show the parts that need re-validation due to the repair. As can be seen, the inconsistency in the left tree ($\gamma_1(wait)$) will be resolved as it is mentioned in the repair ($I \to C$, $I \ldots$ Inconsistent, $C \ldots$ Consistent). But the same modification causes a new inconsistency ($C \to I$) in the right validation tree ($\gamma_1(connect)$) because no *Operation* named `connect` exists anymore in the *Class* `VideoServer`. In this case we talk about a *negative side effect on* $\gamma_1(connect)$.

$$S(r_1(\gamma_1(wait))) \quad := \quad \langle \gamma_1(connect), n \rangle$$

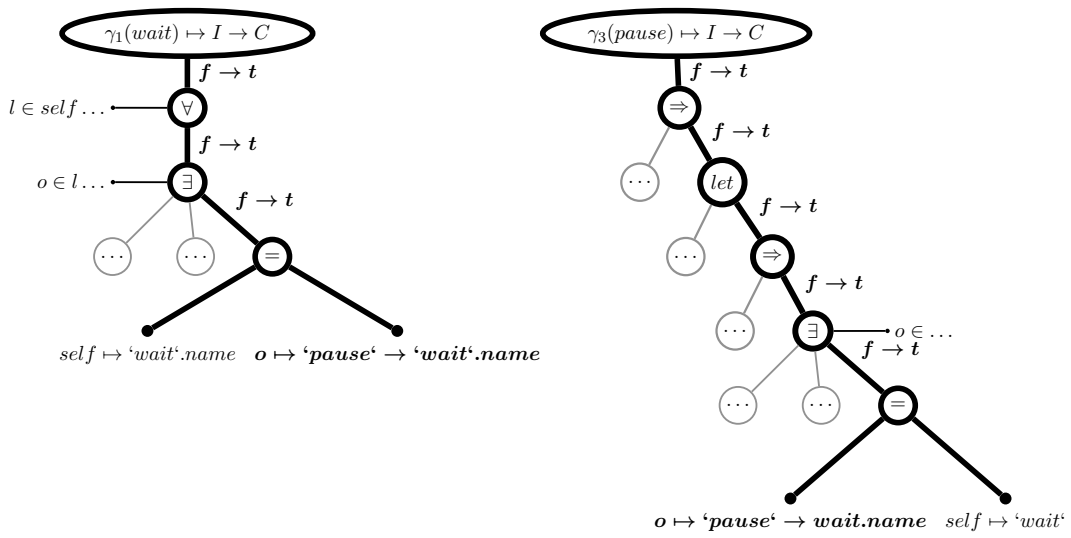


FIGURE 5.22: Positive Side Effect in Validation Tree $\gamma_1(pause)$

As the Boolean parts of the validation trees are kept for fast re-validation, the side effects can be calculated without applying the repair in the model, but instead by *simulating* it in the validation tree. If a result of, for example, a model element property is needed (via a property call expression) and the model element property is not affected by a repair to test, the value for the needed property will be determined from the model. But these model element property accesses are strictly read-only for the determination of the side effects. Thus, the side effects can be calculated very fast. A fast calculation of the side effects is very important because there exists in average eight possible repairs per inconsistency ([40]) that must be tested — an exponential growth regarding to model size.

The side effects of repair actions are not necessarily negative. Instead of applying the repair containing r_1 on to the model we can apply the repair that contains $r_2 := \langle \times o \mapsto$ '$pause$'$.name$, '$wait$'$\rangle$ on to the model. In this case there is no side effect on the validation of the constraint γ_1 (except the inconsistent one that will be repaired using this repair). But there is a side effect on the validation of $\gamma_3(pause)$, a constraint validation that is also violated by the model (for the *Transition* `wait` no *Operation* named 'wait' exists). Figure 5.22 shows the excerpts of the validation trees of the two constraint validations. Again, only parts are shown that are affected by the repair action r_2. As can be seen, both inconsistencies can be resolved by applying the repair consisting of r_2. In this case we talk about a *positive side effect on* $\gamma_3(pause)$.

$$S(r_2(\gamma_1(wait))) \quad := \quad \langle \gamma_3(pause), p \rangle$$

Additional to detect effects of a repair on other constraint validations, the side effects are also used to determine values for repair actions where no direct value can be

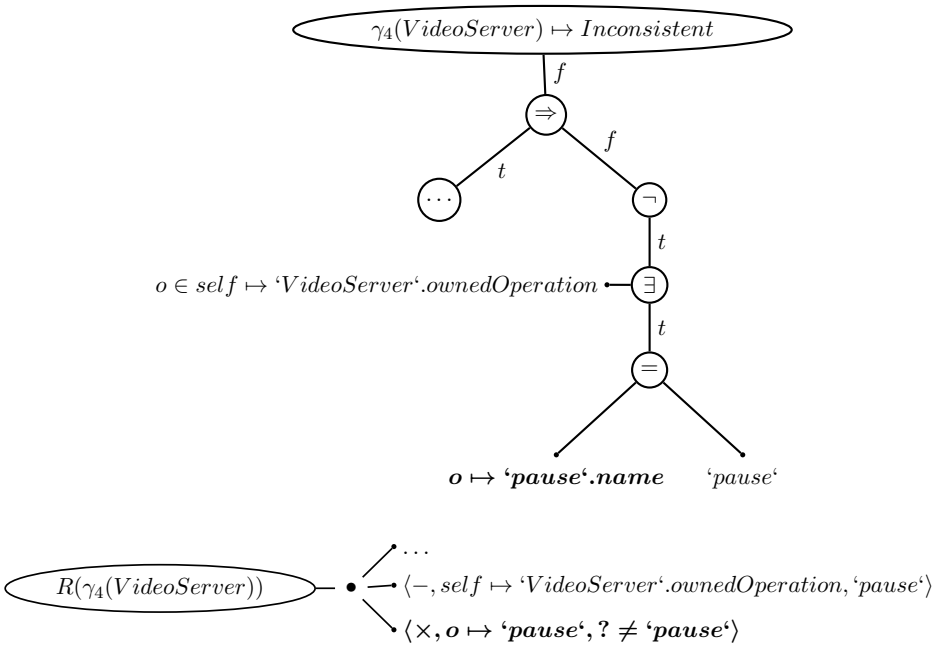


FIGURE 5.23: Validation- and Repair Tree for $\gamma_4(VideoServer)$

determined, i. e., for all those repair actions that are abstract or conditional concrete after the repair tree generation process. For illustration we introduce a constraint that might come from customer requirements that says that there must not be an *Operation* `pause` in the *Class* `VideoServer`.

Constraint 4 There must not be *Operation* `pause` in the *Class* `VideoServer`

```
17 context Class inv:
18 self.name='VideoServer' implies
19     not self.ownedOperation->exists(o:Operation|
20         o.name='pause')
```

In Constraint 4 the customer requirement is expressed in OCL to be applicable in the model. This constraint is validated on all *Class*es, but, due to the implication, only for the *Class* `VideoServer` it is validated, if it does not contain an *Operation* named 'pause'. As there exists an *Operation* named 'pause' in the *Class* `VideoServer`, the validation on this *Class* fails, i. e., an inconsistency has been detected. In Figure 5.23 the validation- (top) and repair (bottom) for the failed constraint validation and its resolution is shown. Only the excerpts of the trees are shown that are relevant to illustrate how a missing value for a repair action can be determined.

Two possibilities to resolve the inconsistency are shown: one is to remove the *Operation* named 'pause' from the *Class* `VideoServer`, and the second one is to rename

the *Operation* `pause` to something different — a conditional concrete repair action ($r_3 := \langle \times, o \mapsto \text{`}pause\text{`}, ? \neq \text{`}pause\text{`} \rangle$). Please note that in this example only one modify action will be generated because the second argument of the equality relation is a constant that contributes to the cause of expressions only, i. e., no scope element that can be modified is provided by this expression.

Theoretically an infinite number of solutions exist to resolve the inconsistency using the conditional concrete repair action because any text combination except 'pause' is sufficient to resolve this inconsistency. But when we generate the side effects of that repair action, we get the following list (for simplicity, only already discussed validations are considered, i. e., $\gamma_1(pause)$ and $\gamma_3(pause)$):

$$\begin{aligned} S_1(r_3(\gamma_4(VideoServer))) &:= \langle \gamma_1(pause), u \rangle \\ S_2(r_3(\gamma_4(VideoServer))) &:= \langle \gamma_3(pause), u \rangle \end{aligned}$$

The two side effects generated for the repair action $r_3(\gamma_4(VideoServer))$ are unknown because no concrete value exists, that can be tested on the affected constraint validations. But the two affected constraint validations detect other inconsistencies that must be resolved, i. e., for these inconsistencies repairs exist. In contrast to $r_3(\gamma_4(VideoServer))$, the repair actions from the other two inconsistencies provide a single value for that particular scope element, the `name` property from the *Operation* named 'pause' from *Class* `VideoServer`. Hence, the value provided by these repairs ('wait') can be taken for $r_3(\gamma_4(VideoServer))$ as one possible solution for this inconsistency. To ensure, whether the provided value is able to resolve the inconsistency, the side effects are generated for this repair. Therefore, one repair can be determined that is able to resolve three inconsistencies.

$$r_3(\gamma_4(VideoServer)) := \langle \times, o \mapsto \text{`}pause\text{`}, \text{`}wait\text{`} \rangle$$

An abstract or conditional concrete repair action might not only have side effects on other constraint validations that cause inconsistencies, but also on constraint validations that are not violated by the model. In such cases a possible contradiction in the constraint set is detected, as there is a value for a model element property that satisfies one constraint but violates another one. To indicate contradictions in the set of constraints the cause of expressions is used, i. e., if a possible contradiction has been detected, the solution for the inconsistencies might be in the constraints and not in the model alone. This is the case if Line 20 of Constraint 4 would be `o.name='wait'`. 'wait' is the only value for the repair actions that modify the *Operation* name but this is not allowed because Constraint 4 (there must not be *Operation* `pause` in the *Class* `VideoServer`) would be violated. As a consequence the modifications of the *Operation* names can be excluded from the list of alternative repairs. Fortunately, other repairs exists that resolve the inconsistency but if not, changing one of the constraints would be the only solution.

5.6 Summary

In this chapter the CIM approach was introduced. It is based on the principles introduced in the last chapter. The CIM approach provides the functionality to 1) define arbitrary constraints, 2) detect violations (inconsistencies) in the set of constraint, 3) generates repair alternatives based on constraint validations only (independent of the used design and modeling language), and provides information about the consequences (side effects) of the proposed repairs on other constraint validations. Furthermore, the CIM approach supports the designer in locating inconsistencies in the model using the cause of scope elements as well as in the set of constraints using the cause of expressions.

Chapter 6

Tool Implementation

"Things do not happen. Things are made to happen."
John F. Kennedy, American President, 1917-1963

To evaluate the working of the CiM approach a prototype tool was implemented. An early version of the prototype that already provides the main functionality was presented in [89]. The tool is implemented as an eclipse based plug-in for the IBM Rational Software Modeler using UML as modeling language and OCL as constraint language and can be downloaded from the institutes web site `http://www.sea.jku.at/tools`.

6.1 IBM Rational Software Architect Integration

The CiM approach implementation is set of plug-ins for the eclipse based modeling tool IBM Rational Software Architect (RSA). It provides additional views and editors to define and visualize the constraints as well as the detailed representation in the form of the validation tree. Figure 6.1 gives an overview of a typical configuration of a RSA workbench.

The top left window shows the diagrams of the software model. In this case the model containing the class, sequence and state machine diagram from our example is shown. Top right all the constraint definitions and their validations are shown. In the tool, red flags indicate an inconsistency and the green flags consistent validations. The lower two windows are views that show the validation tree for Constraint 2 validated for the *Class* `Display` (left) and the repair tree for this inconsistency on the right. Furthermore, the tool provides views to visualize the syntactical structure of a constraint (similar to the validation tree, but with validation results), and to visualize the dependencies of a scope element, i. e., what other constraint validations access a scope element. The arrangement and size of the views can be customized to the designers needs — as in any eclipse based application.

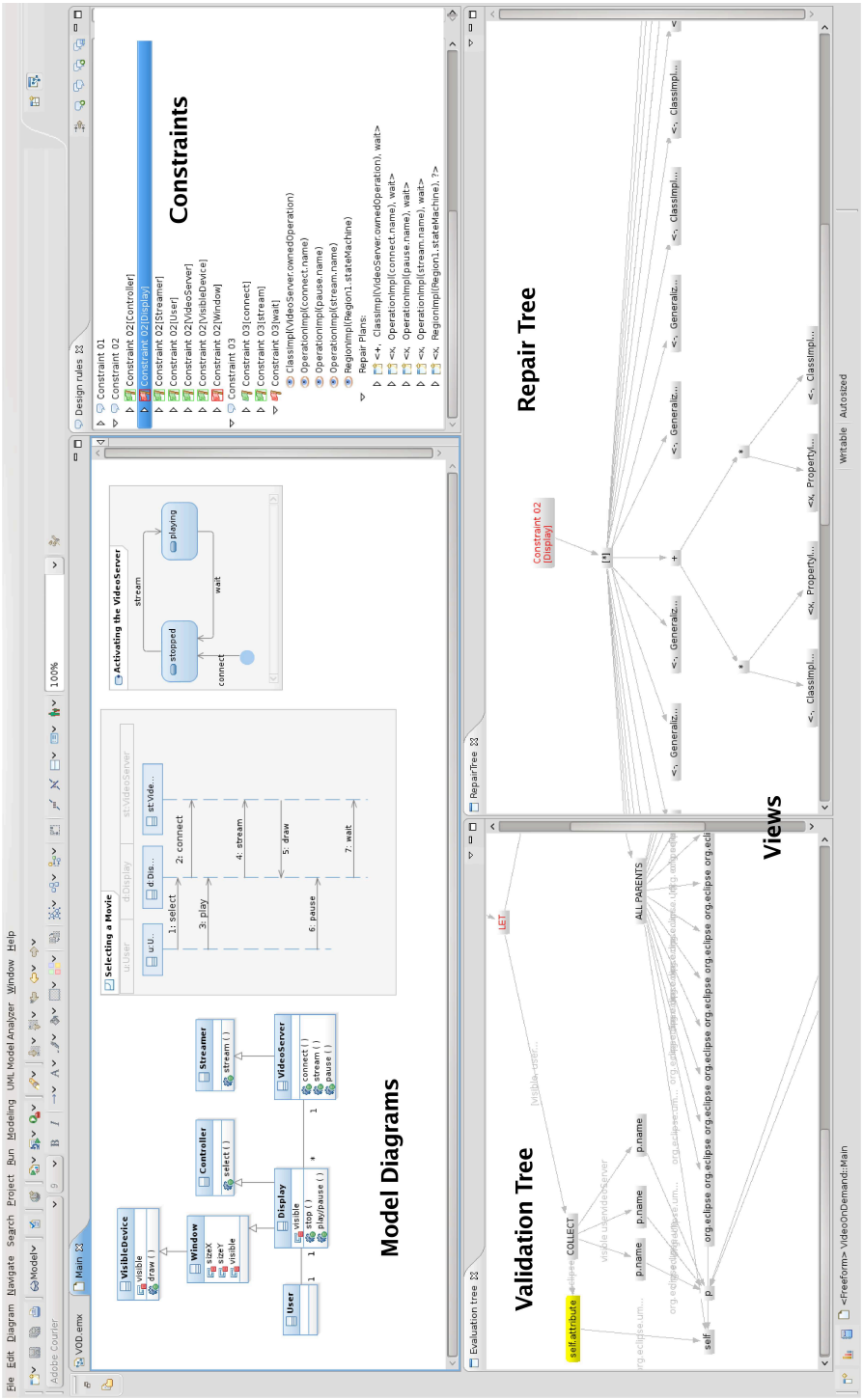


FIGURE 6.1: Overview of the CIM Approach Implementation for the IBM Rational Software Architect

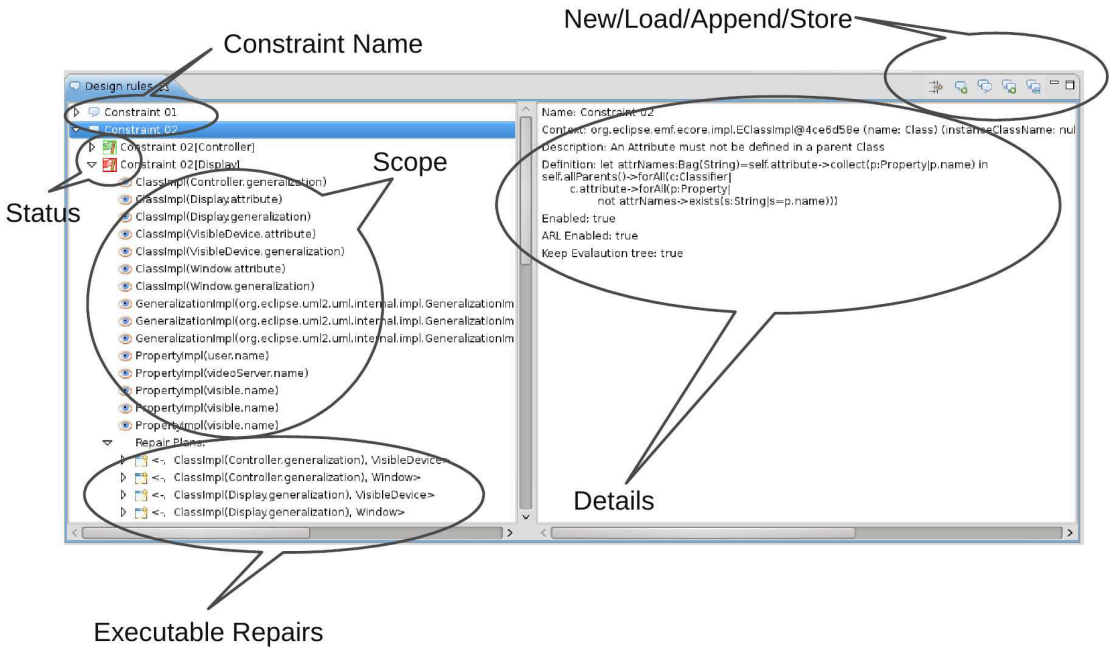


FIGURE 6.2: Constraint View

6.2 CONSTRAINTS

The view for the constraints plays an essential role in the tool because in this view constraints can be added, deleted and modified, as well as the graphical visualizations (syntax-, validation, and repair trees) can be selected. For that purpose a separate view exists (Figure 6.2). This view is split into two parts: the left side lists all the constraints defined in the model and the right side shows the details for the item selected on the left side. Furthermore, the right side is organized tree based, i. e., below the constraint definition the instances (the validation) of those constraints can be found and below one validation the scope, as well as repairs with their side effects, in case of an inconsistent validation, is shown.

When a constraint is selected, on the right side the details for the constraint definition are show. The constraint definition consists of a name (e. g., Constraint 01), a brief description, a formal definition in OCL, as well as some control flags, for example, to enable a constraint (if not enabled, the constraint will not be validated on the model) and other flags that are mainly for debugging and testing purposes.

Below the constraint definition all the validations for the constraint are listed. The validation result of the constraint validation is indicated as a flag — green for consistent and red for inconsistent. Furthermore, a border around the flag indicates if the constraint validation was affected by a model change, i. e., if the model is changed the flags (green and red) of the constraint validations get a border around, if they were re-validated,

independent whether they change their result or not. The flag follows the name of the constraint and the context element (in square brackets) on which the constraint has been validated. The details of a constraint validation show the context element and the validation result.

During a constraint validation model element properties are accessed — scope elements. These scope elements are indicated by an eye symbol and are shown below a constraint validation. Next to the symbol, the type of the model element is shown, followed (in brackets) by the instance name of the model element and the property (separated by a dot). The details of a scope element show the concrete model element instance, the property and the value for that property. The scope elements can be highlighted in the model using the context menu of the scope elements (more than one scope element can be selected). In the case of an inconsistency, the scope is equal to the cause of scope elements.

If a constraint validation fails (inconsistent), the executable repairs are shown below the constraint validation. For each alternative repair a separate item is shown and if an alternative consists of more than one repair action, the repair actions are concatenated by a comma. Below the repair alternatives the side effects of that repair are listed.

On top of the constraint view, behind the four icons the functions to add a constraint, to load constraints from a file (replaces the defined), to append constraints to the existing ones from a file or to store the list of constraint to a file provided. In the preferences of the tool, a file containing constraints can be defined that will be loaded on program start.

To add a constraint or to edit a constraint (context menu of the constraint definition) a separate editor exists. Figure 6.3 shows this editor window. First, a name and a description can be defined for a constraint. In the large text box in the center of the editor the formal definition of the constraint is given. This text box provides syntax highlighting and code completion for OCL. Below this text box, the context for the constraint is defined, i. e., for which type of model element the constraint must be validated (the context definition is slightly different as for standard OCL environments). The four check boxes below the context definition are use to control the validation of the constraint.

The first check box is used to enable the constraint, i. e., if it should be validated in the model. The second check box is used to enable the reasoning engine that provides the proposed functionality (ARL — Abstract Rule Language, the internally used representation of the constraint validations, i. e., validation tree,. . .). Otherwise the OCL environment from eclipse is used. In the case ARL is not used, no repairs are enabled and the re-validation process is restricted. Disabling ARL is only used for evaluation purposes to ensure, if the self-made reasoner is working correctly. The third check box is used to keep the validation tree to achieve a better performance. Disabling this check box reduces the used memory, but the re-validation time will increase. The last check box indicates a repairable constraint, i. e., if repairs should be generated, if this constraint is violated. The deletion of a constraint from the set of constraint is done using the context menu of the constraint definition.

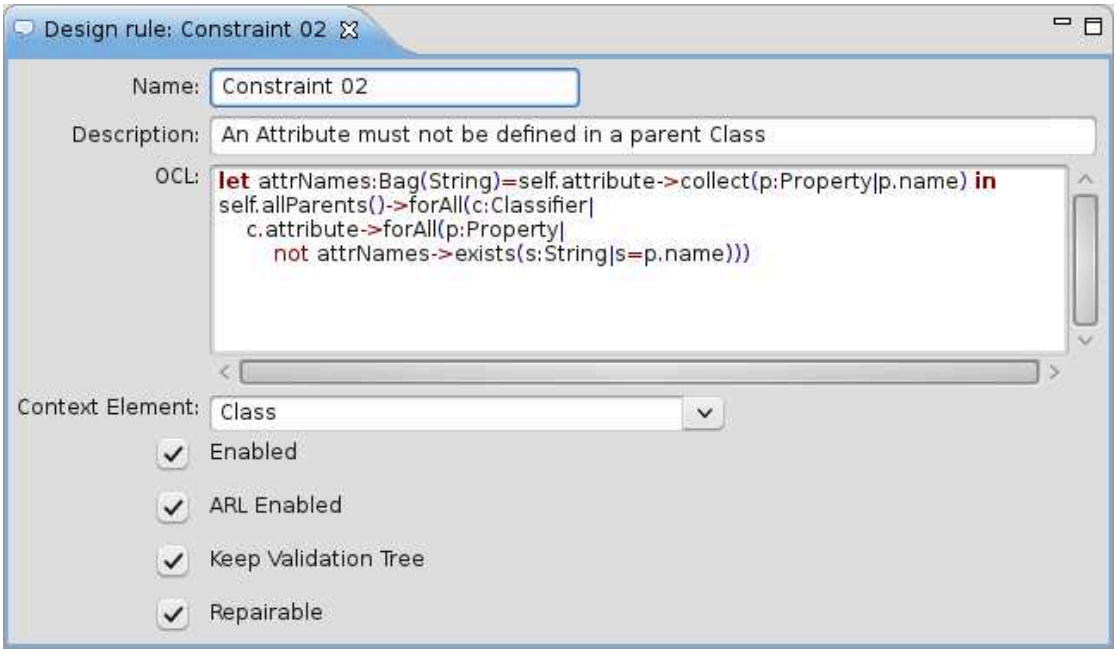


FIGURE 6.3: Constraint Editor

Additional to these combined view for constraint definitions and their validation outcomes, separate views exists that enable the visualization of all constraint validations (not grouped by the constraint definitions) and all scope elements collected from all constraint validations.

6.3 Graphical Visualization

As mentioned before, internally a specialized reasoner for OCL is implemented to provide the functionality of the CIM approach. This reasoner is based on the validation tree — the basic data structure for our approach. This data structure is represented in the Abstract Rule Language (ARL) that is based on first order predicate logic and can be extended with specialized functions provided by the used constraint language, like, for example, the `select` or `collect` functions from OCL. The designer is able to select the appropriate views out of the constraint and constraint validations.

In Figure 6.4 the validation tree for the inconsistency caused by $\gamma_2(Display)$ (Constraint 2 validated on the *Class* `Display`) is shown. The top node of the validation tree represent the constraint validation. The left branch points to the constraint definition and the lower branch to the context element. As can be seen, the structure of the real validation tree is slightly more complex than explained in the approach. This stems from two facts: Firstly, the `allParents` property is a recursive call of the properties `generalization` and `general` on the context class and all its super classes. Secondly,

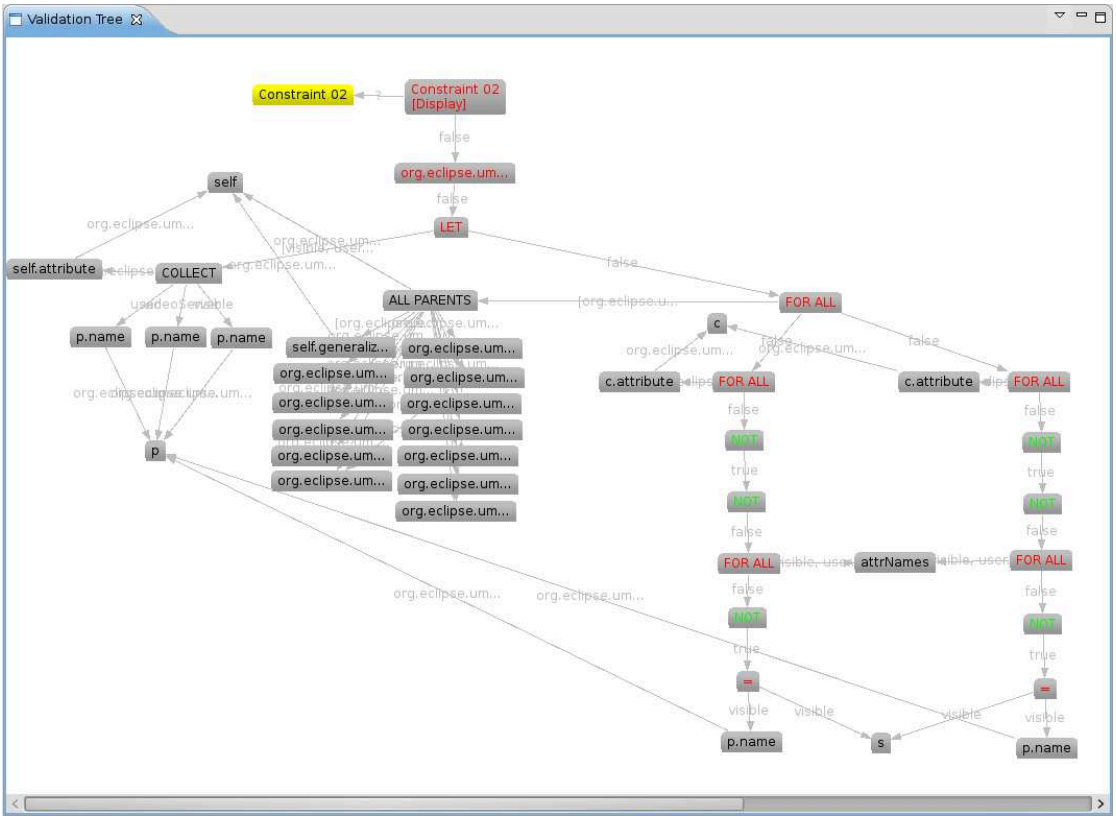


FIGURE 6.4: Validation Tree View

the set of Boolean operations is limited to a negation, conjunction and the universal quantifier and therefore all other Boolean operations are derived from these basic operations. This comes with the advantage that the generation of the scope, cause and the repairs is very similar for all the derived operations (explained in Chapter 4), hence the functionality has to be implemented only once for the Boolean operations — this eliminates code duplication and therefore reduces the potential of making errors. It enables also an easier porting on different application platforms.

The coloring of the nodes representing Boolean expressions indicates their validation result — red for 'false' and green for 'true'. The branches of the tree are attached with the validation result they propagate to the parent. Boolean results for Boolean expressions as well as text results and model elements returned from property calls. The Boolean nodes in the validation also reflect the cause of expressions in case of an inconsistent validation.

Aside the validation tree, the syntax tree reflects the syntactical structure of a constraint definition. The syntax tree can be selected from the context menu of the constraint definition. Furthermore, a view exists to visualize the scope tree to show the dependencies of the scope elements, i. e., where a scope element is accessed and causes a

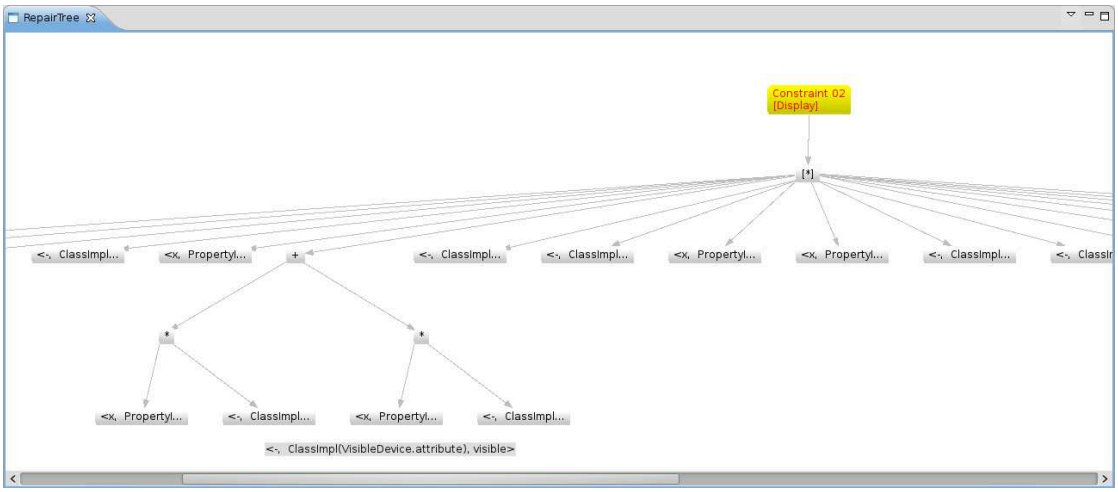


FIGURE 6.5: Repair Tree View

side effect. It shows the parts of the constraint validations where they are accessed and what re-validation a modification of the scope element might trigger. This view can be selected from the context menu of a validated constraint.

Another view is the repair tree that can be selected from the context menu of a constraint validation that indicates an inconsistency. Figure 6.5 shows an excerpt of the repair tree for the inconsistency caused by $\gamma_2(Display)$. The main reason why the repair tree consist of many more alternatives as explained in Chapter 5 is because of the `allParents` property. In the approach section it was assumed that this is a single property that provides all parents (super classes) of the context class. But in UML the `allParents` property is a recursive call of the `generalization` and `general` properties of the context class and all its parents (also expressed in the validation tree). Hence, there exist many more possibilities to resolve this inconsistency because not only the classes can be removed, but also each of the generalizations can be removed.

The top node of the repair tree represents the constraint validation it belongs to. Below that node the root node of the repair tree is shown (indicated using the square brackets around the node type). The alternatives are indicated using an asterisk (*) and the combinations are indicated using a plus (+). The notation of the repair actions is taken from the formal notations introduced in this thesis. Out of this repair tree, the proposed repairs in the constraint view are generated.

The views can be selected from the context menu of the corresponding element in the view for the constraints, i. e., in the context of the constraint the syntax tree can be found, in the context of the constraint validations the validation and, if applicable, the repair tree, and finally in the context of a scope element the scope tree.

6.4 Summary

In this chapter a brief overview about the prototype implementation for the CiM approach is given. It has been shown how the proposed functionality of the approach is realized and presented to the designer as well as how the designer can interact with the system regarding the definition of arbitrary constraints. This prototype is the basis for the evaluations shown in the next chapter.

Chapter 7

Evaluation and Discussion

> *"No amount of experimentation can ever prove me right; a single experiment can prove me wrong."*
> Albert Einstein, German Physicist, 1897-1955

In this chapter the CIM approach is evaluated using the prototype tool introduced in the last chapter. It will be shown how the three research questions are answered and what the limitations of our approach are.

7.1 Generic Applicability — RQ 1

In the following we will answer research question 1, how models and constraints can be generalized to become applicable for managing the consistency in a broad applications scope. It is informally explained how the proposed approach is applicable to a broad scope of applications. The question is answered using applications that actually use (parts) of the proposed approach or work that is currently in progress using our approach.

7.1.1 Design Language

One major concern that we have throughout this thesis is to keep the approach as general as possible. This refers to the used design and modeling language (i. e., the language in which a model, that should be kept consistent, is expressed) and the used constraint language (i. e., the language that is used to express the constraints that define a consistent state of the system). Most state-of-art approaches are limited to one particular design and modeling language and/or constraint language. Nentwich et al. presented the xLinkIt approach [73] that is general applicable on XML documents. However, the limitation to XML documents itself is a restriction on the used design language, even though nowadays many models can be expressed as XML documents.

Since we can apply Definition 1 to UML diagrams, this definition can be applied to various different design and modeling languages. For example, to Entity Relationship (ER) models [25] known from database systems. The elements in these model are *Entities*

and *Relationships*. The entities are connected via relationships and can have attributes, the properties of these elements.

Aside database systems, product lines are also able to be expressed in this notation. A case study [107, 108] is already made for the DOPLER (Decision-Oriented Product Line Engineering for effective Reuse) tool suite [35], a decision oriented model environment for software product lines. This tool is now used in productive work.

The approach presented in this work is also used in environments where the meta-model can be evolved [34], i. e., the constraints against the model are validated (explained in the next section), are defined generic and the concrete form of this constraints depends on the meta-model definition. So, this enables a variable definition of the meta-model syntax for the applicability of the proposed approach in this work.

Aside from modeling languages, our definition of a model can also be applied to programming languages like Java, which are very similar to the UML class diagrams. Java classes are elements that contain attributes and operations. Both, attributes and operations can be elements or attributes. The approach presented in this work is also applied to validate constraints on running Java programs.

Actually the proposed approach is also used in the mechatronics domain [64], where the consistency of interrelated documents is managed. The sources for the documents are spreadsheets (e. g., MS Excel), CAD drawings (e. g., ProEngineer) and calculations from, for example, MathLab. The proposed approach is used to determine the consistency, if the calculations done in Excel correspond to the dimensions used in the drawings. If, for example, the calculated dimensions and number of screws of a flange corresponds in the drawings.

Unfortunately, a formal proof that our definition of a model is applicable on every kind of design and modeling language cannot be made as there exist many modeling languages, but we believe, based on our experience with the mentioned projects ([107, 108], [34], that this definition is applicable to most of the commonly used modeling languages.

7.1.2 CONSTRAINT LANGUAGE

Definition 2 defines a general representation of a constraint. The context can be any element type specified by the domain language, e. g., the model language UML, the DOPLER meta-model [107, 108], the Java programming language, etc. Furthermore, the condition can be expressed in any form as long as it can deal with the design language and validates to a Boolean value. The condition a constraint must fulfill is that the validation can be represented in a tree based form.

The MODELANALYZER approach by Egyed [39] uses hard coded C# or Java constraints. Later, this approach was extended to use OCL as constraint language [51]. Vierhauser et al. [108] use Java only and Demuth et al. [34] use a constraint language that is based on OCL. Actually, most of the applications of the approach use OCL like constraints ([34], [64]) or Java constraints ([108]). As there already exist works asides UML and OCL that uses this technology, research question RQ 1 was answered with the limitations shown in Section 7.4.1.

7.2 Correctness and Appropriateness — RQ 2

The second research question addresses the problem, if it is possible to generate appropriate and directly executable repair alternatives for detected inconsistencies without loosing the general applicability of the approach. To answer this question first the correctness of the detected inconsistencies must be ensured.

7.2.1 Correct Inconsistency Detection

The correctness of the detected results is evaluated using the tool implementation for UML and OCL. Parallel to the reasoner that is based on the proposed approach (CIM), the OCL constraints are validated using the OCL environment provided by the Eclipse MDT environment. Used for the evaluation are the OCL constraints introduced in Chapter 2 and listed in Appendix A. The constraints are applied on the UML models listed in Table 7.1. The first column of Table 7.1 shows the number of the model and the second column the name of the model (some models must be anonymized). From the third to the sixth column are the size of the model (number of model elements), the number of scope elements and the number of total constraint validations listed. In the brackets next to the constraint validations the detected inconsistencies are listed. For all cases the number of the detected inconsistencies and consistencies was equal for the MDT OCL validation and the CIM approach implementation. Therefore, the inconsistency detection is working correctly.

7.2.2 Correct Scope for Re-Validation

The scope of a constraint must be complete and minimal to be considered correct. Minimal means that there is no scope element in the scope of a constraint validation where not at least one value exists that causes a change of the overall validation result or changes the scope of the constraint validation.

$$\forall e.p \in S(\gamma(x))|(\exists e.p.\sigma'|(e.p.\sigma \neq e.p.\sigma' \wedge (\gamma(x) \neq \gamma(x)' \vee S(x) \neq S(x)')) \tag{7.1}$$

Complete means that their does not exist a model element property in the model whose modification has an influence on the overall validation result of the constraint validation. It does not imply that a modification of a model element property that is not in the scope would make a constraint validation obsolete (e. g., the deletion of the context element $\Rightarrow$ deletion of the constraint validation).

$$\nexists e \in M|(e.p.\sigma \neq e.p.\sigma' \wedge \exists x \in M|(e.p \notin S(\gamma(x)) \wedge \gamma(x) \neq \gamma(x)')) \tag{7.2}$$

In Chapter 4 we showed that the first-order logic expressions can be expressed with only three expression types: the conjunction ($\wedge$), negation ($\neg$) and the universal quantifier ($\forall$). Thus, we prove the minimality property (7.1) and completeness property (7.2) on the three expression types. We proof the correctness based on case distinctions. We distinguish four different cases to proof the correctness of the scope calculations. The

Nr	Name	#Model Elements	#Scope Elements	#Constr. Val. (I)
1	Video on Demand	90	127	63 (7)
2	ATM	220	763	304 (64)
3	Microwave Oven	290	296	138 (24)
4	Model View Controller	418	834	393 (74)
5	eBullition	513	892	341 (76)
6	Curriculum	763	1,350	595 (95)
7	Teleoperated Robot	1,115	1,969	885 (86)
8	Dice 3	1,274	1,649	599 (150)
9	ANTS Visualizer	1,282	3,119	1,225 (235)
10	Inventory and Sales	1,296	1,898	803 (134)
11	Course Registration	1,406	1,822	712 (159)
12	UML IOC F05a T12	1,453	2,441	998 (314)
13	VOD 3	1,558	4,652	1,789 (259)
14	Vacation and Sick Leave	1,658	2,681	1,084 (246)
15	Home Appliance	1,707	2,115	754 (157)
16	HDCP Defect Seeding	1,784	2,199	985 (171)
17	DESI 2.3	1,974	4,727	1,838 (373)
18	iTalks	2,212	4,049	2,289 (289)
19	Hotel Management Sys.	2,583	4,244	2,033 (564)
20	Biter Robocup	2,632	6,265	2,334 (476)
21	Calendarium	2,809	6,160	2,694 (708)
22	UML LCA F03a T1	2,983	2,912	1,243 (317)
23	<unnamed>	5,373	6,804	2,906 (872)
24	NPI	7,110	8,536	2,930 (1,192)
25	Word Pad	8,078	17,907	8,186 (1,557)
26	dSpace 3.2	8,761	12,994	5,869 (1,371)
27	OODT	9,828	26,650	11,384 (2,590)
28	Insurance Network Fees	16,255	27,442	10,562 (3,250)
29	<unnamed>	33,347	33,844	16,627 (3,609)
30	<unnamed>	64,061	67,723	40,297 (4,305)

TABLE 7.1: List of Models Used in the Evaluation

proofs for the negation are omitted for case 2 to 4 because the proofs would be the same as for case 1.

Case 1:

$$\begin{aligned} M := \{a,b\}, a := false, b := false \quad &: \quad S(a \wedge b) := \{a\} \; or \; \{b\} \\ &: \quad S(\neg a) := \{a\} \\ &: \quad S(\forall x \in M | x) := \{M, a\} \; or \; \{M, b\} \end{aligned}$$

Proof. 7.1:

$$\begin{aligned} S(a \wedge b) := \{a\}, a' := true \quad &: \quad S(a \wedge b)' := \{a\} \rightarrow \{b\} \\ S(a \wedge b) := \{b\}, b' := true \quad &: \quad S(a \wedge b)' := \{b\} \rightarrow \{a\} \end{aligned}$$

$$
\begin{array}{rcl}
S(\neg a) := \{a\}, a' := true & : & S(\neg a') := \{a\}, \gamma(\neg a') := true \rightarrow false \\
S(\forall x \in M|x) := \{M, a\}, a' := true & : & S(\forall x \in M|x)' := \{M, a\} \rightarrow \{M, b\} \\
S(\forall x \in M|x) := \{M, b\}, b' := true & : & S(\forall x \in M|x)' := \{M, b\} \rightarrow \{M, a\}
\end{array}
$$

□

Proof. 7.2:

$$
\begin{array}{rcl}
S(a \wedge b) := \{a\}, b' := true & : & S(a \wedge b) = S(a \wedge b'), \gamma(a \wedge b) = \gamma(a \wedge b') \\
S(a \wedge b) := \{b\}, a' := true & : & S(a \wedge b) = S(a' \wedge b), \gamma(a \wedge b) = \gamma(a' \wedge b)' \\
S(\neg a) := \{a\}, b' := true & : & S(\neg a) = S(\neg a), \gamma(\neg a) = \gamma(\neg a') \\
S(\forall x \in M|x) := \{M, a\}, b' := true & : & S(\forall x \in M|x) = S(\forall x \in M|x)', \\
 & & \gamma(\forall x \in M|x) = \gamma(\forall x \in M|x)' \\
S(\forall x \in M|x) := \{M, b\}, a' := true & : & S(\forall x \in M|x) = S(\forall x \in M|x)', \\
 & & \gamma(\forall x \in M|x) = \gamma(\forall x \in M|x)'
\end{array}
$$

□

Case 2:

$$
\begin{array}{rcl}
M := \{a, b\}, a := true, b := false & : & S(a \wedge b) := \{b\} \\
 & : & S(\forall x \in M|x) := \{M, b\}
\end{array}
$$

Proof. 7.1:

$$
\begin{array}{rcl}
S(a \wedge b) := \{b\}, b' := true & : & S(a \wedge b') := \{b\} \rightarrow \{a, b\}, \\
 & & \gamma(a \wedge b)' = false \rightarrow true \\
S(\forall x \in M|x) := \{M, b\}, b' := true & : & S(\forall x \in M|x)' := \{M, b\} \rightarrow \{M, a, b\}, \\
 & & \gamma(\forall x \in M|x)' = false \rightarrow true
\end{array}
$$

□

Proof. 7.2:

$$
\begin{array}{rcl}
S(a \wedge b) := \{b\}, a' := false & : & S(a \wedge b) = S(a \wedge b)', \gamma(a \wedge b) = \gamma(a \wedge b)' \\
S(\forall x \in M|x) := \{M, b\}, a' := false & : & S(\forall x \in M|x) = S(\forall x \in M|x)', \\
 & & \gamma(\forall x \in M|x) = \gamma(\forall x \in M|x)'
\end{array}
$$

□

Case 3:

$$
\begin{array}{rcl}
M := \{a, b\}, a := false, b := true & : & S(a \wedge b) := \{a\} \\
 & : & S(\forall x \in M|x) := \{M, a\}
\end{array}
$$

Proof. 7.1:

$$\begin{aligned} S(a \wedge b) := \{a\}, a' := true &: S(a \wedge b)' := \{a\} \rightarrow \{a, b\}, \\ &\quad 2\gamma(a \wedge b)' = false \rightarrow true \\ S(\forall x \in M|x) := \{M, a\}, a' := true &: S(\forall x \in M|x)' := \{M, a\} \rightarrow \{M, a, b\}, \\ &\quad \gamma(\forall x \in M|x)' = false \rightarrow true \end{aligned}$$

□

Proof. 7.2:

$$\begin{aligned} S(a \wedge b) := \{a\}, b' := false &: S(a \wedge b) = S(a \wedge b)', \gamma(a \wedge b) = \gamma(a \wedge b)' \\ S(\forall x \in M|x) := \{M, a\}, b' := false &: S(\forall x \in M|x) = S(\forall x \in M|x)', \\ &\quad \gamma(\forall x \in M|x) = \gamma(\forall x \in M|x)' \end{aligned}$$

□

Case 4:

$$\begin{aligned} M := \{a, b, c\}, A := \{a, b\}, a := true, b := true &: S(a \wedge b) := \{a, b\} \\ &: S(\forall x \in A|x) := \{A, a, b\} \end{aligned}$$

Proof. 7.1:

$$\begin{aligned} S(a \wedge b) := \{a, b\}, a' := false &: S(a \wedge b)' := \{a, b\} \rightarrow \{a\}, \\ &\quad \gamma(a \wedge b)' = true \rightarrow false \\ S(a \wedge b) := \{a, b\}, b' := false &: S(a \wedge b)' := \{a, b\} \rightarrow \{b\}, \\ &\quad \gamma(a \wedge b)' = true \rightarrow false \\ S(\forall x \in A|x) := \{A, a, b\}, a' := false &: S(\forall x \in A|x)' := \{A, a, b\} \rightarrow \{A, a\}, \\ &\quad \gamma(\forall x \in A|x)' = true \rightarrow false \\ S(\forall x \in A|x) := \{A, a, b\}, b' := false &: S(\forall x \in A|x)' := \{A, a, b\} \rightarrow \{A, b\}, \\ &\quad \gamma(\forall x \in A|x)' = true \rightarrow false \end{aligned}$$

□

Proof. 7.2:

$$\begin{aligned} S(a \wedge b) := \{a, b\}, c \rightarrow c' &: S(a \wedge b) = S(a \wedge b)', \gamma(a \wedge b) = \gamma(a \wedge b)' \\ S(\forall x \in A|x) := \{M, a, b\}, c \rightarrow c' &: S(\forall x \in A|x) = S(\forall x \in A|x)', \\ &\quad \gamma(\forall x \in A|x) = \gamma(\forall x \in A|x)' \end{aligned}$$

□

7.2.3 Correct Cause Calculation

For correctness, calculated causes must be complete in that they identify all elements that cause the inconsistency and minimal in the sense that they do not contain any elements that do not cause the inconsistency.

For the proof of correctness, we can make the following assumptions which are true for commonly used modeling languages and constraints: 1) the constraint is well formed (i. e., syntactically correct), 2) the constraint starts its evaluation always at the context element provided, and 3) all model elements accessed during that rule's evaluation must be reached by navigating from this context element (i. e., there must not exists any "floating" model parts that are not connected to other parts but are accessible). All constraints we encountered to date satisfy these conditions.

Algorithm 4 in Section 5.4 computes two causes: the expressions of the rule validation that cause inconsistencies (line 10) and the model element properties that contribute to this cause (line 8). The model element properties are these parts that cause an inconsistency and that can be changed by the designer. Therefore, each model element property that is in the cause of model element properties must be accessed by at least one expression that is in the cause of expressions, i. e., an expression that is in the cause of expression must have an expression that accesses the model element property in its arguments.

$$\exists e.p \in \zeta_{e.p}, \exists \epsilon \in \zeta_{\epsilon} | \epsilon_{e.p} \in \epsilon.\alpha$$

The cause of expressions contains all expressions from inconsistent constraint validations where the validation result differs the expected result. These expressions are the basis for the cause of model element properties because only those model element properties are included in the cause of model element properties that are accessed by expressions where the validated result differs the expected result. Therefore, it is enough to prove that the cause of expressions is correct.

$$\zeta_{\epsilon} \rightarrow correct \quad \Rightarrow \quad \zeta_{e.p} \rightarrow correct$$

To show that the cause of expression is complete and minimal (i. e., is correct) we use an abstract example of constraint validation that is inconsistent shown in Figure 7.1. The top diamond node represents the inconsistent constraint validation. Beneath this node the oval nodes represent the Boolean expressions of the validation tree. In the nodes the result of the expression is shown. Instead of 'true' and 'false' the result of the comparison of the expected and the validation result is show, i. e., 'fail' means that the validated result differs the expected result and 'succeed' means that the validated result equals the expected result.

We distinguish three different cases in a constraint validation that detects an inconsistency: a) there is a continuous path of Boolean expressions that fail, b) one Boolean expression validation in the path to the leaves succeeds and c) all Boolean expressions

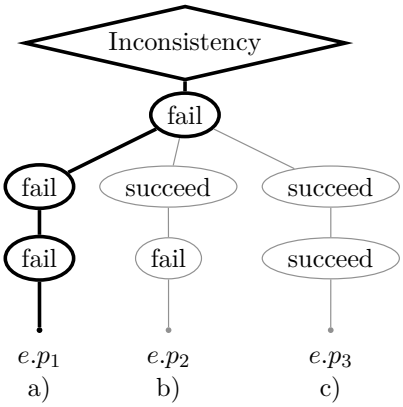


FIGURE 7.1: Combinations of Expression Validations in an Inconsistent Validation Tree

below a failed expressions succeed. A Boolean expression is in the cause of expressions if the validated result differs the expected result (fail) and if its parent is in the cause of expressions (a recursive condition up to the root expression that the validated and expected result of its parent expression differ too). The only exception is the root expression because it has no parent but the root expression expresses the overall validation result of the constraint validation: 'true' for consistency (the expected result) and 'false' for inconsistency.

Case a) of Figure 7.1 shows a continuous path of expression that fail (the validated and expected result differ) and therefore these expression must be in the cause of expressions of an inconsistency. The leaves of a validation tree are the expressions that access model elements or are constants of a constraint condition. Thus, the leaves of a continuous path of expressions that fail must be in the cause of model element properties in the case of a model element property access. Therefore, at least one Boolean expression that fails is a parent of the expressions that access a model element property that is in the cause of model element properties.

On the other hand, case b) illustrated in Figure 7.1 shows a validation path where an Boolean expression that fails is a parent of an expression that accesses a model element property but both are not in the cause of expressions or cause of model element properties. The overall validation result of the inconsistency cannot change due to a change of this model element property because one of the expressions in the path to the root expressions (parents) validates to the expected result. The algorithm to calculate the cause is strictly top-down starting at the root expression and therefore it stops, when the first expression is detected that succeeds.

Case c) shows the case where all expression validate to their expected result and thus these validation can not cause the inconsistency at all. Our algorithm to calculate the cause applied to a consistent validation will generate an empty cause because in a constraint validation that does not detect an inconsistency, only the cases b) and c) can occur.

Please note that there may be very well some model element properties in the cause of an inconsistency that occur in branches that are not in the cause of expressions because a model element property can be accessed in more than one location of a constraint validation. This would be the case if, for example, in Figure 7.1 all three model element accesses would be the same ($e.p_1 = e.p_2 = e.p_3$). Due to case a) the model element property would be in the cause. But because of the strict top-down navigation in the validation tree (from expression to model element properties and not from model element properties to the expressions) for generating the cause, it is guaranteed that the accessed model element properties from the cause can be uniquely associated to the detected inconsistency.

7.2.4 Appropriateness of the Generated Repairs

The repairs are generated out of the cause of expressions and cause of scope elements, which we already have shown that they are correct. We evaluated empirically if the generated repairs are appropriate, i. e., if they are applicable on the models used in the evaluations.

We evaluated how many repairs can be generated and how many of them are concrete. Table 7.2 shows the number of repairs that are generated for each model. The first column shows the model number (referencing to Table 7.1), the second and third column the number of abstract and concrete repairs generated directly out of the validation tree, the fourth and fifth column the repairs after considering the side effects and the last column the improvements due to the consideration of the side effect. The number of improvements is very low. This is because of the use of very small constraints that contain nearly no negations that could improve this result. For the largest model it was not possible to generate repairs, due to memory limitations which are discussed in Section 7.4). Please note that the total number of repairs is not necessarily the same after considering the side effects, because for some abstract or concrete repairs more possible concrete repairs can be generated, or some of them might cause unwanted side effect so they are removed. All concrete repairs can be applied in the model and are able to resolve the inconsistencies. This was proven by executing the repair in the model and observing if the inconsistency was resolved.

We also evaluated how many repair actions a repair contains and in our evaluations we encountered that each repair consists of one repair action only. This stems from the small constraints used in our evaluation. Furthermore, we evaluated how many repairs are generated for each detected inconsistency. In Figure 7.2 the number of repairs per inconsistency depending of the model size is shown. As we can see, the number of repairs generated per inconsistency remains stable between two and twelve repairs (on average eight). Moreover, we split up the repairs in the three types of repair actions. The total number of these repairs is also shown in Figure 7.2. In contrast to the repairs per inconsistency, the total number of repairs increases with the size of the model. The number of repairs for each repair action type increases nearly constant and the distribution is nearly the same for all evaluated models. Most of the repairs are *modify* and *delete* actions (approximately ten times more than *add* repairs).

Nr	Direct [#Repairs]		Side Effect [#Repairs]		Improvement [%]
	Abstract	Concrete	Abstract	Concrete	
1	27	45	27	45	0.0
2	201	138	192	147	6.5
3	183	261	183	261	0.0
4	297	816	297	816	0.0
5	405	1,356	405	1,368	0.9
6	600	1,485	600	1,488	0.2
7	351	390	351	390	0.0
8	1,140	2,133	1,119	2,250	5.5
9	735	1,353	729	1,377	1.8
10	981	4,530	969	4,740	4.6
11	1,515	4,680	1,353	6,018	28.6
12	1,335	1,818	1,329	1,830	0.7
13	579	1,677	579	1677	0.0
14	2,196	3,012	2,196	3,192	6.0
15	972	4,782	972	4,782	0.0
16	108	732	108	732	0.0
17	600	2,553	600	2,553	0.0
18	2,373	12,534	2,343	12,669	1.1
19	3,351	5,931	3,324	6,261	5.6
20	4,614	4,434	4,383	5,064	14.2
21	1,134	3,561	1,134	3,561	0.0
22	3,747	6,396	3,474	6,396	0.0
23	1,707	1,002	1,707	1,170	16.8
24	7,038	5,037	7,038	5,037	0.0
25	2,880	1,761	2,880	1,761	0.0
26	111	393	111	393	0.0
27	765	2,721	765	2,721	0.0
28	15,987	33,369	15,957	33,789	1.3
29	29,625	73,056	29,619	73,068	1.1
30	N.A. out of memory				

TABLE 7.2: Abstract and Concrete Repairs Generated

In Figure 7.3 the average number of repairs depending on the validation tree size is shown. The size of the validation tree corresponds to the number of nodes (expression) in the validation tree. For the evaluation the average number of nodes for each constraint validation is taken. Each plot in the diagram is the average number of repairs generated for a constraint definition. Only those constraints are shown and taken for the calculation of average number of nodes and repairs. As can be seen the number of generated repairs slightly increases depending on the validation tree size. However, a complex and large validation tree does not imply the generation of a large number of repairs as can be seen in Constraint 1 (C01). On average for all constraints the average number of generated repairs is between 1 and 13, hence, at least one repair for each inconsistency was generated and can be executed in the model to resolve the inconsistency. Therefore, research question RQ 2 was answered.

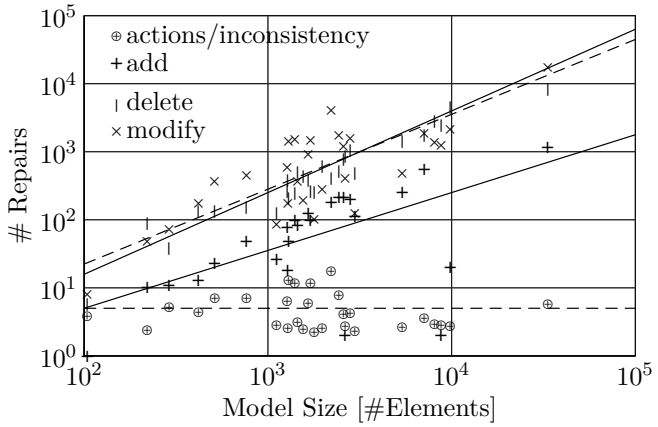


FIGURE 7.2: Repairs depending on the Model Size

7.3 Performance and Scalability — RQ 3

Finally, we evaluated if the approach can be used to work interactively. Two aspects are evaluated to answer research question RQ 3: 1) the memory consumption of the approach depending on the model size, and 2) the response time of the approach depending on the model size and constraint complexity (validation tree size). For the evaluations we used the prototype implementation for the IBM Rational Software Architect for UML and OCL introduced in Chapter 6. The prototype was running on an Intel Core 2 Quad CPU @2.83GHz with 8GB (4GB available for the RSA) RAM and 64bit Linux (3.1.9).

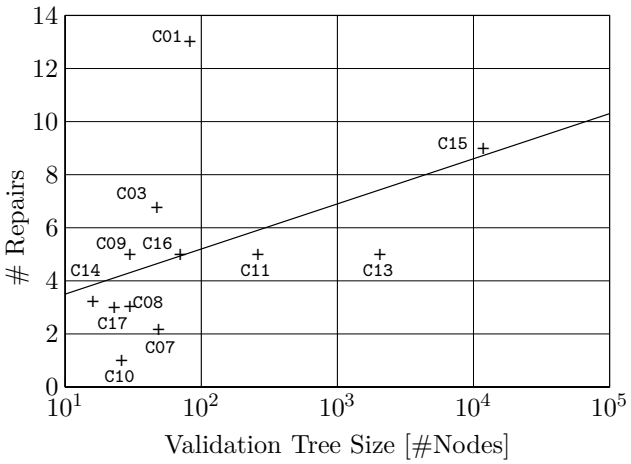


FIGURE 7.3: Repairs depending on the Validation Tree Size

Nr	Memory Overhead [MB]		
	Complete	CIM	MDT OCL
1	2	2	1
2	20	10	7
3	29	13	9
4	16	12	7
5	53	18	10
6	150	43	4
7	97	34	5
8	74	14	3
9	169	93	6
10	250	17	4
11	97	19	4
12	67	23	6
13	175	110	7
14	145	65	5
15	267	53	6
16	72	36	7
17	188	106	7
18	417	130	6
19	359	87	5
20	227	129	8
21	326	79	6
22	108	53	3
23	973	129	7
24	1,353	97	7
25	860	513	20
26	N.A.	259	11
27	752	434	21
28	N.A.	172	58
29	N.A.	382	111
30	N.A.	724	61

TABLE 7.3: Memory Overhead Compared to MDT OCL

7.3.1 MEMORY CONSUMPTION

The basic data structure used in the CIM approach is the validation tree that must be cached in memory. Similar to the RETE algorithm [50], the cached data structure is used primary for the fast re-validation. To generate the repairs it is not necessary to keep all the validation trees but unfortunately to calculate the side effects most of the validation trees must be kept in memory, which results in an increased memory consumption.

The memory consumption was measured using the `Runtime` interface of Java. Before measuring the memory the garbage collector was activated running `gc()`. To reduce the uncertainty the measurements were repeated five times and the median was taken. Additional, on a random basis the results were double-checked using the TPTP profiler for Eclipse. The memory consumption was measured after the initial validation of all constraints on the models.

In Table 7.3 the results of the memory evaluation is shown. In the first column the model number is shown. The second column show the memory usage for the validation trees without optimization, i. e., the complete validation tree without scope and cause optimization was used. This would be the case if the basic idea of the RETE algorithm would be used. As can be seen, the amount of memory increases very fast and for the larger models the amount of available memory was not enough for the tool.

The third column shows the amount of memory used by the validation trees from used by the CIM Approach. Especially for larger models the memory consumption could be reduced significantly, such that it was possible to evaluate even the larger models. Only during the calculation of the side effects for randomly seeded inconsistencies (shown in the next section) for the largest model the tool was running out of memory.

The last column shows the memory consumption using the MDT OCL environment. MDT OCL uses the least amount of memory, however, if MDT OCL is used the incremental characteristics get lost and the generation of appropriate repairs including their side effects will not work.

7.3.2 Response Time and Scalability

Finally we evaluated the response time and the scalability regarding the model size and the validation tree size of the proposed approach. For that purpose we randomly seeded 50 changes in each model and repeated each change ten times to reduce environmental effects such as, for example, the garbage collector. The measurements were done using `System.nanoTime()` (resolution $1\mu s$). We compared the approach to the MDT OCL environment of Eclipse. Comparing this approach to others would not be appropriate because it is very hard to evaluate them using the same environment and models to compare. Thus, we are concentrating on the measurements on a normal desktop environment and evaluate if the times to detect inconsistencies and propose solutions to resolve them are sufficient for the practical use (Miller [72]).

In Figure 7.4 the re-validation times (without repair and side effect generation time) for the random changes are shown. As can be seen the re-validation time of the CIM approach is only the tenth of the re-validation of the MDT OCL environment. The re-validation increases slightly with the size of the model. However, the time never exceeds 1ms.

In Figure 7.5 the re-validation time depending on the validation tree is shown. As can be seen here, the re-validation time for the MDT OCL environment increases much more with the size of the validation tree or in the case of the MDT OCL environment the complexity of the constraint. Therefore, we see the strength of our approach which is rooted in the re-validation of complex constraints, hence it is not only independent of the model size but also independent of the constraint.

Figure 7.6 shows the time it takes to generate the repairs and calculate their side effects. As can be seen the generation of the repairs takes about 1ms independent of the model size. The calculation of the side effects takes much longer, in average about 10ms and up to 30ms. However, the calculation is also nearly independent of the model size, hence the calculation of the side effects scales on large models.

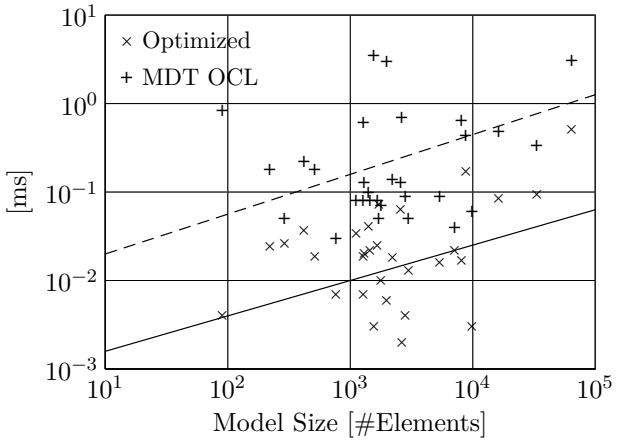


FIGURE 7.4: Re-Validation Time Depending on the Model Size

Finally we analyzed the generation of the repairs and the calculation of the side effects depending on the validation tree size. In Figure 7.7 we can see that the generation of the repairs is in average below 1ms for each constraint. However, the calculation of the side effects exceeds 100ms for particular constraints, in the worst case it takes about 500ms. Fortunately, after manual inspection such cases are very rare (below 1% of all validations) and happened in large models only.

When we summarize all the evaluations (re-validation, repair generation and side effect calculation) we observe that 99% of the validations, generations and calculations are done in less then 100ms and in 90% of all cases even in less than 1ms. With some limitations we also could answer research question RQ 3.

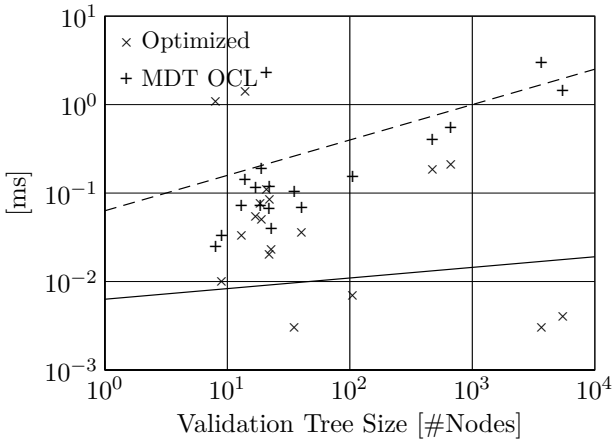


FIGURE 7.5: Re-Validation Time Depending on the Validation Tree Size

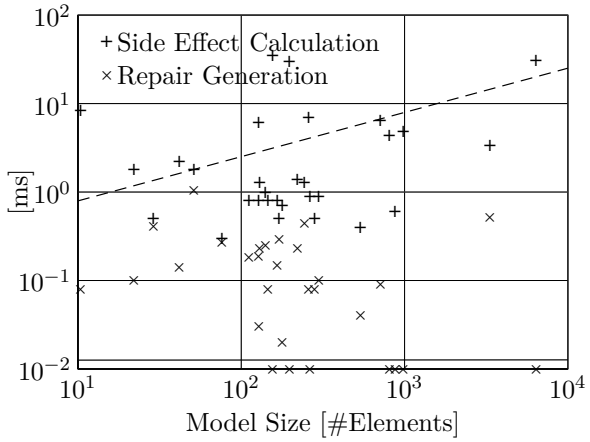


FIGURE 7.6: Repair Generation and Side Effect Calculation Depending on the Model Size

7.4 Limitations of the Approach

The evaluation show the potential of the CIM approach. However, it has also shown its limitations. For all the three research questions that we have answered we found the limitations of the approach.

7.4.1 Limitation in the General Applicability

During the implementation of the prototype the first limitations and difficulties were detected when the constraint language provide some special operations, like, for example, a `select` (from OCL) or the `allParents` property from UML. In these cases special

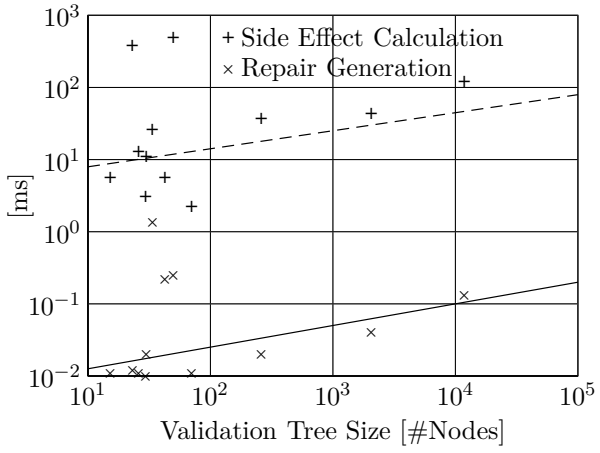


FIGURE 7.7: Repair Generation and Side Effect Calculation Depending on the Validation Tree Size

adaptions to the reasoner had been made, so that these types of expressions can be used in the reasoner. Although, a solution was found, it can be expected that other constraint languages may lay down additional special requirements to the implementation.

7.4.2 Limitations in Appropriateness

For the second research research question we only encountered limitations in the appropriateness. Limitations of the correctness would make the approach obsolete, because false results are not an option. As we use only the validation behavior of the constraint for our reasoner, some potential repairs cannot be provided to the designer. In the case of the example used in Chapter 5 (Constraint 1 validated on the *Message* `wait`), an appropriate repair would be the introduction of a super class that has an *Operation* named 'wait'. But, as the constraint does not explicitly searches for super classes, no repair can be suggested that introduces a new *Class* with an *Operation* `wait`. Considering the design and modeling language could improve this situation, however doing so would restrict the general applicability. Fortunately, the approach itself does not restrict the freedom of extending it in a special implementation. Therefore, the focus of this thesis is a general applicable approach that can be implemented for user specific needs.

7.4.3 Limitations in Performance and Scalability

The needed memory constitutes the main limitation of the presented approach. During developing the approach this problem was encountered very early and therefore we developed the optimization for the scope and cause. Fortunately, this optimization resulted in not only memory improvements but also in improvements for the re-validations (smaller scopes cause fewer re-validations). However, the memory consumption is the main disadvantage of this approach and further optimizations must be made. A tighter integration of the approach in a used tool might also improve this situation, because identical data structures were build twice — one time in the tool itself and one time for the approach (e. g., model elements — in the design tool and the scope — in the prototype implementation).

7.5 Summary

In this chapter the CIM approach introduced in this thesis was evaluated. The evaluations were done mostly empirically based on the prototype tool implementation for UML and OCL. 30 UML models of different sizes were used and 20 OCL well formedness rules to ensure that the two of three research questions can be answered. Furthermore, the encountered limitations of our approach were also discussed.

Chapter 8

Conclusion and Ongoing Work

"You are never too old to set another goal or to dream a new dream."
C. S. Lewis, British Author, 1898-1963

In this chapter we briefly summarize the problems addressed in this thesis and how they were solved, as well as the results from the evaluation of the solution. Furthermore, a brief outlook is given about ongoing work, as well as planned topics based on the introduced approach.

8.1 Conclusion

Inconsistencies or missed expectations in a delivered software product are annoying for both, the customer and the developer. For the customer because she needs the software, had to pay for it and if it does not fulfill the expectations, the result would be increasing costs and a delayed application of the software. For the developer because she must repair the software, which comes also with increasing costs and time that can be spent on new projects. However, it is science fiction to assume that developers will immediately fully understand the customer and environmental requirements. Therefore, in this thesis an approach is introduced that should help to improve the understanding and validation of the customer needs early in the development process to increase the customer and developer satisfaction.

In this thesis an approach for managing the consistency regarding customer requirements and environmental requirements, like for instance the used design or programming languages. The approach was design to be applicable in different domains in the software development process, but with the primary focus on the design and modeling phase at a very early stage in the development process. A key requirement of the presented approach was to provide a fast response about the state of the design model, i. e., instant feedback when an inconsistency was detected and supporting the designer with information how to resolve it. Furthermore, the designer must be able do define constraints based on a common used constraint language do define all the aspects that a software product must fulfill.

Existing work in the domain of consistency management often focuses on one particular design language and constraint language. Moreover, the used constraints are often hard coded or it is very hard to write new constraints because a specially designed constraint language is used. Recent work on consistency management comes with new technologies that address some of the problems that were solved in this thesis. However, they address a single problem, but do not cover the complete spectrum, e. g., an incremental consistency checker that is able to resolve inconsistencies based on an arbitrary set of constraints in a general context – that's what this thesis focused on.

The presented approach works with a constraint language that is based on the basic concepts of first order logic, i. e., any language can be used as a constraint language that is based on the basic principles of first order logic, like, for example, OCL. It is not limited and designed for one specific constraint language, as other work on this domain (e. g., [110], [74]). Furthermore, the only restrictions set for the design language are that it can be expressed as an interrelated set of elements containing properties on the meta level, like the UML. The incremental reasoning is based on a scope based approach.

For the basic logical expressions the parts were determined that are responsible for their validation result (it is not necessary to validate every argument of a logical expression to determine a correct result) to optimize the re-validation effort. Each logical expression comes also with the functionality what parts caused a violation based on an expected result (is true for the whole constraint but can be differ for the arguments) and the actual validation result. Aside the basic logical expression, whose arguments and result are Boolean values, expressions exist that access properties of elements in, for example, the software model. Using the accessed model elements, the expected and validated result, the repairs to resolve a detected inconsistency are generated.

As already mention in existing work, a repair for an inconsistency can have theoretically an infinite number of choices. Therefore, data structures are developed that provide a manageable visualization of the proposed solutions. The basic data structure for the (re)-validation is the *validation tree* and for resolving inconsistencies the *repair tree*. The validation tree represents the validation of a constraint with the expressions as nodes and the branches as their relations. The leaves of the validation tree are expression that access properties of model elements or constant defined in the constraint. Out of a validation tree that is violated (validates to false – inconsistency) the repair tree is generated. The nodes of the repair tree are alternative (one of the alternatives must be chosen to resolve an inconsistency) or combination (from all branches an action must be take to resolve an inconsistency) nodes. The leaves of the repair tree are the actions that must be executed in the model.

An action to repair an inconsistency can be an addition, deletion or modification of an element property. Furthermore, to execute repair actions in a model a value must be known of how to modify the addressed element property. Unfortunately, it is not always possible to determine an unique value that is applicable. Therefore user intervention is needed resolving such inconsistencies and only a suggestion to the designer can be given.

A repair action does not only have an influence on the constraint validation that is violated, but also on the constraint validation that accesses the element property

that will be changed. Such effects we call *side effect*. A side effect can resolve another inconsistency or cause an addition inconsistency. Moreover, side effects are used to determine values for repair actions where no direct value could be determined in the repair tree generation process.

The approach was implemented as an Eclipse based plugin for the IBM Rational Software Architect for UML and OCL. The evaluations of the approach were done using the tool implementation and have shown that it is possible to generate a generalizable approach that is able to detect inconsistencies, propose solutions to resolve them and calculating their side effects in a reasonable amount of time so that it can be used for interactive work. Unfortunately, during the implementation and evaluation, the limitations of that approach were made clear – the memory consumption for larger models. However, further optimization and tighter tool integration can improve the memory management.

8.2 Ongoing Work

The approach presented in this work actually is used in different domains. Since a few years it is used in the domain of product line engineering to validate the consistency of product lines. The most recent work using this approach is on evolving constraints, i. e., where the constraints evolve with changes in the meta model. Another application of that approach is in the mechatronics domain where the consistency among different tools is validated. One work focuses on the validation of Java programs during run-time, if arbitrary definable constraints are fulfilled, without annotating the source code (e. g., assertions). In a very early stage is an application that uses this approach in a multi-user environment, where different users are working at the same time at different but interrelated parts of a model.

Aside from the different application domains, an important topic is the optimization of the approach regarding the memory usage. This would be one of the most important tasks for this approach to be productively applicable in a wide range of domains, not only in the software development process.

References

[1] *S-DAGs: Towards Efficient Document Repair Generation*. CCCT 2004, 2004. 34

[2] **Welcome to NetBeans**. http://netbeans.org/, 2012. 3, 5

[3] **Manifesto for Agile Software Development**. http://agilemanifesto.org/, 2001. 2

[4] Marcos Almeida da Silva, Alix Mougenot, Xavier Blanc, and Reda Bendraou. **Towards Automated Inconsistency Handling in Design Models**. In Barbara Pernici, editor, *Advanced Information Systems Engineering*, **6051** of *Lecture Notes in Computer Science*, pages 348–362. Springer Berlin / Heidelberg, 2010. 34

[5] Kerstin Altmanninger, Gerti Kappel, Angelika Kusel, Werner Retschitzegger, Martina Seidl, Wieland Schwinger, and Manuel Wimmer. **AMOR - Towards Adaptable Model Versioning**. In *1st International Workshop on Model Co-Evolution and Consistency Management (MCCM'08), Workshop at MODELS'08*, Toulouse, France, 2008. 30

[6] Ken Arnold and James Gosling. *The Java Programming Language*. Addison-Wesley, 1996. 30

[7] Robert Balzer. **Tolerating Inconsistency**. In *ICSE*, pages 158–165, 1991. 4, 24, 34

[8] J. Barwise. **An introduction to first-order logic**. *Studies in Logic and the Foundations of Mathematics*, **90**:5–46, 1977. 8, 16

[9] K. Beck and E. Gamma. **Test Infected: Programmers Love Writing Tests**. *Java Report*, **3**(7):51–56, 1998. 36

[10] Kent Beck and Cynthia Andres. *Extreme Programming Explained: Embrace Change (2nd Edition)*. Addison-Wesley Professional, 2004. 2

[11] Matthias Biehl and Welf Löwe. **Automated Architecture Consistency Checking for Model Driven Software Development**. In Raffaela Mirandola, Ian Gorton, and Christine Hofmeister, editors, *QoSA*, **5581** of *Lecture Notes in Computer Science*, pages 36–51. Springer, 2009. 31

[12] Armin Biere, Marijn Heule, Hans van Maaren, and Toby Walsh, editors. *Handbook of Satisfiability*, **185** of *Frontiers in Artificial Intelligence and Applications*. IOS Press, 2009. 31

[13] Xavier Blanc, Isabelle Mounier, Alix Mougenot, and Tom Mens. **Detecting model inconsistency through operation-based model construction**. In *ICSE '08: Proceedings of the 30th international conference on Software engineering*, pages 511–520, New York, NY, USA, 2008. ACM. 33

[14] Barry W. Boehm. **Software Engineering**. *IEEE Trans. Computers*, **C-25**:1226–1241, 1976. 1

[15] Barry W. Boehm. *Software engineering economics*. Prentice-Hall advances in computing science and technology series. Prentice-Hall, 1981. xiii, 2, 3

[16] Barry W. Boehm. **A spiral model of software development and enhancement**. *Computer*, **21**(5):61–72, 1988. 1

[17] Barry W. Boehm and Victor R. Basili. **Software Defect Reduction Top 10 List.** *IEEE Computer*, **34**(1):135–137, 2001. 2

[18] Barry W. Boehm and Alexander Egyed. **WinWin requirements negotiation processes: A multi-project analysis**. In *Proceedings of the 5th International Conference on Software Processes*, 1998. 30

[19] Barry W. Boehm, Paul Grünbacher, and Robert O. Briggs. **EasyWinWin: a groupware-supported methodology for requirements negotiation**. In *Proceedings of the 23rd International Conference on Software Engineering*, ICSE '01, pages 720–721, Washington, DC, USA, 2001. IEEE Computer Society. 5, 16, 30

[20] Chandrasekhar Boyapati, Sarfraz Khurshid, and Darko Marinov. **Korat: automated testing based on Java predicates**. In *ISSTA*, pages 123–133, 2002. 36

[21] Lionel C. Briand, Yvan Labiche, and L. O'Sullivan. **Impact Analysis and Change Management of UML Models**. In *ICSM*, pages 256–265. IEEE Computer Society, 2003. 35

[22] Petra Brosch, Martina Seidl, Konrad Wieland, Manuel Wimmer, and Philip Langer. **We can work it out: Collaborative Conflict Resolution in Model Versioning**, 2009. 30

[23] Jordi Cabot and Ernest Teniente. **Incremental Evaluation of OCL Constraints**. In Eric Dubois and Klaus Pohl, editors, *CAiSE*, **4001** of *Lecture Notes in Computer Science*, pages 81–95. Springer, 2006. 33

[24] Laura A. Campbell, Betty H. C. Cheng, William E. McUmber, and Kurt Stirewalt. **Automatically Detecting and Visualising Errors in UML Diagrams**. *Requir. Eng.*, **7**(4):264–287, 2002. 31

[25] Peter Pin-Shan Chen. **The entity-relationship model: toward a unified view of data**. *SIGIR Forum*, **10**(3):9–9, December 1975. 8, 87

[26] Betty H. C. Cheng, Enoch Y. Wang, and Robert H. Bourdeau. **A Graphical Environment for Formally Developing Object-Oriented Software**. In *ICTAI*, pages 26–32, 1994. 32

[27] Alessandro Cimatti, Edmund M. Clarke, Fausto Giunchiglia, and Marco Roveri. **NUSMV: A New Symbolic Model Checker**. *STTT*, **2**(4):410–425, 2000. 31

[28] Alessandro Cimatti, Marco Roveri, Angelo Susi, and Stefano Tonetta. **Formalizing requirements with object models and temporal constraints.** *Software and System Modeling*, **10**(2):147–160, 2011. 31

[29] Krzysztof Czarnecki and Krzysztof Pietroszek. **Verifying feature-based model templates against well-formedness OCL constraints**. In Stan Jarzabek, Douglas C. Schmidt, and Todd L. Veldhuizen, editors, *GPCE*, pages 211–220. ACM, 2006. 11, 31

[30] Hoa Khanh Dam and Michael Winikoff. **Supporting change propagation in UML models**. In *ICSM*, pages 1–10. IEEE Computer Society, 2010. 35

[31] Khanh Hoa Dam. *Supporting Software Evolution in Agent Systems*. PhD dissertation, RMIT University, Melbourne, Victoria, Australia, School of Computer Science and Information Technology, August 2008. 35

[32] Khanh Hoa Dam and Michael Winikoff. **Cost-based BDI plan selection for change propagation**. In Lin Padgham, David C. Parkes, Jörg P. Müller, and Simon Parsons, editors, *AAMAS (1)*, pages 217–224. IFAAMAS, 2008. 35

[33] M. Davis, G. Logemann, and D. Loveland. **A machine program for theorem-proving**. *Communications of the ACM*, **5**(7):394–397, 1962. 31, 36

[34] Andreas Demuth, Roberto E. Lopez-Herrejon, and Alexander Egyed. **Cross-layer modeler: a tool for flexible multilevel modeling with consistency checking**. In Tibor Gyimóthy and Andreas Zeller, editors, *SIGSOFT FSE*, pages 452–455. ACM, 2011. 88

[35] Deepak Dhungana, Rick Rabiser, Paul Grünbacher, Klaus Lehner, and Christian Federspiel. **DOPLER: An Adaptable Tool Suite for Product Line Engineering**. In *SPLC (2)*, pages 151–152. Kindai Kagaku Sha Co. Ltd., Tokyo, Japan, 2007. 88

[36] S. Easterbrook and B. Nuseibeh. **Using ViewPoints for inconsistency management**. *Software Engineering Journal*, **11**(1):31–43, January 1996. 33

[37] Steve M. Easterbrook and Bashar Nuseibeh. **Managing inconsistencies in an evolving specification**. In *RE*, pages 48–55. IEEE Computer Society, 1995. 32

[38] Inc. Eclipse Foundation. **Eclipse - The Eclipse Foundation open source community website**. http://www.eclipse.org/, 2012. 3, 5

[39] Alexander Egyed. **Instant Consistency Checking for the UML**. ACM, 2006. 4, 5, 9, 11, 20, 22, 33, 59, 88

[40] Alexander Egyed. **Fixing Inconsistencies in UML Design Models**. In *ICSE*, pages 292–301. IEEE Computer Society, 2007. 4, 24, 35, 74

[41] Alexander Egyed, Emmanuel Letier, and Anthony Finkelstein. **Generating and Evaluating Choices for Fixing Inconsistencies in UML Design Models**. In *ASE*, pages 99–108. IEEE, 2008. 4, 5, 35

[42] Bassem Elkarablieh and Sarfraz Khurshid. **Juzi: a tool for repairing complex data structures**. In Wilhelm Schäfer, Matthew B. Dwyer, and Volker Gruhn, editors, *ICSE*, pages 855–858. ACM, 2008. 36

[43] Andy Evans, Robert B. France, Kevin Lano, and Bernhard Rumpe. **The UML as a Formal Modeling Notation**. In Jean Bézivin and Pierre-Alain Muller, editors, *UML*, **1618** of *Lecture Notes in Computer Science*, pages 336–348. Springer, 1998. 30

[44] Carles Farré, Ernest Teniente, and Toni Urpí. **Checking query containment with the CQC method.** *Data Knowl. Eng.*, **53**(2):163–223, 2005. 31

[45] Anthony Finkelstein, Dov M. Gabbay, Anthony Hunter, Jeff Kramer, and Bashar Nuseibeh. **Inconsistency Handling in Multi-Perspective Specifications**. In *ESEC '93: Proceedings of the 4th European Software Engineering Conference on Software Engineering*, pages 84–99, London, UK, 1993. Springer-Verlag. 32

[46] Anthony Finkelstein, Dov M. Gabbay, Anthony Hunter, Jeff Kramer, and Bashar Nuseibeh. **Inconsistency Handling in Multperspective Specifications**. *IEEE Trans. Software Eng.*, **20**(8):569–578, 1994. 4

[47] Anthony Finkelstein and Colin Potts. **Formalizing Requirements Systematically.** In Roland Wagner, Roland Traunmüller, and Heinrich C. Mayr, editors, *EMISA*, **143** of *Informatik-Fachberichte*, pages 44–57. Springer, 1987. 30

[48] Anthony Finkelstein and Ian Sommerville. **The Viewpoints FAQ**. *Software Engineering Journal: Special Issue on Viewpoints for Software Engineering*, **11**(1):2–4, 1996. 29

[49] International Organisation for Standardization. **ISO/IEC 9899:1999**. http://www.iso.org/iso/catalogue_detail.htm?csnumber=29237, 1999. 23

[50] C.L. Forgy. **Rete: A Fast Algorithm for the Many Pattern / Many Object Pattern Match Problem**. *Artificial Intelligence*, **19**:17–37, 1982. 22, 27, 59, 98

[51] Iris Groher, Alexander Reder, and Alexander Egyed. **Incremental Consistency Checking of Dynamic Constraints**. In David S. Rosenblum and Gabriele Taentzer, editors, *FASE*, **6013** of *Lecture Notes in Computer Science*, pages 203–217. Springer, 2010. 9, 33, 88

[52] John Grundy, John Hosking, and Warwick B. Mugridge. **Inconsistency Management for Multiple-View Software Development Environments**. *IEEE Transactions on Software Engineering*, **24**:960–981, 1998. 32

[53] Volker Haarslev and Ralf Möller. **High Performance Reasoning with Very Large Knowledge Bases: A Practical Case Study.** In Bernhard Nebel, editor, *IJCAI*, pages 161–168. Morgan Kaufmann, 2001. 32

[54] A.N. Habermann and D. Notkin. **Gandalf: Software development environments.** *IEEE transactions on software engineering*, **12**(12):1117–1127, 1986. 4

[55] Anders Hejlsberg, Scott Wiltamuth, and Peter Golde. *C# Language Specification*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2003. 30

[56] Daniel Jackson. **Alloy: a lightweight object modelling notation**. *ACM Trans. Softw. Eng. Methodol.*, **11**(2):256–290, April 2002. 4, 8

[57] Simon M. Kaplan and Gail E. Kaiser. **Incremental Attribute Evaluation in Distributed Language-based Environments**. In *5th ACM Symp. on Principles of Distributed Computing*, pages 121–130. ACM press, Calgary, August 1986. See also: report UIUCDCS-R-86-1294, University of Illinois at Urbana-Champaign (September 1986). 4

[58] Brian W. Kernighan. *The C Programming Language*. Prentice Hall Professional Technical Reference, 2nd edition, 1988. 30

[59] Dae-Kyoo Kim and Jon Whittle. **Generating UML Models from Domain Patterns**. In *SERA*, pages 166–173. IEEE Computer Society, 2005. 33

[60] H. Kitapci and Barry W. Boehm. **Using a Hybrid Method for Formalizing Informal Stakeholder Requirements Inputs**. In *Comparative Evaluation in Requirements Engineering, 2006. CERE '06. Fourth International Workshop on*, pages 48–59, sept. 2006. 31

[61] R. Kowalski. **Logic for Problem-solving**. *DCL Memo 75*, 1974. 33

[62] Christian Lange and Michel R. V. Chaudron. **An Empirical Assesment of Completeness in UML Design**. In *Proceedings of the 8th Conference on Empirical Assessment in Software Engineering (EASE04)*, 2004. 30

[63] Gary T. Leavens, Albert L. Baker, and Clyde Ruby. **JML: A Notation for Detailed Design**, 1999. 36

[64] Daniela Lettner, Peter Hehenberger, Alexander Nöhrer, Klaus Anzengruber, Paul Grünbacher, Michael Mayrhofer, and Alexander Egyed. **Variability and Consistency in Mechatronic Design**. In *The Seventh International Workshop on Variability Modelling of Software-intensive Systems, VaMoS 2013*, Under Submission, 2013. 88

[65] Mark H. Liffiton and Karem A. Sakallah. **Algorithms for Computing Minimal Unsatisfiable Subsets of Constraints**. *J. Autom. Reasoning*, **40**(1):1–33, 2008. 4

[66] Robert M. MacGregor. **Inside the LOOM Description Classifier.** *SIGART Bulletin*, **2**(3):88–92, 1991. 32

[67] Alan K. Mackworth. **Consistency in Networks of Relations.** *Artif. Intell.*, **8**(1):99–118, 1977. 4

[68] H. Malgouyres and G. Motet. **A UML model consistency verification approach based on meta-modeling formalization**. In *Proceedings of the 2006 ACM symposium on Applied computing*, SAC '06, pages 1804–1809, New York, NY, USA, 2006. ACM. 31

[69] Muhammad Zubair Malik, Junaid Haroon Siddiqui, and Sarfraz Khurshid. **Constraint-Based Program Debugging Using Data Structure Repair**. In *ICST*, pages 190–199. IEEE Computer Society, 2011. 36

[70] Anders Mattsson, B Lundell, Brian Lings, and Brian Fitzgerald. **Linking Model-Driven Development and Software Architecture: A Case Study**. *Software Engineering, IEEE Transactions on*, **35**(1):83–93, 2009. 30

[71] Tom Mens, Ragnhild Van Der Straeten, and Maja D'Hondt. **Detecting and Resolving Model Inconsistencies Using Transformation Dependency Analysis**. In Oscar Nierstrasz, Jon Whittle, David Harel, and Gianna Reggio, editors, *Model Driven Engineering Languages and Systems*,

4199 of *Lecture Notes in Computer Science*, pages 200–214. Springer Berlin / Heidelberg, 2006. 4, 34

[72] Robert B. Miller. **Response time in man-computer conversational transactions**. In *Proceedings of the December 9-11, 1968, fall joint computer conference, part I*, AFIPS '68 (Fall, part I), pages 267–277, New York, NY, USA, 1968. ACM. 5, 8, 99

[73] Christian Nentwich, Licia Capra, Wolfgang Emmerich, and Anthony Finkelstein. **xlinkit: A Consistency Checking and Smart Link Generation Service**. *ACM Trans. Internet Techn.*, **2**(2):151–185, 2002. 4, 11, 22, 32, 35, 87

[74] Christian Nentwich, Wolfgang Emmerich, and Anthony Finkelstein. **Consistency Management with Repair Actions**. In *ICSE*, pages 455–464. IEEE Computer Society, 2003. 4, 24, 35, 104

[75] Alexander Nöhrer, Armin Biere, and Alexander Egyed. **Managing SAT inconsistencies with HUMUS**. In Ulrich W. Eisenecker, Sven Apel, and Stefania Gnesi, editors, *VaMoS*, pages 83–91. ACM, 2012. 36

[76] Alexander Nöhrer, Alexander Reder, and Alexander Egyed. **Positive effects of utilizing relationships between inconsistencies for more effective inconsistency resolution**. In Richard N. Taylor, Harald Gall, and Nenad Medvidovic, editors, *ICSE*, pages 864–867. ACM, 2011. 26

[77] Ariadi Nugroho, Bas Flaton, and Michel R. V. Chaudron. **Empirical Analysis of the Relation between Level of Detail in UML Models and Defect Density**. In Krzysztof Czarnecki, Ileana Ober, Jean-Michel Bruel, Axel Uhl, and Markus Völter, editors, *MoDELS*, **5301** of *Lecture Notes in Computer Science*, pages 600–614. Springer, 2008. 30

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

[79] OMG. **MOF 2.4.1 Specification**. http://www.omg.org/spec/MOF/2.4.1/, 2012. xiii, 14, 15

[80] OMG. **OCL 2.3.1 Specification**. http://www.omg.org/spec/OCL/2.3.1/, 2012. 8, 16, 31

[81] OMG. **UML 2.1 Specification**. http://www.uml.org/, 2012. 8, 12

[82] Leon J. Osterweil, H. Dieter Rombach, and Mary Lou Soffa, editors. *28th International Conference on Software Engineering (ICSE 2006), Shanghai, China, May 20-28, 2006*. ACM, 2006. 35

[83] Charles Pecheur, Jamie Andrews, and Elisabetta Di Nitto, editors. *ASE 2010, 25th IEEE/ACM International Conference on Automated Software Engineering, Antwerp, Belgium, September 20-24, 2010*. ACM, 2010. 114, 116

[84] Michael Pilato. *Version Control With Subversion*. O'Reilly & Associates, Inc., Sebastopol, CA, USA, 2004. 30

[85] Dan Pilone and Neil Pitman. *UML 2.0 in a Nutshell (In a Nutshell (O'Reilly))*. O'Reilly Media, Inc., 2005. 11

[86] Derek Robert Price. **CVS Concurrent Versions System**. http://www.nongnu.org/cvs. 30

[87] Anna Queralt and Ernest Teniente. **Reasoning on UML Class Diagrams with OCL Constraints**. In David W. Embley, Antoni Olivé, and Sudha Ram, editors, *ER*, **4215** of *Lecture Notes in Computer Science*, pages 497–512. Springer, 2006. 31

[88] Alexander Reder. **Inconsistency management framework for model-based development.** In Richard N. Taylor, Harald Gall, and Nenad Medvidovic, editors, *ICSE*, pages 1098–1101. ACM, 2011. 9, 24

[89] Alexander Reder and Alexander Egyed. **Model/Analyzer: A Tool for Detecting, Visualizing and Fixing Design Errors in UML**. In Pecheur et al. [83], pages 347–348. 10, 24, 79

[90] Alexander Reder and Alexander Egyed. **Computing Repair Trees for Resolving Inconsistencies in Design Models**. In *ASE*. ACM, 2012. 4, 9, 10, 50, 67

[91] Alexander Reder and Alexander Egyed. **Determining the Cause of a Design Model Inconsistency**. *Software Engineering, IEEE Transactions on*, 2012. Major Revision. 10, 24, 62

[92] Alexander Reder and Alexander Egyed. **Incremental Consistency Checking for Complex Design Rules and Larger Model Changes**. In *Models*, Lecture Notes in Computer Science. Springer, 2012. 4, 5, 10, 27

[93] Steven P. Reiss. **Incremental Maintenance of Software Artifacts**. *IEEE Trans. Softw. Eng.*, **32**(9):682–697, 2006. 32

[94] Jason Elliot Robbins. *Cognitive Support Features for Software Development Tools*. PhD dissertation, University of California, Irvine, 1999. 32

[95] Jason Elliot Robbins. **ArgoUML, v0.32.1**. http://argouml.tigris.org/, March 2011. 4, 11, 32

[96] P. Rook. **Controlling software projects**. *Software Engineering Journal*, **1**(1):7, 1986. 1

[97] Winston W. Royce. **Managing the Development of Large Software Systems: Concepts and Techniques**. In *Technical Papers of Western Electronic Show and Convention (WesCon)*, 1970. 1

[98] Jan Scheffczyk, Peter Rödig, Uwe M. Borghoff, and Lothar Schmitz. **Managing inconsistent repositories via prioritized repairs**. In Ethan V. Munson and Jean-Yves Vion-Dury, editors, *ACM Symposium on Document Engineering*, pages 137–146. ACM, 2004. 34

[99] Ken Schwaber. **SCRUM Development Process**. In *Proceedings of the 10th Annual ACM Conference on Object Oriented Programming Systems, Languages, and Applications (OOPSLA*, pages 117–134, 1995. 2

[100] Bran Selic. **A Systematic Approach to Domain-Specific Language Design Using UML**. In *ISORC*, pages 2–9. IEEE Computer Society, 2007. 33

[101] George Spanoudakis and Andrea Zisman. **Inconsistency management in software engineering: Survey and open research issues**. In *in Handbook of Software Engineering and Knowledge Engineering*, pages 329–380. World Scientific, 2001. 29

[102] Ragnhild Van Der Straeten and Maja D'Hondt. **Model refactorings through rule-based inconsistency resolution**. In Hisham Haddad, editor, *SAC*, pages 1210–1217. ACM, 2006. 34

[103] Ragnhild Van Der Straeten, Tom Mens, Jocelyn Simmonds, and Viviane Jonckers. **Using Description Logic to Maintain Consistency between UML Models**. In Perdita Stevens, Jon Whittle, and Grady Booch, editors, *UML*, **2863** of *Lecture Notes in Computer Science*, pages 326–340. Springer, 2003. 32, 34

[104] Ragnhild Van Der Straeten, Jocelyn Simmonds, and Tom Mens. **Detecting Inconsistencies between UML Models Using Description Logic**. In Diego Calvanese, Giuseppe De Giacomo, and Enrico Franconi, editors, *Description Logics*, **81** of *CEUR Workshop Proceedings*. CEUR-WS.org, 2003. 11, 32

[105] Bjarne Stroustrup. *The C++ Programming Language, Third Edition*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 3rd edition, 1997. 30

[106] M. H. Van Emden and R. A. Kowalski. **The Semantics of Predicate Logic as a Programming Language**. *J. ACM*, **23**:733–742, October 1976. 33

[107] Michael Vierhauser, Deepak Dhungana, Wolfgang Heider, Rick Rabiser, and Alexander Egyed. **Tool Support for Incremental Consistency Checking on Variability Models**. In David Benavides, Don S. Batory,

and Paul Grünbacher, editors, *VaMoS*, **37** of *ICB-Research Report*, pages 171–174. Universität Duisburg-Essen, 2010. 88

[108] Michael Vierhauser, Paul Grünbacher, Alexander Egyed, Rick Rabiser, and Wolfgang Heider. **Flexible and scalable consistency checking on product line variability models**. In Pecheur et al. [83], pages 63–72. 9, 88

[109] Jessica Winkelmann, Gabriele Taentzer, Karsten Ehrig, and Jochen Malte Küster. **Translation of Restricted OCL Constraints into Graph Constraints for Generating Meta Model Instances by Graph Grammars**. *Electr. Notes Theor. Comput. Sci.*, **211**:159–170, 2008. 31

[110] Yingfei Xiong, Zhenjiang Hu, Haiyan Zhao, Hui Song, Masato Takeichi, and Hong Mei. **Supporting automatic model inconsistency fixing**. In Hans van Vliet and Valérie Issarny, editors, *ESEC/SIGSOFT FSE*, pages 315–324. ACM, 2009. 4, 11, 35, 104

[111] Yingfei Xiong, Arnaud Hubaux, Steven She, and Krzysztof Czarnecki. **Generating range fixes for software configuration**. In Martin Glinz, Gail C. Murphy, and Mauro Pezzè, editors, *ICSE*, pages 58–68. IEEE, 2012. 36

[112] Chang Xu, Shing-Chi Cheung, and W. K. Chan. **Incremental Consistency Checking for Pervasive Context.** In Leon J. Osterweil, H. Dieter Rombach, and Mary Lou Soffa, editors, *ICSE*, pages 292–301. ACM, 2006. 22, 33

[113] Andrea Zisman and Alexander Kozlenkov. **Knowledge Base Approach to Consistency Management of UML Specification**. In *ASE*, pages 359–363. IEEE Computer Society, 2001. 32

Appendix A

Constraints

Constraint 5 Parent Class should not have an Attribute referring to a Child Class

```
context Class:
    let children:Set(NamedElement) = self.namespace.oclAsType(Package).
        packagedElement->
        select(pe:PackageableElement|pe.oclIsTypeOf(Class) and
            pe.oclAsType(Class).allParents()->includes(self)) in
    self.ownedAttribute->forAll(p:Property|p.type.oclIsTypeOf(Class)
        implies
        children->excludes(p.type.oclAsType(Class)))
```

Constraint 6 Parent Class should not have an Operation with a Parameter referring to a Child Class

```
context Class:
    let children:Set(NamedElement) = self.namespace.oclAsType(Package).
        packagedElement->
        select(pe:PackageableElement|pe.oclIsTypeOf(Class) and
            pe.oclAsType(Class).allParents()->includes(self)) in
    self.ownedOperation->forAll(o:Operation|o.ownedParameter->
        forAll(p:Parameter|p.type.oclIsTypeOf(Class) implies
            children->excludes(p.type.oclAsType(Class))))
```

Constraint 7 Message Direction must match Class Association

```
context Message:
    self.receiveEvent.oclAsType(InteractionFragment).covered->
        exists(let rc:Class=represents.type.oclAsType(Class) in
            self.sendEvent.oclAsType(InteractionFragment).covered->
                exists(let sc:Class=represents.type.oclAsType(Class) in
                    sc.ownedAttribute->exists(association<>null implies
                        type=rc)))
```

Constraint 8 The connected Classifier of the Association End should be included in the Namespace of the Association

```
context Association:
    self.memberEnd<>null and self.memberEnd->
        forAll(p|p.type<>null and p.type.namespace=self.namespace)
```

Constraint 9 AssociationEnds must have unique Names within the Association

```
context Association:
    self.memberEnd->forAll(p1,p2:Property|p1<>p2 implies p1.name<>p2.name
        )
```

Constraint 10 At most one AssociationEnd may be an Aggregation or Composition

```
context Association:
    self.memberEnd->size()>0 implies
        self.memberEnd->select(p|p.aggregation<>AggregationKind::none)->
            size()<=1
```

Constraint 11 A Class may use Unique Attribute Names

```
context Class:
    self.ownedAttribute->forAll(p1,p2:Property|p1<>p2 implies p1.name<>p2
        .name)
```

Constraint 12 A Classifier may not belong by Composition to more than one Composite Classifier

```
context Property:
    (self.association<>null and self.aggregation=AggregationKind::
        composite) implies
        (self.upper>=0 and self.upper<=1)
```

Constraint 13 The Elements owned by a Namespace must have unique Names

```
context Package:
    self.packagedElement->forAll(e1,e2:PackageableElement|(e1<>e2)
        implies (e1.name<>e2.name))
```

Constraint 14 An Interface can only contain Public Operations and no Attributes

```
context Interface:
    self.ownedAttribute->forAll(pr:Property|pr.association<>null or
        pr.visibility=VisibilityKind::public) and
        self.ownedOperation->forAll(o:Operation|o.visibility=
            VisibilityKind::public)
```

Constraint 15 No two Class Operations may have the same Signature

```
context Class:
    self.ownedOperation->forAll(o1,o2:Operation|o1<>o2 implies
        (o1.name <> o2.name or o1.ownedParameter->size() <> o2.
            ownedParameter->size() or
            let ops1:Collection(Type)=o1.ownedParameter->collect(type) in
            let ops2:Collection(Type)=o2.ownedParameter->collect(type) in
                ops1->exists(t:Type|ops2->excludes(t))or ops2->exists(t:
                    Type|ops1->excludes(t))))
```

Constraint 16 Operation Parameters must have unique Names

```
context Operation:
    self.ownedParameter->forAll(p1,p2:Parameter|p1<>p2 implies p1.name<>
        p2.name)
```

Constraint 17 The Type of Operation Parameters must be included in the Namespace of the Operation Owner

```
context Operation:
    self.ownedParameter->forAll(p:Parameter|p.type<>null implies
        p.type.namespace=self.owner.oclAsType(Class).namespace)
```

Constraint 18 The Parent must be included in the Namespace of the Generalizable Element

```
context Generalization:
    self.source->forAll(e1:Element|e1.oclIsKindOf(NamedElement) implies
        self.target->forAll(e2:Element|e2.oclIsKindOf(NamedElement) and
        e1.oclAsType(NamedElement).namespace = e2.oclAsType(NamedElement)
            .namespace))
```

Constraint 19 No circular Inheritance allowed

```
context Class:
    not self.allParents()->includes(self)
```

Constraint 20 An Operation has at most one return Parameter

```
context Operation:
    self.ownedParameter->select(p:Parameter|p.direction=
        ParameterDirectionKind::return)->size()<=1
```

Curriculum Vitae

Dipl.-Ing. Alexander Reder

Contact

University	Johannes Kepler University, Linz
Department	Institute for Systems Engineering and Automation
Phone	+43 732 2468 4394
e-Mail	mailto:alexander.reder@jku.at
web	http://www.sea.jku.at

Education

10/2009-Present	PhD Student, Johannes Kepler University, Linz, Computer Science. *Automated Consistency Management Framework for the Model Based Software Development*
10/2007-09/2009	Dipl.-Ing., Johannes Kepler University, Linz, Software Engineering. *Model Based Productline Engineering: Variability Modeling using the Eclipse Modeling Framework and the Object Constraint Language*
10/2004-09/2007	B.Sc., Johannes Kepler University, Linz, Bachelor Study Computer Science.

Work Experience

04/2009-Present	Researcher, Johannes Kepler University, Linz. *Consistency Management in Model-Based Software Development*
10/2004-02/2012	Software Engineer, Self-employed, Linz. *Development and support of document output solutions*
04/1998-09/2004	Service Support Engineer, Canon GmbH, Linz. *Service, support and installation of document output solutions*

Teaching Experience

Winter Semester 2010/2011	Johannes Kepler University, Linz. *Präsentations und Arbeitstechnik*
Winter Semester 2011/2012	Johannes Kepler University, Linz. *Präsentations und Arbeitstechnik*
Summer Semester 2012	Johannes Kepler University, Linz. *Engineering of Software-Intensive Systems*
Winter Semester 2012/2013	Johannes Kepler University, Linz. *Präsentations und Arbeitstechnik*
Winter Semester 2012/2013	Johannes Kepler University, Linz. *Software Engineering Übung*

Language

German	Mother tongue
English	Proficiency
Spanish	Basic knowledge