Universität Paderborn

Fachbereich 17 - Mathematik/Informatik
Arbeitsgruppe Softwaretechnik
Warburger Straße 100
33098 Paderborn

Realisierung eines diagrammübergreifenden Konsistenzmanagement-Systems für UML-Spezifikationen

Diplomarbeit

für den integrierten Studiengang Informatik
im Rahmen des Hauptstudiums II

Robert Wagner
Am Coesfeld 1
33334 Gütersloh

vorgelegt bei
Prof. Dr. Wilhelm Schäfer

und
Prof. Dr. Gregor Engels

Paderborn, im November 2001

Eidesstattliche Erklärung

Ich versichere, dass ich die Arbeit ohne fremde Hilfe und ohne Benutzung anderer als der angegebenen Quellen angefertigt habe, und dass die Arbeit in gleicher oder ähnlicher Form noch keiner anderen Prüfungsbehörde vorgelegen hat und von dieser als Teil einer Prüfungsleistung angenommen wurde. Alle Ausführungen, die wörtlich oder sinngemäß übernommen wurden, sind als solche gekennzeichnet.

Ort, Datum	Unterschrift

Inhaltsverzeichnis

Abbildungsverzeichnis

1 Motivation

1.1 Integrierter Softwareentwurf

An der Erstellung von großen und komplexen Softwaresystemen sind viele Entwickler aus unterschiedlichen Disziplinen beteiligt. Jede Disziplin beschreibt Teile des Softwaresystems aus verschiedenen Perspektiven und verwendet dazu eine bevorzugte Spezifikationssprache. Die Gesamtspezifikation eines Projektes besteht somit aus unterschiedlichen, sich zum Teil überlappenden Beschreibungen, die nicht nur Redundanzen sondern auch widersprüchliche Aussagen über das Softwaresystem enthalten können (siehe Abbildung 1.1).

Konsistente Spezifikationen sind aber eine grundlegende Voraussetzung für fehlerfreie Software. Eine manuelle Überprüfung der Widerspruchsfreiheit zwischen den Beschreibungen eines Softwareentwurfs durch die Softwareentwickler ist aber nicht nur mühsam und fehleranfällig, sondern auch zeitintensiv und damit unwirtschaftlich. Erschwerend kommt hinzu, dass die Spezifikation und Modellierung von einem Entwicklungsteam durchgeführt wird, dessen Mitglieder parallel an der Erstellung des Systems arbeiten, wodurch die Spezifikation ständigen Änderungen unterliegt.

Um die diagrammübergreifende Konsistenzsicherung zu ermöglichen, müssen die verwendeten Spezifikationssprachen integriert werden. Ein dabei häufig gewählter Ansatz ist die Kombination der unterschiedlichen Spezifikationstechniken auf der Basis eines einheitlichen Metamodells. Ein bekanntes Beispiel hierfür ist die *Unified Modeling Language* (UML) [UML99].

Im Rahmen des ISILEIT[1]-Projektes, das am Lehrstuhl für Softwaretechnik der Universität Paderborn durchgeführt wird, erfolgt die Spezifikation von verteilten Fertigungssystemen in der integrierten Entwicklungsumgebung FUJABA[2] mit ausgewählten Diagrammen der UML und der *Specification and Description Language* (SDL) [SDL96].

1. Integrative Spezifikation von verteilten Leitsystemen der flexibel automatisierten Fertigung, http://www.upb.de/cs/isileit
2. FROM UML TO JAVA AND BACK AGAIN, http://www.fujaba.de

Die SDL wird insbesondere in der Telekommunikationsbranche zur Spezifikation von verteilten, eingebetteten und asynchron kommunizierenden Systemen eingesetzt. Sie hat sich in den vergangenen Jahren zu einem weit verbreiteten und akzeptierten Industriestandard etabliert. Zur Modellierung komplexer Objektstrukturen hat sich die UML durchgesetzt, die insbesondere in den Phasen der Anforderungsanalyse und des objektorientierten Softwareentwurfs verwendet wird. Um die Vorteile beider Spezifikationssprachen nutzen zu können, wurden SDL-Blockdiagramme im Rahmen des ISILEIT-Projektes in FUJABA integriert und ermöglichen zusammen mit den Diagrammen der UML eine durchgängige Spezifikation von Fertigungssystemen.

Das Fertigungssystem besteht aus dezentralen, autonom agierenden Komponenten, die zur Bearbeitung von Fertigungsaufträgen untereinander kommunizieren. Zur Modellierung und Spezifikation der Kommunikationsstruktur werden daher SDL-Blockdiagramme eingesetzt. Die aktiven Komponenten eines Systems werden dabei als Prozesse modelliert und kommunizieren über das Versenden von Signalen entlang von Kommunikationskanälen.

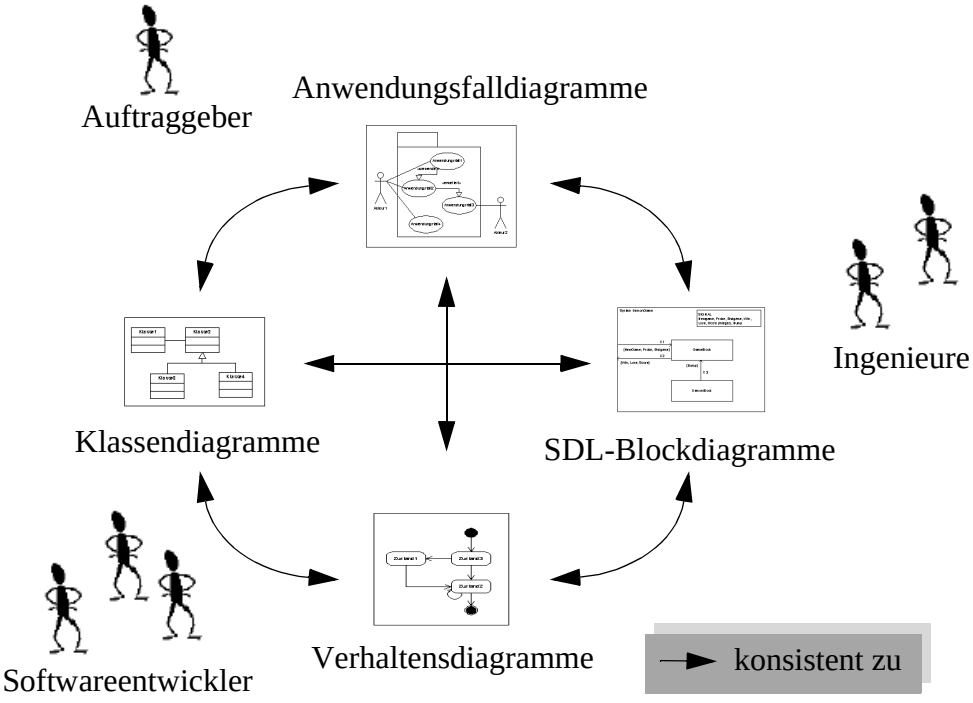


Abbildung 1.1: Integrierter Softwareentwurf

Allgemein werden in der UML zur Anforderungsanalyse Anwendungsfalldiagramme (Use Case Diagrams) verwendet. Die damit spezifizierten Anforderungen an

die zu erstellende Software werden durch weitere Diagrammarten verfeinert. So werden die einzelnen Anwendungsfälle beispielsweise durch Aktivitätsdiagramme näher spezifiziert.

Das Verhalten der Prozesse wird durch eine Kombination aus Zustandsdiagrammen, Aktivitätsdiagrammen und Kollaborationsdiagrammen beschrieben. Die Objektstrukturen hingegen werden mit Klassendiagrammen spezifiziert. Dazu wird das SDL-Blockdiagramm auf ein Klassendiagramm abgebildet, das weiter verfeinert werden kann.

Die durch die Anwendungsdomäne festgelegte Semantik ermöglicht eine durchgängige Methodik für den interdisziplinären Entwurf eines Softwaresystems. Durch den integrierten Entwurf wird die Überlappung von redundanten Informationen verringert und damit das Konsistenzproblem reduziert. Dennoch bestehen zwischen den Diagrammen weiterhin Abhängigkeiten (siehe Abbildung 1.1), so dass Inkonsistenzen entstehen können. So müssen beispielsweise die in einem Zustandsdiagramm spezifizierten Operationen und Signale als Methoden in der Klassendefinition auftauchen. Ebenso können unvollständige oder fehlerhafte Spezifikationen innerhalb eines Diagramms zu mehrdeutigen Interpretationen bei der Implementierung des Systems führen. Es ist offensichtlich, dass hier weiterhin Handlungsbedarf besteht.

Das Ziel dieser Diplomarbeit ist daher die Erarbeitung eines Konzeptes zur diagrammübergreifenden Konsistenzprüfung und Konsistenzerhaltung, das in der integrierten Entwicklungsumgebung FUJABA realisiert werden soll. Hierzu soll mit den bereits bestehenden Formalismen die Spezifikation von statischen Konsistenz- und Integritätsbedingungen auf dem Metamodell ermöglicht und ein Konsistenzmanagement-System erstellt werden, welches die Bedingungen während der Modellierung überwacht und Konsistenzverletzungen mitprotokolliert. Darüber hinaus soll das System Reparaturen soweit möglich und sinnvoll automatisch durchführen oder dem Benutzer zumindest Vorschläge zur Behebung unterbreiten. Zusätzlich sollen unvollständige Spezifikationen um die fehlenden Diagramme beziehungsweise Diagrammelemente automatisch ergänzt werden, womit auch eine Transformation von Diagrammen in andere Beschreibungsformen ermöglicht werden soll.

1.2 Das Konsistenzproblem

Unter *Konsistenz* verstehen wir die Widerspruchsfreiheit sowohl zwischen als auch innerhalb von Beschreibungen. Zwei Beschreibungen sind *konsistent* zueinander, wenn die Aussagen bezüglich der Struktur oder des Verhaltens einer Software im Einklang sind. Dementsprechend bedeutet *Inkonsistenz*, dass ein Widerspruch vorliegt.

Ursachen für Inkonsistenzen können *falsche* oder *unvollständige* Spezifikationen sein. Während eine unvollständige Spezifikation eventuell um die fehlenden Elemente automatisch ergänzt werden kann, ist eine automatische Behebung bei falschen Spezifikationen nicht immer möglich oder sinnvoll, da das System die Semantik eines Modells nicht hinreichend verstehen kann. Generell kann jede Situation als inkonsistent bezeichnet werden, in der Beschreibungen eine geforderte Beziehung nicht einhalten [NKF84].

Konsistenzverletzungen können sowohl syntaktischer als auch semantischer Natur sein. Syntaktische Konsistenzverletzungen betreffen die dem Modell zugrundeliegende abstrakte Syntax, die zum Beispiel in der UML durch das Metamodell und die *Well-Formedness Rules* [UML99] definiert ist. Semantische Konsistenzverletzungen hingegen ergeben sich unter anderem, wenn spezifizierte Modelle widersprüchliche und nicht vereinbare Aussagen bezüglich des Verhaltens eines Systems machen. Semantische Konsistenz ist jedoch schwer überprüfbar, da die Semantik von Modellen oft nicht präzise und formell genug definiert ist. Dies trifft zum Beispiel auf die UML zu [EGHK01].

Des weiteren wird unterschieden, ob Inkonsistenzen innerhalb eines Modells oder zwischen mehreren Modellen bestehen. Die Konsistenz innerhalb eines Modells wird mit *Intra-Modell-Konsistenz* bezeichnet, die Konsistenz zwischen mehreren Modellen mit *Inter-Modell-Konsistenz* [Gli00].

Bei der *Inter-Modell-Konsistenz* wird zusätzlich zwischen der *horizontalen Konsistenz* und der *vertikalen Konsistenz* differenziert [EGHK01]. Die horizontale Konsistenz muss insbesondere dann eingehalten werden, wenn ein System aus verschiedenen Perspektiven modelliert wird und die verwendeten Teilmodelle sich hinsichtlich der dargestellten Informationen überlappen. Beispielsweise stellen Sequenzdiagramme und Kollaborationsdiagramme die Interaktionen eines Systems aus unterschiedlichen Sichten dar. Während in einem Sequenzdiagramm die Interaktionen unter dem zeitlichen Gesichtspunkt betrachtet werden, erfolgt die Modellierung des Kollaborationsdiagramms unter dem Aspekt der Objektstruktur. Hierbei muss sichergestellt werden, dass die Beschreibungen widerspruchsfrei sind.

Mit der vertikalen Konsistenz wird das Problem bezeichnet, das auftaucht, wenn ein Diagramm eine Verfeinerung eines anderen Diagramms ist. Mit der Verfeinerung können sich widersprüchliche Aussagen ergeben, die mit dem ursprünglichen Diagramm nicht vereinbar sind.

1.3 Aspekte der Konsistenzprüfung

In diesem Abschnitt werden einige Aspekte zur Konsistenzprüfung betrachtet. Konsistenzprüfungen können unter verschiedenen Gesichtspunkten erfolgen. Aus diesem Grund werden im Folgenden die verwendete Spezifikationssprache, der Zeitpunkt der Konsistenzprüfung und mögliche Maßnahmen bei Feststellung von Inkonsistenzen näher beleuchtet.

1.3.1 Spezifikationssprache für Konsistenzbedingungen

Die Spezifikation von Konsistenzbedingungen kann auf unterschiedliche Art und Weise erfolgen. Grundsätzlich wird zwischen deklarativen Spezifikationssprachen und prozeduralen Sprachen unterschieden. Eine deklarative Sprache beschreibt, was zu überprüfen ist. Im Gegensatz dazu spezifiziert eine prozedurale Sprache, wie etwas geprüft werden soll.

Weiterhin ist noch zu unterscheiden, ob mit der Sprache ein konsistenter oder inkonsistenter Zustand beschrieben wird [GMT99]. Beispielsweise ist eine statische Integritätsbedingung für eine Assoziation im UML-Metamodell [UML99], dass sie mindestens zwei Assoziationsenden besitzen muss. Die Konsistenz kann überprüft werden, indem die Bedingung formuliert wird, dass die Anzahl der an der Assoziation beteiligten Assoziationsenden größer oder gleich zwei beträgt. Ist diese Bedingung erfüllt, so liegt Konsistenz vor. Ebenso kann aber auch der inkonsistente Zustand beschrieben werden, indem spezifiziert wird, dass die Anzahl der Assoziationsenden kleiner als zwei ist. Ist diese Bedingung erfüllt, so liegt eine Inkonsistenz vor.

Sowohl die Spezifikation als auch die Überprüfung können unterschiedlichen Aufwand verursachen. Daher ist es vorteilhaft, wenn die Spezifikationssprache beide Möglichkeiten unterstützt und der Benutzer die am besten geeignete Beschreibungsform auswählen kann.

1.3.2 Überprüfungszeitpunkt

Ein wesentlicher Aspekt der Konsistenzprüfung ist der Überprüfungszeitpunkt. Es stehen vier Möglichkeiten zur Verfügung, wann eine Konsistenzüberprüfung stattfinden kann.

Die erste Möglichkeit besteht darin, eine Konsistenzprüfung nebenläufig in unbestimmten Zeitintervallen durchzuführen. Dabei wird das Modell auch dann überprüft,

wenn zwischen den Prüfungen keine Veränderungen am Modell vorgenommen wurden.

Die zweite Möglichkeit ist, eine Konsistenzprüfung durchzuführen, sobald eine Änderung an einem Modellelement vorgenommen wurde. Hierdurch erreicht man eine ständige Überprüfung der Konsistenz.

Eine andere Variante führt nicht nach jeder Änderung eine Konsistenzprüfung durch, sondern wartet bis eine gewisse Anzahl von Änderungen stattgefunden hat. Dies kann insbesondere bei komplexeren Bearbeitungen und Modifikationen des Modellelements sinnvoll sein, bei denen ein konsistenter Zustand erst nach mehreren Bearbeitungsschritten erreicht werden kann. Eine Konsistenzprüfung nach jeder Modifikation hätte in diesem Fall den Bearbeitungsvorgang unnötig erschwert beziehungsweise verzögert.

Die vierte Möglichkeit bezüglich des Zeitpunktes der Überprüfung überläßt dem Benutzer die Entscheidung, wann er eine Konsistenzprüfung durchführen möchte. Der Benutzer der Entwicklungsumgebung ruft dazu ein speziell ausgezeichnetes Kommando auf, mit dem er die Konsistenzprüfung startet.

1.3.3 Maßnahmen beim Auftreten von Inkonsistenzen

Tauchen in einem Projekt Inkonsistenzen auf, so gibt es verschiedene Möglichkeiten mit ihnen umzugehen. Inkonsistenzen können entweder behoben, ignoriert oder toleriert werden [NER00]. Werden Inkonsistenzen toleriert, so kann man weiterhin unterscheiden, ob sie nur verzögert und zu einem späteren Zeitpunkt behandelt werden, oder ob man die Inkonsistenz nur partiell verbessert und sich damit einer konsistenten Lösung nähert. Zusätzlich können Konsistenzen vereitelt werden, indem zum Beispiel Ausnahmen von der Regel zugelassen werden. Das Vereiteln von Inkonsistenzen bedeutet, dass die Konsistenzbedingung entweder ganz ausser Kraft gesetzt wird oder nur in einer abgeschwächten Form überprüft wird. In dem zuletzt genannten Fall muss die Spezifikation der Bedingung geändert werden.

In diesem Zusammenhang stellt sich noch die Frage, wie Inkonsistenzen dem Benutzer präsentiert werden. Das Einblenden von speziellen Symbolen in den betroffenen Diagrammen ist eine Möglichkeit, um den Benutzer auf vorhandene Konsistenzverletzungen aufmerksam zu machen. Eine weitere Variante ist die Darstellung inkonsistenter Modellelemente in einer anderen Farbe. Dies kann entweder automatisch erfolgen oder aber auf Anfrage. Darüberhinaus sollte dem Benutzer die Möglichkeit gegeben werden, Anfragen an das Konsistenzmanagement-System zu stellen, um zum Beispiel alle Inkonsistenzen eines speziellen Elementes zu erfragen.

1.4 Anforderungen

Konsistenzverletzungen treten in vielfältiger Weise auf und sind, insbesondere zwischen unterschiedlichen Diagrammen, für den Entwickler nicht immer sofort zu erkennen. Die manuelle Behebung von Inkonsistenzen durch den Softwareentwickler ist nicht nur fehleranfällig, sondern auch zeitaufwändig. Daher ist es wünschenswert, dass möglichst viele Konsistenzverletzungen durch die Entwicklungsumgebung automatisch erkannt, verwaltet und gegebenenfalls auch automatisch behoben werden. In diesem Abschnitt werden die Anforderungen an ein solches Konsistenzmanagement-System formuliert.

1.4.1 Anforderungen an den Überprüfungsmechanismus

Eine notwendige Bedingung für das Management von Konsistenzverletzungen ist deren Erkennung. Hierzu muss das Modell untersucht werden. Komplexe Software bedingt aber meistens sehr grosse Modelle. Daher kann eine Konsistenzprüfung sehr aufwändig sein und den Benutzer in seiner Arbeit mit der Entwicklungsumgebung beeinträchtigen. Aus diesem Grund sollte eine Prüfungsstrategie verfolgt werden, die diesen Aufwand minimiert. Dies kann beispielsweise durch eine inkrementelle Prüfung erreicht werden, die nur die von einer Änderung betroffenen Elemente überprüft und somit nur ein Teilmodell untersucht.

Konsistenzprüfungen innerhalb einer Entwicklungsumgebung sollen sowohl automatisch als auch auf Anforderung durchgeführt werden können. Beide Strategien haben ihre Berechtigung. So ist es durchaus sinnvoll, zu Beginn einer Spezifikation die automatische Konsistenzprüfung auszuschalten, um nicht von Fehlermeldungen überflutet zu werden, die sich aus dem noch unvollständigen Modell ergeben. Zu einem vom Benutzer bestimmten Zeitpunkt kann dann eine Konsistenzprüfung durchgeführt und die erkannten Inkonsistenzen behoben werden. Ein weiterer Vorteil von Konsistenzprüfungen auf Anforderung ist, dass sie nicht ständig durchgeführt werden und somit auch keine Ressourcen verbrauchen. Dies ist insbesondere bei selten gebrauchten oder sehr aufwändigen Prüfungen von Nutzen.

1.4.2 Anforderungen an die Spezifikation von Konsistenzbedingungen

Die Basis für ein Konsistenzmanagement-System bilden Konsistenzregeln. Sie beschreiben, welche Bedingungen in einem Diagramm gelten müssen und ermöglichen damit eine automatische Überprüfung durch die Entwicklungsumgebung. Bevor eine Überprüfung stattfinden kann, müssen diese Regeln jedoch zuerst identifiziert und spezifiziert werden.

Ein Konsistenzmanagement-System muss das Hinzufügen, Ändern und Löschen von Konsistenzregeln erlauben. Die Gründe, die zu dieser Anforderung führen, sind unterschiedlich. Zum einen ergibt sich diese Anforderung aus der Tatsache, dass Konsistenzbedingungen nicht vollständig im Vorfeld der Entwicklung eines solchen Systems identifiziert werden können. Dies liegt unter anderem an der Komplexität des zugrundeliegenden Metamodells. So besteht beispielsweise der abstrakte Syntaxgraph (ASG) von FUJABA aus ca. 90 Klassen und über 100 Assoziationen [FNT98]. Damit können Konsistenzbedingungen leicht übersehen werden. Selbst die UML-Spezifikation [UML99] enthält nicht alle Konsistenzbedingungen und erlaubt damit inkonsistente Modelle.

Ein anderer Grund liegt in der ständigen Anpassung und Erweiterung der Entwicklungsumgebung selbst. Durch die Integration von neuen Beschreibungselementen und Diagrammen ergeben sich auch neue Bedingungen, um die das Konsistenzmanagement-System ergänzt werden muss. Durch die Änderungen am Metamodell und der enthaltenen Metamodellelemente können neue Bedingungen hinzukommen und bestehende Bedingungen sich ändern oder ganz entfallen.

Ein weiteres Argument für eine modulare und veränderbare Regelmenge ergibt sich aus der Tatsache, dass unterschiedliche Anwendungsdomänen auch unterschiedliche Konsistenzbedingungen an die Modelle stellen. So ist es vorstellbar, dass die im ISILEIT-Projekt formulierten Konsistenzbedingungen in einer anderen Anwendungsdomäne nicht mehr gelten oder um zusätzliche Bedingungen ergänzt werden müssen. So unterliegen beispielsweise eingebettete Systeme oft hardwarespezifischen Einschränkungen, die sich auf die Modellierung auswirken und eingehalten werden müssen. Ein Beispiel hierfür ist die Forderung, dass alle zur Laufzeit benötigten Objekte bereits nach der Initialisierung erzeugt sein müssen. Dies erfordert eine neue Konsistenzbedingung, die das Erzeugen von Objekten innerhalb von Kollaborations- und Sequenzdiagramm verbietet.

1.4.3 Anforderungen an die Benutzerschnittstelle

Das Hauptziel ist die Realisierung eines Konsistenzmanagement-Systems, das den Entwickler bei der Erkennung und Beseitigung von Konsistenzverletzungen unterstützt. Damit der Entwickler sich auf seine Hauptaufgabe, nämlich die Spezifikation des Softwaresystems, konzentrieren kann, sollte das Konsistenzmanagement-System möglichst automatisch agieren und den Benutzer nicht unnötig bei seiner Arbeit stören.

Andererseits darf man aber auch nicht vergessen, dass ein Konsistenzmanagement-System ein Hilfswerkzeug für den Entwickler darstellt und ihn daher auch nicht bevormunden darf. Ebenso muss ein „Leben mit Inkonsistenzen“ erlaubt sein [Bal91]. Vor diesem Hintergrund wurden auch die nachfolgenden Anforderungen an die Benutzerschnittstelle formuliert.

Neben der Anzeige und Verwaltung der erkannten Inkonsistenzen muss ein Konsistenzmanagement-System konfigurierbar und an die Anforderungen des Benutzers anpassbar sein.

Beispielsweise ist der vom Anwender gewünschte formale Spezifikationsgrad seiner Modelle ein Aspekt, der vom Konsistenzmanagement-System beachtet werden sollte. Die Erstellung eines konzeptionellen Modells erfolgt auf einem wesentlich höheren Abstraktionsniveau, bei dem viele Details während der Modellierung vernachlässigt werden. Dadurch entsteht unter Umständen ein unvollständiges Modell. Eine Konsistenzprüfung würde hier sehr viele Fehler erkennen, für die sich der Entwickler jedoch nicht interessiert. Daher sollte dem Entwickler die Möglichkeit gegeben werden, die automatische Konsistenzprüfungen auszuschalten.

Ebenso sollte eine Deaktivierung der automatischen Fehlerkorrektur unterstützt werden. Das ist insbesondere dann sinnvoll, wenn vorhandene Modelle untersucht werden sollen. So kann man in FUJABA vorhandenen JAVA-Quellcode einlesen und die verwendeten Klassen, ihre Attribute, Methoden und Beziehungen in einem Klassendiagramm darstellen. Eine automatische Korrektur würde diese Struktur, sofern sie nicht den Bedingungen entspricht, verändern und damit das zu analysierende Modell verfälschen. Ein Round-Trip Engineering sollte, sofern vom Benutzer keine Änderungen am Modell vorgenommen worden sind, auch wieder zum ursprünglichen Quellcode führen.

1.5 Aufbau der Diplomarbeit

In der Softwaretechnik existieren zu verschiedenen Konsistenzproblemen unterschiedliche Lösungsansätze, die sich mit einem spezifischen Teilaspekt der Konsistenz innerhalb oder zwischen unterschiedlichen Diagrammen beschäftigen. Ein genereller Ansatz, der die Gesamtproblematik berücksichtigt, existiert jedoch nicht. In Kapitel 2 erfolgt daher eine kurze Darstellung ausgewählter Arbeiten zur Konsistenzprüfung. Daneben werden zwei Spezifikationssprachen zur Formulierung von Konsistenzbedingungen und die bereits realisierten Konsistenzanalysen in FUJABA vorgestellt.

Das Kapitel 3 befasst sich mit der Spezifikation von Konsistenzbedingungen innerhalb des abstrakten Syntaxgraphen von FUJABA. Dabei werden zur diagrammübergreifenden Konsistenzsicherung Tripel-Graph-Grammatiken und das in der AG Softwaretechnik der Universität Paderborn entwickelte und auf Graphgrammatiken basierende *Story Driven Modeling* vorgestellt.

Nach der Spezifikation von Konsistenzbedingungen werden in Kapitel 4 die verwendeten Konzepte zur Verwaltung und Überprüfung der Konsistenzregeln sowie der Behandlung von Inkonsistenzen erläutert.

Die technische Realisierung und Umsetzung der zuvor vorgestellten Konzepte wird in Kapitel 5 beschrieben und anhand eines Beispiels in Kapitel 6 verdeutlicht.

In Kapitel 7 erfolgt eine Zusammenfassung der erzielten Ergebnisse. Darüber hinaus werden offene Probleme und ein Ausblick auf noch mögliche und sinnvolle Ergänzungen des Konsistenzmanagement-Systems gegeben.

Den Abschluß der Arbeit bilden die spezifizierten Konsistenzregeln, die im Anhang A präsentiert werden.

2 Existierende Ansätze

In diesem Kapitel wird eine Auswahl von Arbeiten zu dem Thema der Konsistenz präsentiert. In Abschnitt 2.1 wird ein Framework für das (In-)Konsistenzmanagement vorgestellt, in dem verschiedene Möglichkeiten zum Umgang mit Inkonsistenzen aufgezeigt werden. Der nachfolgende Abschnitt 2.2 beschäftigt sich mit zwei speziellen Konsistenzprüfungsproblemen und deren Lösungen. Dabei handelt es sich bei dem ersten Ansatz um die horizontale Konsistenz zwischen Klassen- und Sequenzdiagrammen. Die zweite Arbeit beschäftigt sich mit der vertikalen Konsistenz zwischen Diagrammverfeinerungen. Im darauf folgenden Abschnitt 2.3 werden Konsistenzanalysen der Entwicklungsumgebung FUJABA vorgestellt und die damit verbundenen Nachteile aufgezeigt. Schließlich widmet sich der Abschnitt 2.4 zwei möglichen Spezifikationssprachen zur Formulierung von Konsistenzbedingungen.

2.1 Ein Framework für das Konsistenzmanagement

Während der Entwicklung von Software entstehen viele unterschiedliche Dokumente. Diese Dokumente können natürlichsprachlich oder auch in Form von formalen Spezifikationen vorliegen und verschiedene Abstraktionsgrade aufweisen. Das Überprüfen von Konsistenzbedingungen zwischen diesen Dokumenten ist ein schwieriges Problem, da die Menge von Beschreibungen stetig wächst und ständigen Änderungen unterzogen wird. Während es viele Arbeiten zu speziellen Teilproblemen gibt, die sich beispielsweise mit der Überprüfung von einzelnen Dokumenten beschäftigen [GN98, GN99], existiert kein genereller Ansatz, wie mit inkonsistenten Spezifikationen umgegangen werden soll und wie eine Unterstützung durch Werkzeuge aussehen könnte. Die meisten existierenden Werkzeuge erwarten, sofern sie überhaupt eine Konsistenzüberprüfung anbieten, dass Inkonsistenzen vom Benutzer nach ihrer Erkennung sofort aufgelöst werden. Dieses ist aber nicht immer sinnvoll, da die Auflösung einer Inkonsistenz weitere Konsistenzverletzungen in anderen Teilen der Spezifikation nach sich ziehen kann. Nach Nuseibeh et. al [NER00] ist daher eine vollkommene Widerspruchsfreiheit zwischen den einzelnen Dokumenten sogar nur mit großem Aufwand zu erreichen, der in der Praxis in keinem Verhältnis zum Nutzen

steht. Daher schlagen Nuseibeh et al. in ihrer Arbeit [NER00] ein Framework vor, das die Erkennung von Inkonsistenzen unterstützt und dem Entwickler viele Freiheiten im Umgang mit Inkonsistenzen gewährt. Abbildung 2.1 zeigt das dort vorgeschlagene Framework.

Abbildung 2.1: Framework, entnommen aus [NER00]

Den zentralen Ausgangspunkt für das Framework bilden die Konsistenzregeln, die verschiedene Beschreibungen auf ihre Konsistenz prüfen. Da die Regeln nie vollständig und im Vorfeld eines großen Softwareprojekts definiert werden können, ist es möglich, die Konsistenzregeln dynamisch zu erweitern, zu verfeinern und an neue Anforderungen anzupassen.

Wird eine Inkonsistenz gefunden, so erfolgt eine Diagnose, in der die inkonsistenten Stellen lokalisiert werden und der Grund für die Inkonsistenz festgestellt wird. Ebenso erfolgt hier auch eine Klassifizierung der festgestellten Inkonsistenz, da der Grad und die Art der Konsistenzverletzung als Entscheidungshilfe für den Entwickler dienen kann, ob überhaupt, und wenn ja, wann die gefundene Konsistenzverletzung aufgelöst wird.

Nach der Charakterisierung der Konsistenzverletzung stehen dem Entwickler verschiedene Möglichkeiten zur Behandlung der Inkonsistenz zur Verfügung. Grundsätzlich kann die Inkonsistenz behoben, vollkommen ignoriert oder für eine bestimmte Zeit toleriert werden. Die Wahl der Behandlungsroutine beim Auftreten einer Inkonsistenz ist abhängig von dem Kontext und dem Einfluß der Konsistenzverletzung auf das Gesamtprojekt. Die Auflösung der Konsistenzverletzung kann in einigen Fällen sehr einfach durch das Hinzufügen oder Entfernen von Informationen zu der Beschreibung durchgeführt werden. Oft ergeben sich daraus aber weitere Inkonsistenzen oder grundlegende Konflikte, so dass Entwurfsentscheidungen neu über-

dacht und revidiert werden müssen. In solchen Fällen ist eine sofortige Auflösung der Inkonsistenz nicht immer die beste Wahl.

So steht in einigen Fällen der Aufwand, den eine Beseitigung der Konsistenzverletzung verursacht, in keinem Verhältnis zum Nutzen. Ist zum Beispiel das Risiko sehr gering, dass die Inkonsistenz nachteilige Auswirkungen auf das Projekt und die spezifizierte Software hat, so ist es vorteilhafter, die Inkonsistenz gänzlich zu ignorieren.

Eine Aufschiebung der Behandlung von Konsistenzverletzungen verschafft dem Entwickler mehr Zeit, um die Ursachen für die Inkonsistenz zu analysieren. Inkonsistenzen entstehen aber auch durch unvollständige Beschreibungen, die sich im Verlauf eines Projektes durch das Hinzufügen und Verfeinern von Informationen auflösen und nicht speziell behandelt werden müssen. Gerade wenn viele Softwareentwickler am Entwicklungsprozess beteiligt sind und Teilspezifikationen sich erst im Nachhinein zu einer grossen und vollständigen Beschreibung des Systems zusammensetzen, erweist sich die Möglichkeit, die Behandlung von Inkonsistenzen auf einen späteren Zeitpunkt zu verschieben, als sehr nützlich.

Nach Nuseibeh [NER00] werden die von einem Werkzeug gemeldeten Inkonsistenzen von Entwicklern nicht immer auch als solche akzeptiert. Dies kann daran liegen, dass Regeln falsch spezifiziert wurden oder aber, dass die Inkonsistenz eine Ausnahmesituation bezüglich der spezifizierten Regel darstellt. Daher sieht das Framework auch die Möglichkeit vor, Inkonsistenzen zu vereiteln und inkonsistente Spezifikationen als konsistent zu markieren. Dies kann zum Beispiel durch das Deaktivieren von Regeln für einen speziellen Kontext geschehen.

Da in vielen Fällen eine vollständige Auflösung der Konsistenzverletzung weitreichende Folgen für das Gesamtprojekt haben kann und unter Umständen sogar nachteilige Konsequenzen mit sich bringt, ist die schrittweise Verbesserung von Konsistenzverletzungen oft ein geeignetes Mittel, um einer konsistenten Gesamtspezifikation näher zu kommen, ohne Inkonsistenzen wirklich aufzulösen. Eine Verbesserung der Situation kann zum Beispiel durch das Hinzufügen weiterer Teilinformationen erreicht werden. Ein oft beobachteter Seiteneffekt dieser Methode ist, dass andere Inkonsistenzen damit ebenfalls beseitigt werden.

Werden Inkonsistenzen ignoriert oder toleriert, so schließt das jedoch nicht aus, dass bei Änderungen der Anforderungen das damit verbundene Risiko neu überprüft und eingeschätzt werden muss. Allgemein gilt, dass unabhängig von der gewählten Behandlungsroutine die Inkonsistenzen und ihre Auswirkungen auf das Gesamtprojekt weiter beobachtet werden müssen. Hierzu gehört auch eine Bewertung der gefundenen Konsistenzverletzungen. Diese kann dem Entwickler Aufschluss über die Qualität seines Modells geben, um damit zum Beispiel den Fortschritt eines Projektes und die Qualität bezüglich eines fest definierten Standards einzuschätzen.

Das vorgestellte Framework gibt einen Überblick über die anfallenden Aufgaben, um ein praktikables (In-)Konsistenzmanagement in einem Werkzeug zu ermöglichen. Allerdings werden auch viele Aspekte vernachlässigt. So wird nicht weiter darauf eingegangen, auf welche Art und Weise der Benutzer mit dem System interagiert und wie ihm Konsistenzverletzungen gemeldet oder präsentiert werden. Es wird nicht näher beschrieben, in welcher Form die spezifizierten Modelle verwaltet und gespeichert und zu welchem Zeitpunkt Konsistenzprüfungen durchgeführt werden. Ebenso fehlt eine konkrete Beschreibung, wie Konsistenzbedingungen spezifiziert, verwaltet und überprüft werden. Dies ist aber die Basis für ein effizientes Konsistenzmanagement.

Das vorgestellte Framework besitzt einen konzeptionellen Charakter und wurde bisher nur anhand von Fallstudien untersucht. Eine praktische Umsetzung in einem Entwicklungswerkzeug wurde noch nicht realisiert. Da die Umsetzung und implementierung der gesamten Funkionalität den Rahmen dieser Diplomarbeit sprengen würde, werden nur die wichtigsten Konzepte übernommen. Dies betrifft beispielsweise die Verwaltung von Konsistenzregeln in einer gemeinsamen Datenbasis, die jederzeit ergänzt und angepasst werden kann. Ebenso wird die Möglichkeit der Konfiguration der Konsistenzprüfung ermöglicht, so dass einzelne Konsistenzregeln aktiviert und deaktiviert werden können. Damit lässt sich die Konsistenzprüfung an die Bedürfnisse eines Entwicklers individuell anpassen.

2.2 Spezielle Konsistenzprüfungsprobleme

2.2.1 Konsistenzanalysen zwischen Sequenz- und Klassendiagrammen

In [TE00] wird ein auf attributierten Typgraphen und Graphgrammatiken basierender Ansatz zur Überprüfung von Konsistenzbedingungen zwischen Klassen- und Sequenzdiagrammen vorgestellt. Dazu wird das Klassendiagramm in einen attributierten und typisierten Graphen transformiert.

Die Klassen werden auf Knoten und die Assoziationen auf Kanten des Typgraphen abgebildet. Die Multiplizität einer Assoziation wird nicht im Typgraph gespeichert, sondern durch grafische Bedingungen formuliert. Die untere Kardinalitätsgrenze wird durch eine Implikation dargestellt. Die obere Grenze wird durch eine negative Bedingung formuliert. Kardinalitäten von 0 oder unendlich unterliegen keiner speziellen Semantik zum Ausführungszeitpunkt und müssen daher

nicht durch Bedingungen weiter spezifiziert werden. Schließlich werden aus Sequenzdiagrammen Graphersetzungsregeln gebildet. Mit Hilfe dieser Diagramme und Bedingungen werden Existenz-, Sichtbarkeits- und Multiplizitätsprüfungen zwischen beiden Diagrammarten durchgeführt.

Die referentielle Integrität zwischen den Diagrammen wird durch die Existenzprüfung validiert. Die in einem Sequenzdiagramm verwendeten Objekte und Links müssen in dem Klassendiagramm als Klassen und Assoziationen vorhanden sein. Darüber hinaus setzen Methodenaufrufe in einem Sequenzdiagramm die Existenz dieser Methoden in der Zielklasse voraus.

Die Existenz von Klassen und Methoden ist jedoch nicht das einzige Kriterium. Die verwendeten Objekte und Methodenaufrufe müssen im Sequenzdiagramm sichtbar sein. Dies wird durch die Sichtbarkeitsprüfung validiert.

Eine weitere Prüfung befasst sich mit der in den Assoziationen spezifizierten Multiplizität von Objektinstanzen. In einem Sequenzdiagramm ist die Anzahl der Objekte nicht immer statisch. Neue Objekte können während einer Interaktion erzeugt und bereits vorhandene Objekte gelöscht werden. Dadurch kann ein Objekt zu wenige oder aber auch zu viele Beziehungen zu einem anderen Objekt besitzen. Diese Prüfung wird als Multiplizitätsprüfung bezeichnet.

Die beschriebenen Konsistenzprüfungen wurden durch die Implementierung eines Prototyps validiert. Dieser erwartet als Eingabe syntaktisch korrekte Klassendiagramme und Sequenzdiagramme. Hier ergibt sich aber bereits ein Problem beim Einsatz einer Entwicklungsumgebung, die dem Entwickler viele Freiheiten einräumt und temporär inkonsistente und unvollständige Modelle zulässt. Um eine Basis für die Überprüfung solcher spezieller Konsistenzprobleme zu schaffen, müssen zunächst statische Konsistenz- und Integritätsbedingungen im Metamodell untersucht und eventuelle Konsistenzverletzungen behoben werden. Hierfür wird die Idee, Konsistenzbedingungen in Form von Graphgrammatiken zu formulieren, aus dem vorgestellten Ansatz aufgegriffen und für die Überprüfung des astrakten Syntaxgraphen eingesetzt.

2.2.2 Konsistenzprüfung einer Diagrammverfeinerung

In [Egy00] wird eine Methode vorgestellt, um Inkonsistenzen zwischen zwei Klassendiagrammen festzustellen, wobei das eine Diagramm eine Verfeinerung des anderen Diagramms ist. Die Überprüfung besteht aus drei Schritten. Zuerst werden die Klassen des abstrakten Diagramms den konkreten Klassen des verfeinerten Diagramms zugeordnet. Diese Beziehungen werden in einer Tabelle gespeichert und dienen als Eingaben für den nächsten Schritt. Hier erfolgt eine schrittweise

Transformation des konkreten Modells in Richtung des abstrakten Modells, um beide anschließend besser vergleichen zu können. Dabei wird mit Hilfe von vorher definierten Abstraktionsregeln versucht, möglichst viele Klassen und Assoziationen auf abstrakte Modellelemente abzubilden. Die Abbildung der Modellelemente des konkreten Diagrams auf die abstrakteren Elemente wird gespeichert, um die Zuordnung zwischen dem konkreten Diagramm und dem transformierten Diagramm zu ermöglichen. Das so entstandene Diagramm spiegelt die Struktur des abstrakten Diagramms wieder und ist mit diesem wesentlich leichter vergleichbar. Der letzte Schritt besteht dann aus der Auswertung von verschiedenen Konsistenzregeln, die zwischen einem abstrakten Diagramm und dessen Verfeinerung gelten müssen. Dazu wird das abstrakte Diagramm mit dem transformierten Diagramm verglichen. In Abbildung 2.2 ist das Vorgehen schematisch dargestellt.

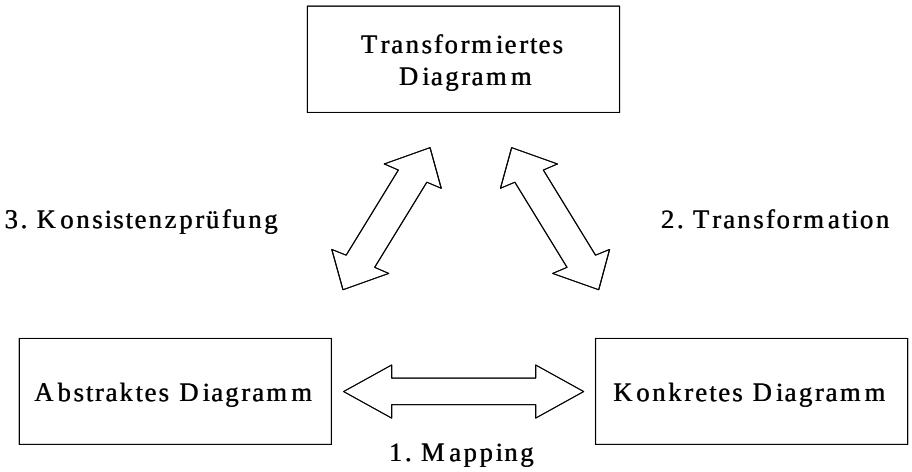


Abbildung 2.2: Das Prüfungsschema

Obwohl in dieser Diplomarbeit die Überprüfung der vertikalen Konsistenz zwischen Diagrammen nicht weiter untersucht wird, stellt der oben vorgestellte, transformationsbasierte Ansatz zur Überprüfung von Konsistenz einen interessanten Aspekt dar. Die Transformation eines Modells in eine andere Darstellungsform ist durchaus hilfreich und erleichtert ansonsten nur schwer realisierbare Konsistenzprüfungen. Zusätzlich muss jedoch bedacht werden, dass eine weitere Darstellung eines Modells weitere Überlappungen und damit auch Konsistenzprobleme mit sich bringt. So ist ohne Unterstützung das transformierte Diagramm ungültig, sobald der Benutzer das abstrakte Modell verändert. Um eine erneute Transformation zu umgehen, wird daher die in Abschnitt 2.4.2 vorgestellte Tripel-Graph-Grammatik verwendet, die zur Spezifikation von Diagrammtransformationen und anschliessender Konsistenzsicherung eingesetz wird.

2.3 Analysen in Fujaba

FUJABA verfügt bereits über einige Konsistenzanalysen, die zum Beispiel die Einhaltung von Namenskonventionen und Kardinalitätsangaben in Assoziationen sicherstellen. Die realisierten Konsistenzüberprüfungen basieren auf dem in [FNT98] vorgeschlagenen Mechanismus, der neben der reinen Konsistenzprüfung auch eine automatische Korrektur von Konsistenzverletzungen ermöglicht.

Die Überprüfung der Konsistenz erfolgt durch sogenannte Analysemaschinen, die nebenläufig ausgeführt werden. Abbildung 2.3 zeigt das Klassendiagramm mit den am Analyseprozess beteiligten Klassen. Die Klasse *InheritanceCheckerEngine* ist eine konkrete Analysemaschine, die für die Überprüfung von Generalisierungsbeziehungen zuständig ist.

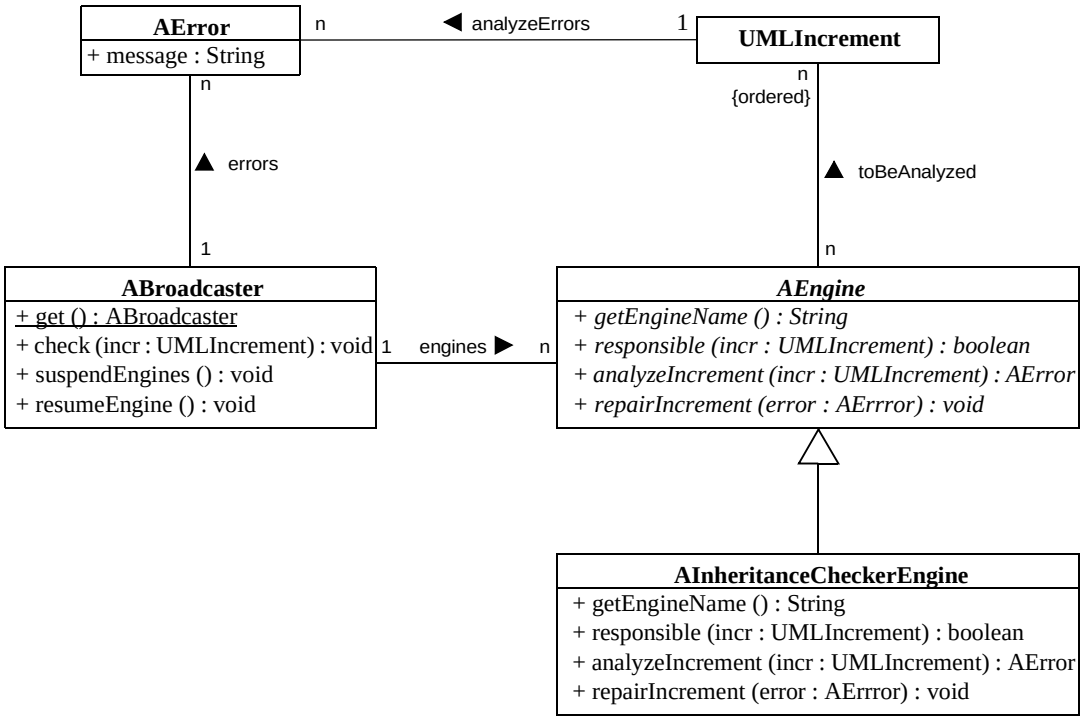


Abbildung 2.3: Architektur zur Konsistenzanalyse in FUJABA

Im Mittelpunkt des Analyseprozesses steht die Klasse *Broadcaster*. Diese Klasse ist als „Singleton-Pattern“ [GHJV95] realisiert. Damit gibt es zur Laufzeit genau eine Instanz dieser Klasse, auf die über die Klassenmethode *get* zugegriffen werden kann. Der *Broadcaster*[1] ist für die Verwaltung der Analysemaschinen und die Verteilung der modifizierten Elemente an die Analysemaschinen verantwortlich.

Soll ein Element des abstrakten Syntaxgraphen (ASG), das eine Spezialisierung der Klasse *UMLIncrement* ist, überprüft werden, so wird das betreffende Element dem *Broadcaster* als Parameter der Methode *check* übergeben. Über die Assoziation *engines* reicht der *Broadcaster* das zu überprüfendende Modellelement an die einzelnen Maschinen weiter. Jede Analysemaschine entscheidet dabei selbständig, ob sie für die Analyse des ihr übergebenen Elementes zuständig ist. Die Überprüfung der Zuständigkeit findet in der Methode *responsible* statt. Liefert *responsible* den Wert *true* zurück, so wird das Element in die Warteschlange *toBeAnalyzed* der Maschine eingefügt.

Die Überprüfung der Konsistenz findet in der Methode *analyzeIncrement* statt. Dabei wird ein Element der Warteschlange entnommen und als Parameter an diese Methode weitergegeben. Nach erfolgreicher Konsistenzprüfung wird das nächste Element aus der Warteschlange genommen und überprüft. Im Fehlerfall liefert die Methode eine Instanz der Klasse *Error* zurück. Die Klasse *Error* unterhält neben einem Attribut *message* zur Fehlerbeschreibung die beiden Assoziationen *analyseErrors* und *errors*.

Neben der Verwaltung der Analysemaschinen ist der *Broadcaster* auch für die Synchronisation der Zugriffe auf den abstrakten Syntaxgraphen zuständig. Da die Prüfungen und Korrekturen sowohl untereinander als auch zu den Benutzerinteraktionen nebenläufig ablaufen, müssen sowohl lesende als auch schreibende Zugriffe auf den ASG koordiniert werden, um Probleme wie zum Beispiel „dirty read“ [FNT98] zu vermeiden. Hierzu stellt der *Broadcaster* die Methoden *suspendEngines* und *resumeEngines* zur Verfügung. Da Veränderungen an einem Objekt weitere Modifikationen an beteiligten Objekten nach sich ziehen und es nicht einfach ist, all diese Objekte im Vorfeld zu identifizieren, erfolgt eine Sperre auf dem gesamten ASG. Vor jeder Benutzerinteraktion werden mit dem Aufruf von *suspendEngines* die laufenden Analysen unterbrochen um einen möglichen Zugriffskonflikt zu vermeiden. Nach der Beendigung des Editiervorgangs werden die Analysen mit *resumeEngines* wieder aufgenommen. Die Synchronisation der Analysemaschinen untereinander erfolgt durch den Synchronized-Mechanismus von JAVA [Lea97].

Der in Fujaba realisierte Analysemechanismus birgt einige Nachteile. Zum einen betrifft dies die Spezifikation der Konsistenzregeln. Dazu muss für jede neue Konsistenzbedingung eine neue Analysemaschine erstellt werden. Diese wird per Hand implementiert, so dass sowohl die Erstellung als auch die Wartung recht aufwä

ndig sind und spezielle Kenntnisse über den Analyseprozess erfordern.

Ein anderer Aspekt sind die recht eingeschränkten Möglichkeiten des Benutzers, Konsistenzanalysen dynamisch um weitere Bedingungen zu erweitern oder nicht ge-

1. Aus Gründen der besseren Lesbarkeit wird auf das vorangestellte ’A’ in den Bezichnern der Klassen verzichtet.

wünschte Überprüfungen von der Analyse auszuschließen. Darüber hinaus sind die erzeugten Informationen wenig hilfreich, wenn es keine automatische Korrektur für die Verletzung gibt und der Benutzer diese manuell durchführen muss.

Ein weiteres Problem stellen die nebenläufigen Prüfungen dar. Der Grundgedanke ist dabei, die Überprüfungen im Hintergrund und ohne Einfluss auf die Performance der Entwicklungsumgebung stattfinden zu lassen. Allerdings ist die Nebenläufigkeit nicht gegeben, da bei einer durch eine Maschine durchgeführten automatischen Reparatur alle anderen Analyseprozesse blockiert werden, um konkurierende Schreib- und Lesezugriffe auf den AST zu verhindern. Durch die Synchronisation der Konsistenzanalysemaschinen untereinander ist die anscheindene Nebenläufigkeit der Analysemaschinen aufgelöst und die Analyse findet wieder sequenziell statt.

Zusätzlich entsteht durch die nebenläufigen Konsistenzprüfungen eine starke Synchronisationsproblematik zwischen den Benutzeraktionen und den Analyseprozessen. Die Sychronisation wird bei der Erstellung eines neuen Benutzerkommandos dem Programmierer überlassen. Der Aufruf der dafür benötigten Methoden wird jedoch oft vergessen, was zu Deadlockproblemen, Konflikten bei Zugriffen auf den ASG und Ausführungsfehlern führt. Diese Fehler tauchen jedoch oft erst wesentlich später auf, wenn beispielsweise andere Hardware oder ein anderes Betriebssystem mit einer anderen Scheduling-Strategie benutzt wird, so dass die Zuordnung des Fehlers sich als recht schwierig erweist.

Ein anderes Problem betrifft den undefinierten Zeitpunkt einer Konsistenzprüfung. Da die Analysen unabhängig von der Benutzeraktion durchgeführt werden, ist die Ursache für eine Konsistenzverletzung nicht nachvollziehbar. Es ist also nicht möglich festzustellen, durch welche Aktion die Konsistenzverletzung verursacht wurde.

2.4 Formulierung von Konsistenzbedingungen

2.4.1 Object Constraint Language

In der UML können statische Integritätsbedinungen durch die *Object Constraint Language* (OCL) an ein Modell formuliert werden. Die OCL ist eine formale Sprache zur Beschreibung zusätzlicher Eigenschaften, die ein Modell erfüllen muss. So lassen sich zum Beispiel präzise Zusicherungen über Zusammenhänge zwischen unterschiedlichen Modellelementen formulieren, Invarianten aufstellen, deren Eigenschaf-

ten immer gelten müssen, sowie Vor- und Nachbedingungen für Operationen angeben.

Da der UML ein formales Metamodell zugrunde liegt, werden Bedingungen an das Metamodell ebenfalls mit der OCL formuliert. Die sogenannten *Well-Formedness Rules* sind fester Bestandteil der UML-Spezifikation [UML99]. Ein Beispiel für eine solche Konsistenzbedingung ist in Abbildung 2.4 dokumentiert.

Diese einfache Bedingung besagt, dass in einer Assoziation höchstens ein Assoziationsende eine Aggregation oder Komposition sein darf.

```
// Dieser OCL-Ausdruck gilt im Kontext einer Assoziation, auf die mit
// self Bezug genommen wird.
context Association inv:
self.allConnections->select (aggregation < #none)->size <= 1

// Zusatzoperation allConnections; liefert alle Assiziationsenden zu einer
// Assoziation über den Pfad self.connection
allConnections : Set (AssociationEnd);
allConnections = self.connection
```

Abbildung 2.4: OCL-Ausdruck und Hilfsoperation

Die OCL besteht aus vordefinierten Datentypen, Operationen und Konstrukten zur Navigation innerhalb von Modellen. Dennoch ist die OCL keine Programmiersprache im herkömmlichen Sinn, da sie keine Zuweisungsoperatoren oder andere Formen zur Manipulation von Strukturen bereit stellt. Zum einen ist dies von Vorteil, da dadurch keine unbeabsichtigten Veränderungen an den Benutzermodellen als Seiteneffekt einer Auswertung auftreten können. Andererseits ist es aber damit auch nicht möglich, Korrekturmaßnahmen bei Verletzungen von Bedingungen zu spezifizieren. Daher eignet sich diese Sprache nur zur Überprüfung von Konsistenzbedingungen aber nicht für automatische Konsistenzsicherung.

Ein anderer Nachteil ist die textliche und wenig intuitive Formulierung von Regeln. Zwar können einige Bedingungen durch den Einsatz von zusätzlich definierten Operationen auch sehr klein und handlich formuliert werden, der Grossteil der in [UML99] definierten Regeln ist jedoch nicht so kurz und prägnant wie das hier vorgestellte Beispiel. In [BKPT00] wird daher ein graph-basierter Ansatz zur visuellen Darstellung und Überprüfung von OCL-Ausdrücken vorgestellt.

2.4.2 Tripel-Graph-Grammatik

Lefering und Schürr stellen in ihren Arbeiten [Lef95, Sch94] die auf *Pair-Graph-Grammatiken* [Prat71] beruhende *Tripel-Graph-Grammatik* (TGG) vor, die die Nachteile von Graph-Grammatiken bezüglich der Spezifikation von Transformationen zwischen unterschiedlichen Dokumenten beseitigen. Ein besonderes Merkmal der *Tripel-Graph-Grammatik* ist die klare Unterscheidung der beiden zu transformierenden Datenstrukturen der Dokumente und die Speicherung der Zusatzinformationen über die durchgeführte Transformation in der sogenannten Integrationsstruktur. Diese dritte Datenstruktur definiert dabei die Beziehungen zwischen den Objekten der einzelnen Dokumente und ermöglicht eine Konsistenzüberprüfung.

Eine TGG-Regel ist in Abbildung 2.5 zu sehen. Sie besteht aus einer linken und einer rechten Regelseite. Jede Regelseite enthält die zu transformierenden Objekte eines Diagrammes. Die Integrationsstruktur besteht aus sogenannten Map-Objekten und verbindet die in Beziehung stehenden Objekte beider Regelseiten über Links. Dadurch wird der Zusammenhang zwischen Objekten der beiden Dokumente hergestellt und deklarativ beschrieben. Die grauen Linien gehören nicht zur Spezifikation und dienen nur zur besseren Unterscheidung der drei Datenstrukturen.

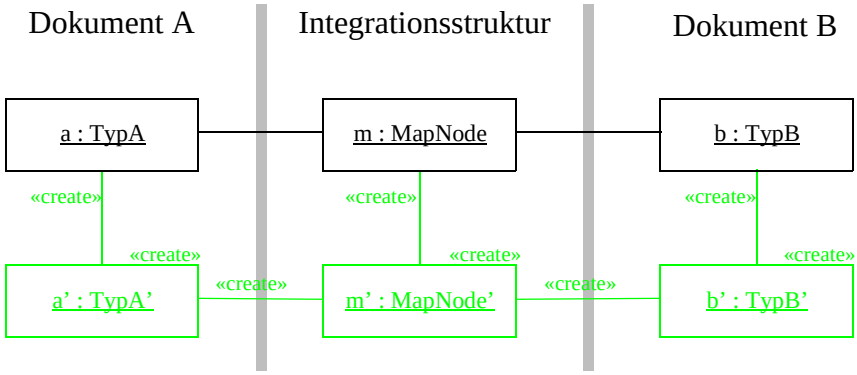


Abbildung 2.5: Eine TGG-Regel

Die dargestellte TGG-Regel ist Teil einer Grammatik und spezifiziert einen konsistenten Zustand zwischen den beiden Objekten der Dokumente. Dabei stellen die schwarzen Objekte und Links der TGG-Regel den Ausgangsgraphen und die durch den Stereotype «create» gekennzeichneten Objekte sowie Links den Erweiterungsgraphen dar. Wird also der Ausgangsgraph in einer Objektstruktur gefunden, so wird er um den Erweiterungsgraphen ergänzt.

Sind alle TGG-Regeln spezifiziert, so können beide Dokumente und die Integrationsstruktur durch die Anwendung der Grammatik erzeugt werden. Abbildung 2.6 zeigt schematisch die durch die Grammatik erzeugten Dokumente sowie die Integrationsstruktur.

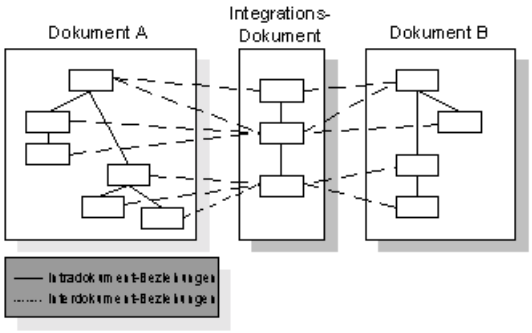


Abbildung 2.6: Integration durch Tripel-Graph-Grammatiken

Der parallele Aufbau der beiden Dokumentstrukturen samt zugehöriger Integrationsstruktur ist jedoch nicht der wichtigste Aspekt der Tripel-Graph-Grammatik. Entscheidender ist die Tatsache, dass aus den TGG-Regeln drei Graphersetzungsregeln abgeleitet werden können, die eine Transformation des linken Dokumentes in das rechte Dokument vornehmen und umgekehrt [Lef95, Sch94].

Die erste Regel ist die Vorwärts-Regel. Sie generiert zu einem Objekt des linken Dokuments das zugehörige Objekt des rechten Dokuments sowie ein Integrationsobjekt, das beide Objekte verbindet. Die Graphersetzungsregel wird aus der TGG-Regel abgeleitet, indem der Ausgangsgraph und alle Objekte sowie Links des linken Dokumentes als schwarze Objekte übernommen werden. Die verbleibenden Objekte werden als zu erzeugende Objekte gekennzeichnet und ebenfalls in die Graphersetzungsregel übernommen. Abbildung 2.7 zeigt die abgeleitete Graphersetzungsregel.

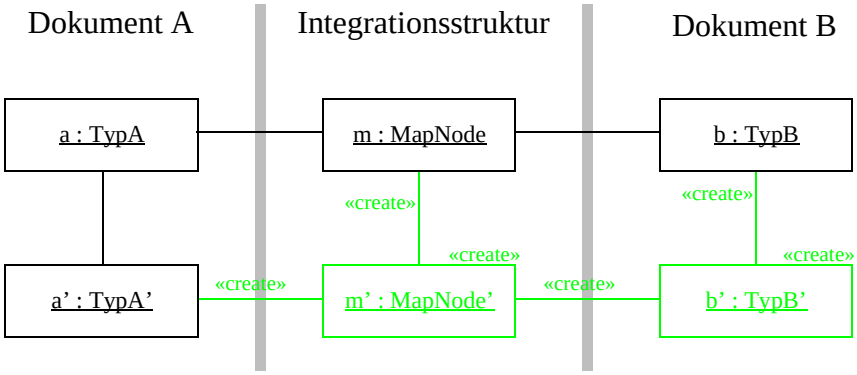


Abbildung 2.7: Vorwärts-Regel

Die zweite Regel ist die Rückwärts-Regel. Sie wird genauso abgeleitet wie die Vorwärts-Regel, nur dass die Rollen der linken und rechten Dokumentseiten vertauscht sind. Mit der in Abbildung 2.8 dargestellten Rückwärtsregel wird zu einem Objekt des rechten Dokumentes das zugehörige Objekt des linken Dokumentes und das Integrationsobjekt erstellt.

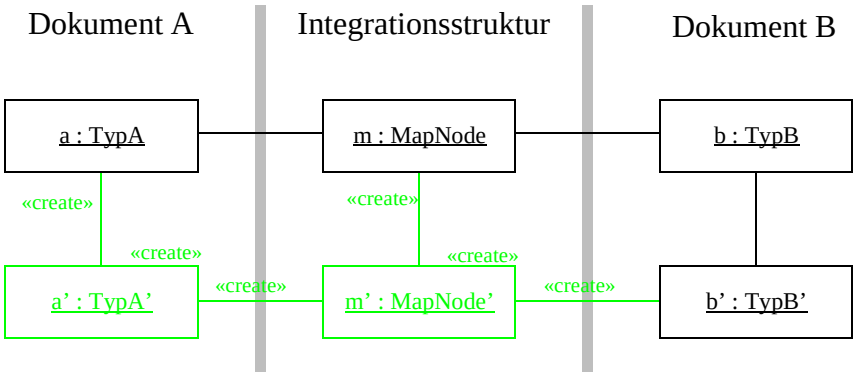


Abbildung 2.8: Rückwärts-Regel

Die letzte Graphersetzungsregel ist die Konsistenzregel. Sie wird eingesetzt, wenn bereits beide Dokumente vorhanden sind und nur noch das Integrationsobjekt erstellt werden muss. Sie wird aus der TGG-Regel abgeleitet, indem alle Objekte des Ausgangsgraphen und des Erweiterungsgraphen als zu suchende Objekte übernommen werden und nur das Map-Objekt des Erweiterungsgraphen als zu erzeugendes Objekt gekennzeichnet wird. Abbildung 2.9 zeigt die aus der TGG-Regel abgeleitete Konsistenzregel.

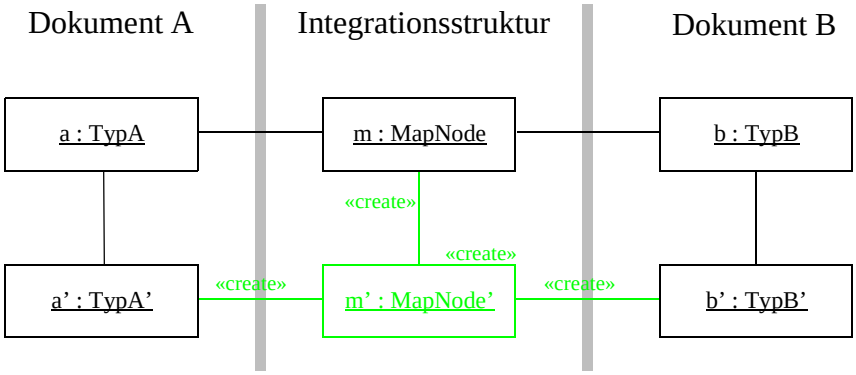


Abbildung 2.9: Konsistenz-Regel

In der Diplomarbeit von Jens H. Mühlenhoff [Müh00] wurden Tripel-Graph-Grammatiken zur Integration und Konsistenzprüfung zwischen unterschiedlichen Dokumenten bereits vorgestellt und verwendet. Das dort vorgestellte Verfahren ist jedoch nur dann anwendbar, wenn die kleinste mögliche Änderung an einem Dokument das Löschen oder Erzeugen von Objekten ist. Werden die Attributwerte eines Objektes verändert, so wird dies durch die dort vorgestellten TGG-Regeln und dem sogenannten TGG-Controler jedoch nicht erkannt [Müh00]. Ändert sich beispielsweise der Name einer Klasse, so wird dies von dem sogenannten TGG-Controler nicht erkannt, da dieser nur erzeugte oder gelöschte Objekte berücksichtigt und auf Änderungen der Attribute nicht reagiert.

Im Laufe dieser Arbeit wird die *Tripel-Graph-Grammatik* zur Spezifikation von Konsistenzbedingungen aufgegriffen und die oben genannte Einschränkung beseitigt. Dazu wird die Umsetzung der TGG-Regeln in *Story-Diagramme* an die neuen Anforderungen angepasst und ein Steuerungsmechanismus entwickelt, der die *Story-Diagramme* integriert. Die Steuerungslogik erkennt die Modifikation eines Objektzustands und startet eine inkrementelle Überprüfung der Konsistenz. Der Steuerungsmechanismus wird in Abschnitt 4 vorgestellt. Die dafür benötigte Umsetzung der TGG-Regeln in die *Story-Diagramme* wird in Abschnitt 3 anhand eines Beispiels beschrieben.

3 Spezifikation von Konsistenzbedingungen

Dieses Kapitel stellt Techniken zur Spezifikation von Konsistenzbedingungen vor. In Abschnitt 3.1 wird die Repräsentation von Modellen durch einen abstrakten Syntaxgraphen erläutert. Abschnitt 3.2 erläutert die Spezifikation von einfachen Konsistenzbedingungen mit der Beschreibungsmethode *Story Driven Modeling* [FNTZ98]. Die Spezifikation von Konsistenzbedingungen zwischen unterschiedlichen Diagrammen erfolgt mit der *Tripel-Graph-Grammatik* [Sch94] und wird in Abschnitt 3.3 vorgestellt.

3.1 Das FUJABA-Metamodell

In der integrierten Entwicklungsumgebung FUJABA erfolgt die Modellierung eines Systems üblicherweise mit unterschiedlichen Diagrammen und Diagrammelementen, denen ein werkzeugspezifisches Metamodell zugrunde liegt. Das Metamodell von FUJABA ist an das UML-Metamodell [UML99] angelehnt und besteht aus circa 90 Metaklassen und über 100 Metaassoziationen [FNT98]. Erstellt der Benutzer ein Diagramm, so werden die benutzten Diagrammelemente durch Instanzen der Metaklassen repräsentiert. Sie bilden damit eine mögliche Ausprägung des Metamodells, die auch als Metamodellexemplar bezeichnet wird.

In Abbildung 3.1 ist ein kleiner Ausschnitt aus dem Metamodell von FUJABA dargestellt. Die Basisklasse für alle Modellelemente ist die Klasse *UMLIncrement*. Alle ASG-Objekte müssen direkt oder indirekt von dieser Klasse abgeleitet werden. Die Basis für Diagramme und Diagrammelemente wird durch die beiden Klassen *UMLDiagram* und *UMLDiagramItem* hergestellt. Diese Klassen stehen über die Assoziation *items* in Beziehung zueinander, da ein Diagramm verschiedene Diagrammelemente enthalten kann.

Um eine neue Diagrammart in dieses Metaschema zu integrieren, müssen die neuen Metaklassen von *UMLDiagram* und *UMLDiagramItem* abgeleitet werden. Wie in

Abbildung 3.1 zu sehen ist, wurden auf diese Weise SDL-Blockdiagramme und UML-Klassendiagramme integriert.

Hierbei ist zu beachten, dass die Syntax des Metamodells es nicht verbietet, eine Instanz von *SDLBlock* einer Instanz von *UMLClassDiagram* zuzuweisen, was nicht beabsichtigt ist. Solche impliziten Konsistenzbedingungen können jedoch bereits bei der Eingabe berücksichtigt werden, indem nur die in einem Diagramm erlaubten Elemente zur Auswahl gestellt werden. Obwohl solche Konsistenzprüfungen mit den im Folgenden vorgestellten Konzepten ebenfalls realisierbar sind, werden sie in dieser Arbeit nicht weiter berücksichtigt.

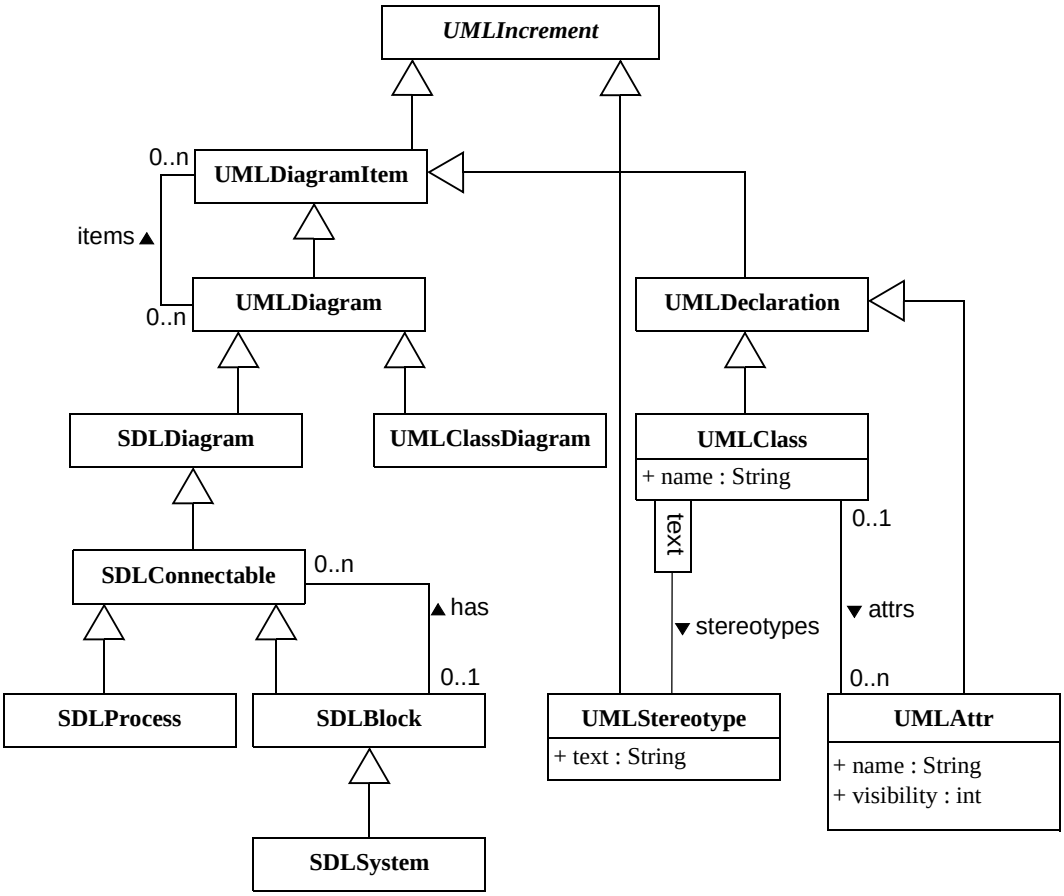


Abbildung 3.1: Ausschnitt aus dem ASG von FUJABA

Um ein syntaktisch korrektes Modell zu erhalten, müssen neben der oben genannten Bedingung, dass ein Diagramm nur dafür bestimmte Elemente enthalten darf, zusätzliche Konsistenzbedingungen an das Metamodell formuliert werden. Dabei können die Konsistenzbedingungen in zwei Klassen eingeteilt werden.

Die erste Konsistenzform stellt Bedingungen an die Eigenschaften von Objekten, zu deren Überprüfung entweder andere Objekte nicht herangezogen werden müssen oder diese sehr einfach über Assoziationen erreichbar sind.

Ein Beispiel für solche Bedingungen sind Klassennamen, die immer groß geschrieben werden müssen oder die Sichtbarkeit von Attributen, die durch einen *Integer*-Datentyp repräsentiert werden. Ein Beispiel für abhängige Objekte stellen die beiden über die Assoziation *attrs* verbundenen Metaklassen *UMLClass* und *UMLAttr* dar. Eine weitere Konsistenzbedingung ist beispielsweise, dass die Attribute einer Klasse unterschiedliche Namen haben müssen. Um diese Bedingung zu überprüfen, können im Metamodellexemplar alle mit der Klasse verbundenen Attribute besucht und die Namen auf Eindeutigkeit überprüft werden.

Die zweite Klasse von Konsistenzbedingungen ähnelt der vorhergegangenen Beschreibung. Im Gegensatz dazu existieren jedoch zwischen den Objekten keine direkten Beziehungen, so dass ein Navigieren in dem Metamodellexemplar nicht möglich ist. Versuche, diese Abhängigkeit beispielsweise über die Namensgleichheit von Elementen herzustellen, scheitern, sobald der Name eines der Elemente vom Benutzer geändert wird.

Eine Konsistenzbedingung hierfür ist beispielsweise die Abhängigkeit zwischen Blöcken in einem SDL-Blockdiagramm und Klassen in einem UML-Klassendiagramm. Jeder Block eines SDL-Blockdiagrammes wird auf eine Klasse in einem UML-Klassendiagramm abgebildet. Die betroffenen Metaklassen *UMLClass* und *SDLBlock* haben jedoch keine direkte Assoziation zueinander.

Zur Formulierung von Konsistenzbedingungen können die Objekte und Beziehungen eines Metamodellexemplars auch als Knoten und Kanten interpretiert und als abstrakter Syntaxgraph (ASG) aufgefasst werden. Die Spezifikation von Änderungsoperationen zur Manipulation von Graphen kann auf Grundlage der Theorie von Graph-Grammatiken [Roz97] erfolgen. Daher wird zur Überprüfung der beiden Konsistenzarten das auf Graphgrammatiken basierte *Story Driven Modeling* [FNTZ98] eingesetzt. Für Konsistenzprüfungen zwischen Objekten ohne direkte Beziehung wird auf das Konzept der *Tripel-Graph-Grammatiken* [Sch94] zurückgegriffen.

3.2 Story Driven Modeling

3.2.1 Grundlagen

Das *Story Driven Modeling* (SDM) [FNTZ98] wurde am Lehrstuhl für Softwaretechnik an der Universität Paderborn entwickelt und ist eine visuelle Modellierungssprache zur Manipulation von komplexen Objektstrukturen. Den Hauptbestandteil von SDM bilden *Story-Diagramme*, die eine Kombination aus Aktivitäts- und Kollaborationsdiagrammen sind und deren Semantik auf Graph-Grammatiken beruht.

Eine Graph-Grammatik besteht im wesentlichen aus einem Satz von Produktionen und Graphersetzungsregeln. Jede Regel setzt sich dabei aus einer rechten und einer linken Regelseite zusammen. Die linke Seite beschreibt dabei einen Teilgraphen, der die aktuelle Situation im Ausgangsgraphen darstellt. Die rechte Regelseite beschreibt den Zustand des Teilgraphen nach der Anwendung der Ersetzungsregel. Es wird also zuerst versucht, den linken Teilgraphen durch eine Teilgraphensuche zu finden und ihn durch das Hinzufügen und Löschen von Objekten in den Teilgraphen der rechten Seite zu überführen. Graph-Grammatiken eignen sich somit sowohl zur Beschreibung von gültigen und damit konsistenten Objektstrukturen als auch zur Spezifikation von Änderungsoperationen, die zu einer konsistenten und fehlerfreien Objektstruktur führen.

Ein *Story-Diagramm* besteht aus *Story-Pattern*, die auch als Aktivitäten bezeichnet und durch ein Rechteck mit abgerundeter linker und rechter Seite dargestellt werden. Die erste auszuführende Aktivität wird durch genau einen Startzustand gekennzeichnet. Wird die Aktivität in einer Schleife durchlaufen, so wird dies durch zwei übereinander liegende Aktivitäten repräsentiert. Die Aktivitäten sind durch Transitionen miteinander verbunden. Eine Transition wird durch einen Pfeil dargestellt und kann zusätzlich Bedingungen und Verzweigungen enthalten. Transitionen werden verwendet, um den Kontrollfluss zwischen den Aktivitäten zu modellieren. Die Beendigung einer Ausführung wird durch Endzustände kenntlich gemacht.

In einem Story-Pattern werden Bedingungen an die Objektstruktur gestellt und Operationen zur Modifikation der Objekte spezifiziert. Die Objekte werden als Rechtecke dargestellt und können durch einen Namen und einen Typ näher beschrieben werden. Objekte können gebunden und ungebunden sein. Während gebundene Objekte bereits bekannt sind, müssen ungebundene Objekte erst im abstrakten Syntaxgraphen gesucht werden. Objekte besitzen außerdem Attribute, deren Werte abgefragt und verändert werden können. Zusätzlich können Bedingungen an diese Werte formuliert werden.

Im Gegensatz zu den in PROGRES[1] [Zün96] realisierten Graph-Grammatik-Spezifikationen werden die linke und die rechte Seite einer Graphersetzungsregel in einem *Story-Pattern* zusammengefasst. Zur Unterscheidung zwischen einer Objektstruktur, die als Vorbedingung in dem abstrakten Syntaxgraphen gefunden werden muss, um anschließend Veränderungen vorzunehmen, werden die zu suchenden, zu löschenden oder zu erzeugenden Objekte durch unterschiedliche Farben und Stereotypen gekennzeichnet. Objekte, die gefunden werden müssen, werden schwarz dargestellt. Ein Objekt, das erzeugt werden soll, wird in der Farbe grün dargestellt und mit dem Stereotyp *«create»* versehen. Ein Objekt, das zu löschen ist, wird mit dem Stereotyp *«destroy»* annotiert. Das Schlüselwort wird dabei in rot gekennzichnet.

Weiterhin können Objekte durch die Farbe grau als optional gekennzeichnet werden. Ein optionales Objekt wird im abstrakten Syntaxgraphen gesucht, stellt aber keine notwendige Bedingung für die Ausführung des *Story-Patterns* dar. In Verbindung mit dem Schlüsselwort *«create»* wird ein als optional gekennzeichnetes Objekt nur dann erzeugt, wenn es im Syntaxgraphen nicht gefunden wurde. Im Gegensatz dazu darf ein durchgestrichenes Objekt in der Objektstruktur auf keinen Fall existieren, damit die Ausführung des *Story-Patterns* fortgesetzt wird.

Diese Erweiterungen und Einschränkungen gelten auch für Beziehungen zwischen Objekten. Diese Beziehungen werden als Kanten dargestellt und entsprechen Assoziationen im Klassendiagramm der betrachteten Objektwelt. Sie werden auch als *Links* bezeichnet.

Ein großer Vorteil des *Story Driven Modeling* ist die automatische Generierung von JAVA-Quellcode aus den spezifizierten *Story-Diagrammen*, wovon in dieser Arbeit Gebrauch gemacht wird. Da das *Story Driven Modeling* und die Generierung von Quellcode aus den *Story-Diagrammen* bereits in [FNT98] ausführlich beschrieben worden ist, wird auf eine detailliertere Darstellung an dieser Stelle verzichtet.

3.2.2 Beispiel einer Konsistenzverletzung

Eine Regel, die in UML-Klassendiagrammen eingehalten werden muss und in [UML99] durch die *Well-Formedness Rules* formuliert wurde, fordert, dass alle Attribute eines Interfaces öffentlich deklariert werden müssen. Das Interface *Process* in Abbildung 3.2 enthält das private Attribut *running* und verletzt somit diese Konsistenzbedingung.

1. PROgramierte GraphErsetungsSysteme

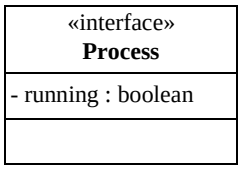


Abbildung 3.2: Konsistenzverletzung in einem Interface

Ein Interface wird im abstrakten Syntaxgraphen durch eine Instanz der Klasse *UMLClass* und *UMLStereotype* repräsentiert. Beide Objekte stehen über die qualifizierte Assoziation *stereotypes* miteinander in Beziehung. Stereotypen klassifizieren die Verwendung eines Modellelementes und können unter anderem eingesetzt werden, um die äußere Darstellung eines Elementes zu verändern oder um das Sprachkonstrukt zu präzisieren.

Im ASG werden Attribute einer Klasse und damit auch eines Interfaces durch Instanzen der Klasse *UMLAttr* repräsentiert. Die Sichtbarkeit eines Attributs wird durch das *visibility* Attribut festgelegt, das die Werte *PUBLIC*, *PROTECTED* und *PRIVATE* annehmen kann. Alle Attribute einer Klasse sind über die Assoziation *attrs* zu erreichen. Abbildung 3.3 zeigt die Ausprägung des abstrakten Syntaxgraphen, der das Interface aus dem obigen Beispiel repräsentiert.

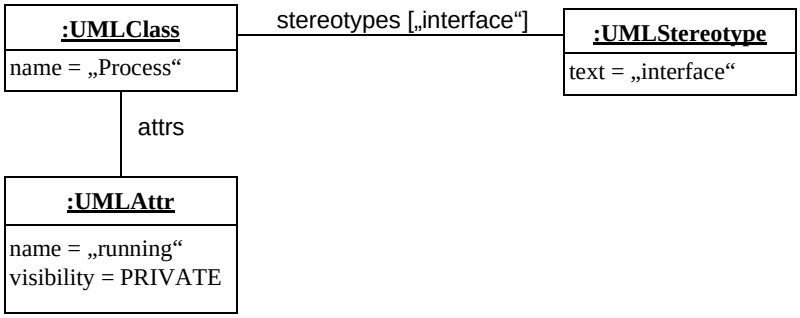


Abbildung 3.3: Abstrakter Syntaxgraph zum Interface *Process*

3.2.3 Spezifikation der Konsistenzbedingung

Die Spezifikation einer Konsistenzbedingung zur Erkennung und Behebung der Inkonsistenz aus dem oben dargestellten Beispiel kann mit dem *Story-Diagramm* aus Abbildung 3.4 erfolgen.

In dem *Story-Diagramm* finden sich die oben beschriebenen Objekte, die zur Repräsentation eines Interfaces im abstrakten Syntaxgraphen von FUJABA erzeugt wer-

den. Es existieren das *UMLClass*-Objekt und das *UMLStereotype*-Objekt. Beide Objekte sind über den qualifizierten Link *stereotypes* verbunden. Ebenso ist ein *UMLAttr*-Objekt vorhanden, das über den Link *attrs* mit dem Objekt *clazz* in Beziehung steht. An das Attribut wird mit *visibility != PUBLIC* die Bedingung gestellt, dass es sich hierbei nicht um ein nicht öffentliches Attribut handeln darf. Dies beudeutet, dass das Attribut gebunden wird, sofern die Sichtbarkeit *PRIVATE* oder *PROTECTED* ist.

Das *Story-Diagramm* spezifiziert eine Methode *checkConsistency*, die als Parameter ein Objekt vom Typ *UMLIncrement* erwartet. Um die dargestellte Objektwelt im ASG wiederzufinden, wird zuerst versucht, den Parameter *incr* in ein *UMLClass*-Objekt umzuwandeln und dem *clazz*-Objekt zuzuweisen. Konnte die Umwandlung durchgeführt werden, so wird im nächsten Schritt über den Qualifizierer „interface“ geprüft, ob ein entsprechender Stereotyp in Beziehung zu der Klasse steht. In diesem Fall repräsentiert das Klassenobjekt ein Interface. Schließlich wird noch ein nicht öffentliches Attribut der Klasse gesucht. Konnte ein Attribut gefunden werden, das diese Bedingung erfüllt, so liegt eine Konsistenzverletzung vor. Diese Inkonsistenz wird im letzten Schritt durch die Zuweisung *visibility := PUBLIC* behoben.

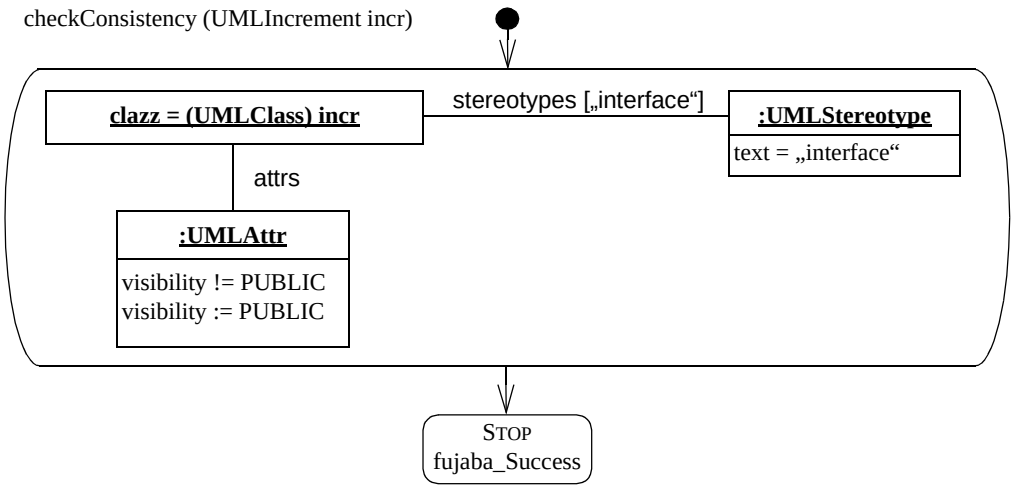


Abbildung 3.4: Spezifikation einer Konsistenzbedingung

Sobald alle im *Story-Diagramm* spezifizierten Objekte im ASG gefunden und die Sichtbarkeit des Attributes korrigiert wurde, wird der internen Variablen *fujaba_Success* der Wert *true* zugewiesen und beim Verlassen des Story-Patterns über den Endzustand *STOP* zurückgegeben.

An dieser Stelle stellt die Semantik des Story-Patterns ein nicht zu unterschätzendes Problem dar. Ein *Story-Pattern* liefert den Wert *true*, wenn alle Objekte gefunden werden konnten und die spezifizierten Operationen erfolgreich ausgeführt wurden. Konnte eine der Aktionen nicht durchgeführt werden, so liefert es den Wert *false*. In

diesem Fall muss das Konsistenzmanagement-System aber den Grund für das Scheitern der Ausführung eines *Story-Pattern* erkennen, um zwischen nicht anwendbaren Konsistenzbedingungen aufgrund von nicht erfüllten Vorbedingungen, erkannten Inkonsistenzen oder Fehlern bei der automatischen Korrektur unterscheiden zu können.

Das Story-Pattern würde also den Wert *false* zurückliefern, wenn das als Parameter übergebene Objekt nicht vom Typ *UMLClass* ist. In diesem Fall liegt aber keine Inkonsistenz vor, da die Konsistenzregel auf andere Objekte nicht anwendbar ist. Ebenso signalisiert das Story-Pattern eine Inkonsistenz, wenn der Klasse kein „interface“-Stereotyp zugeordnet ist. Da die Klasse aber ohne einen entsprechenden Stereotyp aber kein Interface ist, kann auch hier die Konsistenzregel nicht angewendet werden. Eine weitere, nicht vorhandene Konsistenzverletzung wird gemeldet, wenn das Interface keine Attribute besitzt, oder alle Attribute öffentlich sind. In beiden Fällen ist das Interface bezüglich der Regel aber konsistent.

Um diese Problematik zu beheben, erfolgt die Spezifikation der Konsistenzbedingung daher durch die drei *Story-Diagramme responsible*, *check* und *repair*. Durch die Aufteilung der oben vorgestellten Spezifikation in diese drei Methoden kann zwischen nicht anwendbaren Konsistenzbedingungen aufgrund von nicht erfüllten Vorbedingungen, erkannten Inkonsistenzen sowie Fehlern bei der automatischen Korrektur unterschieden werden. Im Folgenden werden die drei Methoden und ihre *Story-Diagramme* anhand des bereits bekannten Beispiels erläutert.

Die *responsible* Methode

Die Spezifikation der *responsible* Methode überprüft die Anwendbarkeit der Konsistenzregel auf das ihr übergebene Objekt und stellt somit die Vorbedingung für eine Konsistenzprüfung dar. Abbildung 3.5 zeigt die Spezifikation der Vorbedingung für die Überprüfung der bereits bekannten Konsistenzregel.

In dem *Story-Diagramm* finden sich die oben beschriebenen Objekte, die zur Repräsentation eines Interfaces im abstrakten Syntaxgraphen von FUJABA erzeugt werden. Es existieren das *UMLClass*-Objekt und das *UMLStereotype*-Objekt. Beide Objekte sind über den qualifizierten Link *stereotypes* verbunden. Ebenso ist ein *UMLAttr*-Objekt vorhanden und über den Link *attrs* mit dem Objekt *clazz* verbunden.

In der Spezifikation werden keine Bedingungen an die Werte des Objektes *attr* gestellt, da diese erst durch die nachfolgende Methode überprüft werden. Die Vorbedingung überprüft, ob die Konsistenzregel auf das zu überprüfende Objekt anwendbar ist. Dies schließt falsche Ergebnisse bei der Konsistenzprüfung aus. Ohne Vorbedingung würde eine Konsistenzprüfung einen inkonsistenten Zustand auch dann melden, wenn das Interface keine Attribute besitzt. In diesem Fall ist das Interface bezüglich der Regel aber konsistent. Ebenso würde die Konsistenzprüfung falsche Ergebnisse

liefern, wenn die Klasse überprüft würde aber keinen Link zu *stereotype* hätte. Dann ist die Klasse aber kein Interface und die Konsistenzregel nicht anwendbar.

Werden alle Objekte und die zugehörigen Links in dem abstrakten Syntaxgraphen gefunden, so ist die Vorbedingung erfüllt und die Aktivität liefert den Rückgabewert *true*. In diesem Fall kann die Überprüfung der Konsistenzregel mit der Methode *check* durchgeführt werden.

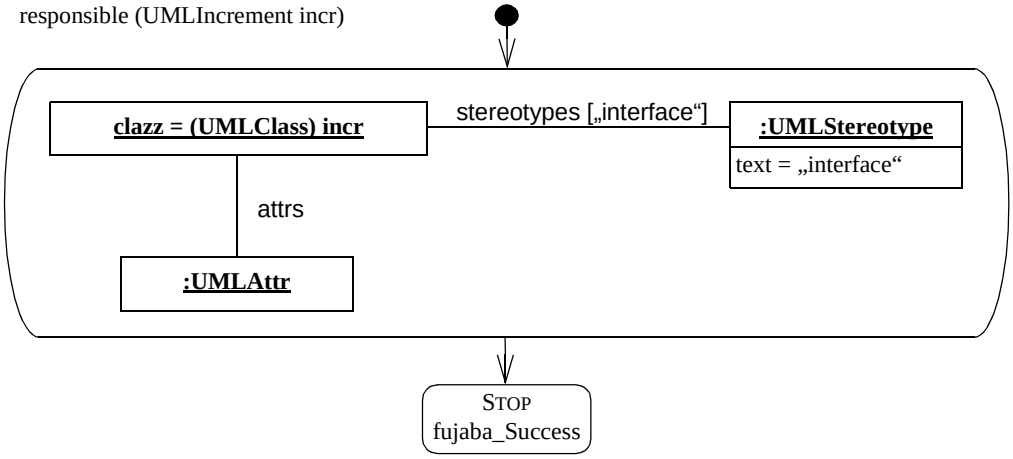


Abbildung 3.5: Vorbedingung zur Konsistenzprüfung

Die *check* Methode

Abbildung 3.6 zeigt das *Story-Diagramm* der *check* Methode. Hier wird die eigentliche Konsistenzbedingung an das Interface formuliert und überprüft.

Da bereits bekannt ist, dass das zu überprüfende Objekt ein Interface ist, wird das Objekt *stereotype* nicht mehr aufgeführt. Es sind lediglich die beiden Objekte *clazz* und *attr* sowie der Link *attrs* vorhanden. Zusätzlich wird jedoch die Bedingung *visibility != PUBLIC* an das Attribute *attr* gestellt. Da das Interface nur öffentliche Attribute enthalten darf, liegt eine Inkonsistenz vor, wenn die Sichtbarkeit des Attributs nicht öffentlich ist. Enthält *clazz* ein Attribut, das diese Bedingung erfüllt, so liegt eine Konsistenzverletzung vor, die mit der *repair* Methode behoben werden kann.

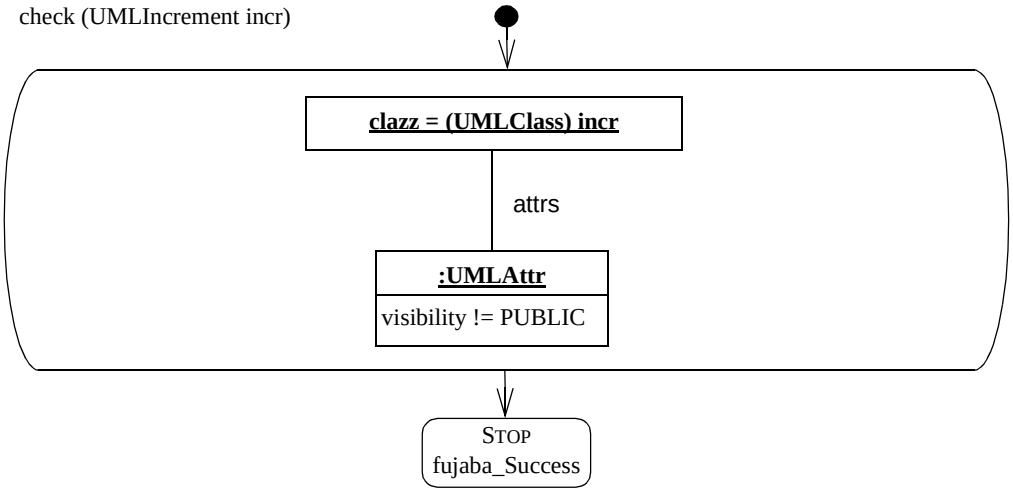


Abbildung 3.6: Konsistenzregel

Die *repair* Methode

Die *repair* Methode wird zur automatischen Behebung von Konsistenzverletzungen aufgerufen und wird nur ausgeführt, wenn eine Überprüfung durch die *check* Methode den Wert *false* zurückliefert. In dem vorgestellten Beispiel gibt es drei Möglichkeiten, die Inkonsistenz zu beheben. Sie werden in Abbildung 3.7, Abbildung 3.8 und Abbildung 3.9 vorgestellt.

Die erste Möglichkeit besteht darin, das Objekt *stereotype* zu löschen und damit das Interface in eine Klasse umzuwandeln. Dabei kann sich jedoch eine weitere Konsistenzverletzung ergeben, wenn beispielsweise eine Klasse das Interface implementiert und zusätzlich von einer anderen Klasse erbt. Wird das Interface jetzt in eine Klasse umgewandelt, so liegt anschließend eine Mehrfachvererbung vor, die jedoch von JAVA nicht unterstützt wird.

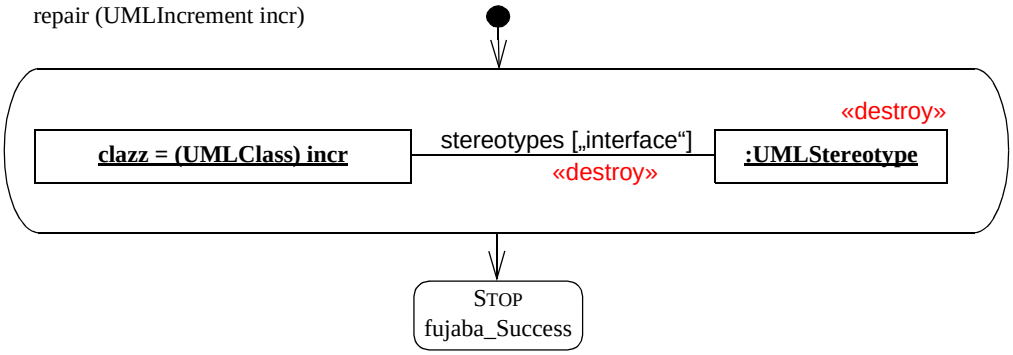


Abbildung 3.7: Erste Korrekturmöglichkeit zur Behebung der Inkonsistenz

Eine weitere Korrekturmöglichkeit ist, das Interface zu belassen und lediglich das fehlerhafte Attribut zu löschen. Dabei gehen jedoch die vom Benutzer modellierten Informationen gänzlich verloren. Daher ist die Korrekturmethode, die in Abbildung 3.8 dargestellt wird, vorzuziehen.

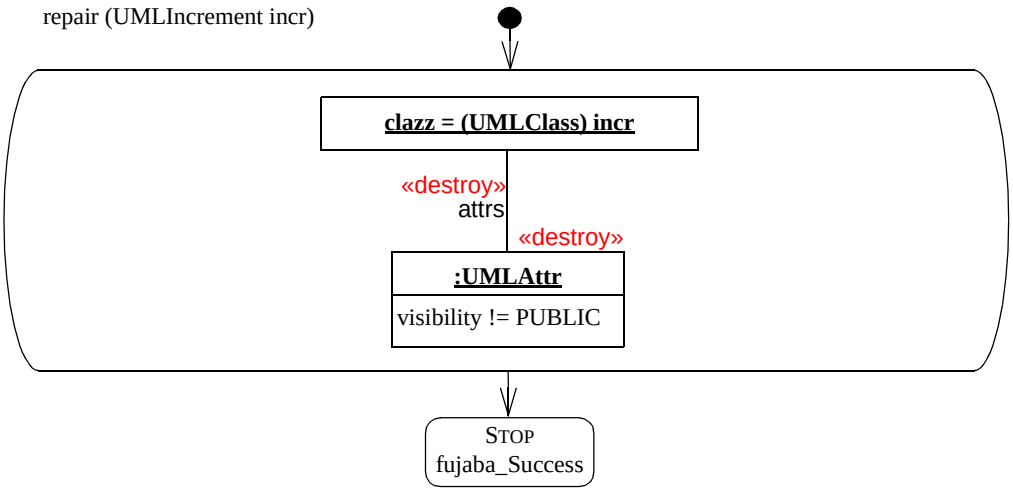


Abbildung 3.8: Zweite Korrekturmöglichkeit zur Behebung der Inkonsistenz

Die letzte *repair* Methode wandelt das nicht öffentliche Attribut in ein öffentliches Attribut um. Öffentliche Attribute sind in der JAVA-Programmiersprache implizit als *static* und *final* deklariert und stellen damit Konstanten dar. Diese Korrekturmethode hat den Vorteil, dass keine Objekte gelöscht werden. Dazu wird in dem Story-Pattern die Sichtbarkeit der Attribute einer Klasse auf öffentlich gesetzt. Durch die Bedingung *visibility != PUBLIC* werden nur nicht öffentliche Attribute korrigiert. Die Korrektur erfolgt durch die Zuweisung *visibility := PUBLIC*. Da es sich bei dem Story-Pattern um eine *for each* Aktivität handelt, werden bei der Korrektur alle nicht öffentlichen Attribute repariert.

Das *Story Driven Modeling* kann eingesetzt werden, um Objektstrukturen innerhalb eines Diagramms zu überprüfen und zu modifizieren. Ebenso können diagrammübergreifende Konsistenzbedingungen spezifiziert werden, falls die Objekte der Diagramme über Assoziationen miteinander verbunden sind.

Das *Story Driven Modeling* eignet sich jedoch weniger, um diagrammübergreifende Konsistenzprüfungen durchzuführen, wenn die Objekte der Diagramme in keiner Beziehung zueinander stehen. Zwar könnten solche Beziehungen beispielsweise über die Namensgleichheit zweier Objekte definiert und in *Story-Diagrammen* spezifiziert werden, allerdings schlägt dieser Ansatz fehl, sobald sich der Name eines der Objekte ändert.

Um diese Problematik zu lösen, wurde im Laufe dieser Arbeit die *Tripel-Graph-Grammatik* zur Spezifikation von Konsistenzbedingungen aufgegriffen, die nun im folgenden Abschnitt vorgestellt wird.

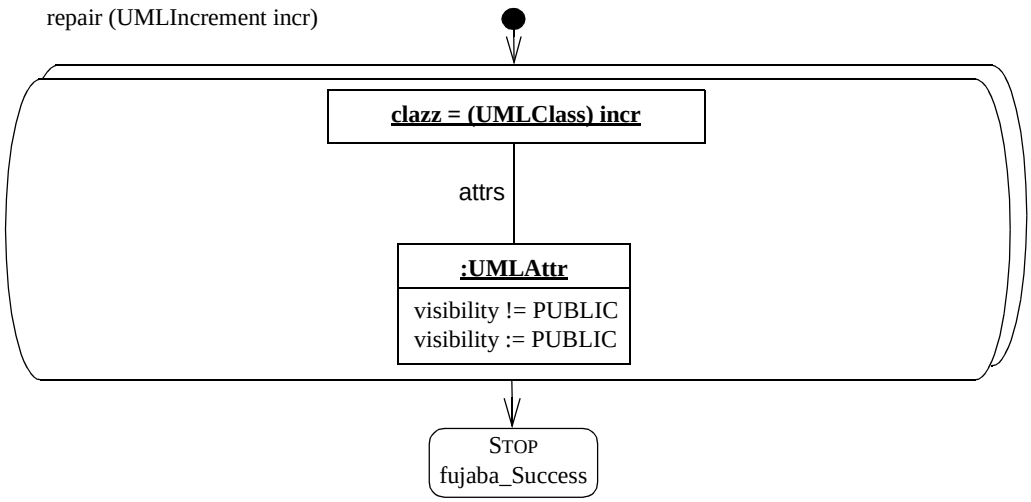


Abbildung 3.9: Dritte Korrekturmöglichkeit zur Behebung der Inkonsistenz

3.3 Tripel-Graph-Grammatik

In Abschnitt 2.4.2 wurden die Grundlagen der Tripel-Graph-Grammatik bereits erläutert. Im Rahmen der Diplomarbeit von Jens H. Mühlenhoff [Müh00] ist außerdem ein Editor zur Spezifikation von TGG-Regeln entstanden, der aus der Spezifikation drei *Story-Diagramme* erzeugt. Der Editor wird in dieser Arbeit weiter verwenden, allerdings musste die Übersetzung der TGG-Regeln in die *Story-Diagramme* angepasst werden, um die Einbettung in das Konsistenzmanagement-System und die damit ver-

bundene Steuerungslogik zu ermöglichen. In den folgenden Abschnitten werden die gegenüber den Arbeiten [Lef95, Sch94] sowie [Müh00] durchgeführten Änderungen und Anpassungen anhand eines Beispiels erläutert.

3.3.1 Ein Anwendungsbeispiel

Im ISILEIT-Projekt wurde eine durchgängige Methodik für den integrierten Entwurf, die Analyse und die Validierung von verteilten Fertigungssystemen entwickelt. Dabei werden zur Modellierung der statischen Struktur SDL-Blockdiagramme [SDL96] verwendet, die in ein UML-Klassendiagramm transformiert werden, um anschließend das dynamische Verhalten des Systems mit UML-Zustandsdiagrammen spezifizieren zu können [NW01].

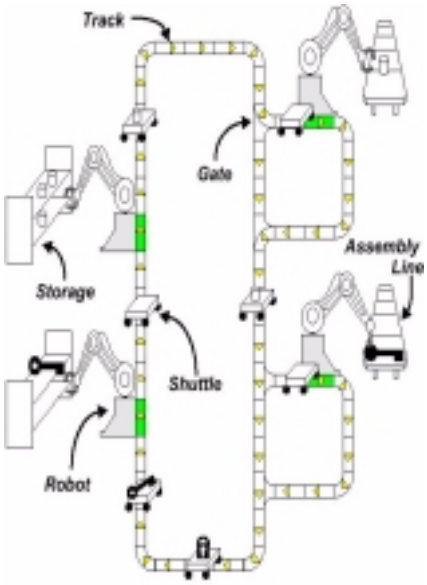


Abbildung 3.10: Fertigungssystem einer Fabrik, entnommen aus [KNNZ00]

Das Beispiel in Abbildung 3.10 wurde [KNNZ00] entnommen und zeigt ein Fertigungsystem einer Fabrik. Die in dem Beispiel modellierte Fabrikanlage besteht aus einem schienengebundenen Transportsystem, auf dem Transportfahrzeuge Werkstoffe und Waren zwischen Fertigungsanlagen befördern. Über das Schienensystem können zwei Montagestationen, ein Materiallager und ein Produktlager erreicht werden. Die beiden Montagestationen befinden sich auf zwei Nebenstrecken und können nur durch das Stellen von Weichen erreicht werden.

Erhält ein Transportfahrzeug einen neuen Auftrag, so fährt es zuerst das Materiallager an. Dort legt ein Roboterarm den gewünschten Werkstoff auf das Transportfahrzeug. Im nächsten Produktionsschritt sucht das Transportfahrzeug die Montagestation, die am wenigsten ausgelastet ist und am schnellsten erreicht werden kann. Damit der Transporter die richtige Montagestation erreicht, prüft er vor jeder Weiche, ob sie korrekt gestellt ist und ändert gegebenenfalls ihren Zustand. Wenn das Transportfahrzeug die ausgewählte Montagestation erreicht hat, nimmt der Produktionsroboter das Rohmaterial und erstellt daraus das gewünschte Produkt. Nach der Fertigstellung legt er das Produkt auf den Transporter. Dieser setzt sich in Bewegung und liefert es im Produktlager ab. Damit ist der Auftrag für das Transportfahrzeug abgeschlossen und es ist bereit neue Aufträge entgegen zu nehmen. Aus dieser Beschreibung wird nun das Softwaresystem unter Verwendung verschiedener Diagramme entworfen.

Anhand des skizzierten Szenarios können die am Fertigungsprozess beteiligten Komponenten und die notwendigen Kommunikationsstrukturen identifiziert und mit einem SDL-Blockdiagramm beschrieben werden. Dazu werden die Transportfahrzeuge, Weichen, Montagestationen und Lager als Prozesse modelliert.

Abbildung 3.11 zeigt das zugehörige SDL-Blockdiagramm, das aus dem Systemblock *Factory*, dem Block *Assembly*, sowie den Prozessen *Shuttle*, *AssemblyLine*, *Gate* und *Storage* besteht. Die Kommunikation erfolgt durch das Versenden von Signalen über die eingezeichneten Kommunikationskanäle. Besteht zwischen zwei Prozessen eine Verbindung, so können diese Prozesse Nachrichten austauschen. Existiert ein Pfad zwischen einem Prozess und dem System, so kann dieser Prozess Nachrichten auch mit der Umgebung, zum Beispiel dem Benutzer oder einem Produktionsplanungssystem (PPS), austauschen. Beispielsweise kann der Prozess *AssemblyLine* die Nachricht *produce* von dem Prozess *Shuttle* und die Nachrichten *restart* und *stop* von einem Benutzer empfangen.

Aus dieser Beschreibung wird das in Abbildung 3.12 dargestellte UML-Klassendiagramm abgeleitet. Dazu wird zu jedem Block und Prozess eine gleichnamige Klasse erstellt und durch einen Stereotypen klassifiziert. Die hierarchische Struktur des Blockdiagramms wird mit Hilfe von Aggregationen modelliert. So aggregiert die Klasse *Assembly* die Prozessklassen *Gate*, *Shuttle*, *Storage* und *AssemblyLine* und wird selbst von der Klasse *Factory* aggregiert. Die Kommunikationspfade werden hingegen durch Assoziationen modelliert. Signale, die von den Prozessen empfangen werden können, werden auf Methoden abgebildet. Der Methodenname entspricht dabei dem Signalnamen.

Das Klassendiagramm kann nun verfeinert und um neue Objektstrukturen ergänzt werden. Die Verfeinerungen sind für dieses Beispiel jedoch nicht relevant und werden aus diesem Grund in der Abbildung 3.12 nicht explizit dargestellt.

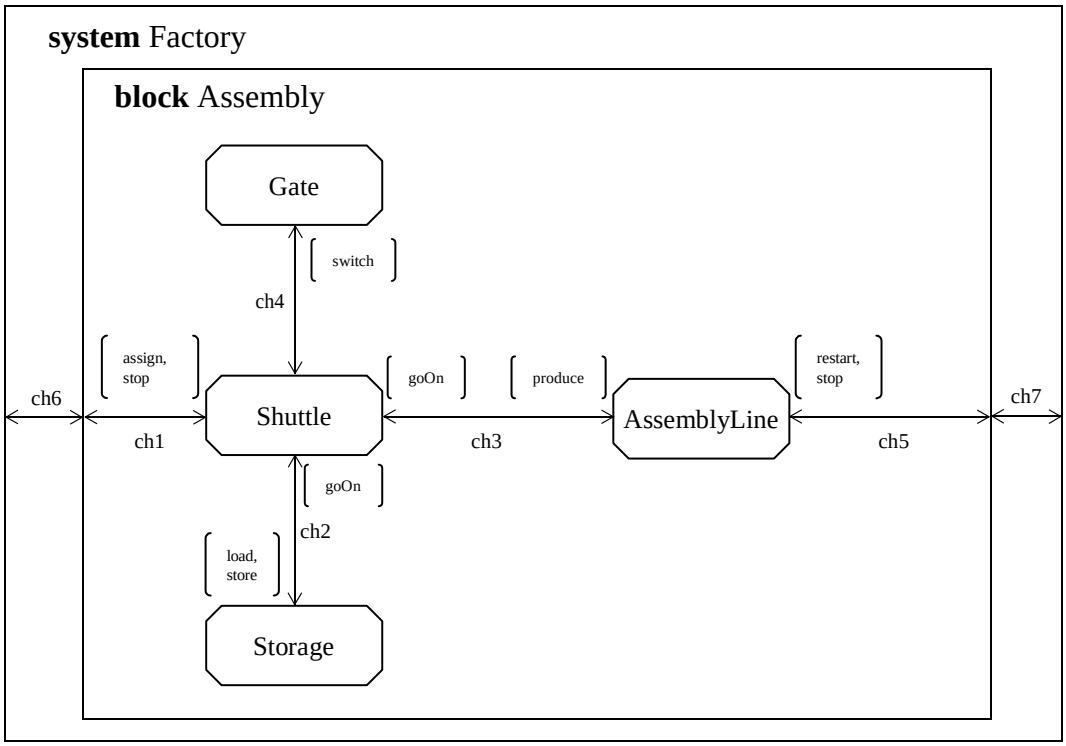


Abbildung 3.11: SDL-Blockdiagramm

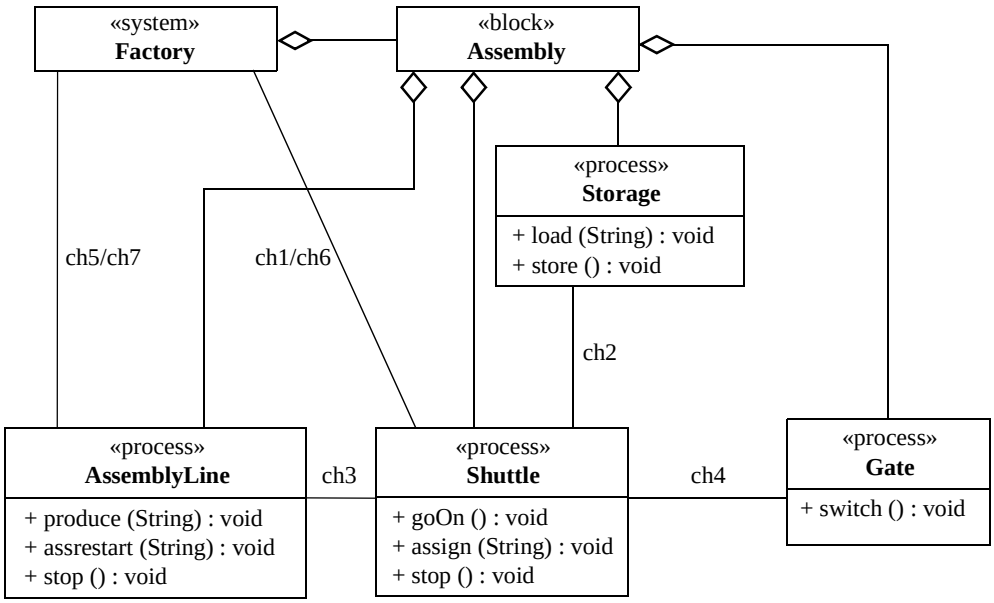


Abbildung 3.12: UML-Klassendiagramm

Um die Spezifikation der Fabrikanlage abzuschließen wird noch das Verhalten der Prozessklassen durch Zustandsdiagramme spezifiziert. Ein UML-Zustandsdiagramm definiert die erlaubten Zustandsfolgen eines Objektes oder einer in Ausführung befindlichen Operation. Ein UML-Zustandsdiagramm repräsentiert dabei einen erweiterten Zustandsautomaten nach David Harel [Har87] und setzt sich im wesentlichen aus Zuständen, Zustandsübergängen (Transitionen), Ereignissen und Aktivitäten zusammen.

Die vorgestellten Diagramme werden in Fujaba durch ASG-Objekte repräsentiert. Auf der Metamodellebene besitzen die Objekte der unterschiedlichen Diagramme untereinander keine Verbindungen. Trotzdem muss sichergestellt werden, dass das Blockdiagramm konsistent zu dem Klassendiagramm ist. So ist zu gewährleisten, dass beispielsweise die zu einem Prozess oder Block erstellte Klasse gleich benannt ist. Ebenso müssen die in einem Blockdiagramm definierten Signale in den zugehörigen Klassendefinitionen als Methoden auftauchen. Diese Bedingungen müssen auch eingehalten werden, nachdem der Benutzer eines der Diagramme verändert hat. Dazu müssen die Änderungen an einem ASG-Objekt an das dazu in Beziehung stehende ASG-Objekt weitergereicht werden. Dies muss in beide Richtungen möglich sein. Verändert der Benutzer beispielsweise den Namen einer Klasse, die einen SDL-Prozess repräsentiert, so muss der SDL-Prozess den Namen der Klasse annehmen. Erfolgt die Änderung im Blockdiagramm, so muss die Klasse den Namen des SDL-Prozesses annehmen.

Ebenso können Inkonsistenzen zwischen den Klassen in einem Klassendiagramm und einem Zustandsdiagramm auftauchen. Da die Signale beziehungsweise Ereignisse, die eine Transition auslösen, Methoden der Klasse sein müssen, kann es beim Erstellen und Bearbeiten des Zustandsdiagramms dazu kommen, dass Methoden verwendet werden, die noch gar nicht in der Klasse deklariert wurden. Diese Inkonsistenz ist durch das Einfügen der Methode in die betroffene Klasse schnell und einfach aufzulösen. Diese Ergänzung ist aber nur unidirektional möglich. Enthält eine Klasse eine Methode, die nicht im Zustandsdiagramm verwendet wird, so kann man keine Transition mit dem fehlenden Ereignis im Zustand erstellen, da die Semantik dieser Methode unbekannt ist. Ebenso wenig kann man die Methode aus der Klasse löschen, da sie womöglich an anderer Stelle verwendet wird. Die einzig sinnvolle Reaktion ist, den Benutzer auf diese potentielle Inkonsistenz aufmerksam zu machen.

Um die Konsistenzsicherung zwischen ASG-Objekten, die keine explizite Verbindung zueinander haben, zu automatisieren, wurden in dieser Arbeit TGG-Regeln aufgestellt und in das Konsistenzmanagement-System integriert.

3.3.2 Allgemeine Problematik

In der Originalarbeit [Sch94] werden zur Übersetzung einer Dokumentstruktur in eine andere Dokumentstruktur aus einer TGG-Regel drei Graphersetzungsregeln abgeleitet. Dabei handelt es sich um die Vorwärts-Regel, die Rückwärts-Regel und die Konsistenzregel. Zur Konsistenzsicherung zweier Dokumentstrukturen wurden diese Regeln in [Müh00] um die sogenannte Vorwärts-Lösch-Regel und Rückwärts-Lösch-Regel erweitert und ein sogenannter Kontroller entwickelt, der die Ausführung dieser Regeln steuert. Allerdings werden durch das dort vorgestellte Verfahren Änderungen am Zustand eines Objektes, der durch die aktuellen Werte der Attribute definiert ist, nicht erkannt. Die Konsistenzsicherung in [Müh00] ist daher nur anwendbar, wenn die Änderungen aus dem Löschen oder Erzeugen von Objekten besteht. Damit wird aber nur die Konsistenz der Objektstruktur berücksichtigt.

Dies reicht jedoch nicht aus, wenn neben der Objektstruktur auch die Attributwerte der Objekte geändert werden und die Konsistenzbedingungen davon abhängig sind. Um beispielsweise die Konsistenz zwischen den im Anwendungsbeispiel vorgestellten SDL-Prozessen und UML-Klassen sicherzustellen, muss die Konsistenzprüfung auch dann ausgeführt werden, wenn der Benutzer den Namen der Klasse oder des zugehörigen Prozesses ändert. Die Namen sind aber als Attribute in den entsprechenden ASG-Objekten hinterlegt.

Im Zusammenhang mit der zu realisierenden automatischen Korrektur von Konsistenzverletzungen existiert noch eine weitere Problematik. Unterscheidet sich nach einer Benutzermodifikation beispielsweise der Klassenname von dem Prozessnamen, so kann zwar die Inkonsistenz festgestellt werden, jedoch nicht, welcher der beiden Namen zuletzt geändert wurde und damit gültig ist.

Die Festlegung einer starren Strategie führt zu einem unbefriedigenden Verhalten des Entwicklungswerkzeugs. Wird zum Beispiel festgelegt, dass der Prozessname immer Vorrang hat, so würde nach der Modifikation des Klassennamens dieser wieder im Rahmen der automatischen Korrektur durch den Prozessnamen ersetzt werden. Damit hätten die Änderungsoperationen des Benutzers an der Klasse keine Wirkung und er wäre gezwungen, Änderungen immer nur an dem Prozess durchzuführen. Ein natürlicheres Verhalten wird erzielt, indem die Modifikationen an das jeweils andere Objekt weitergereicht werden.

Um die vorgestellten Probleme zu lösen und eine inkrementelle Konsistenzsicherung bei Änderung des Objektzustands zu verwirklichen, wird die in Abschnitt 4 beschriebene Steuerungslogik entwickelt. Zusätzlich wird die Umsetzung der TGG-Regel in die Graphersetzungsregeln angepasst, wodurch einerseits die Integration in die Steuerungslogik ermöglicht und andererseits das zuvor beschriebene und für den Benutzer natürlichere Verhalten der automatischen Korrektur erreicht wird.

3.3.3 Umsetzung in die Story-Diagramme

Um eine Konsistenzsicherung zu ermöglichen, werden zur Spezifikation von TGG-Regeln zusätzliche Klassen benötigt. Mit Hilfe dieser Klassen wird die Integrationsstruktur aufgebaut, die Objekte der beiden Diagramme zueinander in Beziehung setzt. In Abbildung 3.13 ist das Klassendiagramm für die im Anwendungsbeispiel benötigten Integrationsobjekte dargestellt. Dabei wird jede Integrationsklasse von *RootMapNode* abgeleitet und zur eindeutigen Abbildung eines SDL-Objektes auf ein UML-Objekt verwendet. Die *RootMapNode*-Klasse unterhält die beiden Assoziationen *left* und *right* zur Klasse *UMLIncrement*. Da alle ASG-Objekte von dieser Klasse abgeleitet werden müssen, können somit die unterschiedlichen Diagrammelemente zueinander in Beziehung gesetzt werden, auch wenn sie keine direkten Assoziationen aufweisen oder über einen Pfadausdruck erreichbar sind. Wie an den Kardinalitäten der Assoziationen zu erkennen ist, können ASG-Objekte in einer n:m-Beziehung zueinander stehen.

Die Klasse *RootMapNode* ist von der Klasse *UMLIncrement* abgeleitet. Damit sind alle Integrationsobjekte, die von *RootMapNode* erben, auch in den Registrierungsmechanismus eingebunden. Dies ist notwendig, um auf gelöschte Objekte im ASG reagieren zu können. Der Registrierungsmechanismus wird in Abschnitt 4.3.1 ausführlich erklärt.

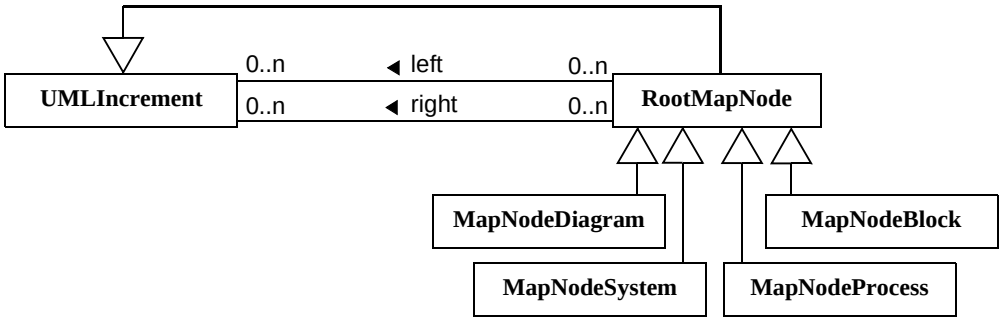


Abbildung 3.13: Klassendiagramm der Integrationsklassen

Mit Hilfe der oben dargestellten Integrationsklassen können ASG-Objekte zueinander in Beziehung gesetzt und Konsistenzbedingungen spezifiziert werden. Die Spezifikation einer *Tripel-Graph-Grammatik* wird an dem bereits bekannten Beispiel zur Transformation und Konsistenzsicherung zwischen einem SDL-Prozess und einer UML-Klasse gezeigt. In Abbildung 3.14 wird die vereinfachte TGG-Regel präsentiert, in der die Objekte zur Erzeugung der Aggregation zu Gunsten der Übersichtlichkeit weggelassen wurden. Die vollständige Regel ist im Anhang zu finden.

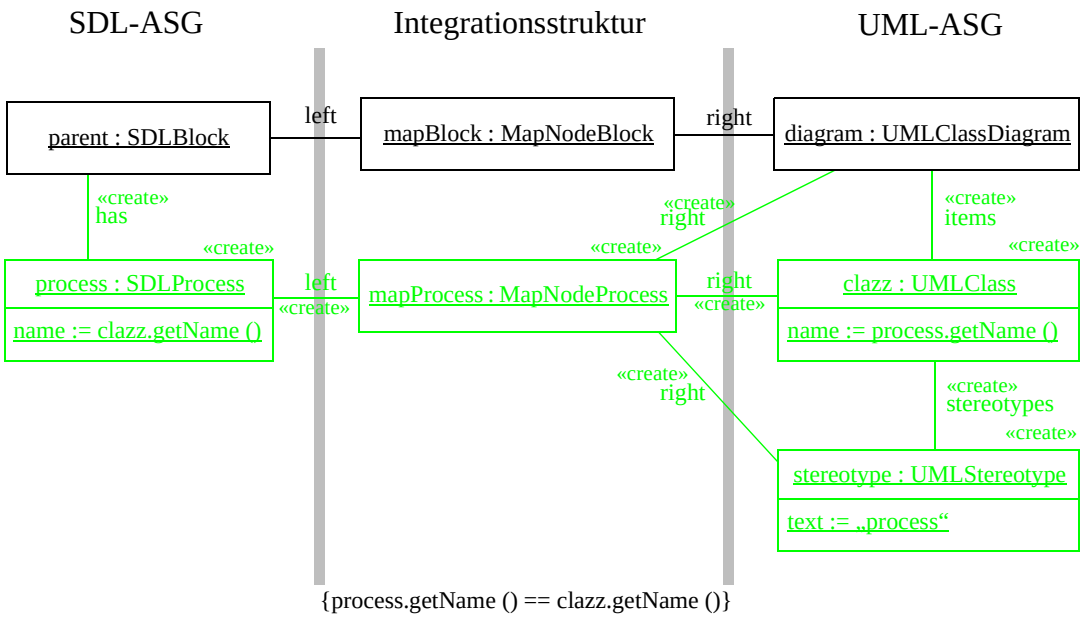


Abbildung 3.14: Eine TGG-Regel

Grundsätzlich gilt, dass eine TGG-Regel den konsistenten Zustand zwischen den in Beziehung stehenden Objekten beschreibt. Die Semantik wurde bereits in Abschnitt 2.4.2 erläutert.

Aus einer TGG-Regel lassen sich Graphersetzungsregeln ableiten, die sowohl zur Konsistenzprüfung als auch zur Konsistenzsicherung eingesetzt werden. Im Gegensatz zu [Müh00] werden die abgeleiteten Graphersetzungsregeln nicht direkt in ein *Story-Diagramm* übernommen, sondern auf die bereits erwähnten *responsible*, *check* und *repair Story-Diagramme* aufgeteilt, um zwischen nicht erfüllten Vorbedingungen, tatsächlichen Inkonsistenzen und Fehlern bei der automatischen Korrektur unterscheiden zu können. Die veränderten Graphersetzungsregeln und die Aufteilung auf die drei *Story-Diagramme* werden nun im Folgenden beschrieben.

Die Vorwärts-Regel

Die Vorwärts-Regel wird eingesetzt, um zu einem Objekt die fehlenden Objektstrukturen zu erzeugen. Dazu wird in dem *responsible Story-Diagramm* zuerst überprüft, ob in dem ASG eine Ausprägung des spezifizierten Teilgraphes zu finden ist.

In der Abbildung 3.15 wird in *responsible* die Suche begonnen, indem zuerst das übergebene Parameterobjekt in ein Objekt von Typ *SDLProcess* umgewandelt wird. Schlägt die Umwandlung fehl, so ist diese Regel nicht für die Überprüfung zuständig

und wird nicht weiter ausgeführt. Nach einer erfolgreichen Umwandlung hingegen werden alle anderen Objekte gesucht. Wurden alle Objekte der spezifizierten Objektstruktur im ASG gefunden, so wird *check* ausgeführt.

In dem *check Story-Diagramm* wird jetzt die Existenz der übrigen Objekte überprüft. Da *responsible* erfolgreich ausgeführt wurde und alle Objekte der Vorbedingung vorhanden sind, kann das Story-Diagramm nur dann nicht erfolgreich ausgeführt werden, wenn mindestens eines der zusätzlichen Objekte nicht vorhanden ist. In diesem Fall wird der boolsche Rückgabewert des *Story-Diagramms* auf *false* gesetzt und damit eine Inkonsistenz signalisiert.

Bei einer Konsistenzverletzung wird die *repair* Methode ausgeführt. Hier wird die Inkonsistenz durch die Erzeugung der noch fehlenden Objektstruktur behoben. Dabei wird die Erzeugung der Objekte als optional gekennzeichnet. Dadurch wird sichergestellt, dass bereits vorhandene Objekte nicht erneut erzeugt werden.

Da Korrekturen auch zu einem späteren Zeitpunkt stattfinden können und die Struktur sich in der Zwischenzeit verändert haben könnte, werden alle Objekte der Vorbedingung erneut gesucht und gebunden. Werden diese Objekte nicht gefunden, so schlägt die automatische Korrektur in diesem Fall fehl und *fujaba_Success* wird auf den Wert *false* gesetzt.

Die Vorwärts-Lösch-Regel

Die Vorwärts-Lösch-Regel wird eingesetzt, wenn ein Objekt des linken Diagrammes gelöscht wurde. In dem Anwendungsbeispiel ist dies ein SDL-Prozess. Durch das Löschen werden auch die Verbindungen zu dem Integrationsobjekt entfernt und damit der Zustand des Integrationsobjektes verändert. Dies führt zu der bereits erwähnten Registrierung des Integrationsobjektes bei dem Konsistenzmanagement-System, welches die Überprüfung der Konsistenz veranlasst. Dazu wird das Integrationsobjekt an das *responsible Story-Diagramm* übergeben.

Ein gelöschter Prozess ist im abstrakten Syntaxgraphen dadurch gekennzeichnet, dass zwar das Integrationsobjekt *mapProcess* und das Objekt *clazz* existieren, aber kein zugehörigers Objekt vom Typ *SDLProcess*. Dies wird in der Spezifikation des *Story-Diagramms* durch einen negativen Knoten dargestellt. Dabei ist zu beachten, dass in *responsible* mit der erfolgreichen Überprüfung der Vorbedingung bereits die Inkonsistenz festgestellt wird. Daher braucht das *check* nur noch über den Endzustand verlassen und durch den Rückgabewert *false* die Konsistenzverletzung signalisiert werden.

Wurde eine Konsistenzverletzung festgestellt, so wird im *repair Story-Diagramm* durch das Löschen des Integrationsobjektes und der in Beziehung stehenden Objekte *clazz* und *stereotype* ein konsistenter Zustand wiederhergestellt.

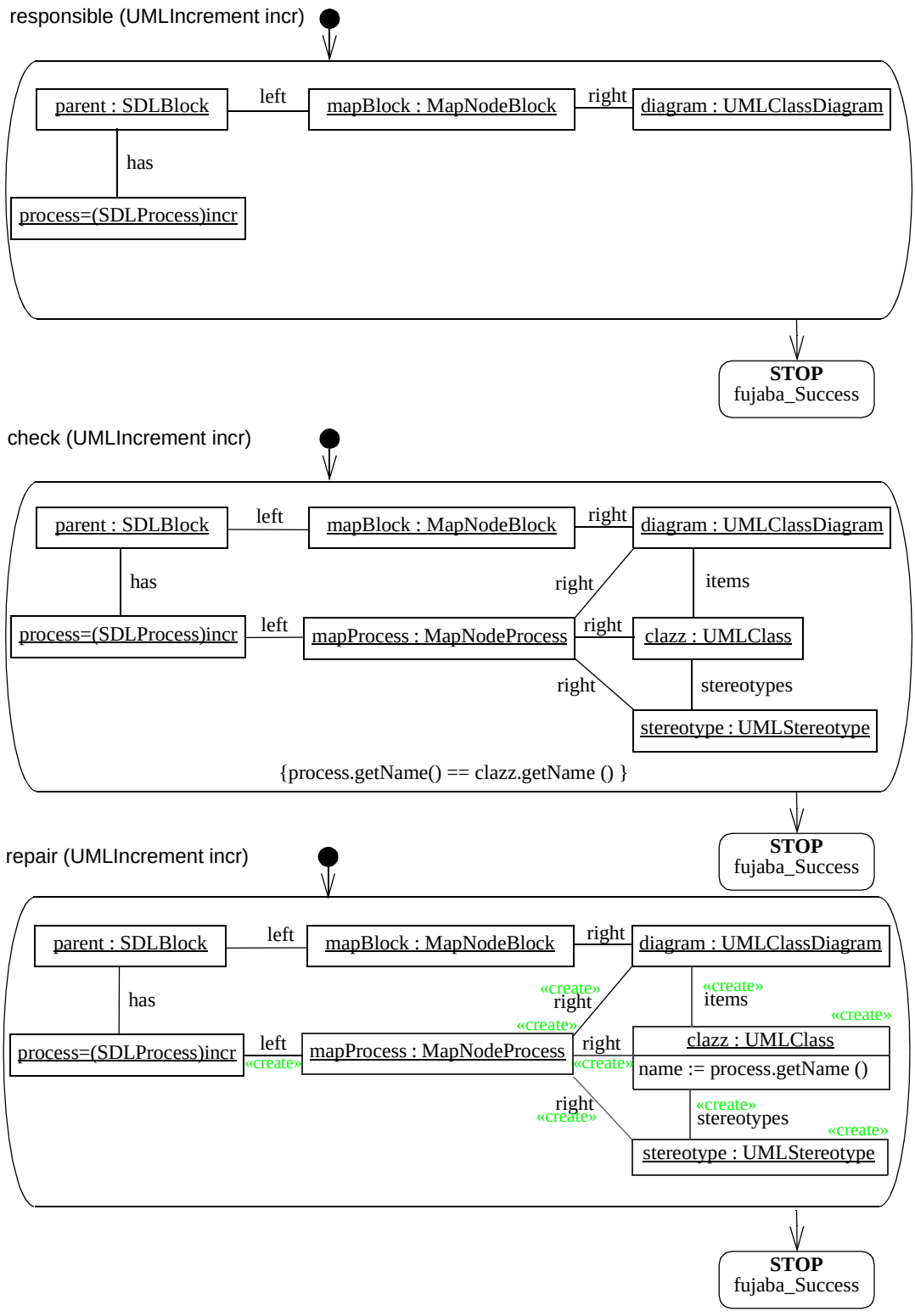
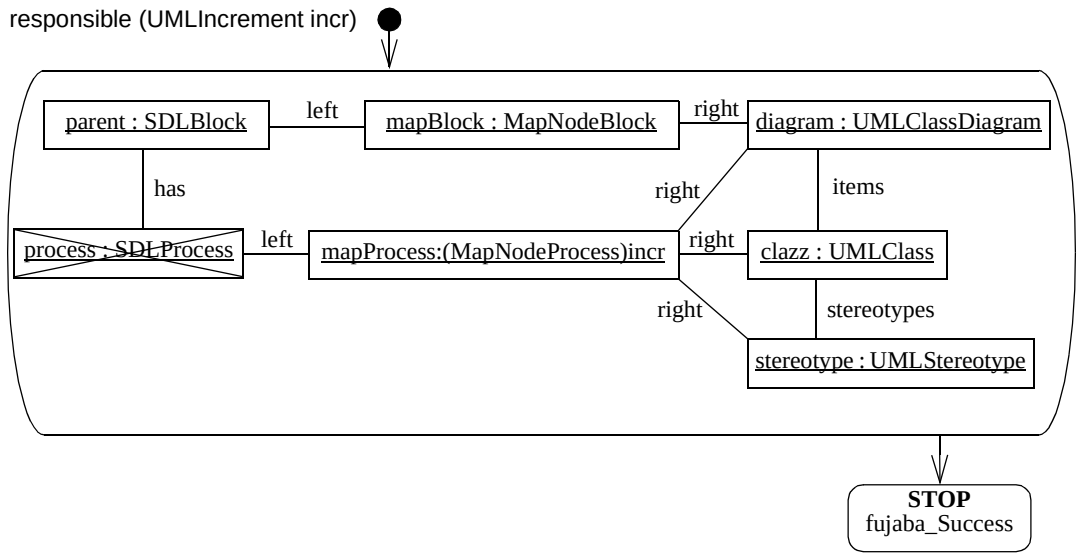
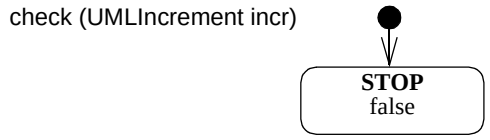


Abbildung 3.15: Story-Diagramme zur Vorwärts-Regel





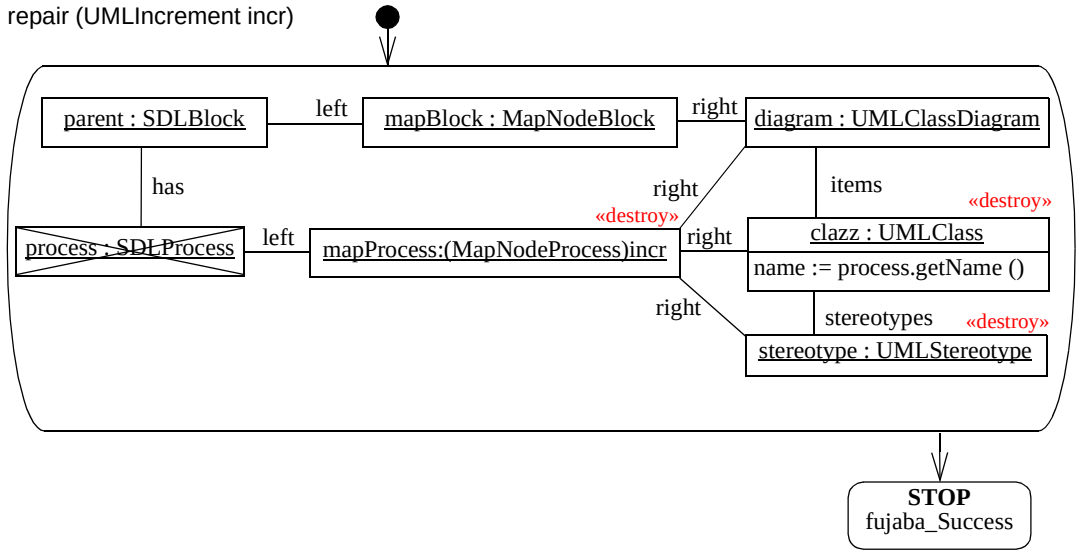


Abbildung 3.16: Story-Diagramme zur Vorwärts-Lösch-Regel

Die Vorwärts-Konsistenz-Regel

In [Sch94] und [Müh00] wird die Konsistenz-Regel verwendet, um die Existenz des Integrationsobjektes sicherzustellen. Bei der Anwendung der Regel werden keine Veränderungen an den Objekten durchgeführt. Es entsteht lediglich ein neues Integrationsobjekt.

Um das bereits beschriebene, natürliche Verhalten bei der Modifikation und automatischen Korrektur zu erzielen, wurde die alte Konsistenzregel durch zwei neue Konsistenzregeln ersetzt. Die alte Konsistenzregel wurde verworfen, da das Integrationsobjekt zwischen zwei bereits existierenden Objekten durch die Vorwärts-Regel erzeugt wird. Dies wird durch die Spezifikation von optional zu erzeugenden Objekten erreicht. Zusätzlich wurde die Semantik der beiden neuen Konsistenzregeln, der sogenannten Vorwärts-Konsistenz-Regel und der Rückwärts-Konsistenz-Regel, geändert. Abbildung 3.17 zeigt die Vorwärts-Konsistenz-Regel.

Das *responsible Story-Diagramm* unterscheidet sich zu der *check* Spezifikation nur durch das Fehlen der Bedingung, in der gefordert wird, das zwischen einem SDL-Prozess und der zugehörigen UML-Klasse Namensgleichheit bestehen muss. Es wird in der Vorbedingung also nur überprüft, ob die dargestellte Objektstruktur im ASG vorhanden ist.

Wenn *responsible* erfolgreich ausgeführt worden ist und alle Objekte gefunden wurden, kann im *check Story-Diagramm* nur noch die zusätzliche Bedingung für das Scheitern der Überprüfung zuständig sein. Ist die Bedingung falsch, so stimmt der Klassenname mit dem Prozessnamen nicht überein. In diesem Fall wird der boolsche Rückgabewert des *Story-Diagramms* auf *false* gesetzt und damit eine Inkonsistenz signalisiert.

Bei einer Konsistenzverletzung wird die *repair* Methode ausgeführt. Hier wird die Inkonsistenz durch die Zuweisung des Prozessnamens an den zugehörigen Klassennamen aufgelöst. Da Korrekturen auch zu einem späteren Zeitpunkt stattfinden können und die Struktur sich in der Zwischenzeit verändert haben könnte, werden die Objekte erneut gesucht und gebunden. Werden die Objekte nicht gefunden, so schlägt die automatische Korrektur in diesem Fall fehl und *fujaba_Success* wird auf den Wert *false* gesetzt.

Die Ableitung der Regeln für die Rückwärtsrichtung wird analog zu den vorgestellten Regeln der Vorwärtsrichtung durchgeführt. Dazu werden lediglich die Rollen der Objekte des linken und des rechten Diagramms vertauscht. Daher wird auf eine Darstellung der Rückwärtsrichtung verzichtet.

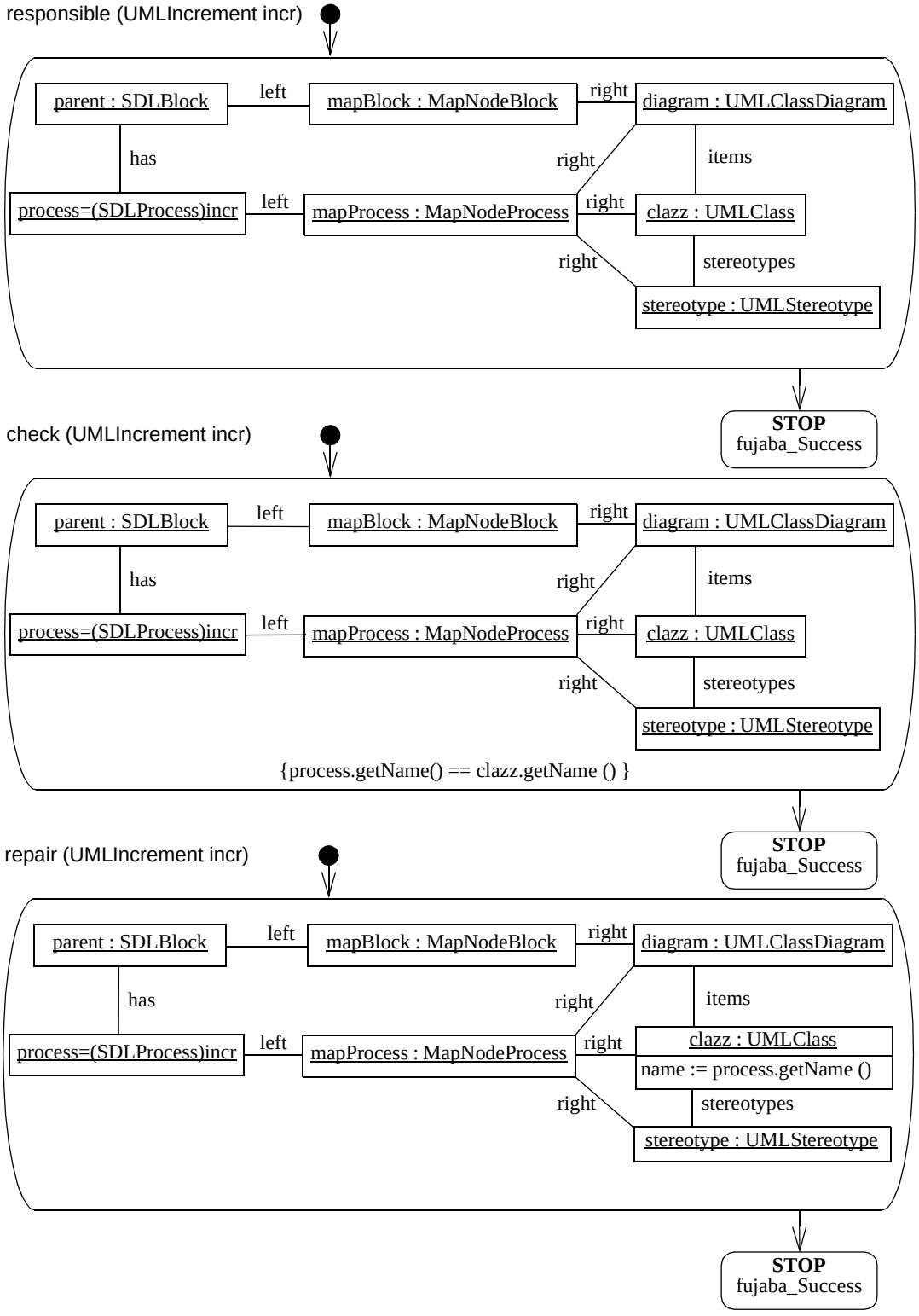


Abbildung 3.17: Story-Diagramme zur Vorwärts-Konsistenz-Regel

3.3.4 Zusammenfassung

In diesem Abschnitt wurde das Konzept der *Tripel-Graph-Grammatik* zur diagrammübergreifenden Konsistenzsicherung vorgestellt. Zur Integration in den noch vorzustellenden Steuerungsmechanismus wurden die aus der *Tripel-Graph-Grammatik* ableitbaren Graphersetzungsregeln in die drei *Story-Diagramme responsible, check* und *repair* eingeteilt.

Um inkrementelle Konsistenzprüfungen bei der Änderung von Objektzuständen durchführen zu können, wurde die Semantik der Graphersetzungsregeln angepasst. Insbesondere wurde die alte Konsistenzregel durch die sogenannte Vorwärts-Konsistenz-Regel und die Rückwärts-Konsistenz-Regel ersetzt.

Damit werden also zu einer TGG-Regel insgesamt sechs Graphersetzungsregeln abgeleitet. Dies sind für die Vorwärts-Richtung die Vorwärts-Regel, die Vorwärts-Lösch-Regel und die Vorwärts-Konsistenz-Regel. Für die Rückrichtung dementsprechend die Rückwärts-Regel, die Rückwärts-Lösch-Regel und die Rückwärts-Konsistenz-Regel. Damit entstehen also zu einer TGG-Regel insgesamt 18 *Story-Diagramme,* die jedoch alle automatisch aus der TGG-Regel erzeugt werden.

In der Umsetzung von J.H. Mühlenhoff wurde eine festgelegte Reihenfolge bei der Ausführung der einzelnen Graphersetzungsregeln gefordert, die durch einen sogenannten Kontroller eingehalten werden musste. In dem in dieser Arbeit vorgestellten Ansatz können die Graphersetzungsregeln in beliebiger Reihenfolge ausgeführt werden. Dieses wird durch den Einsatz des *responsible Story-Diagramms* und den Steuerungsmechanismus des Konsistenzmanagement-Systems erreicht, der in dem folgenden Kapitel vorgestellt wird.

4 Das Konsistenzmanagement-System

Nach der Spezifikation der Konsistenzregeln muss nun ein Mechanismus bereit gestellt werden, der diese Bedingungen überprüft und Maßnahmen bei erkannten Konsistenzverletzungen einleitet. In Abschnitt 4.1 werden die Probleme des in FUJABA bereits realisierten Analysemechanismus noch einmal kurz aufgezeigt. Um die bestehenden Probleme des alten Mechanismus zu beseitigen und den Anforderungen eines flexiblen und modularen Konsistenzmanagements nachzukommen, liegt dem Konsistenzmanagement-System ein Regelwerk zugrunde, das Regeln in Katalogen und Kategorien verwaltet. Dieses Regelwerk wird in Abschnitt 4.2 erklärt. Der Steuerungsmechanismus besteht aus der Registrierung von Objekten zur Konsistenzprüfung, der Konsistenzprüfung und der automatischen Korrektur. Außerdem ermöglicht der Mechanismus auch Konsistenzprüfungen auf Anforderung. Er wird in Abschnitt 4.3 vorgestellt.

4.1 Konsistenzanalysen in FUJABA

Bisherige Konsistenzanalysen werden in FUJABA durch eine zentrale Maschine, den *Broadcaster*, gesteuert. Für jede Konsistenzbedingung existiert eine eigene Analysemaschine, die im *Broadcaster* eingetragen ist und gestartet wird, sobald dieser erzeugt wird. Wird ein Element modifiziert, so wird es zuerst an den *Broadcaster* weitergeleitet. Dieser verteilt das Element an alle Analysemaschinen, die anhand des Elementtyps entscheiden, ob sie für die Überprüfung zuständig sind. Signalisiert eine Analysemaschine dem *Broadcaster*, dass sie das Objekt überprüfen kann, so trägt der *Broadcaster* das Element in die Warteschlange der Analysemaschine ein. Diese arbeitet die Warteschlange nach dem FIFO[1]-Prinzip ab. Jede Konsistenzprüfung findet in einem eigenen Thread statt und wird mit den Benutzerinteraktionen synchronisiert, um Zugriffskonflikte auf dem abstrakten Syntaxgraphen zu verhindern. Da die Ana-

1. First In First Out

lysemaschinen innerhalb der Autokorrektur nicht nur lesend, sondern auch schreibend auf den abstrakten Syntaxgraphen zugreifen, werden die Threads untereinander ebenfalls synchronisiert.

Die bisherigen Konsistenzanalysen haben einige gravierende Nachteile. Zunächst einmal müssen Änderungen von Konsistenzbedingungen direkt im Quellcode der betreffenden Analysemaschine vorgenommen werden. Nach der Modifikation muss die gesamte Entwicklungsumgebung neu übersetzt werden.

Eine Übersetzung der Entwicklungsumgebung muss auch dann erfolgen, wenn Konsistenzanalysemaschinen hinzugefügt oder entfernt werden, da jede Analysemaschine bei dem *Broadcaster* ein- beziehungsweise ausgetragen werden muss. Ein Hinzufügen oder Entfernen von Konsistenzbedingungen zur Laufzeit ist damit nicht möglich.

Um das Problem zu lösen, wird ein Regelwerk eingeführt, in dem die Konsistenzbedingungen in Form von Konsistenzregeln grafisch beschrieben werden können. Aus den grafischen Spezifikationen werden die neuen Konsistenzanalysemaschinen automatisch generiert und können zur Laufzeit in das Konsistenzmanagement-System eingebunden werden, ohne die Entwicklungsumgebung neu übersetzen zu müssen.

Ein weiteres Problem der bisherigen Konsistenzanalyse ist, dass die Konsistenzprüfung nicht an die Bedürfnisse eines Entwicklers anpassbar ist. Es werden immer alle Konsistenzprüfungen durchgeführt. Lediglich die Autokorrektur kann ausgeschaltet werden. Die Konfigurationsmöglichkeiten sind somit sehr eingeschränkt. Darüber hinaus sind kaum Informationen über geprüfte Elemente verfügbar. Es wird nur die verletzte Konsistenzbedingung mit einem kurzen Hinweis auf das betreffende Element dem Benutzer präsentiert. Zusatzinformationen, wie beispielsweise der Zeitpunkt oder die Aktion, die zur Konsistenzverletzung geführt haben, fehlen. Ebenso fehlt eine Beschreibung der möglichen Maßnahmen zur Behebung der Inkonsistenz.

Diesem Problem wird dadurch begegnet, dass zu jeder Konsistenzregel weitere Informationen angegeben werden können. Dabei wird unterschieden in statische Informationen zur Beschreibung von Konsistenzverletzungen, die nur zum Spezifikationszeitpunkt angegeben werden können und zur Laufzeit nicht veränderbar sind, und solchen, die zur Konfiguration und Parametrisierung der Konsistenzprüfung verwendet werden. Diese sind auch zur Laufzeit änderbar und werden zur Steuerung der Konsistenzprüfung verwendet. Dadurch wird die Anpassung der Konsistenzprüfung an die individuellen Bedürfnisse des Benutzers ermöglicht.

Ein weiteres Problem entsteht durch die nebenläufige Ausführung der Analysemaschinen und die fehlende Möglichkeit, Konsistenzprüfungen in einer festgelegten Reihenfolge durchzuführen. Die Reihenfolge bei der Überprüfung von Konsistenzbedingungen spielt erst dann eine Rolle, wenn Konsistenzverletzungen automatisch

korrigiert werden. Die Korrektur einer Konsistenzverletzung an einer Stelle kann als Seiteneffekt an einer anderen Stelle sowohl neue Inkonsistenzen verursachen als auch bestehende Inkonsistenzen auflösen. Dies trifft insbesondere auf die automatische Ergänzung von unvollständigen Spezifikationen zu. Werden diese Ergänzungen in der falschen Reihenfolge durchgeführt, so kann dies einerseits zu falschen Ergebnissen führen und andererseits unnötig viel Aufwand verursachen, da das vervollständigte Modell in seiner Gesamtheit ebenfalls auf Konsistenz geprüft werden muss.

Aus diesem Grund wurde in dieser Arbeit ein Ansatz gewählt, der die im Regelwerk spezifizierte Reihenfolge bei der Konsistenzprüfung berücksichtigt und auf die nebenläufige Ausführung der einzelnen Konsistenzanalysemaschinen verzichtet. Damit der Prüfungsaufwand jedoch vertretbar bleibt und den Benutzer in seiner Arbeit nicht behindert, wurde ein Mechanismus entwickelt, der sowohl die Menge der zu überprüfenden Objekte als auch die Menge der zu überprüfenden Regeln minimiert. Es wird zusätzlich skizziert, wie der vorgestellte Mechanismus in einer nebenläufigen Konsistenzprüfung und Autokorrektur eingesetzt werden kann.

Die oben vorgestellten Ansätze und Konzepte zur Realisierung des Konsistenzmanagementsystems werden in den folgenden Abschnitten nun detailliert beschrieben.

4.2 Das Regelwerk

4.2.1 Organisation der Konsistenzregeln

Konsistenzregeln werden in Katalogen zusammengefasst. Jeder Katalog kann verschiedene Kategorien enthalten, die wiederum aus Unterkategorien zusammengesetzt sein können. Außer den Konsistenzregeln können im Katalog auch Verweise angelegt werden. Ein Verweis zeigt entweder auf eine Regel oder auf eine Kategorie. Ein Verweis auf den Katalog selbst oder andere Kataloge ist nicht möglich.

Durch die Verwendung von Kategorien und Unterkategorien ergibt sich eine baumartige, hierarchische Katalogstruktur, in der die Konsistenzregeln nach unterschiedlichen Kriterien organisiert und verwaltet werden können. Beispielsweise können Konsistenzregeln nach Regeln der Objekttechnologie, der verwendeten Modellierungssprache, der Diagrammart oder der einzelnen Diagrammelemente angeordnet werden. Ebenso ist eine Strukturierung nach Regeln der Zielsprache, Regeln der Anwendungsdomäne, vorhandener oder nicht vorhandener Autokorrektur oder

der Fehlerkategorie möglich. Da die Kategorien nicht festgelegt sind und vom Benutzer beliebig erstellt werden können, sind hier keine Grenzen gesetzt.

Des weiteren können aus den spezifizierten Regeln unterschiedliche Prüfungsprofile erstellt werden. Jedes Profil kann durch eigene Kategorien strukturiert werden und enthält Verweise auf bereits definierte Konsistenzregeln. Hat der Benutzer unterschiedliche Profile erstellt, so kann er zur Laufzeit dann ein geeignetes Profil auswählen. Beispielsweise kann man ein Profil für den Entwurf und ein anderes Profil für die Analyse von Modellen erstellen. Da während der Analyse nur ein konzeptionelles Modell erstellt wird, in dem viele Details vernachlässigt werden können, kann aus dem entsprechenden Profil auf nur wirklich wichtige Konsistenzregeln verwiesen werden. Dadurch wird der Benutzer nicht mit unnötigen Informationen überhäuft, für die er sich in dieser Phase der Modellierung nicht interessiert.

Ein Profil kann aber auch verwendet werden, um eine Reihenfolge festzulegen, in der die Konsistenzregeln überprüft werden. So kann der Katalog durch die Verwendung von Kategorien unabhängig von der Ausführungsreihenfolge der Regeln nach frei wählbaren Kriterien strukturiert werden. Die Reihenfolge wird dann erst in einem Ausführungsprofil festgelegt und ergibt sich dann aus der Anordnung der Regeln innerhalb der hierarchischen Struktur des Ausführungsprofils. Dabei werden Konsistenzregeln, die weiter oben in der Hierarchie zu finden sind, auch zuerst überprüft.

4.2.2 Informationen für den Benutzer

Um das Ergebnis einer Konsistenzprüfung mit weiteren Informationen anzureichern und diese dadurch aussagekräftiger zu gestalten, können zusätzliche Angaben zu jeder Regel im Katalog hinterlegt werden. Diese Angaben dienen als zusätzliche Hilfen für Benutzerinteraktionen und werden in verschiedenen Dialogen aufgeführt.

Regelname und Beschreibung

Außer einem aussagekräftigen Regelnamen, der bereits einen ersten Hinweis auf den Sinn der Konsistenzregel geben sollte, kann eine ausführliche Beschreibung der durch die Regel geprüften Bedingung angegeben werden. Diese Beschreibung wird sowohl bei der Konfiguration der Konsistenzprüfung als auch bei der Präsentation von Prüfungsergebnissen angezeigt.

Meldung bei Konsistenzverletzung

Diese Beschreibung wird dem Benutzer zur Meldung einer Inkonsistenz präsentiert und spezifiziert die verletzte Bedingung detaillierter. Hier werden mögliche Ursachen, die zur Inkonsistenz des Objektes geführt haben, beschrieben.

Maßnahmen zur Behebung der Inkonsistenz

Obwohl die zuvor genannte Meldung bereits erste Hinweise zur Behebung der Inkonsistenz geben kann, ist es möglich, eine weitere und konkretere Beschreibung in der Konsistenzregel zu hinterlegen, die mögliche Maßnahmen zur Auflösung der Konsistenzverletzung vorschlägt.

Beschreibung der Autokorrektur

Wurde zur Konsistenzregel eine automatische Korrekturmaßnahme spezifiziert, so können die durchgeführten Aktionen zur Behebung der Inkonsistenz an dieser Stelle beschrieben werden.

Fehlerkategorie

Schließlich kann der Regel noch eine Fehlerkategorie zugeordnet werden. Die unterschiedlichen Kategorien erlauben, den Schweregrad einer Konsistenzverletzung einzuordnen. Zur Auswahl stehen die Kategorien *Fehler*, *Warnung* und *Optimierung*. Ein *Fehler* liegt vor, wenn grundsätzliche Fehler bei der Modellierung gemacht worden sind. *Warnungen* sind für unvollständige Spezifikationen und kleinere Fehler gedacht. Die Kategorie der *Optimierung* ist für Konsistenzverletzungen vorgesehen, die im herkömmlichen Sinne keine Fehler sondern nicht eingehaltene Standards darstellen.

4.2.3 Konfiguration

Die Konsistenzprüfung ist weitgehend konfigurierbar. Hierfür kann im Katalog für jede Konsistenzregel eine Standardkonfiguration hinterlegt werden. Diese Konfiguration kann zur Laufzeit des Systems an die individuellen Bedürfnisse des Benutzers angepasst werden. Hierfür stehen verschiedene Optionen zur Verfügung, die im Folgenden aufgezählt werden.

Prüfungsmodus

Eine Konsistenzregel kann entweder automatisch überprüft werden oder erst nach Aufforderung durch den Benutzer. Ebenso ist ein Deaktivieren der Konsistenzregel möglich. Bei einer automatischen Konsistenzprüfung muss zusätzlich festgelegt werden, nach welchen Aktionen die Regel überprüft werden kann. Hierzu kann der Benutzer aus einer definierten Menge von möglichen Aktionen ein oder mehrere Aktionen der Regel zuweisen.

Korrekturmodus

Wurde eine Korrekturmaßnahme zu einer Konsistenzregel spezifiziert, so muss der Korrekturmodus festgelegt werden. Nach der Erkennung einer Inkonsistenz kann die Fehlerkorrektur entweder automatisch oder interaktiv ausgeführt werden. Während bei der automatischen Fehlerkorrektur die Inkonsistenz sofort behoben wird, informiert die interaktive Konsistenzprüfung den Benutzer über die gefundene Inkonsistenz ohne diese sofort zu beheben und schlägt neben der Autokorrektur weitere Maßnahmen zur Inkonsistenzauflösung vor. Der Benutzer hat damit die Möglichkeit, in den Korrekturprozess einzuschreiten und entweder die vorgeschlagene Autokorrekturmaßnahme auszuführen oder aber eine der anderen Maßnahmen einzuleiten.

Fehlerbenachrichtigung

Unabhängig davon, ob eine automatische Korrekturmaßnahme spezifiziert wurde, kann zusätzlich festgelegt werden, ob bei einer erkannten Inkonsistenz der Benutzer sofort benachrichtigt werden soll. Die sofortige Benachrichtigung unterbricht den Benutzer durch das Einblenden eines Dialogs, der Informationen zu der Konsistenzverletzung enthält. Diese Maßnahme kann sehr störend sein und sollte daher nur für besonders schwere Konsistenzfehler benutzt werden.
Sind sowohl die Fehlerbenachrichtigung als auch die automatische Korrektur eingeschaltet, so wird der Fehler automatisch behoben und der Benutzer trotzdem informiert. Ist hingegen der interaktive Korrekturmodus eingestellt, so erfolgt keine zusätzliche Meldung, da dies bereits durch den Interaktionsdialog geschehen ist.

Die oben vorgestellten Einstellungen werden für jede Regel einzeln vorgenommen. Möchte der Benutzer die automatische Korrektur oder die Fehlerbenachrichtigung zeitweise für alle Regeln ausschalten, so kann er die Fehlerbenachrichtigung und die Autokorrektur in zwei gesonderten Einstellungsoptionen regelübergreifend unterdrücken. Dabei werden die regelspezifischen Einstellungen nicht überschrieben und sind nach der Deaktivierung der Unterdrückung wieder wirksam.

4.3 Ausführungsmechanismus

4.3.1 Registrierung zur automatischen Konsistenzprüfung

Ein Softwareentwurf besteht aus mehreren Diagrammen und Diagrammelementen, die zusammen das Gesamtmodell der Anwendung ergeben. Geht man von einem kon-

sistenten Zustand des Gesamtmodells aus, so können Inkonsistenzen im Modell erst dann entstehen, wenn das Modell verändert wird. In der Entwicklungsumgebung FUJABA geschieht dies durch das Ausführen von Benutzerkommandos. Ein Benutzerkommando ist eine festgelegte Aktion, die aus mehreren Anweisungen bestehen kann. Während der Ausführung der Anweisungen können temporär Inkonsistenzen auftreten, die jedoch unter Umständen wieder aufgelöst werden. Eine Konsistenzprüfung nach jeder Modifikation des ASG wäre damit verfrüht und würde falsche Ergebnisse liefern. Daher findet die Konsistenzprüfung erst statt, wenn alle Anweisungen des Benutzerkommandos abgearbeitet worden sind.

In dem bisherigen Analysemechanismus wurde dies durch den Synchronisationsmechanismus erreicht, der auch verwendet worden ist, um Konflikte zwischen konkurrierenden Schreib- und Lesezugriffen zwischen einer Benutzeraktion und der Konsistenzprüfung bzw. Autokorrektur auf dem ASG zu verhindern.

Dazu wurde vor der Ausführung einer Benutzeraktion eine Nachricht an eine zentrale Instanz, den *Broadcaster*, geschickt und solange gewartet, bis dieser signalisiert hat, dass er alle Analysemaschinen angehalten hat. Erst dann wurde die Benutzeraktion ausgeführt.

Innerhalb der Benutzeraktion modifizierte ASG-Objekte wurden mit Hilfe des Alterungsmechanismus [FNT98] bei dem *Broadcaster* registriert und an die Analysemaschinen weitergeleitet. Da zu diesem Zeitpunkt jedoch alle Analysemaschinen inaktiv waren, wurde eine zu frühe Konsistenzprüfung verhindert.

Am Ende der Benutzeraktion mussten die Analysemaschinen durch eine Nachricht an den *Broadcaster* wieder gestartet werden und haben erst dann die durch die Benutzeraktion modifizierten Objekte auf Konsistenz überprüft.

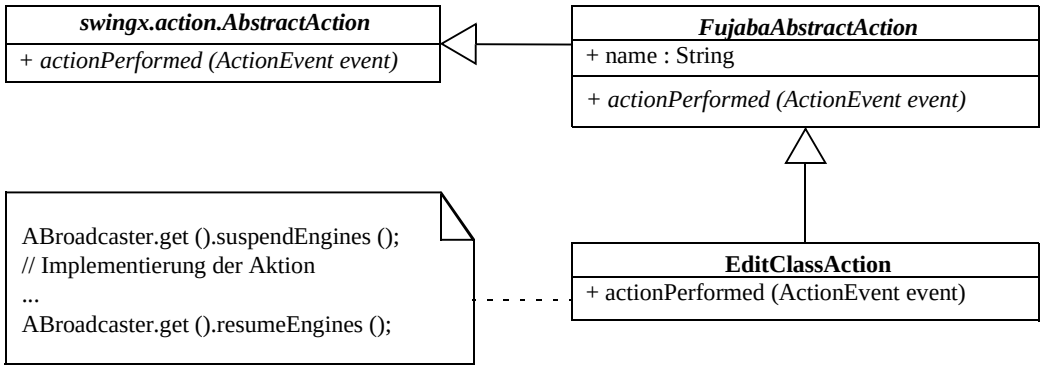


Abbildung 4.1: Verhinderung von Analysen während der Modifikation

In Abbildung 4.1 ist ein Ausschnitt aus dem Klassendiagramm der alten Implementierung zu sehen. Das Klassendiagramm enthält die abstrakte Klasse *FujabaAb-*

stractAction, die von der Klasse *swingx.action.AbstractAction* abgeleitet ist, sowie die konkrete Benutzeraktion *EditClassAction*. Diese Klasse erbt von *FujabaAbstractAction* und implementiert die Methode *actionPerformed*. Diese Methode wird als Reaktion auf das Ausführen eines Menüeintrags oder das Drücken einer Schaltfläche durch die GUI[1] automatisch aufgerufen. Der Aufruf der beiden Methoden *suspendEngines* und *resumeEngines* liegt in der Verantwortung des Programmierers. Fehlen diese Nachrichten in dem Benutzerkommando, so ist die Synchronisation nicht mehr gegeben.

Um die oben genannte Problematik zu umgehen wurde der Ablauf und die Registrierung der ASG-Objekte geändert. In Abbildung 4.2 ist der neue Ansatz schematisch dargestellt.

Wird ein Benutzerkommando aufgerufen, so erzeugt es zuerst ein sogenanntes *Kontext*-Objekt. Der *Kontext* einer Benutzeraktion ist ein *Container*-Objekt, das ASG-Objekte aufnehmen kann. Da Benutzerkommandos auch verschachtelt ablaufen können, wird der neue *Kontext* auf einen Stack gelegt.

Nach dieser Vorbereitung werden nun die Anweisungen der Benutzeraktion ausgeführt. Innerhalb der Benutzeraktion wird sowohl lesend als auch schreibend auf den abstrakten Syntaxgraphen zugegriffen. Bei schreibenden Zugriffen werden die ASG-Objekte im *Kontext* registriert. Für die Registrierung wird weiterhin der Alterungsmechanismus benutzt. Mit dessen Hilfe werden die modifizierten ASG-Objekte im *Kontext* gespeichert. Ein mehrfach modifiziertes ASG-Objekt wird aber nur einmal erfasst.

Wurden alle Anweisungen ausgeführt, so wird vor der Beendigung der Benutzeraktion die Konsistenzprüfung gestartet. Der Prüfungsmechanismus nimmt das oberste *Kontext*-Objekt vom Stapel und überprüft alle dort gespeicherten ASG-Objekte. Da die Konsistenzprüfung innerhalb des Benutzerkommandos stattfindet, können in dieser Zeit keine weiteren Benutzeraktionen stattfinden.

Jedem Benutzerkommando wird ein sogenannter Kontextname zugewiesen. Dabei kann entweder jedem Kommando ein eigener oder aber auch mehrern Benutzerkommandos ein einziger Name zugewiesen werden. In der Regelspezifikation können dann die Kontextnamen angegeben werden, bei denen eine Überprüfung der Regel stattfinden soll. Die Regel wird nur dann überprüft, wenn der Kontextname der Benutzeraktion in der Liste der Kontextnamen der Konsistenzregel enthalten ist. Somit wird die Überprüfung einer Konsistenzregel an die Ausführung eines Benutzerkommandos gebunden und die zu überprüfende Regelmenge eingeschränkt. Durch dieses Verfahren wird die Antwortzeit des Systems niedrig gehalten.

1. Graphical User Interface

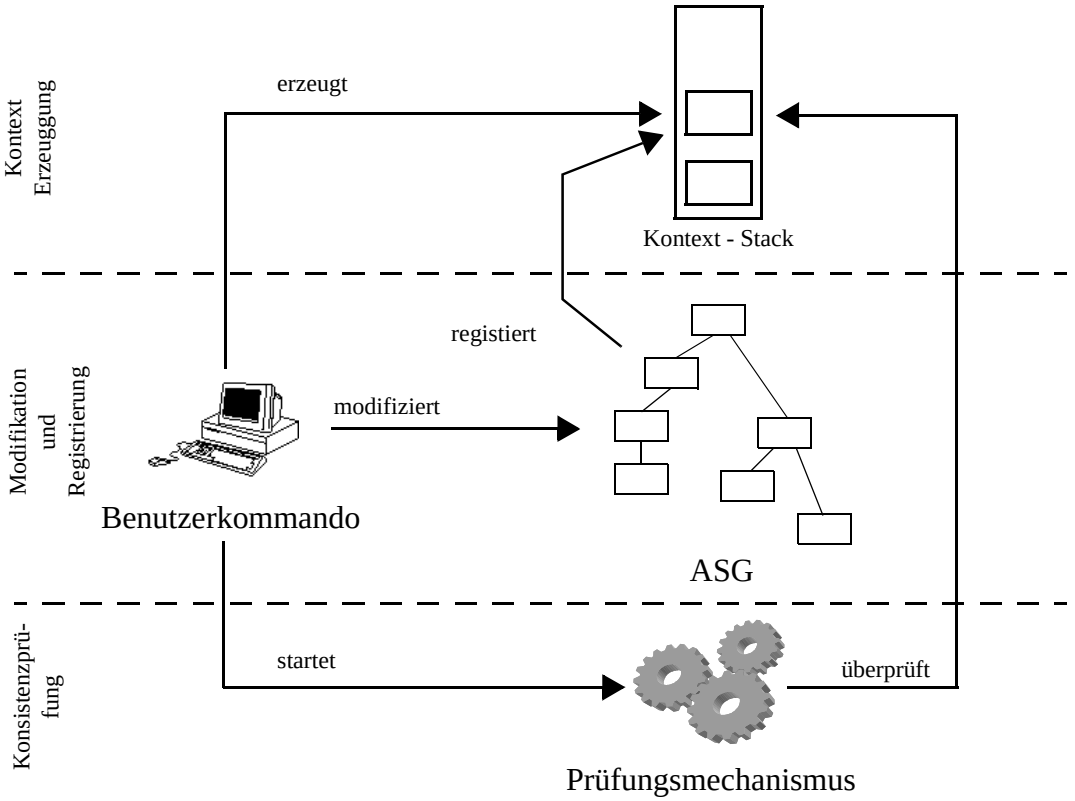


Abbildung 4.2: Registrierung zur Konsistenzprüfung

4.3.2 Registrierung zur manuellen Konsistenzprüfung

Neben der automatischen Konsistenzprüfung werden auch manuelle Konsistenzprüfungen angeboten, die erst auf Anforderung des Benutzers stattfinden. Dabei kann der Benutzer zwischen den im Folgenden vorgestellten Möglichkeiten wählen.

Modifizierte Elemente

Es werden alle modifizierten Elemente geprüft, die nicht von der automatischen Konsistenzprüfung bereits erfasst worden sind. Dies ist insbesondere dann der Fall, wenn die automatische Konsistenzprüfung ausgeschaltet worden ist.

Alle Diagramme

Bei dieser Konsistenzprüfung wird das gesamte Projekt mit allen Diagrammen und den darin enthaltenen Diagrammelementen überprüft. Die Prüfung der Elemente findet unabhängig davon statt, ob die Diagrammelemente modifiziert wurden oder nicht.

Daher ist diese Form der Konsistenzprüfung besonders dazu geeignet, die Konsistenz des Gesamtmodells zu untersuchen.

Ausgewähltes Diagramm

Entscheidet sich der Benutzer für diese Variante, dann werden nur das zur Zeit aktive Diagramm und die darin enthaltenen Diagrammelemente überprüft.

Ausgewählte Elemente

Bei dieser Prüfung werden nur die ausgewählten Elemente überprüft. Damit hat der Benutzer die Möglichkeit, einzelne Elemente auf Konsistenz zu überprüfen. In Verbindung mit der Möglichkeit, einzelne Konsistenzregeln ein- und auszuschalten, kann also eine gezielte Konsistenzprüfung stattfinden.

Zur Erfassung der modifizierten Diagrammelemente wird der bereits vorgestellte Mechanismus der automatischen Registrierung benutzt. Dazu wird bei der Initialisierung des Konsistenzmanagement-Systems ein separates *Kontext*-Objekt erzeugt und gesondert verwaltet. Jedes modifizierte Objekt wird zusätzlich zu dem vom Benutzerkommando erzeugten Kontext in diesen speziellen Kontext eingetragen. Dieser Kontext wird für die manuelle Überprüfung der modifizierten Elemente benutzt.

Für die letzten drei Überprüfungsoptionen wird das Design-Pattern *Visitor* nach Gamma et al. [GHJV95] benutzt. Die Struktur des Visitor-Patterns ist in Abbildung 4.3 zu sehen. Das Entwurfsmuster *Visitor* kapselt eine auszuführende Aufgabe in einer separaten Klasse und ermöglicht somit die Trennung von Datenstruktur, Traversierung und Algorithmus. Dabei muss jede Klasse der Datenstruktur von einer gemeinsamen Basisklasse abgeleitet sein, die eine *accept* Methode implementiert. Dieser Methode wird ein konkretes *Visitor*-Objekt übergeben, dass für eine spezielle Aufgabe verantwortlich ist. Dazu implementiert jeder *Visitor* für jedes Objekt der Datenstruktur eine eigene Methode, die aufgerufen wird, wenn das Objekt besucht wird. In dieser Methode wird die Traversierung und die Funktionalität des *Visitors* implementiert.

Für die Traversierung des abstrakten Syntaxgraphen und die Registrierung der zu überprüfenden Modellelemente wird solch ein Visitor benutzt. Dazu wird vorher ein Kontext-Objekt erzeugt, in das der Visitor die besuchten ASG-Objekte einträgt.

Abhängig davon, auf welchem ASG-Objekt die Traversierung mit dem Visitor gestartet worden ist, ergeben sich die drei zuletzt beschriebenen Varianten. Startet der Visitor also auf der Wurzel des abstrakten Syntaxgraphen, so werden alle Diagramme und ihre Elemente in den Kontext eingetragen und damit das gesamte Projekt überprüft. Wird der Visitor auf dem aktiven Diagramm gestartet, so wird dieses Diagramm-Objekt sowie dessen Elemente in den Kontext eingefügt. Da nicht alle

Objekte eines Diagrammes separat auswählbar sind, wird bei der letzten Variante ebenfalls der Visitor gestartet. Dieser besucht dann die untergeordneten Objekte. Somit kann man beispielsweise Attribute und Methoden einer Klasse durch die Auswahl der Klasse ebenfalls überprüfen.

Ist die Traversierung abgeschlossen, so befinden sich die zu überprüfenden ASG-Objekte in dem Kontext-Objekt, das dann durch den Konsistenzprüfungsmechanismus überprüft wird.

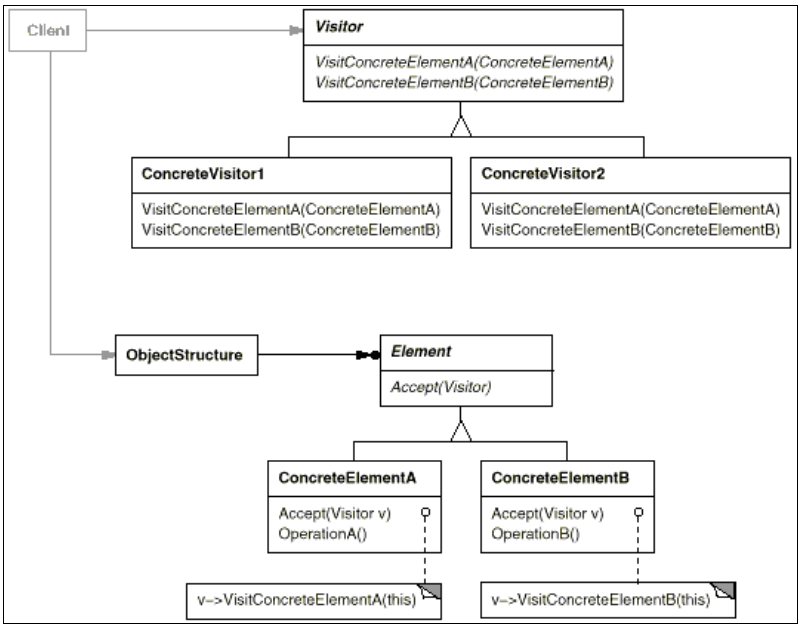


Abbildung 4.3: Das Design-Pattern Visitor nach Gamma et al. [GHJV95]

4.3.3 Konsistenzprüfung und Korrektur

Die Konsistenzprüfung wird am Ende einer Benutzeraktion von einer zentralen Instanz gestartet, die das Rahmenwerk für die Prüfung und die automatische Korrektur bereit stellt. Die Basis für die Prüfungen bildet der Regelkatalog und die darin enthaltenen Konsistenzregeln. In einem Durchlauf wird über den gesamten Katalog iteriert und jede Regel mit der zugehörigen Analysemaschine überprüft.

Die automatische Korrektur von Konsistenzverletzungen ist in dem Prüfungsprozess integriert, kann aber auch einzeln zu einem späteren Zeitpunkt gestartet werden. Durch die Korrektur eines Objektes werden Veränderungen an dem betreffenden Ob-

jekt selbst sowie an beteiligten Objekten vorgenommen, die zu Inkonsistenzen führen können. Daher werden diese Objekte während eines Durchlaufs durch den Regelkatalog gesammelt und wieder einer Konsistenzprüfung unterzogen.

Da nicht auszuschließen ist, dass in einem Katalog fehlerhafte oder sich widersprechende Konsistenzregeln und Korrekturmaßnahmen spezifiziert worden sind, kann dabei ohne zusätzliche Maßnahmen ein Zyklus auftreten, der immer wieder Konsistenzverletzungen behebt, die durch eine vorherige Korrektur entstanden sind. Der Algorithmus wäre in einer Endlosschleife und würde niemals terminieren.

Um dies zu verhindern, werden die reparierten Objekte in einer Tabelle gespeichert, in der auch vermerkt ist, von welcher Regel die Korrektur vorgenommen wurde. Wird bei einem nachfolgendem Prüfungszyklus wieder eine Inkonsistenz des Elementes bezüglich dieser Regel festgestellt, so wird die Korrektur verhindert und der Benutzer über den Konflikt informiert.

Das Rahmenwerk für die Konsistenzprüfung wurde in der Klasse *RTExecMgr* realisiert. Die Iteration über den Katalog findet in der Methode *verifyAutomatically* der Klasse *RTCategory* statt. Der Katalog selbst ist von *RTCategory* abgeleitet und erbt damit ebenfalls diese Methode. Die Konsistenzprüfung und Korrektur wird von jeder Regel eigenständig gesteuert. Die Steuerung wurde in der Methode *verifyAutomaically* der Klasse *RTRule* realisiert, die Methoden zur Überprüfung und Korrektur sind in der zugehörigen Analysemaschine implementiert. Die an der Konsistenzprüfung beteiligten Klassen und die wichtigsten Methoden sind in Abbildung 4.4, 4.5 und 4.6 in Pseudocode dargestellt. Sie beschreiben den Konsistenzprüfungsmechanismus mit der integrierten Korrektur von Konsistenzverletzungen.

Die Konsistenzprüfung wird durch den Aufruf der Methode *checkContext* des *ExecutionManagers* (*RTExecMgr*) gestartet und nur durchgeführt, falls die automatische Konsistenzprüfung eingeschaltet ist (Abbildung 4.4, Zeile 1).

Der *ExecutionManager* verwaltet neben den Regelkatalogen und der Korrekturtabelle auch einen Stack, der vor dem Aufruf der Konsistenzprüfung ein Kontext-Objekt enthalten muss. Wie bereits erwähnt worden ist, verwaltet das Kontext-Objekt alle zu überprüfenden Elemente, die entweder durch das in Abschnitt 4.3.1 oder Abschnitt 4.3.2 dargestellte Verfahren dort eingetragen wurden. Der Kontext kann jedoch auch leer sein, falls beispielsweise Benutzeraktionen abgebrochen worden sind. In diesem Fall findet ebenfalls keine Konsistenzprüfung statt (Abbildung 4.4, Zeile 2 und 3).

Zur Konsistenzprüfung wird das Kontext-Objekt, das im Folgenden als Prüfungskontext bezeichnet wird, vom Stack entfernt (Zeile 5) und die Umgebung für die automatische Korrektur vorbereitet, indem ein Korrekturkontext erzeugt und auf den Stapel gelegt wird (Zeile 6 und 7). Da die während einer automatischen Korrektur modifizierten Elemente ebenfalls über den bereits in Abschnitt 4.3.1 beschriebenen

Registrierungsmechanismus erfasst werden, wird der Korrekturkontext als Seiteneffekt der automatischen Korrektur mit modifizierten Elementen gefüllt.

```
RTExecMgr::checkContext ()
 1:    if automatic checking is enabled
 2:        context = contextStack.top ()
 3:        if context not empty
 4:            do
 5:                checkContext = contextStack.pop ()
 6:                repairContext = new RTContext (checkContext.getName ())
 7:                contextStack.push (repairContext)
 8:                for all catalogs do
 9:                    if catalog is enabled
10:                        catalog.verifyAutomatically (checkContext)
11:            while repairContext not empty
12:            contextStack.pop ();
13:            removeAllFromAutoRepairTable ()
```

Abbildung 4.4: Die Methode *checkContext* der Klasse *RTExecMgr*

Nach dieser Vorbereitung startet die eigentliche Konsistenzprüfung, indem über alle Kataloge iteriert und auf jedem Katalogeintrag die Methode *verifyAutomatically* aufgerufen wird (Abbildung 4.4, Zeile 8 bis Zeile 10). Diese Methode erhält den Prüfungskontext als Parameter.

Ein Katalog ist ein Speziallfall einer Kategorie und benutzt deshalb die geerbte Methode *verifyAutomatically* der Klasse *RTCategory*. In dieser Methode wird über alle Einträge iteriert und auf jedem Eintrag wiederum *verifyAutomatically* aufgerufen (Abbildung 4.5, Zeile 1 bis Zeile 3). Dadurch wird die Kataloghierarchie rekursiv durchlaufen und jeder Eintrag besucht.

```
RTCategory::verifyAutomatically (RTContext context)
1:    for all entries do
2:        if entry is enabled
3:            entry.verifyAutomatically (context)
```

Abbildung 4.5: Die Methode *verifyAutomatically* der Klasse *RTCategory*

Ist der Eintrag eine Konsistenzregel, so wird die Konsistenzbedingung nur dann überprüft, wenn die Regel für diesen Kontext spezifiziert worden ist (Abbildung 4.6, Zeile 1). Dabei wird nacheinander jedes Element aus dem Kontext betrachtet und festgestellt, ob die der Regel zugeordnete Analysemaschine für das betrachtete Element zuständig ist (Abbildung 4.6, Zeile 4). Erst wenn das der Fall ist, überprüft die Analysemaschine die Konsistenz des Elementes (Abbildung 4.6, Zeile 5).

```
RTRule::verifyAutomatically (RTContext context)
 1:    if rule matches context
 2:        engine = getRuleEngine ()
 3:        for all examinees in context do
 4:            if engine.responsible (examinee)
 5:                engine.check (examinee)
 6:                if examinee is inconsistent and AutoRepair is specified
 7:                and RepairMode is automatic and not suppress AutoRepair
 8:                    if not RTExecMgr.get ().hasInAutoRepairTable (examinee, this)
 9:                        engine.repair (examinee)
10:                        RtExecMgr.get ().addToAutoRepairTable (examinee, this)
11:                    else
12:                        notify user about rule conflict
13:                updateCheckingResult (examinee, this)
```

Abbildung 4.6: Die Methode *verifyAutomatically* der Klasse *RTRule*

Wird eine Konsistenzverletzung bezüglich der Regel festgestellt, so wird vor der Reparatur überprüft, ob in dieser Konsistenzprüfung das Element bezüglich dieser Regel bereits korrigiert worden ist. Dazu wird in der bereits erwähnten Tabelle des *ExecutionManagers* nachgeschaut (Abbildung 4.6, Zeile 8). Da dies die erste Iteration über den Katalog ist, kann das Element noch nicht repariert worden sein und ist deshalb nicht in der Tabelle eingetragen. Damit kann die Korrektur stattfinden und das Element samt zugehöriger Konsistenzregel in der Tabelle eingetragen werden (Abbildung 4.6, Zeile 9 und 10).

Sind alle Kataloge und die darin enthaltenen Regeln überprüft worden, so wird im *ExecutionManager* die Bedingung der Schleife überprüft (Abbildung 4.4, Zeile 11). Haben während der Konsistenzprüfung Korrekturen stattgefunden, so wurden alle modifizierten Elemente automatisch in den auf dem Stack liegenden Korrekturkontext eingetragen und es erfolgt eine weitere Iteration. Dabei wird der Korrekturkontext zum neuen Prüfungskontext (Abbildung 4.4, Zeile 5).

Wird in den nachfolgenden Prüfungszyklen eine Inkonsistenz bezüglich einer Konsistenzregel und eines Elementes festgestellt, die bereits in der Korrekturtabelle als repariert vermerkt wurde (Abbildung 4.6, Zeile 8), so findet keine Korrektur statt und der Benutzer wird auf den Konflikt hingewiesen (Abbildung 4.6, Zeile 12).

Die Konsistenzprüfung ist beendet, wenn nach einem Prüfungszyklus der Korrekturkontext keine zu überprüfenden Elemente enthält. Zuvor wird noch die Korrekturtabelle gelöscht und für nachfolgende Konsistenzprüfungen vorbereitet.

4.3.4 Nebenläufige Konsistenzprüfung

Es ist vorstellbar, dass die Menge der zu überprüfenden Konsistenzregeln so groß wird, dass die Grenze der Skalierbarkeit mit dem oben vorgestellten Mechanismus igrendwann einmal erreicht wird. In diesem Fall lässt sich der vorgestellte Mechanismus jedoch auch nebenläufig ausführen.

Um die Konsistenzprüfung nebenläufig ausführen zu können, sind nur zwei Modifikationen nötig. Die Registrierung der modifizierten Elemente in einem Prüfungskontext bleibt weiterhin erhalten. Statt jedoch den Kontext im Anschluss an ein Benutzerkommando direkt zu überprüfen, wird der Kontext erst in eine Warteschlange eingereiht. In Abbildung 4.7 ist die nebenläufige Prüfung konzeptionell dargestellt.

Der Konsistenzprüfungsmechanismus läuft als ein eigener Thread und arbeitet die Warteschlange sequentiell nach dem FIFO-Prinzip ab. Dazu entnimmt er der Warteschlage ein Kontextobjekt und überprüft alle dort enthaltenen Modellelemente. Die Überprüfung findet weiterhin nach dem bereits vorgestellten Prinzip statt.

Da jetzt während einer Konsistenzprüfung weitere Benutzeraktionen stattfinden können, muss zusätzlich noch eine Synchronisation zwischen den Benutzeraktionen und der Konsistenzprüfung stattfinden. Die Synchronisation ist an den bisherigen Analysemechanismus angelehnt und hält vor der Ausführung des Kommandos die Konsistenzprüfung an. Im Unterschied zum bisherigen Mechanismus muss jetzt allerdings nur noch ein Thread angehalten werden.

Der Thread darf nicht an jeder beliebigen Stelle unterbrochen werden, da beispielsweise nach einer Unterbrechung innerhalb der *responsible* Methode und anschließender Wiederaufnahme des Threads die Vorbedingung durch die ausgeführte Benutzeraktion nicht mehr gelten könnte.

Außerdem findet die Synchronisation bereits in der abstrakten Klasse *FujabaAbstractAction* statt, so dass beim Hinzufügen von neuen Kommandos die Funktionalität bereits gegeben ist und der Programmierer sich um die Synchronisation nicht mehr kümmern muss.

Während der automatischen Korrektur werden veränderte Objekte weiterhin in einen separaten Korrekturkontext eingetragen. Der Kontext liegt dabei als oberstes Element auf dem Stack. Wird nun ein Benutzerkommando während einer Konsistenzprüfung aufgerufen, so wird die Konsistenzprüfung angehalten und ein neuer Prüfungskontext durch das Benutzerkommando erzeugt und auf den Stack gelegt. Die innerhalb des Kommandos modifizierten Elemente werden in diesen Prüfungskontext eingetragen. Ist das Kommando abgeschlossen, so wird der Prüfungskontext vom Stack entfernt, in die Warteschlange eingereiht und die Konsistenzprüfung fortgesetzt. Hier ist die Verwendung eines Stacks für das Kontext-Objekt besonders nützlich, da die modifizierten Elemente einer Korrektur wieder in dem Korrekturkontext eingetragen werden.

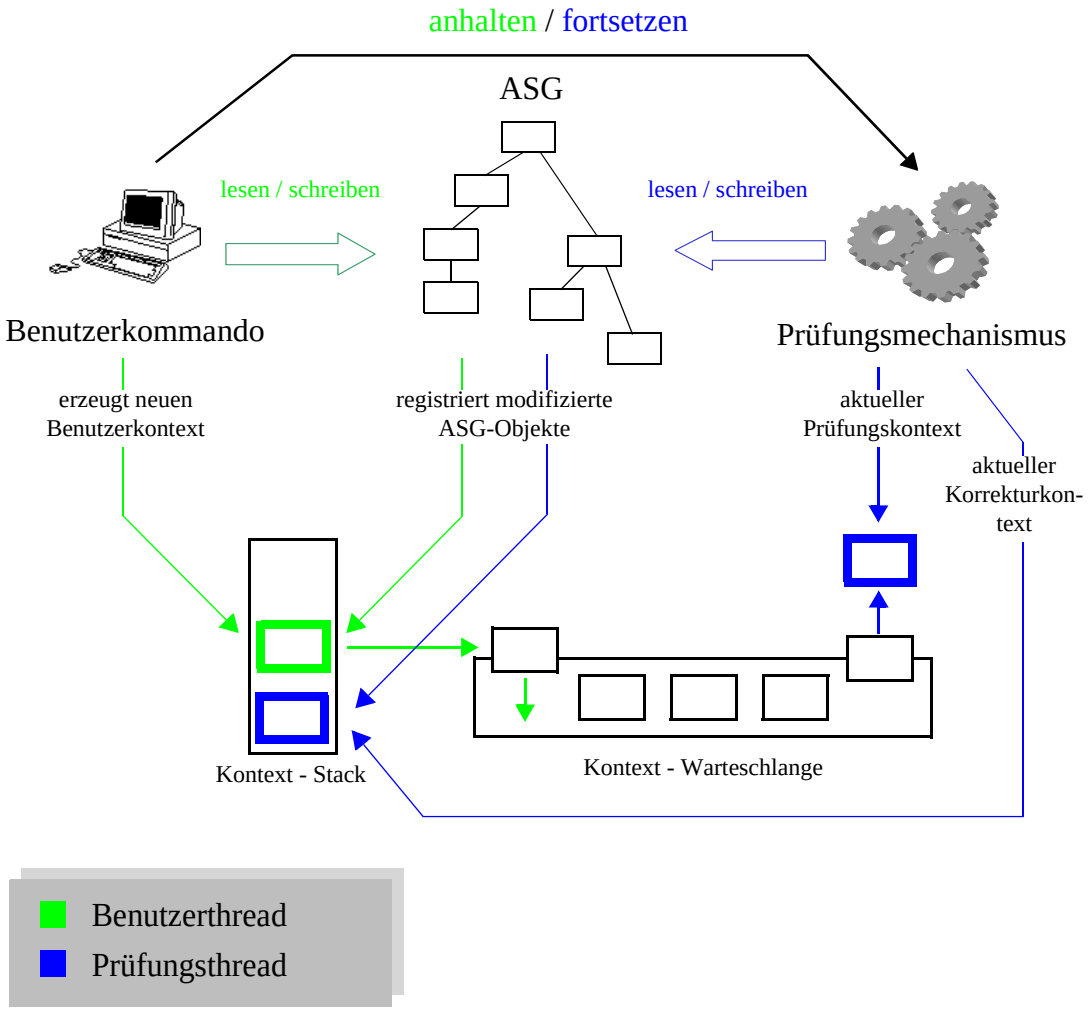


Abbildung 4.7: Schematische Darstellung der nebenläufigen Konsistenzprüfung

5 Technische Realisierung

In diesem Kapitel wird die technische Realisierung des Konsistenzmanagements-Systems in der integrierten Entwicklungsumgebung FUJABA vorgestellt. Zuerst wird in Abschnitt 5.1 ein grober Überblick über die Architektur des Systems gegeben. Anschließend wird in Abschnitt 5.2 das Design des Regelkataloges vorgestellt, der zur Spezifikation und Verwaltung von Konsistenzbedingungen implementiert wurde. Aus den im Katalog erstellten Spezifikationen werden Analysemaschinen für die Konsistenzprüfung erstellt. Dieser Vorgang wird ebenfalls in diesem Abschnitt erläutert. Im letzten Abschnitt wird schließlich das Design der Laufzeitumgebung vorgestellt, die für die Steuerung der Konsistenzprüfung und die automatische Korrektur verantwortlich ist.

5.1 Architektur

Für die technische Realisierung und Integration des Konsistenzmanagement-Systems in FUJABA sind die benötigten Klassen in vier Pakete eingeteilt worden. Die Pakete werden relativ zu dem Paket *de.uni_paderborn.fujaba* angegeben. Die zur Spezifikation und Verwaltung des Regelkatalogs realisierten Klassen sind in dem Paket *cms*[1] untergebracht. Das Paket *cms.rt*[2] enthält die Klassen für den Steuerungsmechanismus sowie eigene Katalogklassen, die zusätzlich zur Katalogstruktur auch Konfigurationsinformationen für die Konsistenzprüfung beinhalten. Die aus der Spezifikation erzeugten Konsistenzanalysemaschinen sind im Paket *cms.rt.engines* untergebracht. Schließlich sind die zur Benutzerinteraktion benötigten Dialoge im Paket *cms.dialogs* zu finden.

Ein Überblick über die Architektur des Konsistenzmanagement-Systems gibt die Abbildung 5.1. Die Basis der Konsistenzprüfung stellen Kataloge mit den darin spezifizierten Regeln dar. Aus jeder Regelspezifikation wird eine Konsistenzanalysemaschine erzeugt und im Paket *cms.rt.engines* abgelegt. Zusätzlich wird aus einem

1. Consistency Management System
2. Runtime

Katalog eine XML-Datei generiert. Sie enthält die hierarchische Struktur des Spezifikationskataloges und die statischen Beschreibungen jeder Regel. Außerdem wird dort die Standardkonfiguration für die Ausführung einer Regel festgelegt.

Zur Konsistenzprüfung können beliebig viele XML-Katalogdateien eingelesen werden. Anhand der Informationen in der Katalogdatei lädt das Konsistenzmanagement-System die für diesen Katalog spezifizierten Konsistenzanalysemaschinen aus dem Paket *cms.rt.engines*. Vor der Konsistenzprüfung kann die Ausführung einer Konsistenzregel neu konfiguriert werden.

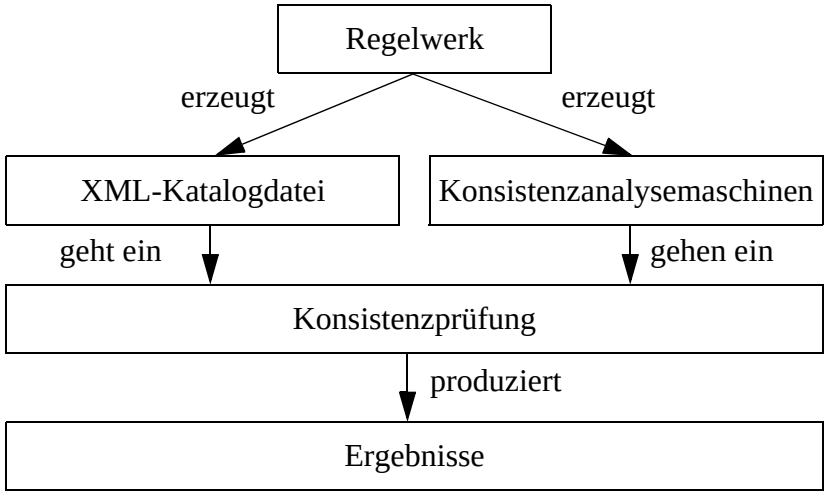


Abbildung 5.1: Architektur des Konsistenzmanagement-Systems

Je nach Konfiguration des Konsistenzmanagement-Systems erfolgt eine automatische oder manuelle Überprüfung der Konsistenz. In beiden Fällen werden Informationen über das Ergebnis der Konsistenzprüfung generiert und im Konsistenzmanagement-System hinterlegt. Diese Ergebnisse können vom Benutzer erfragt und Konsistenzverletzungen nachträglich behoben werden.

5.2 Der Spezifikationskatalog

5.2.1 Design des Kataloges

In Abbildung 5.2 ist das Design des Kataloges abgebildet, der die Regelspezifikationen enthält. Dem Design liegt das Composite-Pattern aus [GHJV95] zugrunde. Die

abstrakte Klasse *CMSEntry* stellt hier die *Component*-Klasse dar, die Klasse *CMSCategory* die *Composite*-Klasse. Beide stehen über die Assoziation *entries* in Beziehung. Die Wurzel eines Katalogs wird durch die *CMSCatalog*-Klasse realisiert. Der Katalog beinhaltet genauso wie eine Kategorie Regeln, Verweise und Kategorien und erbt daher von der *CMSCategory*-Klasse.

Ein Verweis auf einen Katalogeintrag wird durch die Klasse *CMSLink* und die Assoziation *linkedTo* realisiert. Die Kardinalitäten der Assoziationen ergeben sich aus der Tatsache, dass ein Verweis immer nur auf einen Eintrag zeigen darf, auf den Eintrag aber mehrmals verwiesen werden darf. Die Klasse *CMSLink* ist im Sinne des *Composite*-Patterns eine Blattklasse.

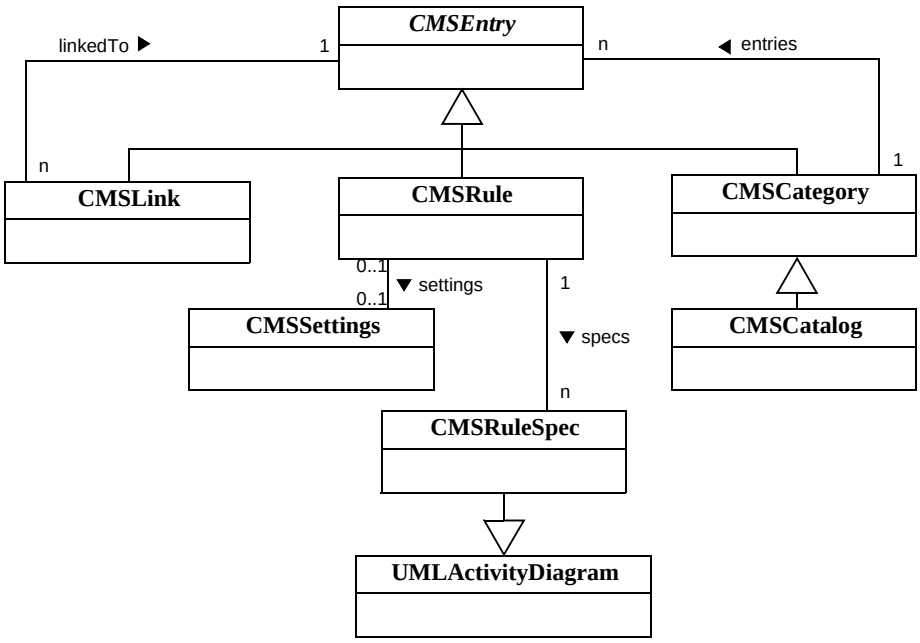


Abbildung 5.2: Design des Spezifikations-Katalogs

Eine weitere Blattklasse ist die *CMSRule*-Klasse. Sie repräsentiert die Konsistenzregel. Zur Spezifikation einer Konsistenzregel gehören *Story-Diagramme*. Um die Darstellung und Spezifikation der Konsistenzbedingungen innerhalb des Konsistenzmanagement-Systems zu ermöglichen, wurde die Klasse *CMSRuleSpec* eingeführt, die von *UMLActivityDiagram* erbt und eine Assoziation zur *CMSRule*-Klasse unterhält. Die Standardkonfiguration der Konsistenzregel kann zur Spezifikationszeit festgelegt werden. Diese Eigenschaften werden in der Klasse *Settings* abgelegt und während einer Konsistenzprüfung abgefragt.

5.2.2 Erzeugung von Konsistenzanalysemaschinen

Beim Erzeugen eines neuen Kataloges wird automatisch ein UML-Klassendiagramm angelegt. Für jede neue Regel wird in dem Klassendiagramm automatisch eine Klasse generiert, die von der abstrakten Klasse *RuleEngine* erbt. Diese Klasse wird durch den Stereotyp *«consistency rule»* für die spätere Generierung von JAVA-Quellcode gekennzeichnet. Die *RuleEngine* besitzt die drei abstrakten Methoden *responsible*, *check* und *repair*. Diese Methoden werden in der konkreten Regelklasse redefiniert. Zu jeder der drei Methoden wird zusätzlich ein *Story-Diagramm* erstellt, das bereits einen Startzustand und eine Transition zum Endzustand besitzt. Abbildung 5.3 zeigt das UML-Klassendiagramm mit den konkreten Regelklassen *InterfaceNoAttr*, *InterfaceAllOpPublic*, *OpUniqueParamName* und *AssocOneAggregateEnd*.

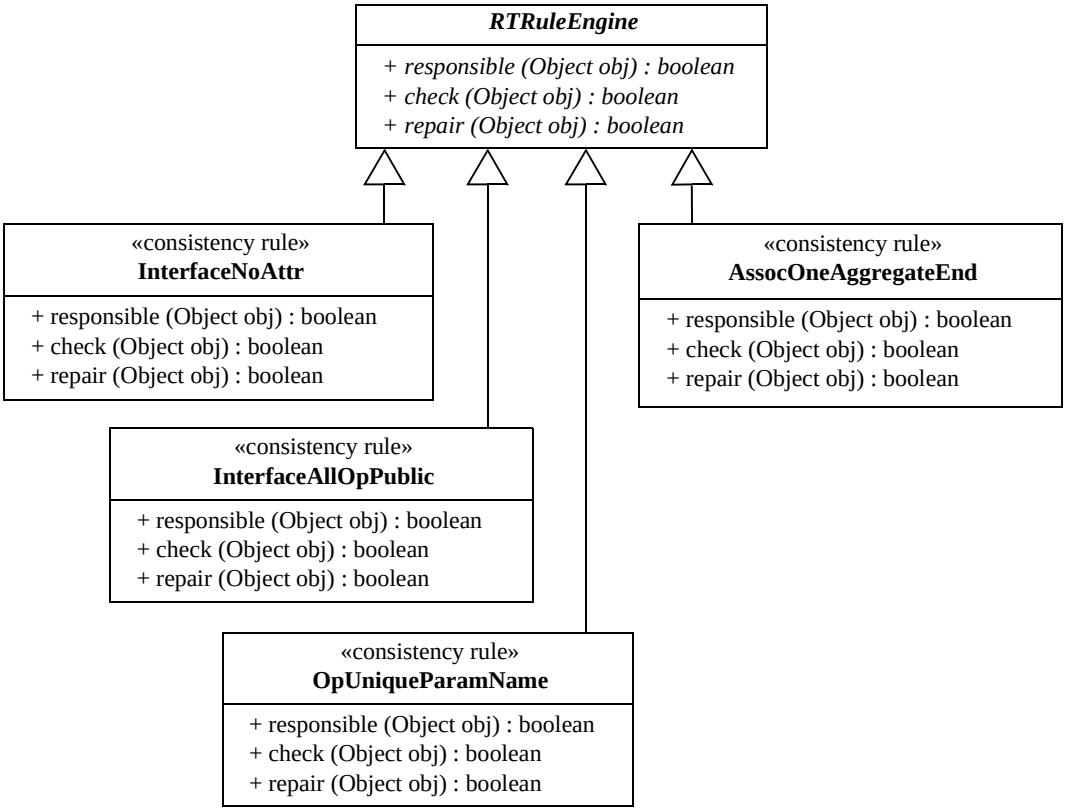


Abbildung 5.3: Klassendiagramm der Konsistenzanalysemaschinen

Im nächsten Schritt muss die Konsistenzregel spezifiziert werden, indem die *Story-Diagramme* der Regelklasse mit Inhalt gefüllt werden. Dies kann auf zwei verschiedenen Wegen erfolgen.

Falls die Konsistenzbedingung an ein einziges Objekt oder an eine Objektstruktur formuliert wird, die alle benötigten Assoziationen besitzt, so dass keine weiteren Abhängigkeiten formuliert werden müssen, erfolgt die Spezifikation direkt durch das *Story Driven Modeling* [FNT98].

Wird eine Konsistenzbedingung zwischen Objekten formuliert, die in keiner direkten Assoziation zueinander stehen und ein Navigieren zwischen den Objekten nicht möglich ist, so erfolgt die Spezifikation unter Verwendung der Tripel-Graph-Grammatik. Dazu existiert ein TGG-Editor [Müh00], aus dem nach erfolgreicher Spezifikation der TGG-Regel die *Story-Diagramme* automatisch erstellt werden. Dazu muss die Regelklasse und die gewünschte Richtung angegeben werden. Für bidirektionale Konsistenzbedingungen, wie sie beispielsweise zwischen SDL-Blockdiagrammen und UML-Klassendiagrammen gelten, müssen also sowohl für die Vorwärts-Richtung als auch für die Rückwärts-Richtung zwei Regelklassen erstellt werden. Die Spezifikation der TGG-Regel erfolgt jedoch nur einmal.

Um die Konsistenzanalysemaschinen ausführen zu können, wird aus dem Katalog eine XML[1]-Katalogdatei mit Laufzeitinformationen erzeugt. Aus dem UML-Klassendiagramm und den SDM-Diagrammen wird automatisch JAVA-Quellcode erzeugt, der nur noch übersetzt werden muss.

5.3 Der Registrierungsmechanismus

Während in dem bisherigen Analysemechanismus die Konsistenzprüfung nebenläufig ausgeführt und vor jedem Benutzerkommando angehalten wurde, wird in dem neuen Ansatz eine Konsistenzprüfung erst gestartet, wenn ein Benutzerkommando abgeschlossen ist. Dazu wurde die Struktur für Benutzerkommados geändert. Abbildung 5.4 zeigt einen Ausschnitt aus dem neuen Klassendiagramm für Benutzeraktionen.

Ein zentraler Bestandteil des Konsistenzprüfungsmechanismus ist die neue Klasse *AbstractContextAction* mit der *actionPerformed* Methode, in der vor der Ausführung der eigentlichen Benutzerkommandos durch die Anweisung *enterContext* das Konsistenzmanagement-System über ein bevorstehende Ausführung eines Benutzerkommandos informiert wird. Die Anweisungen des Benutzerkommandos werden in *executeAction* implementiert. Nach der Ausführung wird dem Konsistenzmanagement-System durch *checkContext* die Beendigung der Benutzeraktion signalisiert und die Konsistenzprüfung gestartet.

1. Extensible Markup Language

AbstractContextAction definiert ferner die zwei Attribute *contextName* und *contextDescription*. Der Kontextname wird für die Bindung von Regeln an die Benutzeraktionen benutzt. Die Kontextbeschreibung ist lediglich eine zusätzliche Hilfe bei der Zuweisung von Benutzerkontexten an verschiedene Konsistenzregeln. Dazu muss in *registerNotifier* der Kontextname und die Kontextbeschreibung gesetzt werden. Bei der Instanziierung eines Benutzeraktions-Objektes wird dann diese Methode aufgerufen und das Objekt registriert, um eine Auswahl in den Regeleigenschaften zu ermöglichen.

Wie in Abbildung 5.4 zu erkennen ist, definieren die Benutzeraktionen *EditClassAction* und *EditAttributeAction* jeweils einen eigenen Kontext, während die Benutzeraktionen *NewBlockAction*, *EditBlockAction* und *NewProcessAction* nur einen einzigen Kontext definieren. Durch diese Architektur kann bereits zur Entwicklungszeit festgelegt werden, wie feingranular Konsistenzregeln an Benutzeraktionen gebunden werden können. Weiterhin ist es möglich Benutzerkommandos zu implementieren, deren Ausführung keine Konsistenzprüfung nach sich zieht. Dies ist beispielsweise für Konfigurationsaktionen der Entwicklungsumgebung sinnvoll, da hier keine Modifikationen am ASG vorgenommen werden.

Zur Identifikation der modifizierten Modellelemente wird weiterhin der Alterungsmechanismus benutzt, der für die Synchronisation zwischen den Objekten des ASG und des für die Anzeige verantwortlichen FUJABA-Displaygraphen zuständig ist [FNT98]. Eine Grundvoraussetzung für die Anwendbarkeit des Alterungsmechanismus für die Konsistenzprüfung ist, dass alle schreibenden Zugriffe auf Attribute nur über zugehörige Zugriffsmethoden stattfinden. Dies ist aber auch Voraussetzung für die Aktualisierung des Displaygraphen und wird daher in FUJABA konsequent umgesetzt. Innerhalb einer jeden schreibenden Zugriffsmethode wird *incrementAge* aufgerufen. An dieser Stelle greift auch das Konsistenzmanagement-System ein und registriert das modifizierte Objekt.

Der Alterungsmechanismus wurde auch im alten Analysemechanismus verwendet. Statt wie bisher die zu überprüfenden Modellelemente jedoch bei jeder Analysemaschine einzeln anzumelden, wird ein sogenannter Prüfungskontext verwendet, bei dem diese Elemente registriert werden. Dieser Kontext wird durch den Aufruf der *enterContext* Methode erzeugt und auf einen Stack gelegt, da Benutzeraktionen verschachtelt ablaufen können.

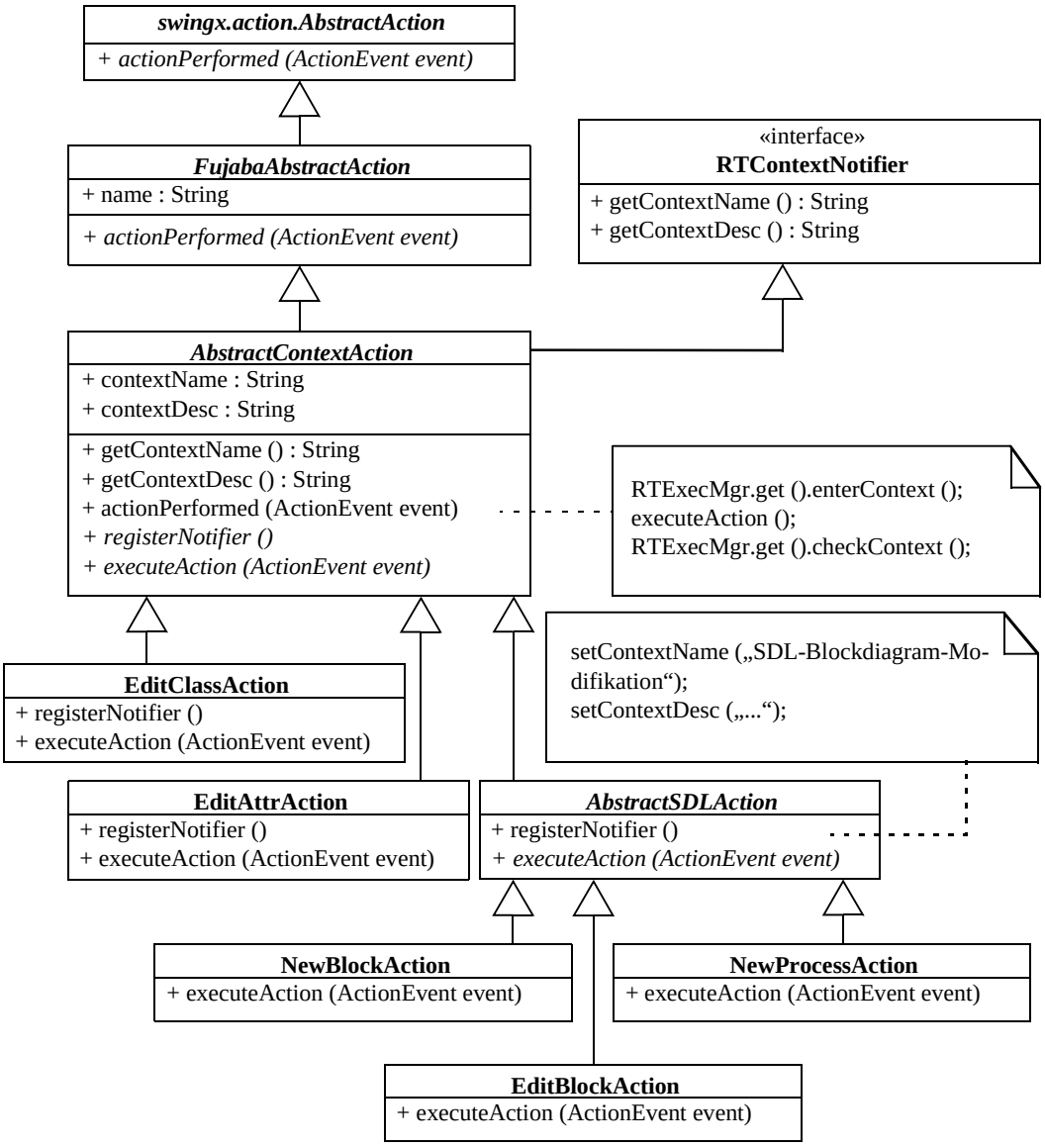


Abbildung 5.4: Klassendiagramm für Benutzeraktionen

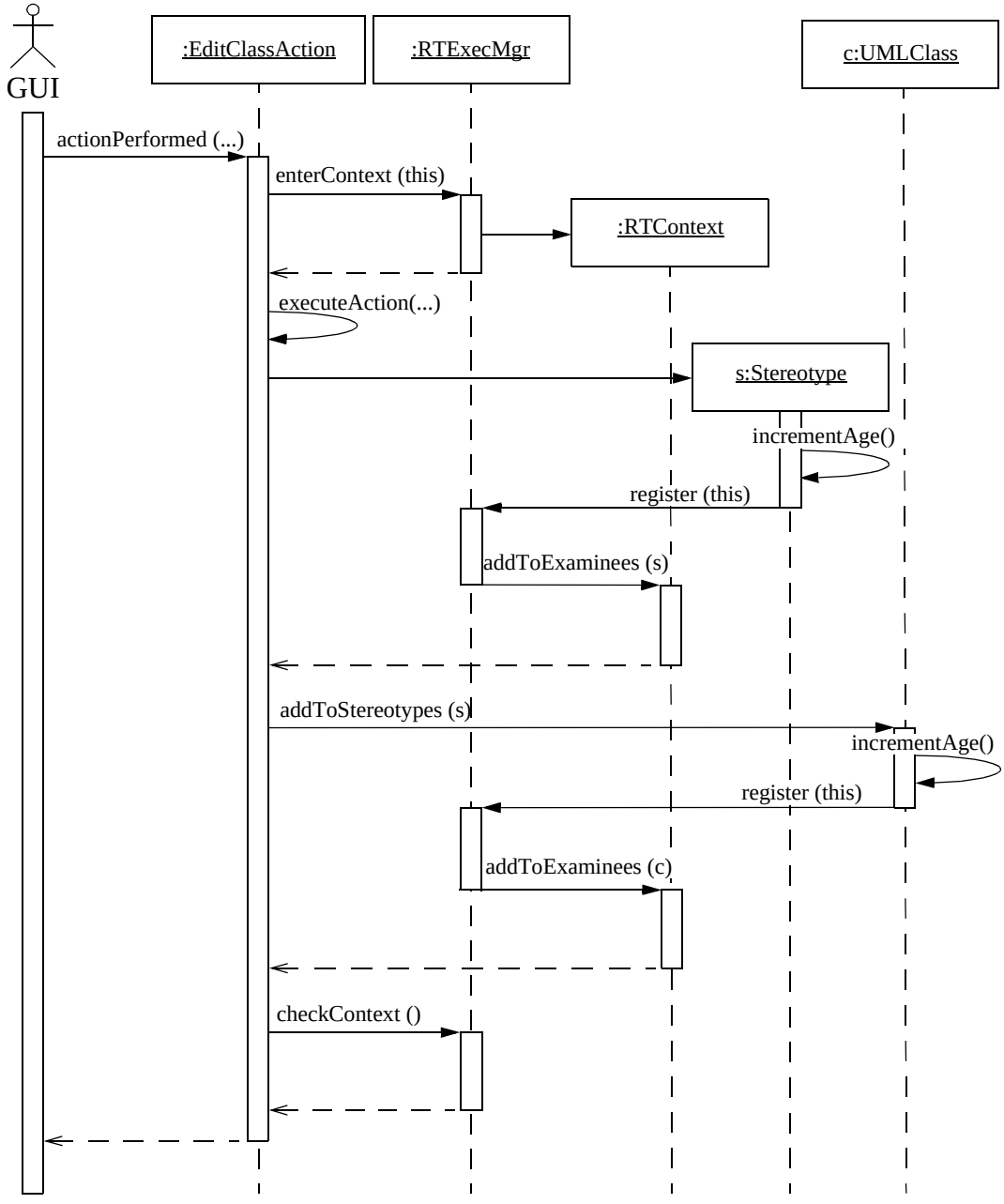


Abbildung 5.5: Sequenzdiagramm der automatischen Registrierung

In Abbildung 5.5 ist der Ablauf der Registrierung durch ein Sequenzdiagramm dargestellt. In diesem Szenario ändert der Benutzer eine Klasse in ein Interface. Dazu wählt er aus dem Bearbeitungsmenü das Kommando *EditClass*. Als Reaktion ruft die GUI[1] auf dem *EditClassAction*-Objekt die *actionPerformed* Methode auf, die über eine Instanz von *RTExecMgr* ein neues Kontextobjekt erzeugt und auf einen Stack legt. Nun wird *executeAction* aufgerufen, die einen Dialog erzeugt, in dem der Benutzer die gewünschten Änderungen machen kann. Die Modifikationen an den ASG-Objekten werden erst nach dem Schließen des Dialoges durchgeführt.

In diesem Beispiel wird eine neue Instanz der Klasse *Stereotype* angelegt und über *addToStereotypes* in Beziehung zum *UMLClass*-Objekt gesetzt. Jede Modifikation einschließlich der Erzeugung eines neuen Objektes resultiert im Aufruf von *incrementAge* und *register*. Durch die Ausführung von *register* wird das modifizierte Objekt in dem Kontext-Objekt mit *addToExaminees* eingetragen. Das *RTContext*-Objekt speichert die modifizierten ASG-Objekte in einer Menge. Daher wird ein mehrfach modifiziertes ASG-Objekt in dem *RTContext*-Objekt nur einmal registriert.

Am Ende der Ausführung von *executeAction* erfolgt ein Aufruf von *checkContext*, der dem *RTExecMgr* signalisiert, dass die Bearbeitung abgeschlossen ist. Dieser entfernt das *RTContext*-Objekt vom Stack und startet die Konsistenzprüfung. Ist die Konsistenzprüfung beendet, so wird die Kontrolle an die GUI wieder abgegeben.

5.4 Design der Laufzeitumgebung für Konsistenzprüfungen

An der Konsistenzprüfung sind mehrere Klassen beteiligt. Die wichtigsten dieser Klassen sind in der Abbildung 5.6 zu sehen. Dabei spielt der *ExecutionManager*, implementiert durch die Klasse *RTExecMgr*, eine zentrale Rolle. Zur Implementierung des *ExecutionManagers* wurde das Entwurfsmuster *Singleton* nach Gamma et. al [GHJV95] verwendet, so dass es immer nur eine zentrale Instanz dieser Klasse geben kann. Der *ExecutionManager* verwaltet verschiedene Kataloge, die zur Laufzeit der Anwendung hinzugefügt und auch wieder entfernt werden können. Dabei lädt der *ExecutionManager* mit Hilfe von weiteren, hier nicht dargestellten Hilfsklassen, einen Katalog und dessen Einträge aus einer XML-Katalogdatei, die auch Konfigurationseinstellungen für den Ablauf der Konsistenzprüfung enthält. Um die in einem Katalog gespeicherten Regeln zur Konsistenzprüfung ausführen zu können, trägt der *ExecutionManager* den Katalog in die Assoziation *catalogs* ein.

1. Graphical User Interface

Durch das Laden des Katalogs wird die Katalogstruktur aufgebaut, die aus Kategorien, Regeln und Verweisen besteht. Zu jeder dieser Katalogkomponenten existieren die entsprechenden Klassen *RTCategory*, *RTRule* und *RTLink*. Jede dieser Klassen ist von der abstrakten Klasse *RTEntry* abgeleitet. Diese Klasse deklariert die beiden abstrakten Methoden *verifyAutomatically* und *verifyManually*, die zur Traversierung der Katalogstruktur und zur Überprüfung der Konsistenzregeln verwendet werden und von jeder Subklasse implementiert werden müssen. Jeder Katalogeintrag besitzt einen Namen und eine Beschreibung, die in den Attributen *name* und *description* abgelegt werden. Darüber hinaus kann jeder Eintrag innerhalb der Konfiguration einer Konsistenzprüfung aktiviert und deaktiviert werden. Der aktuelle Zustand wird in dem boolschen Attribut *enabled* festgehalten.

Die Klasse *RTLink* modelliert einen Verweis auf einen Katalogeintrag und unterhält hierzu die Assoziation *linkedTo*. Mit Hilfe dieser Assoziation delegieren die beiden Methoden *verifyAutomatically* und *verifyManually* die Überprüfung an den Eintrag, auf den der Verweis zeigt.

Die hierarchische, baumartige Katalogstruktur wird durch ein *Composite* Design-Pattern [GHJV95] realisiert. Die Klasse *RTCategory* verwaltet weitere Katalogeinträge über die Assoziation *entries*. Dies können auch Unterkategorien sein. Während einer Konsistenzprüfung wird über alle Einträge iteriert und auf diesen die Methode *verifyAutomatically* beziehungsweise *verifyManually* aufgerufen.

Ein Katalog ist die Wurzel einer Kataloghierarchie und ein Spezialfall einer Kategorie. Daher erbt *RTCatalog* von *RTCategory* und verwendet deren Methoden *verifyAutomatically* und *verifyManually* zur Traversierung des Regelkatalogs.

Die Steuerung der Konsistenzprüfung ist in den Methoden *verifyAutomatically* und *verifyManually* der *RTRule* Klasse implementiert. Dazu wird der Konsistenzregel die zugehörige Konsistenzanalysemaschine über die Assoziation *engine* zugeordnet. Diese Analysemaschine implementiert die bereits in den vergangenen Abschnitten vorgestellten Methoden *responsible*, *check* und *repair*. Zur Steuerung der Konsistenzprüfung enthält die *RTRule* Klasse verschiedene Attribute, die in der Entwicklungsumgebung konfiguriert werden können. Zwei dieser Parameter sind das boolsche Attribut *notifyImmediately* und das Attribut *repairMode*, das den Modus für die automatische Korrektur von Konsistenzverletzungen festlegt. Die übrigen Attribute zur Konfiguration wurden aus Platzgründen in der Abbildung 5.6 ausgelassen.

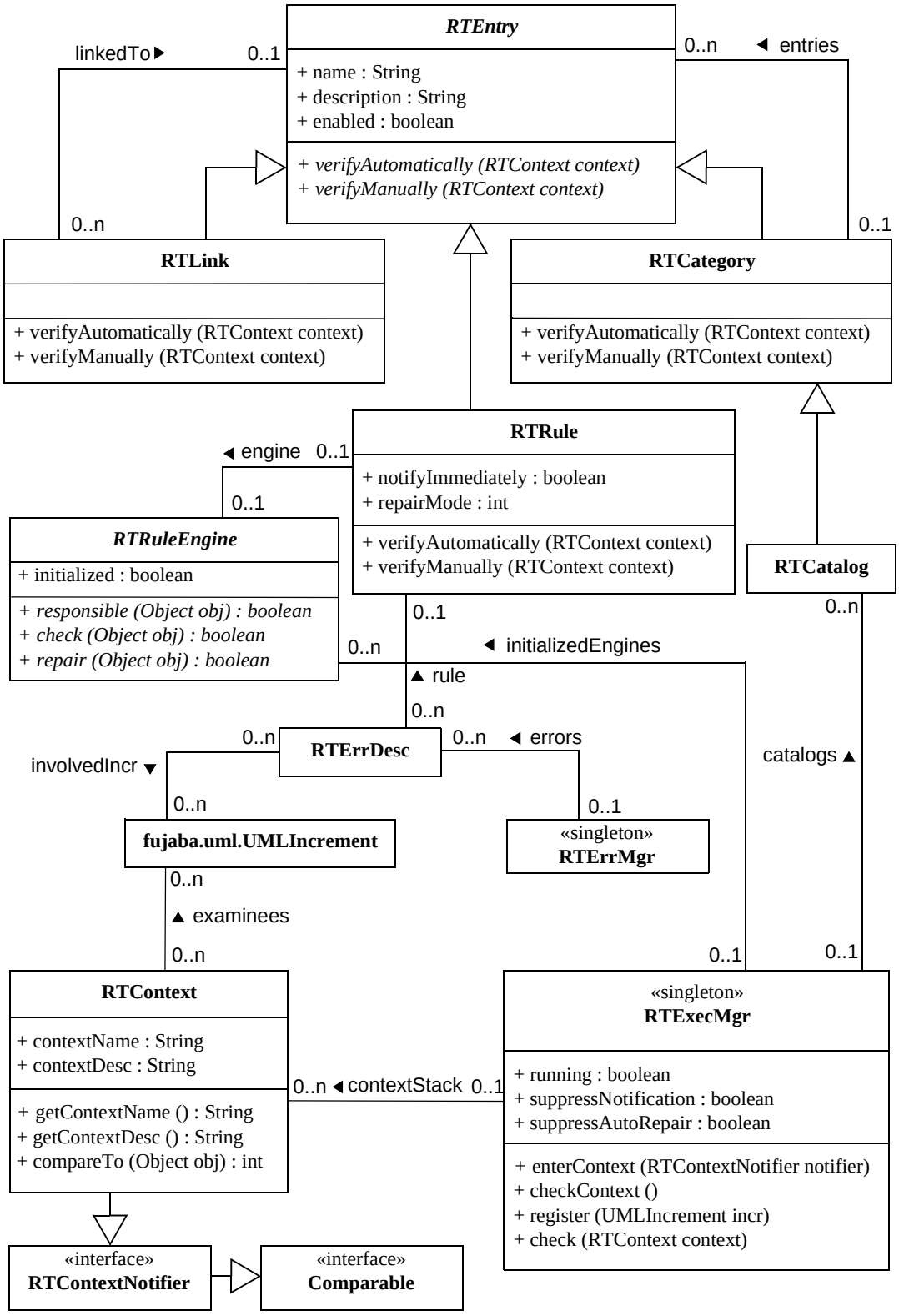


Abbildung 5.6: Klassendiagramm der Laufzeitumgebung

Nach dem Laden des Katalogs erfolgt die Initialisierung des Laufzeitsystems. Dazu werden aus den Kataloginformationen die Regelnamen für zu ladende Konsistenzanalysemaschinen verwendet. Wie bereits in Abschnitt 5.2.2 beschrieben wurde, erben alle Analysemaschinen von *RTRuleEngine*. Die geladene Konsistenzanalysemaschine wird über die Assoziation *engine* mit der zugehörigen Regel des Katalogs in Beziehung gesetzt. Außerdem wird bei einer erfolgreichen Initialisierung die geladene Analysemaschine zur einfacheren Verwaltung über die Assoziation *initializedEngines* in dem *ExecutionManager* eingetragen. Konnte die Analysemaschine nicht geladen oder initialisiert werden, so wird dies vermerkt und während einer Konsistenzprüfung berücksichtigt.

Wie bereits in den vergangenen Abschnitten beschrieben worden ist, wird zur automatischen Konsistenzprüfung vor der Ausführung eines Benutzerkommandos die Methode *enterContext* des *ExecutionManager* aufgerufen. Diese Methode erzeugt ein neues *RTContext*-Objekt und legt es auf den Stack des *ExecutionManagers*. Anschließend werden durch die Benutzeraktion modifizierte Elemente dem *ExecutionManager* über die Methode *register* bekannt gegeben. Diese Methode erhält ein *UMLIncrement*-Objekt als Parameter und trägt es in den obersten Kontext des Stacks ein. Die Klasse *UMLIncrement* ist die Basisklasse für alle in FUJABA integrierten Diagramme und Diagrammelemente. Am Ende einer Benutzeraktion wird die Methode *RTExecMgr::checkContext* aufgerufen, in der die Konsistenzprüfung vorbereitet und gestartet wird.

Während einer Prüfung erkannte Konsistenzverletzungen werden in einem Objekt der Klasse *RTErrDesc* gespeichert und bei der zentralen Verwaltungsinstanz der Klasse *RTErrMgr* eingetragen. Zusätzlich werden über die Assoziation *rule* die Regel, die den Fehler erkannt hat, und über die Assoziation *involvedIncr* die an der Konsistenzverletzung beteiligten ASG-Objekte eingetragen.

6 Beispielsitzung

In den folgenden Abschnitten wird die Verwendung des in dieser Arbeit realisierten Konsistenzmanagement-Systems anhand von Beispielen vorgestellt. Zunächst wird in Abschnitt 6.1 der Umgang mit dem Regelkatalog erläutert. In diesem Zusammenhang wird auch auf die Spezifikation von Konsistenzbedingungen mit dem *Story Driven Modeling* und mit der *Tripel-Graph-Grammatik* eingegangen. Im darauf folgenden Abschnitt 6.2 wird die Konfiguration und automatische Konsistenzprüfung des Systems dargestellt. Schließlich werden im Abschnitt 6.3 die interaktive Konsistenzprüfung und die Verwaltung von Prüfungsergebnissen gezeigt.

6.1 Erstellung und Verwaltung von Konsistenzregeln

Der Umgang mit dem SDM-Editor zur Erstellung von *Story-Diagrammen* wurde bereits in [FNT98] ausführlich beschrieben. Ebenso wurde in der Diplomarbeit von Jens H. Mühlenhoff [Müh00] der TGG-Editor zur Spezifikation von Tripel-Graph-Grammatiken detailliert behandelt. Daher wird in diesem Abschnitt nur kurz auf die Spezifikation von Konsistenzregeln mit diesen Editoren eingegangen und der Schwerpunkt auf den Umgang mit dem Regelkatalog gelegt.

Abbildung 6.1 zeigt das Menü *Consistency Management*, in dem alle Operationen für die Erstellung, Modifikation und Verwaltung eines Regelkatalogs zu finden sind. Ebenso sind dort die Operationen für die Konfiguration und Ausführung von Konsistenzprüfungen zu finden.

Ein neuer Regelkatalog wird über den Menüeintrag *New Specification Catalog* erzeugt. Ein bereits vorhandener Katalog kann mit Hilfe des Menüeintrags *Open Specification Catalog* geöffnet und anschließend bearbeitet werden. In Abbildung 6.1 ist der Regelkatalog für UML-Spezifikationen bereits geöffnet worden, dessen Regeln auf der linken Seite zu sehen sind. Der Katalog ist zweigeteilt. In der oberen Hälfte ist die hierarchische Struktur des Katalogs zu sehen. In der unteren Hälfte sind die Re-

gelspezifikationen untergebracht, zu denen aus einem selektierten Katalogeintrag über ein Kontextmenü navigiert werden kann.

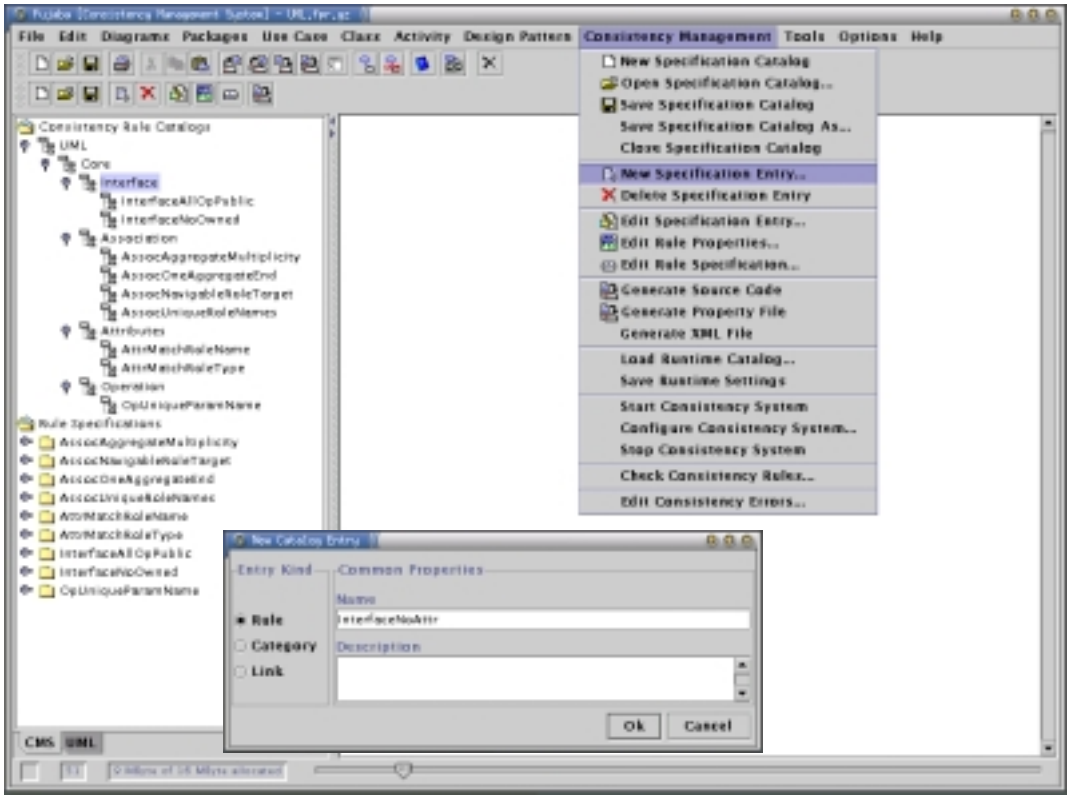


Abbildung 6.1: Anlegen einer neuen Regel

Der Regelkatalog wird nun um die bereits in Abschnitt 3.2 beschriebene Konsistenzregel *InterfaceNoAttr* ergänzt. Hierzu wird über den Menüpunkt *New Specification Entry* die neue Konsistenzregel angelegt und in dem zugehörigen Dialog der Name und eine Beschreibung der neuen Konsistenzregel angegeben. Der neue Katalogeintrag wird in der zuvor selektierten Kategorie erstellt und eingefügt. In dem Beispiel wird also die *InterfaceNoAttr* Regel in der Kategorie *Interface* erzeugt. Mit dem Anlegen einer neuen Regel werden automatisch die *Story-Diagramme* zur Spezifikation der Regel vorbereitet und dem Katalog hinzugefügt.

Mit dem gleichen Verfahren können Unterkategorien und Verweise erstellt werden. Hierzu kann in dem Dialog zwischen einer neuen Regel, einer neuen Kategorie oder einem neuen Verweis ausgewählt werden.

Zu einer Konsistenzregel müssen weitere Eigenschaften und Beschreibungen angegeben werden, die zur Konsistenzprüfung benötigt werden. Hierzu wird über den

Menüeintrag *Edit Rule Properties* der in Abbildung 6.2 dargestellte Dialog aufgerufen. Die Eigenschaften sind auf drei Registerkarten aufgeteilt.

In der in Abbildung 6.2 abgebildeten Registerkarte können der Name und die Beschreibung der Regel nochmals verändert sowie weitere Beschreibungen angegeben werden. Dazu gehört die Meldung bei Konsistenzverletzungen und Vorschläge zur Auflösung von Inkonsistenzen. Diese textuellen Beschreibungen werden dem Benutzer beim Auftreten von Konsistenzverletzungen als zusätzliche Informationen angezeigt. Darüber hinaus kann die Fehlerkategorie festgelegt werden. Die Fehlerkategorie erleichtert die Einordnung des Schweregrades einer Konsistenzverletzung. Zur Auswahl stehen die Kategorien *Fehler*, *Warnung* und *Optimierung*.

In der darauf folgenden Registerkarte können Eigenschaften bezüglich der automatischen Fehlerkorrektur angegeben werden. Hierzu zählt beispielsweise eine textuelle Beschreibung der spezifizierten Korrekturmethode und der Korrekturmodus.

In der letzten Registerkarte können Einstellungen zur Ausführung der Konsistenzregel festgelegt werden. Dabei kann spezifiziert werden, ob die Regel automatisch oder nur auf Anforderung überprüft wird. Bei einer automatischen Konsistenzprüfung muss zusätzlich angegeben werden, nach welchen Benutzeraktionen die Regel aktiviert und überprüft werden soll. Hierzu können aus einer Liste ein oder mehrere von Benutzeraktionen beziehungsweise Kontexte ausgewählt und der Regel zuordnen werden.

Die Spezifikation der Konsistenzbedingungen erfolgt durch Editieren der drei *Story-Diagramme responsible*, *check* und *repair*. Die bereits aus Abschnitt 3.2 bekannten Spezifikationen sind in der Abbildung 6.3 zu sehen.

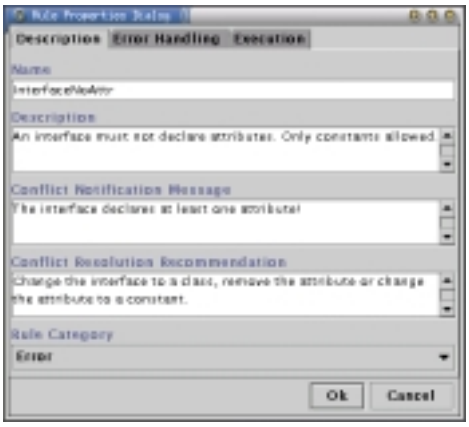


Abbildung 6.2: Eingabe weiterer Regel-Eigenschaften

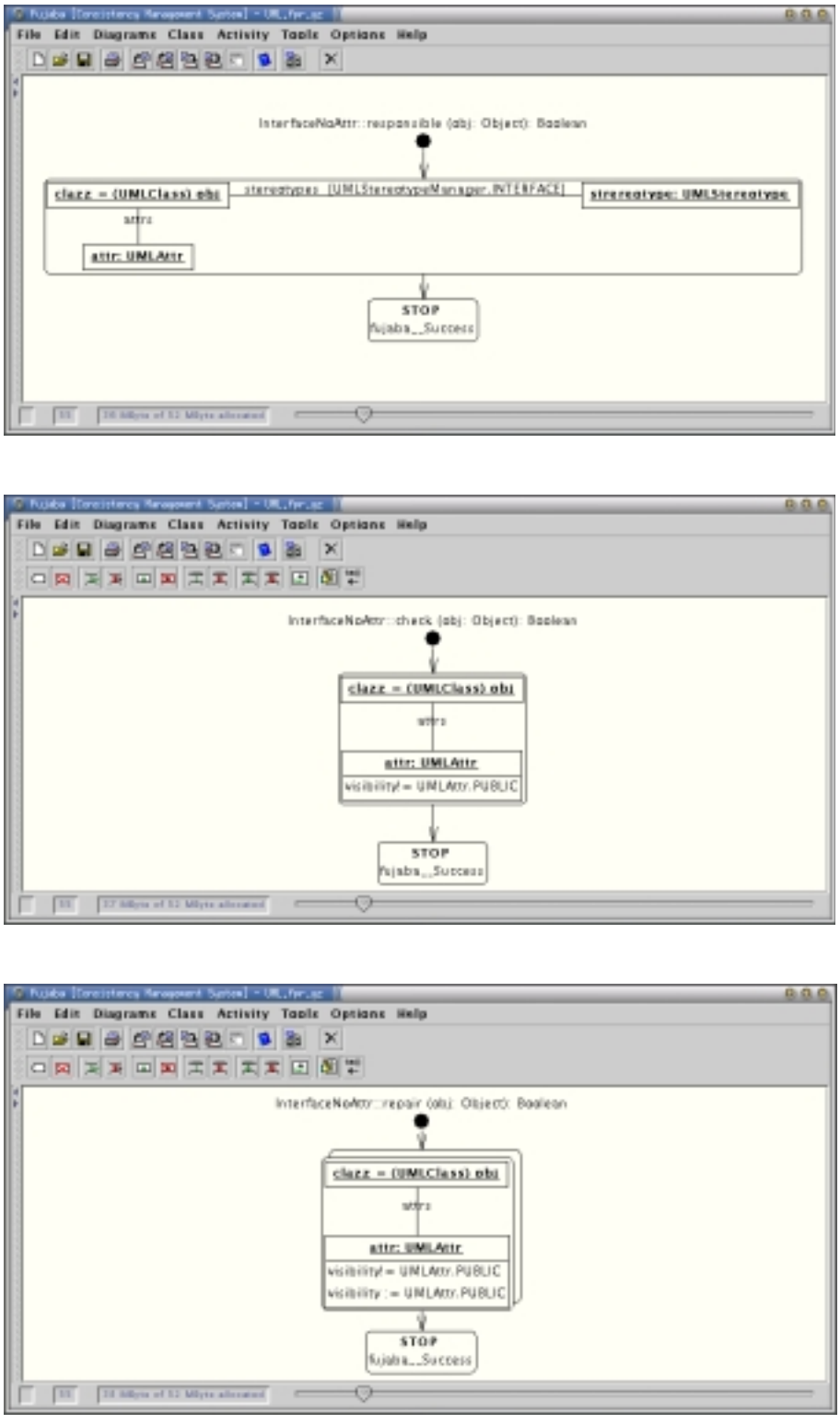


Abbildung 6.3: Regelspezifikation mit Story Driven Modeling

Während in dem vorherigen Beispiel die Konsistenzregel direkt in den *Story-Diagrammen* spezifiziert worden ist, findet die Spezifikation von Konsistenzregeln zur Transformation und anschließender Konsistenzsicherung durch eine Tripel-Graph-Grammatik in dem TGG-Editor statt. Dazu wird in dem Menü *Diagrams* der Eintrag *New TGG Diagram* selektiert und unter Angabe einer Diagrammbezeichnung ein neues TGG-Diagramm erstellt.

Zur Spezifikation der Konsistenzregeln müssen zusätzlich die Metamodelle der beteiligten Diagramme eingelesen und die Klassen der Integrationsstruktur erstellt werden. In Abbildung 6.4 sind die zur Spezifikation relevanten Klassen des UML-Metamodells, des SDL-Metamodells sowie die Klassen der Integrationsstruktur dargestellt.

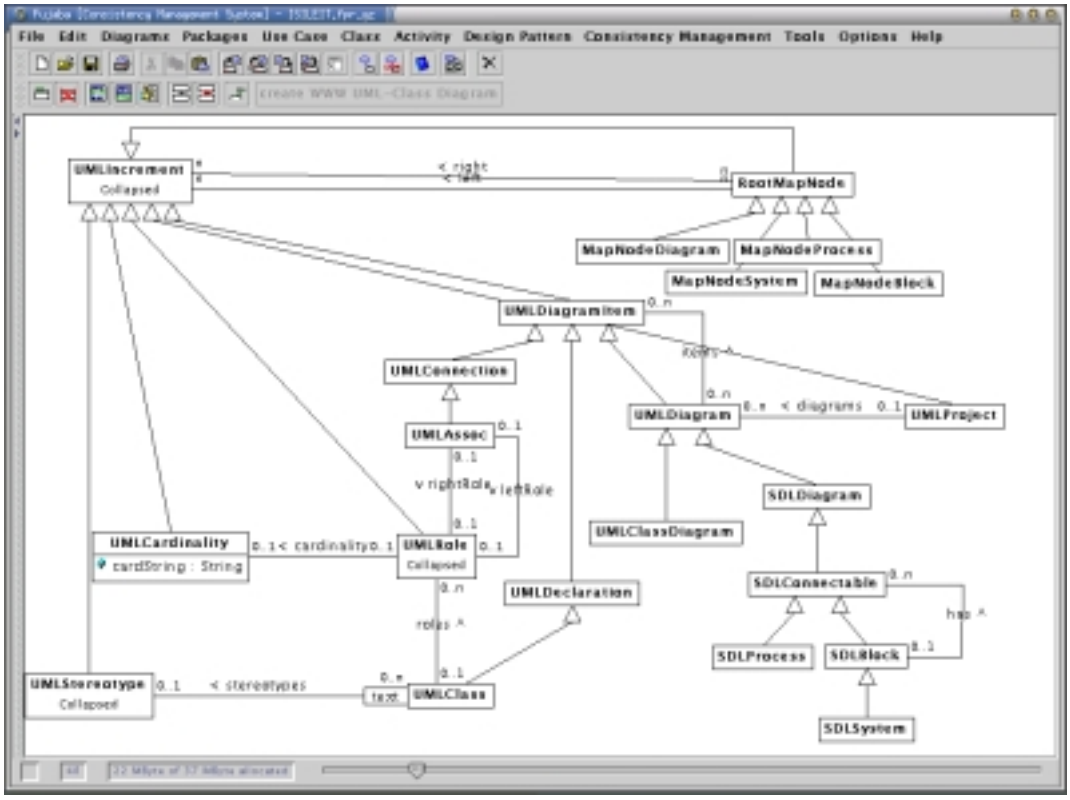


Abbildung 6.4: UML-Metamodell, SDL-Metamodell und Integrationsklassen

Nun kann in dem TGG-Diagramm die TGG-Regel spezifiziert werden. In Abbildung 6.5 ist die bereits aus Abschnitt 3.3 bekannte TGG-Regel zu sehen, die die Beziehung zwischen SDL-Blöcken und UML-Klassen modelliert. Um zwischen den

Objekten der unterschiedlichen Diagramme unterscheiden zu können, werden die Objekte durch die Schlüsselwörter *«left»*, *«map»* und *«right»* gekennzeichnet [Müh00].

Zur Einbindung der TGG-Regel in das Konsistenzmanagement-System werden über ein Kontextmenü und den darin enthaltenen Menüeintrag *Create Consistency Rule* die benötigten *Story-Diagramme* erstellt. Dabei wird der Name der Konsistenzregel im Regelkatalog, die zu erstellende Regel und die gewünschte Transformationsrichtung festgelegt. Abbildung 6.5 zeigt den Auswahldialog.

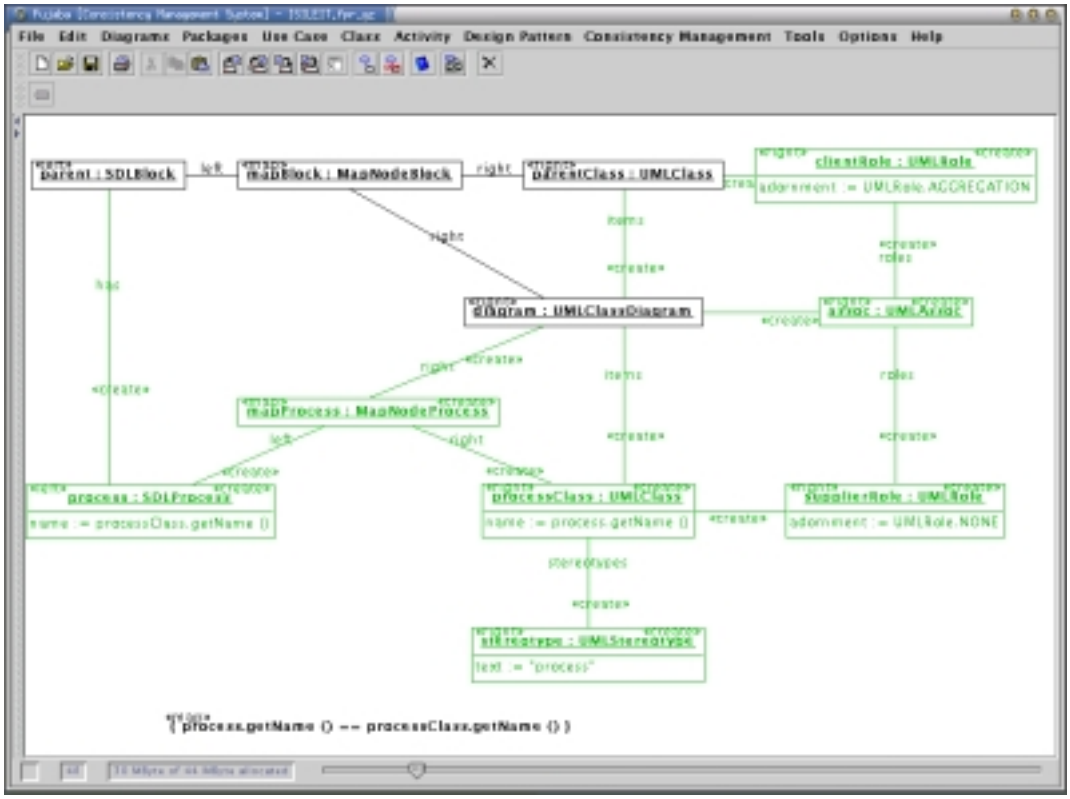


Abbildung 6.5: Regelspezifikation mit TGG-Regeln

Bei der zu erstellenden Regel stehen zur Auswahl *Create-Rule*, *Delete-Rule* und *Consistency-Rule*. Bei der Transformationsrichtung kann zwischen *forward* und *backward* gewählt werden. Durch eine geeignete Kombination können somit die Vorwärts-Regel, die Vorwärts-Lösch-Regel, die Vorwärts-Konsistenz-Regel und die entsprechenden Rückwärts-Regeln erstellt werden.

Nach der Spezifikation der Konsistenzregeln und der Festlegung weiterer Regeleigenschaften, müssen nun aus den Spezifikationen Konsistenzanalysemaschinen und eine XML-Katalogdatei erzeugt werden, die gemeinsam zur Konsistenzprüfung verwendet werden.

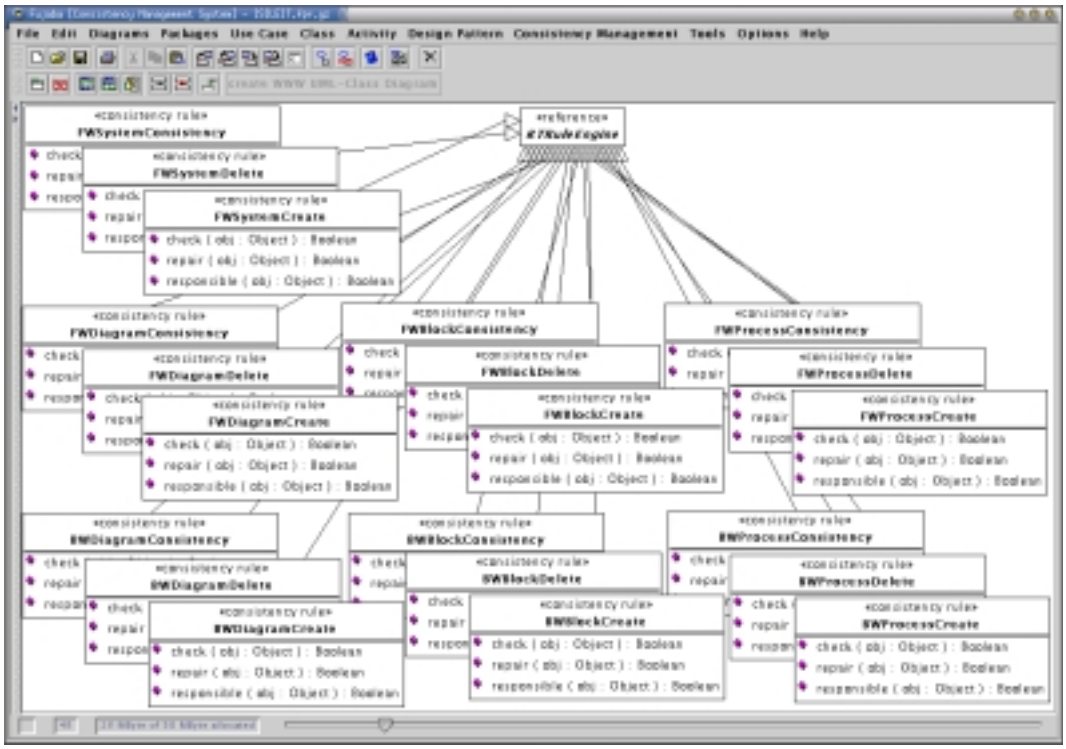


Abbildung 6.6: Klassendiagramm der generierten Konsistenzanalysemaschinen

Zu jeder Konsistenzbedingung des Regelkatalogs werden automatisch Klassen in dem *CMS-Engines* Klassendiagramm erzeugt. Diese Klassen erben alle von *RTRuleEngine* und implementieren die Methoden *responsible*, *check* sowie *repair* aus den spezifizierten *Story-Diagrammen*. Jede Konsistenzanalysemaschine, die durch diese Klassen repräsentiert wird, ist mit dem Stereotyp *«consistency rule»* gekennzeichnet. In Abbildung 6.6 sind die Analysemaschinen zur Transformation und Konsistenzsicherung zwischen SDL-Blockdiagrammen und UML-Klassendiagrammen dargestellt.

Über den Menüeintrag *Generate Source Code*, der in Abbildung 6.1 zu sehen ist, wird für jede Klasse eine übersetzbare JAVA-Quellcodedatei generiert und in dem Paket *cms.rt.engines* abgelegt. Zusätzlich müssen die Eigenschaften jeder Regel und die XML-Katalogdatei erstellt werden. Dies geschieht durch den Aufruf der Menüeinträge *Generate Property File* und *Generate XML File*. Je nachdem, ob nur eine Regel, eine Kategorie oder der gesamte Katalog ausgewählt worden sind, werden einzelne Regeln, alle Regeln einer Kategorie oder alle Regeln des Katalogs erzeugt. Um unab-

hängig von der Auswahl alles auf einmal zu generieren, kann die Schaltfläche der Werkzeugleiste benutzt werden.

6.2 Automatische Konsistenzsicherung

Bevor die neu erzeugten Konsistenzanalysemaschinen gestartet werden können, müssen die zugehörigen JAVA-Quelltexte übersetzt werden. Anschließend wird mit *Load Runtime Catalog* die XML-Katalogdatei geladen. In der Katalogdatei sind die Namen der zu ladenden Konsistenzanalysemaschinen hinterlegt, so dass die Klassendateien dynamisch vom Konsistenzmanagement-System aus dem Paket *cms.rt.engines* geladen und instanziiert werden.

Der Benutzer hat die Möglichkeit, den Katalog und damit die zu überprüfenden Regeln zu konfigurieren und an seine Bedürfnisse anzupassen. Dazu wird der Konfigurationsdialog über den Menüeintrag *Configure Consistency System* geöffnet.

In diesem Dialog werden die Einstellungen für jeden Katalogeintrag einzeln vorgenommen. Hierzu wird der Katalog auf der linken Seite in einer Baumstruktur aus Kategorien und Konsistenzregeln dargestellt. In diesem Katalog kann nun ein Katalogeintrag selektiert werden. Die aktuellen Einstellungen werden dann in der rechten Hälfte dargestellt und können, bis auf den Namen und die Beschreibung, verändert werden.

Beispielsweise können Kategorien und Regeln aktiviert und deaktiviert werden. Ist eine Kategorie deaktiviert, so werden die Einträge in dieser Kategorie bei der Konsistenzprüfung nicht berücksichtigt und damit die enthaltenen Konsistenzregeln auch nicht überprüft. Zusätzlich kann der Korrekturmodus und die Art der Benachrichtigung bei Konsistenzverletzungen gewählt werden.

Eine besondere Bedeutung kommt den beiden Einstellungen in dem oberen Bereich des Dialoges zu. Hier können die Einstellungen für die Benachrichtigung und die automatische Korrektur katalogübergreifend gesetzt werden. Ist beispielsweise, wie in Abbildung 6.7 zu sehen, der Eintrag *No Notification* markiert, so findet unabhängig von der lokalen Einstellung einer jeden Regel keine Benachrichtigung des Benutzers bei Konsistenzverletzungen statt. Ebenso können damit automatische Korrekturen verhindert werden.

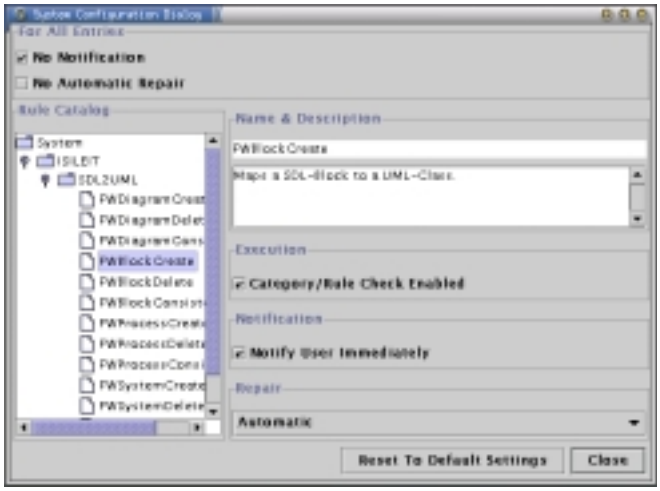


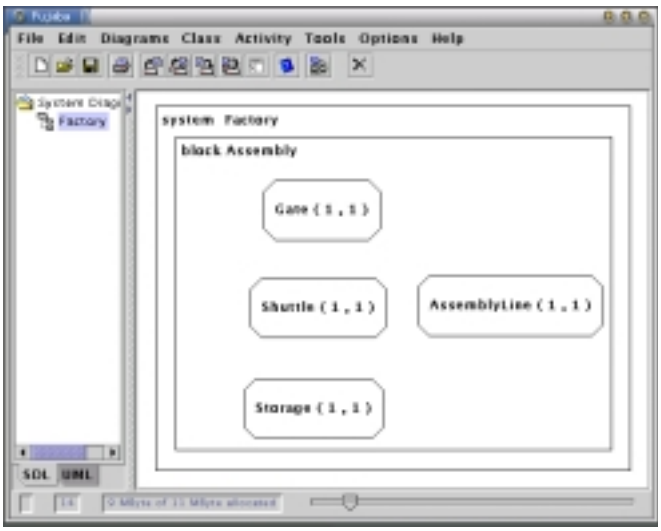
Abbildung 6.7: Konfiguration der Konsistenzprüfung

Nach dem Laden und der Konfiguration des Kataloges mit den Konsistenzregeln zur Transformation und Konsistenzsicherung zwischen einem SDL-Blockdiagramm und einem UML-Klassendiagramm wird das Konsistenzmanagement-System gestartet und ein neues SDL-Blockdiagramm angelegt.

Dieses Diagramm wird nun editiert. Es werden ein SDL-Systemblock sowie weitere Blöcke und Prozesse erstellt. In Abbildung 6.8 ist das spezifizierte SDL-Blockdiagramm dargestellt.

Während des Editiervorgangs werden nun nach jeder Benutzeraktion die Konsistenzregeln überprüft. Da nun beispielsweise zu einem SDL-Block noch keine UML-Klasse existiert, wird durch eine Vorwärts-Regel eine gleichnamige Klasse erzeugt und über ein Integrationsobjekt mit dem SDL-Block verbunden. Da die Benachrichtigung des Benutzers im Konfigurationsdialog ausgeschaltet wurde, kann das SDL-Blockdiagramm erstellt werden, ohne von dem Konsistenzmanagement-System gestört zu werden. Gleichzeitig wird das zugehörige und in Abbildung 6.8 ebenfalls abgebildete UML-Klassendiagramm erstellt.

Änderungen am UML-Klassendiagramm werden durch die Rückwärts-Regeln an das SDL-Blockdiagramm weiterpropagiert. Ändert der Benutzer beispielsweise den Namen einer UML-Klasse, die über das Integrationsobjekt mit einem SDL-Block verbunden ist, so wird durch die Rückwärts-Konsistenz-Regel auch der SDL-Block umbenannt. Mit Hilfe der Konsistenzregeln werden also beide Diagramme konsistent zueinander gehalten.



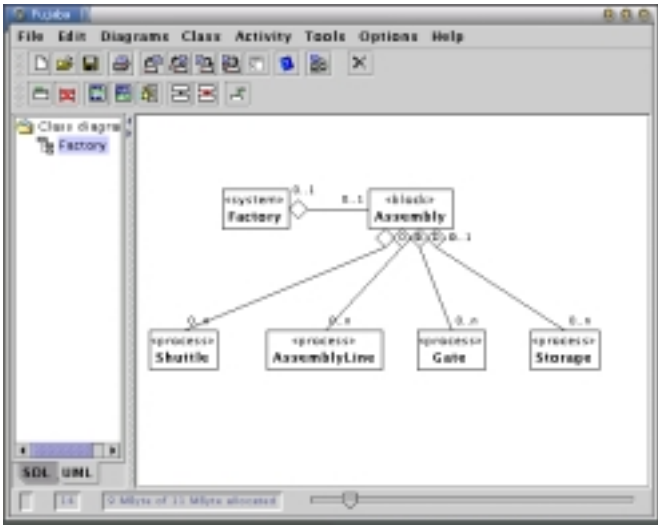


Abbildung 6.8: Automatische Ergänzung und Konsistenzsicherung

6.3 Interaktive Konsistenzprüfung

Nach der Spezifikation des SDL-Blockdiagrammes soll nun das durch das Konsistenzmanagement automatisch ergänzte UML-Klassendiagramm bearbeitet und verfeinert werden. Dazu wird der UML-Regelkatalog geladen und die Benachrichtigung bei Konsistenzverletzungen eingeschaltet, die automatische Korrektur von Konsistenzregeln für diesen Katalog aber ausgeschaltet. Mit diesen Einstellungen findet eine interaktive Konsistenzprüfung statt.

Angenommen, der Benutzer fügt ein neues Interface *Process* zu dem Klassendiagramm hinzu, das alle SDL-Prozessklassen implementieren müssen, und erweitert das Interface um ein Attribut.

Da die Konsistenzregel *InterfaceNoAttr* aus Abschnitt 3.2 Attribute in Schnittstellen verbietet und nur Konstanten erlaubt, erscheint der in Abbildung 6.9 gezeigte Benachrichtigungsdialog, der den Benutzer auf die Inkonsistenz aufmerksam macht.

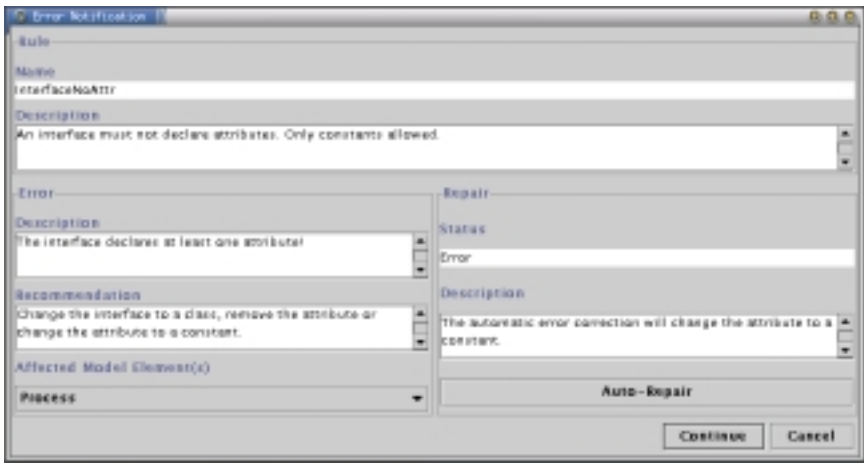


Abbildung 6.9: Benutzerinteraktion bei Erkennung einer Konsistenzverletzung

Da die automatische Korrektur ausgeschaltet wurde, ist die Konsistenzverletzung noch nicht behoben worden. Der Benutzer kann nun die Situation in aller Ruhe analysieren und entweder eine der vorgeschlagenen Methoden zur Behebung der Inkonsistenz manuell durchführen, die automatische Korrektur starten oder die Konsistenzverletzung ignorieren.

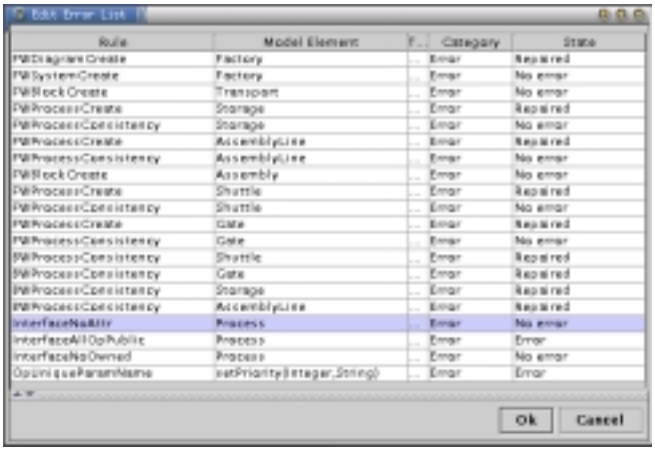


Abbildung 6.10: Tabellarische Ansicht aller Prüfungsergebnisse

Entscheidet sich der Benutzer für die letztgenannte Möglichkeit, so kann er zu einem späteren Zeitpunkt über den Menüeintrag *Edit Consistency Errors* alle erkannten Inkonsistenzen anzeigen und nachträglich die Inkonsistenz noch beheben.

Die Konsistenzverletzungen werden alle in einer Tabelle präsentiert. Es werden unter anderem der Name der verletzen Konsistenzregel, die betroffenen Elemente, die Fehlerkategorie und des Status der Konsistenzverletzung angezeigt.

Die Tabelle kann nach den einzelnen Spalten sortiert werden, so dass beispielsweise eine Sortierung nach den Modellelementen die Elemente gruppiert und eine Übersicht über alle verletzten Konsistenzbedingung eines Elementes bietet. Durch einen Doppelklick auf eine Zeile der Tabelle kann der Benutzerinteraktionsdialog aus Abbildung 6.9 wieder aufgerufen werden, um weitere Informationen zu der Inkonsistenz zu erhalten.

7 Zusammenfassung und Ausblick

In dieser Arbeit ist ein diagrammübergreifendes Konsistenzmanagement-System in der integrierten Entwicklungsumgebung FUJABA realisiert worden. Das Konsistenzmanagement-System unterstützt den Entwickler bei der Spezifikation von widerspruchsfreien und syntaktisch korrekten Modellen, die eine notwendige Grundlage für die Erstellung von fehlerfreier Software sind.

Zur Formulierung von Konsistenzbedingungen wurde das auf der Theorie von Graphgrammatiken basierende *Story Driven Modeling* verwendet, mit dem die gesamte formale Semantik von FUJABA spezifiziert worden ist. Für das Management von Konsistenzverletzungen ist es notwendig, zwischen tatsächlichen Inkonsistenzen, nicht erfüllten Vorbedingungen und Fehlern bei der automatischen Korrektur zu differenzieren, damit dem Benutzer nicht irrtümlich Konsistenzverletzungen gemeldet werden, die aber in Wirklichkeit keine sind. Da die Semantik einer Graphersetzungsregel in einem *Story-Diagramm* diese Unterscheidung nicht zuläßt, wurde die Spezifikation einer Konsistenzbedingung auf drei *Story-Diagramme* aufgeteilt.

Um unterschiedliche ASG-Objekte explizit in eine Beziehung setzen zu können, obwohl im Metamodell keine Assoziationen zwischen diesen Objekten vorhanden sind, wurde das Konzept der *Tripel-Graph-Grammatik* verwendet. Da der bestehende Formalismus der *Tripel-Graph-Grammatik* nicht ausreichend war, um inkrementelle Prüfungen bei Änderung von Attributwerten der ASG-Objekte auszuführen, mussten die aus der *Tripel-Graph-Grammatik* abgeleiteten Graphersetzungsregeln angepasst werden. Zur Einbindung in die Konsistenzprüfung wurden diese Graphersetzungsregeln anschließend in das *Story Driven Modeling* integriert.

Zur Überprüfung der spezifizierten Bedingungen wurde ein Steuerungsmechanismus entwickelt, der automatische Konsistenzprüfungen durchführt. Erkannte Konsistenzverletzungen werden dabei, soweit sinnvoll und möglich, automatisch behoben. Der realisierte Konsistenzsicherungsmechanismus ist weitgehend parametrisiert und kann an die individuellen Bedürfnisse des Benutzers angepasst werden. So lassen sich beispielsweise die automatischen Korrekturen ausschalten, so dass auch ein „Leben mit Inkonsistenzen“ möglich ist. Das Konsistenzmanagement-System basiert auf einem Regelwerk, in dem die spezifizierten Konsistenzregeln verwaltet werden. Konsistenzregeln können dabei zur Laufzeit des Systems hinzugefügt werden, so dass die

Konsistenzprüfung an die Bedürfnisse einer Anwendungsdomäne jederzeit angepasst werden kann.

Der Schwerpunkt dieser Arbeit wurde auf die Konsistenzsicherung im abstrakten Syntaxgraphen von FUJABA gelegt, um die syntaktische Korrektheit und die Konsistenz zwischen Diagrammen zu überprüfen und Verletzungen automatisch zu korrigieren. Damit können vom Benutzer hinsichtlich der Struktur konsistente Modelle erstellt werden.

Es gibt jedoch auch Konsistenzbedingungen, die nicht oder nur mit großem Aufwandt statisch in einem Modell überprüft werden können. Dies trifft insbesondere auf Invarianten und Integritätsbedingungen zu, die das System zur Laufzeit zusichern soll. Solche Bedingungen werden beispielsweise in der UML mit Hilfe der *Object Contraint Language* formuliert.

Daher muss untersucht werden, inwieweit das vorgestellte Konsistenzmanagement-System an die Anwendung gekoppelt werden kann, um die damit formulierten Bedingungen zur Laufzeit der Anwendung zu überprüfen. Die Formulierung von Konsistenzbedingungen, die zur Spezifikationszeit überprüft werden können, ist in dieser Diplomarbeit bereits ermöglicht worden. Die erstellten Regeln befinden sich in einem Regelkatalog, so dass sie dynamisch zur Überprüfung der Anwendung hinzugeladen werden können. Ebenso steht ein Mechanismus zur Verfügung, mit dem modifizierte Objekte der Konsistenzprüfung zugeführt werden. Um diesen Mechanismus jedoch nutzen zu können, muss das Modell der Anwendung eventuell geringfügig verändert werden. Zum Beispiel könnten Zugriffsmethoden auf Attribute und Assoziationen automatisch um Operationsaufrufe erweitert werden, mit denen das Konsistenzmanagement-System über Zustandsänderungen der Anwendungsobjekte benachrichtigt wird.

Es ist weiterhin zu untersuchen, zu welchem Zeitpunkt die Konsistenzprüfung stattfindet. In FUJABA ist der Zeitpunkt der Anwendung der Regeln durch Kontexte festgelegt worden, welche an die Interaktion des Benutzers mit FUJABA gebunden sind. Dieses Konzept muss auf die Ausführung der spezifizierten Anwendung übertragen werden. Hierzu gibt es mindestens zwei Möglichkeiten. Das Konsistenzmanagement-System könnte als eigenständiger Prozess im Hintergrund ablaufen. Dadurch würde es eine Beobachterposition einnehmen und die Prüfung durchführen, sobald ein Objekt modifiziert wurde. Hierbei ist jedoch die Synchronisationsproblematik zu beachten, da der Ausführungszeitpunkt nicht eindeutig definiert ist. Die zweite Möglichkeit besteht darin, zu untersuchen, inwieweit Kontexte definiert werden können, die sich nicht auf die Interaktion des Benutzers, sondern auf die Ausführung des Programms selbst beziehen. Dies könnten beispielsweise Kontexte sein, die den Zeitpunkt der Konsistenzprüfung an den Aufruf von Zugriffsmethoden binden.

A Spezifizierte Konsistenzregeln

A.1 TGG-Konsistenzregeln

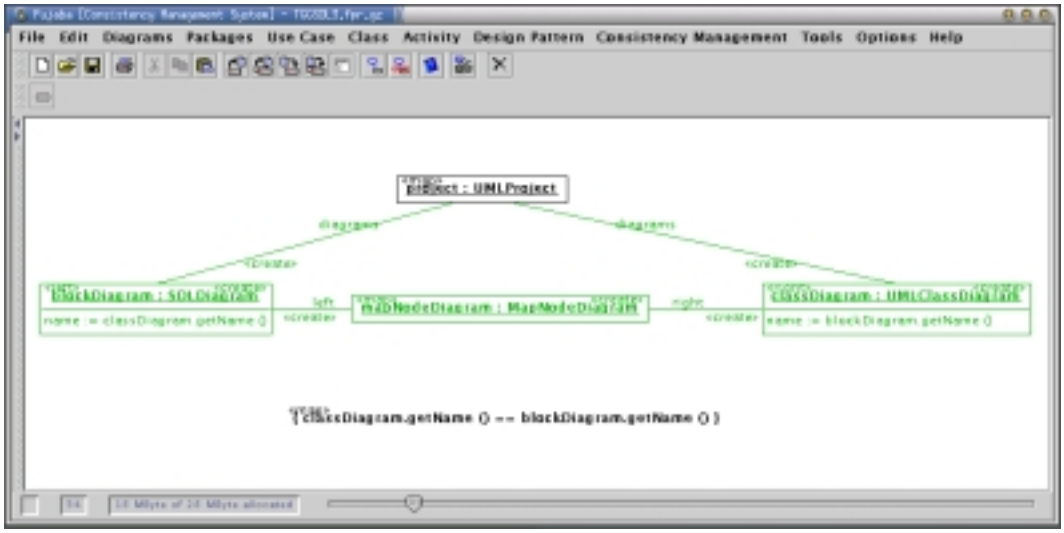


Abbildung A.1: TGG für SDL-Blockdiagramme und UML-Klassendiagramme

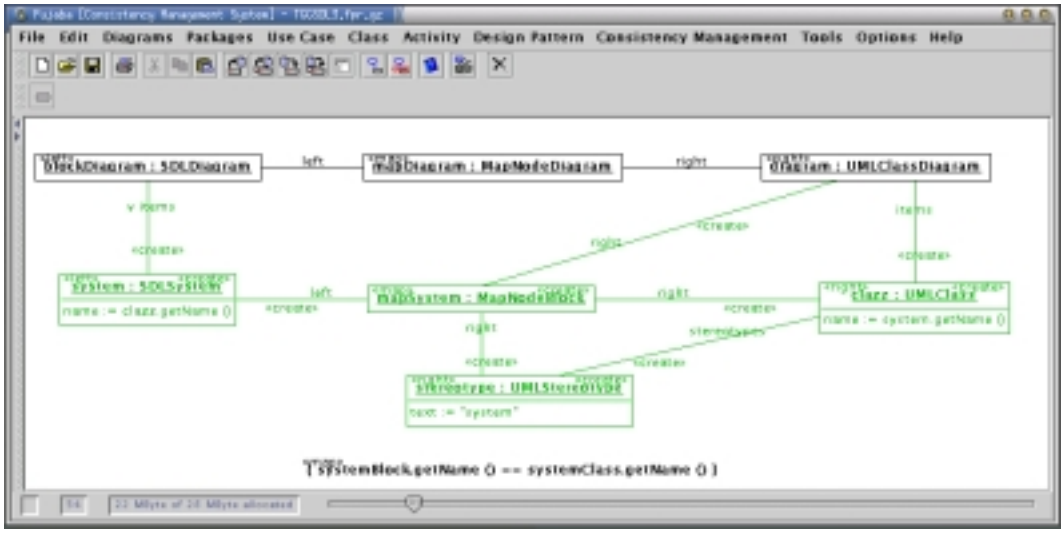


Abbildung A.2: TGG für SDL-System und UML-Klasse

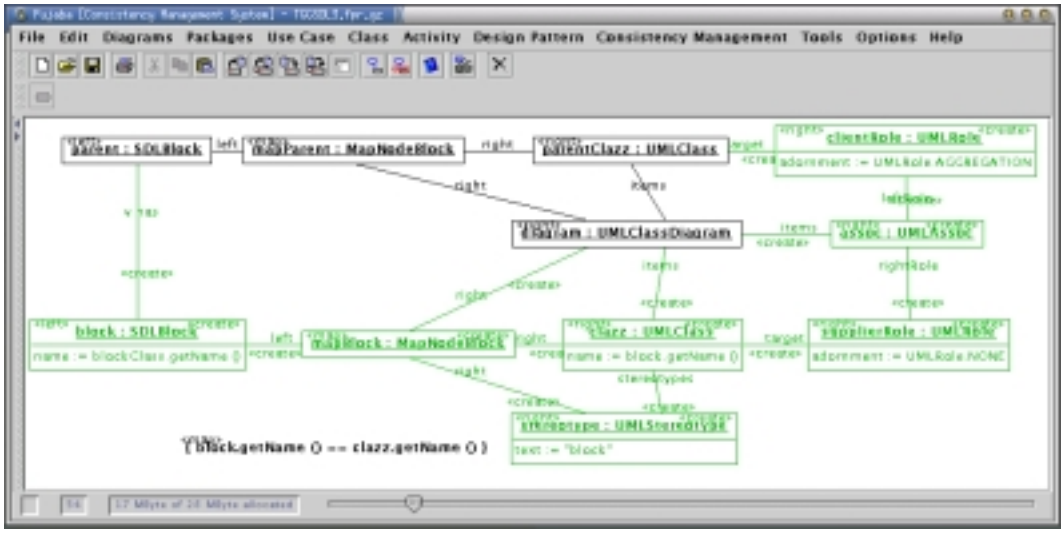


Abbildung A.3: TGG für SDL-Block und UML-Klasse

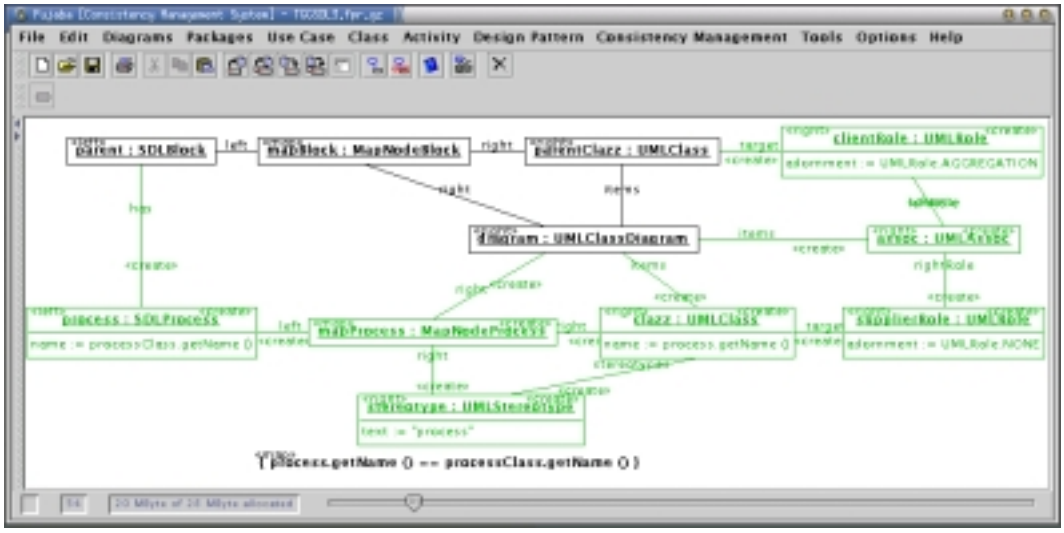


Abbildung A.4: TGG für SDL-Prozess und UML-Klasse

A.2 SDM-Konsistenzregeln

Tabelle 8: Kurzbeschreibung der Konsistenzregeln

Bezeichnung der Konsistenzregel	Abbildung	Kurzbeschreibung
AssocAggregateMultiplicity	A.5	Eine Instanz darf nicht von mehreren Objekten aggregiert werden.
AssocNavigableRoleTarget	A.6	Eine Assoziation darf von einem Interface nicht navigierbar sein.
AssocOneAggregateEnd	A.7	Eine Assoziation darf nur ein Assoziationsende besitzen, dasss eine Aggregation oder Komposition ist.
AssocUniqueRoleNames	A.8	Die Rollen einer Assoziation müssen unterschiedlich benannt sein.
AttrMatchRoleName	A.9	Ein Attribut, das eine Rolle in einer zu eins Assoziation implementiert, muss den gleichen Namen wie die Rolle haben.
AttrMatchRoleType	A.10	Ein Attribut, das eine Rolle in einer zu eins Assoziation implementiert, muss vom Typ der Klasse sein, mit der die Rolle verbunden ist.
InterfaceAllOpPublic	A.11	Alle Methoden eines Interfaces müssen öffentlich definiert sein.
InterfaceNoAttr	A.12	Ein Interface darf keine Attribute definieren. Es sind nur Konstanten erlaubt.
InterfaceNoOwned	A.13	Ein Interface darf keine inneren Klassen definieren.
OpUniqueParamName	A.14	Alle Parameter einer Methode müssen unterschiedliche Namen haben.

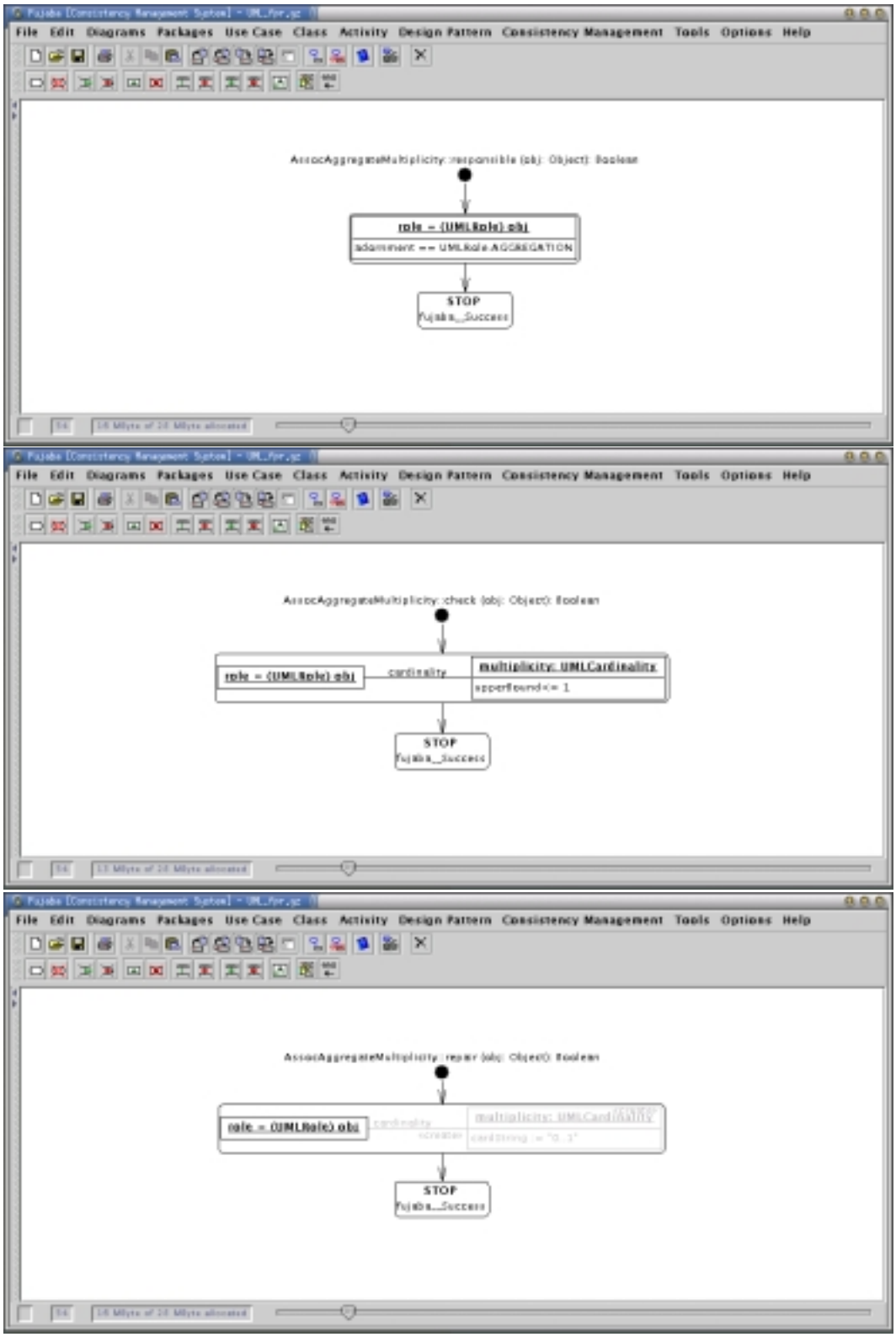


Abbildung A.5: AssocAggregateMultiplicity

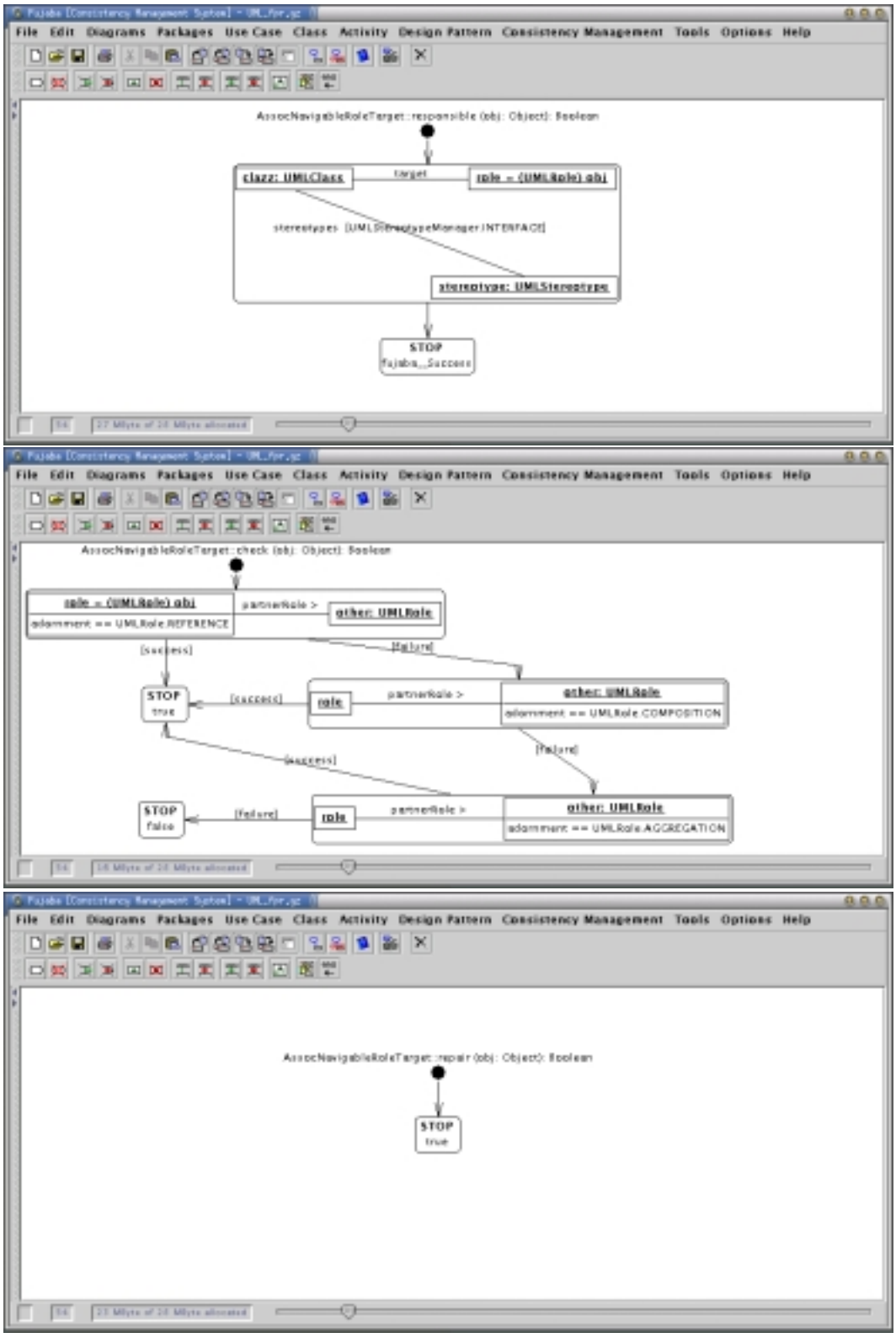


Abbildung A.6: AssocNavigableRoleTarget

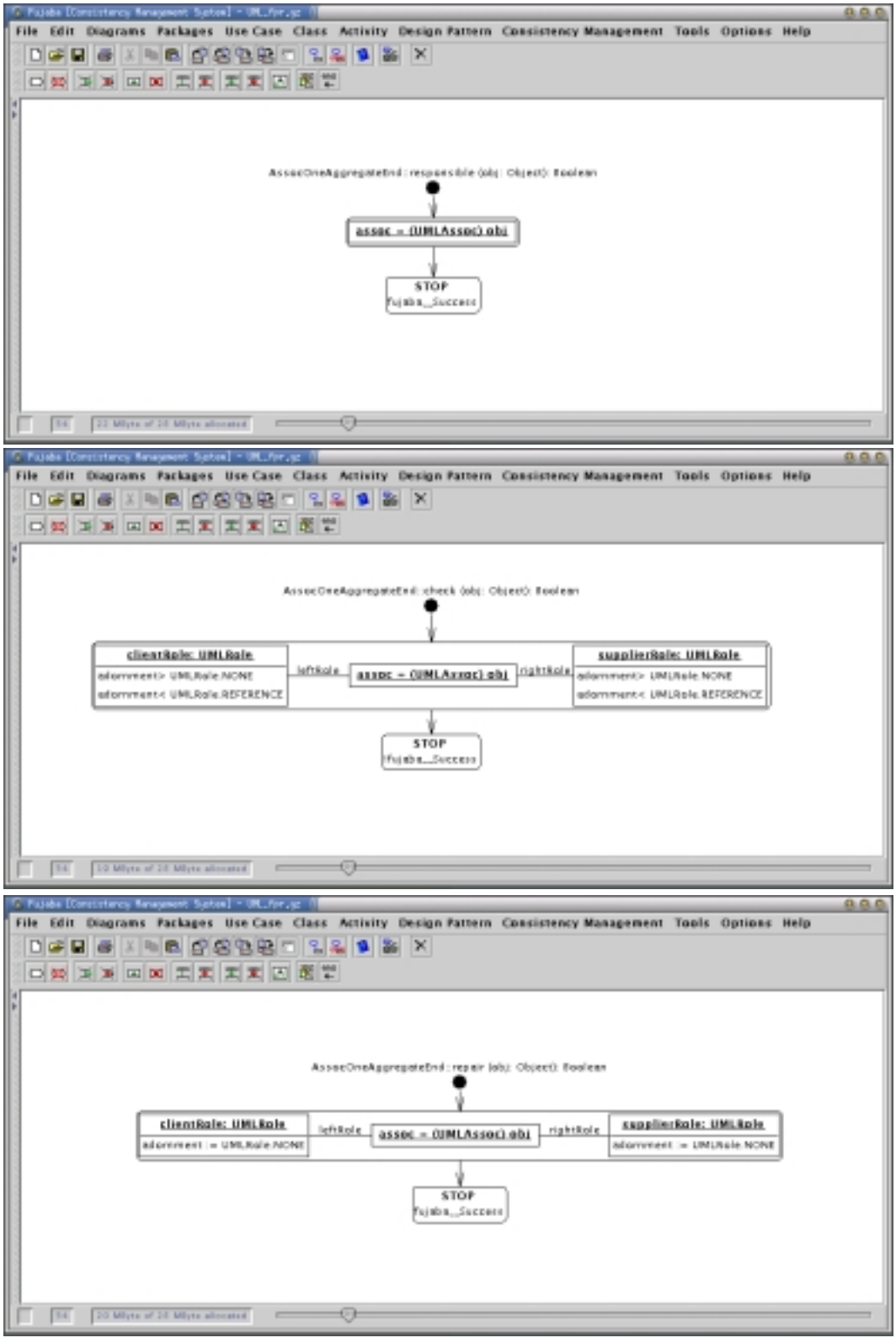


Abbildung A.7: AssocOneAggregateEnd

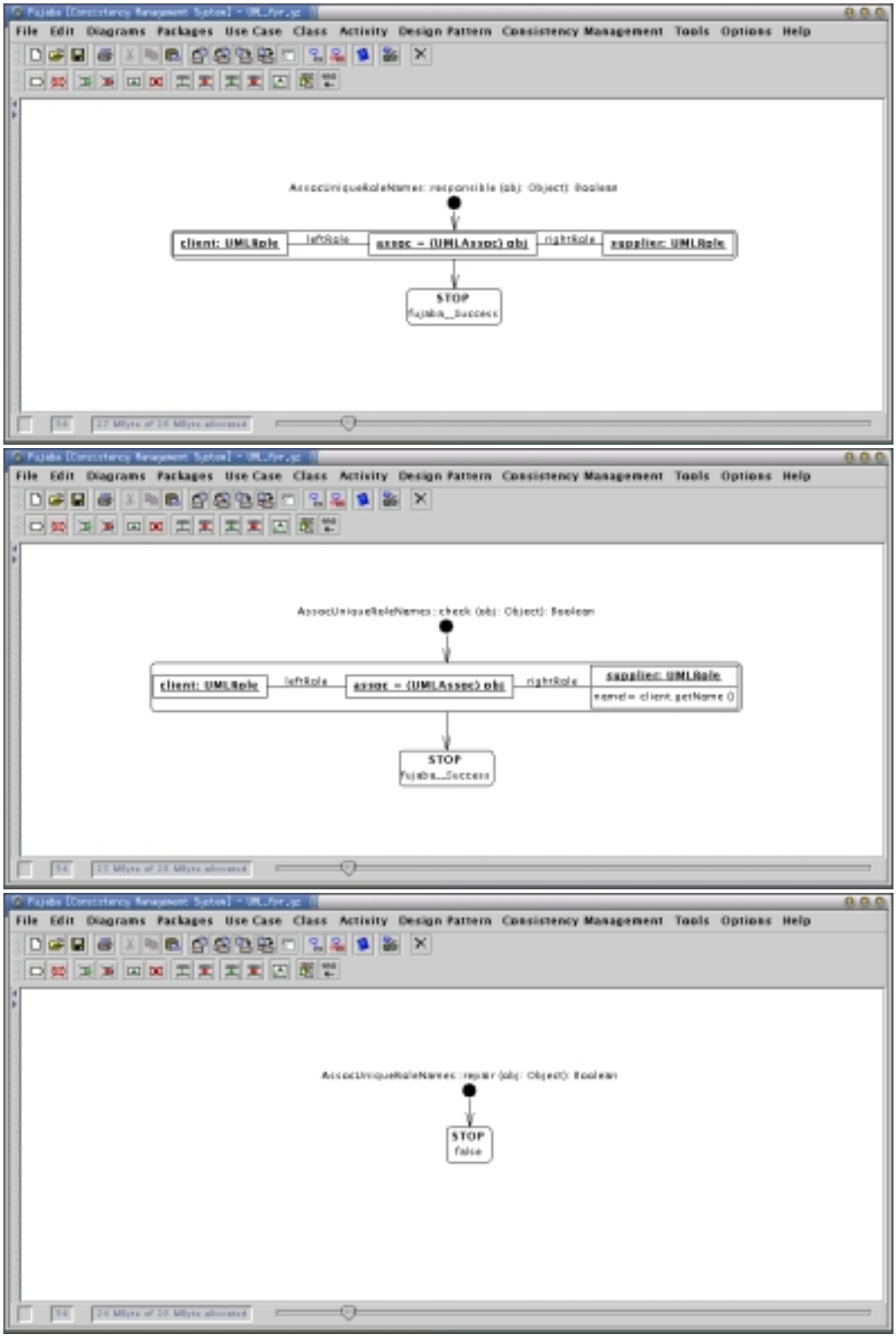


Abbildung A.8: AssocUniqueRoleNames

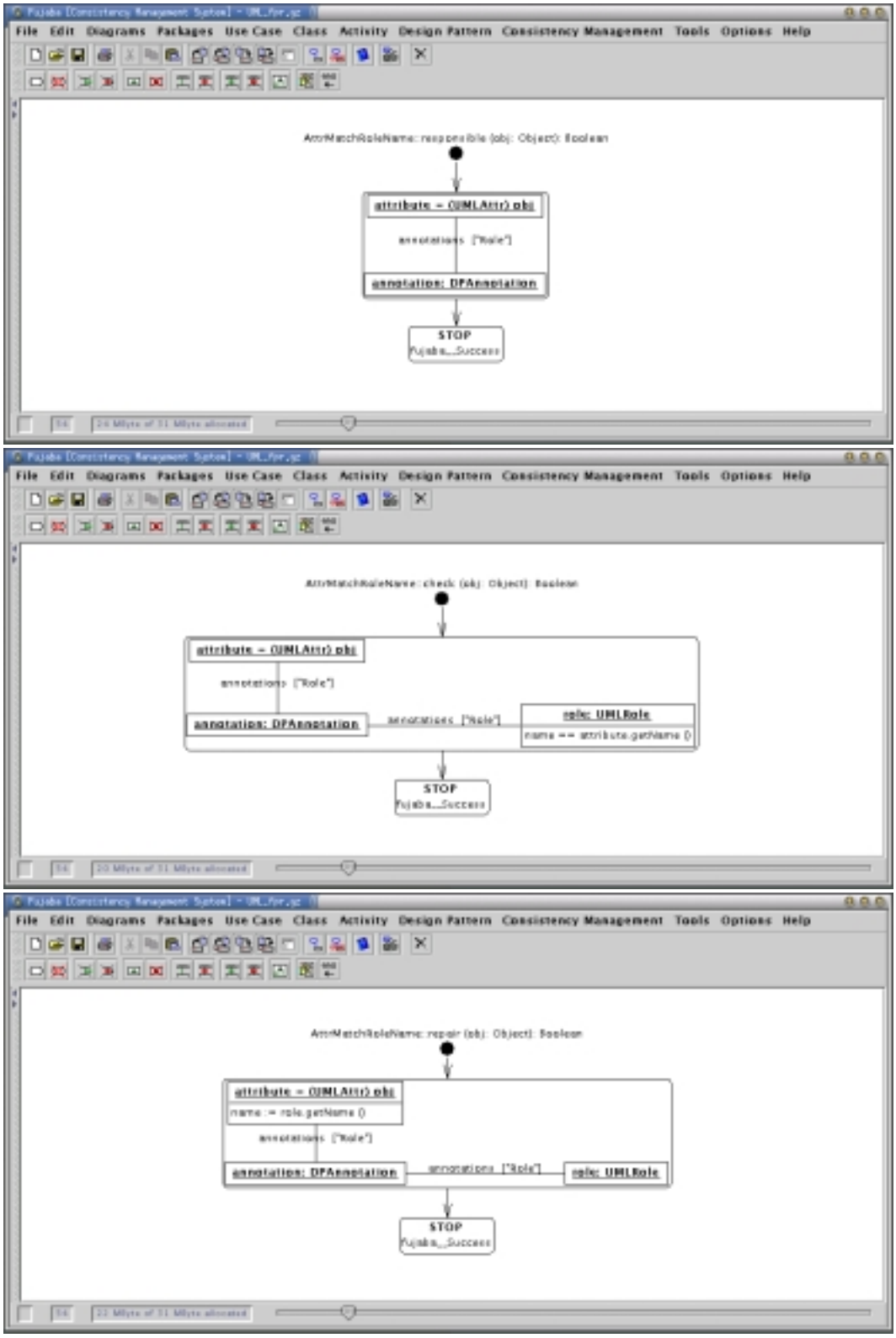


Abbildung A.9: AttrMatchRoleName

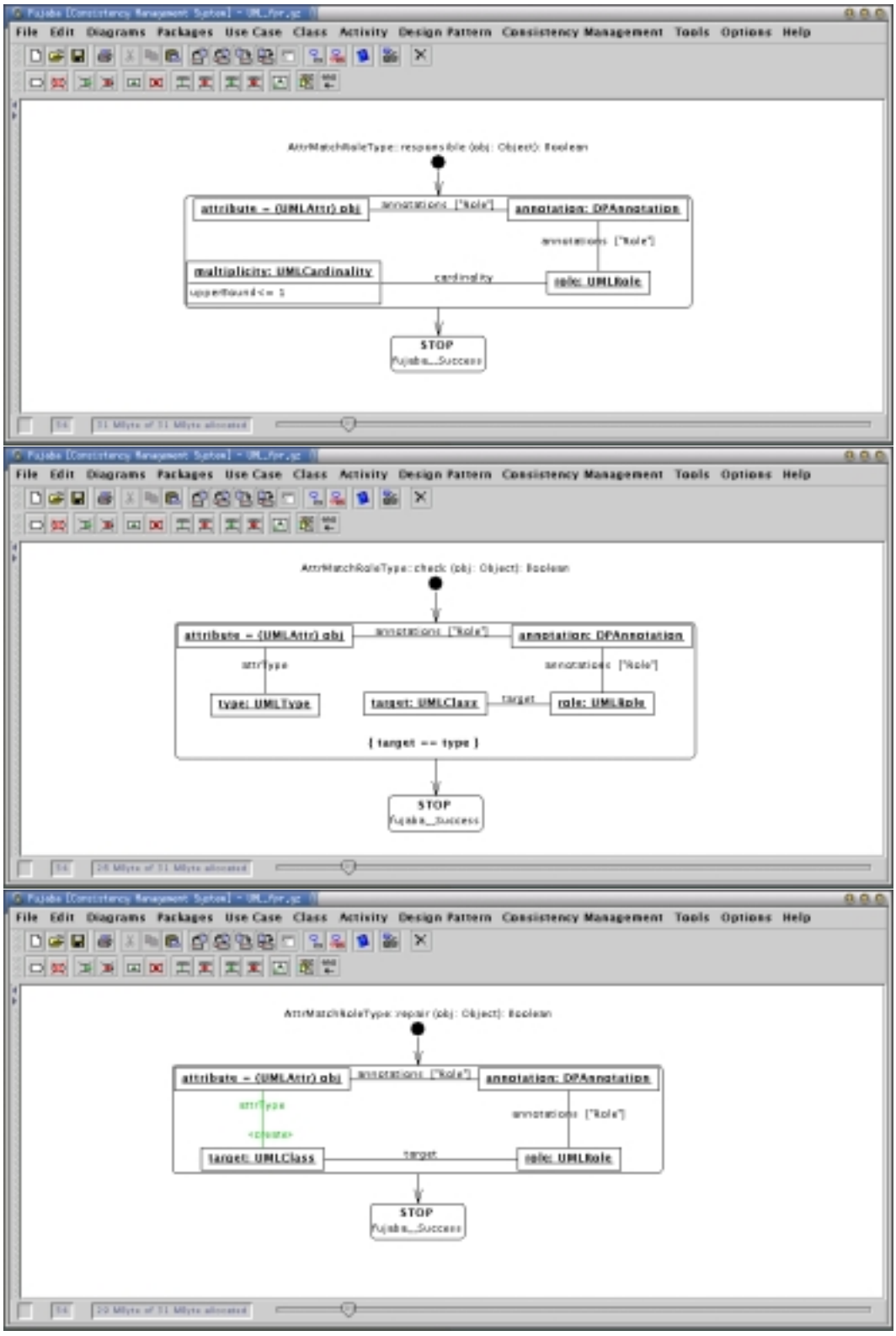


Abbildung A.10: AttrMatchRoleType

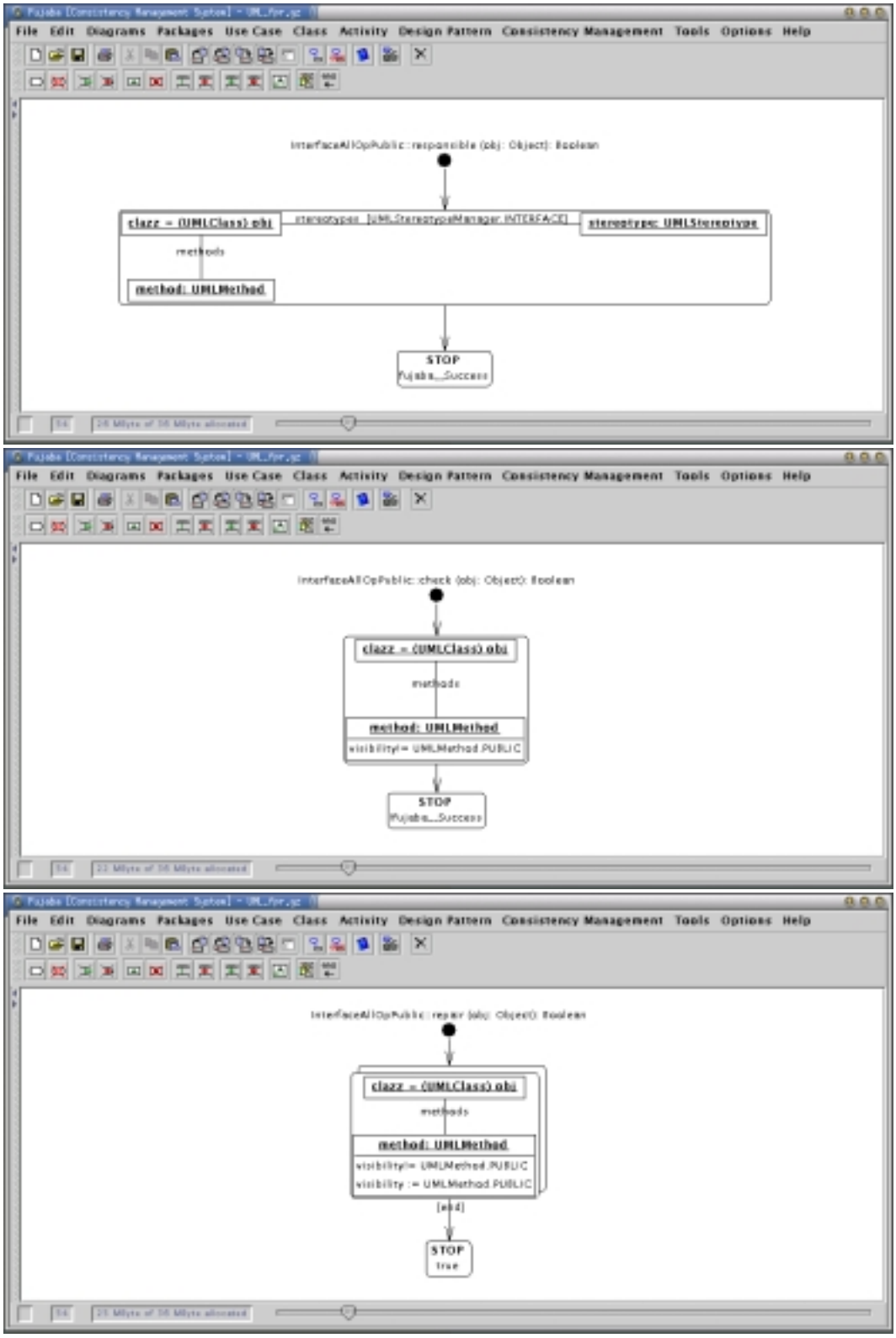


Abbildung A.11: InterfaceAllOpPublic

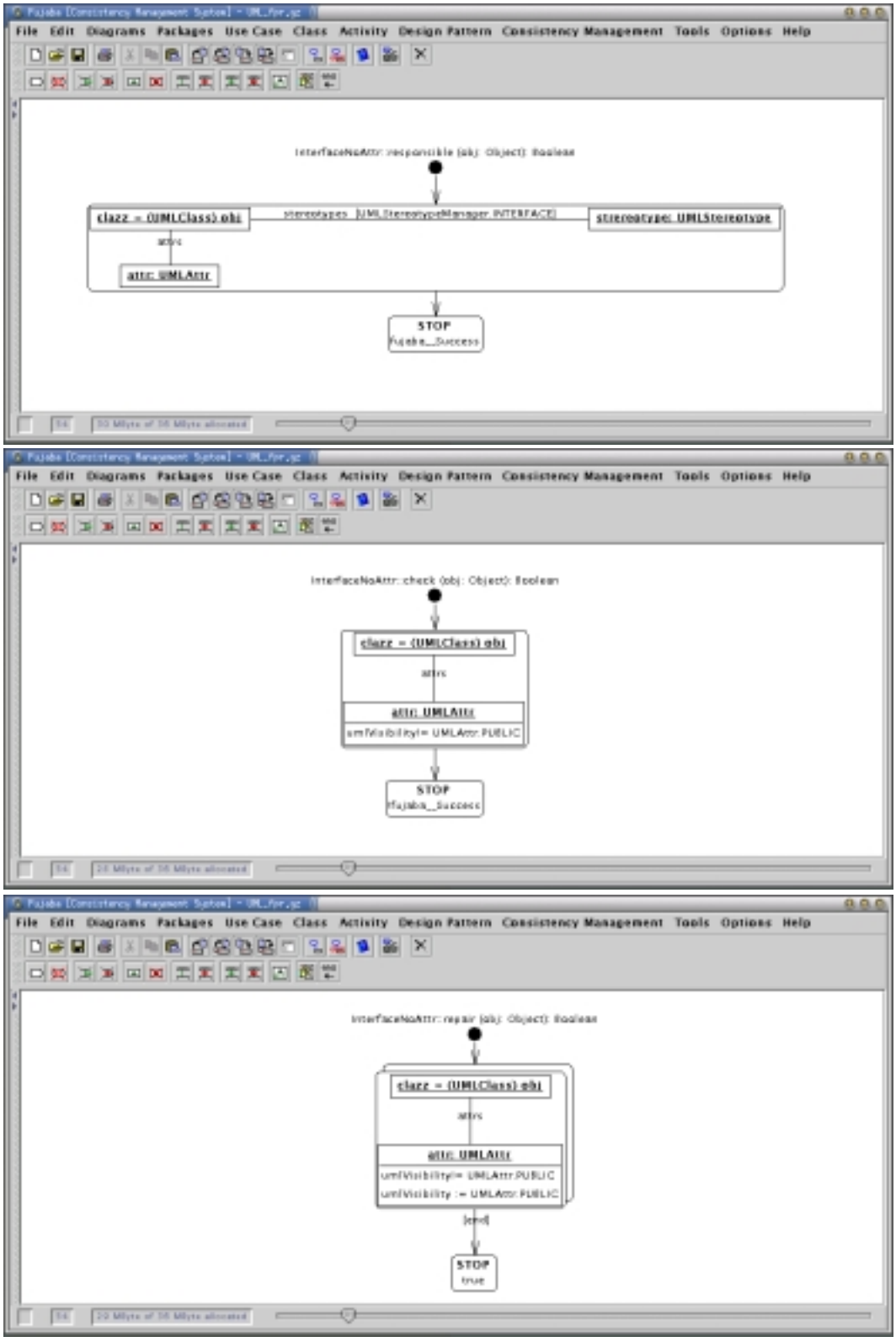


Abbildung A.12: InterfaceNoAttr

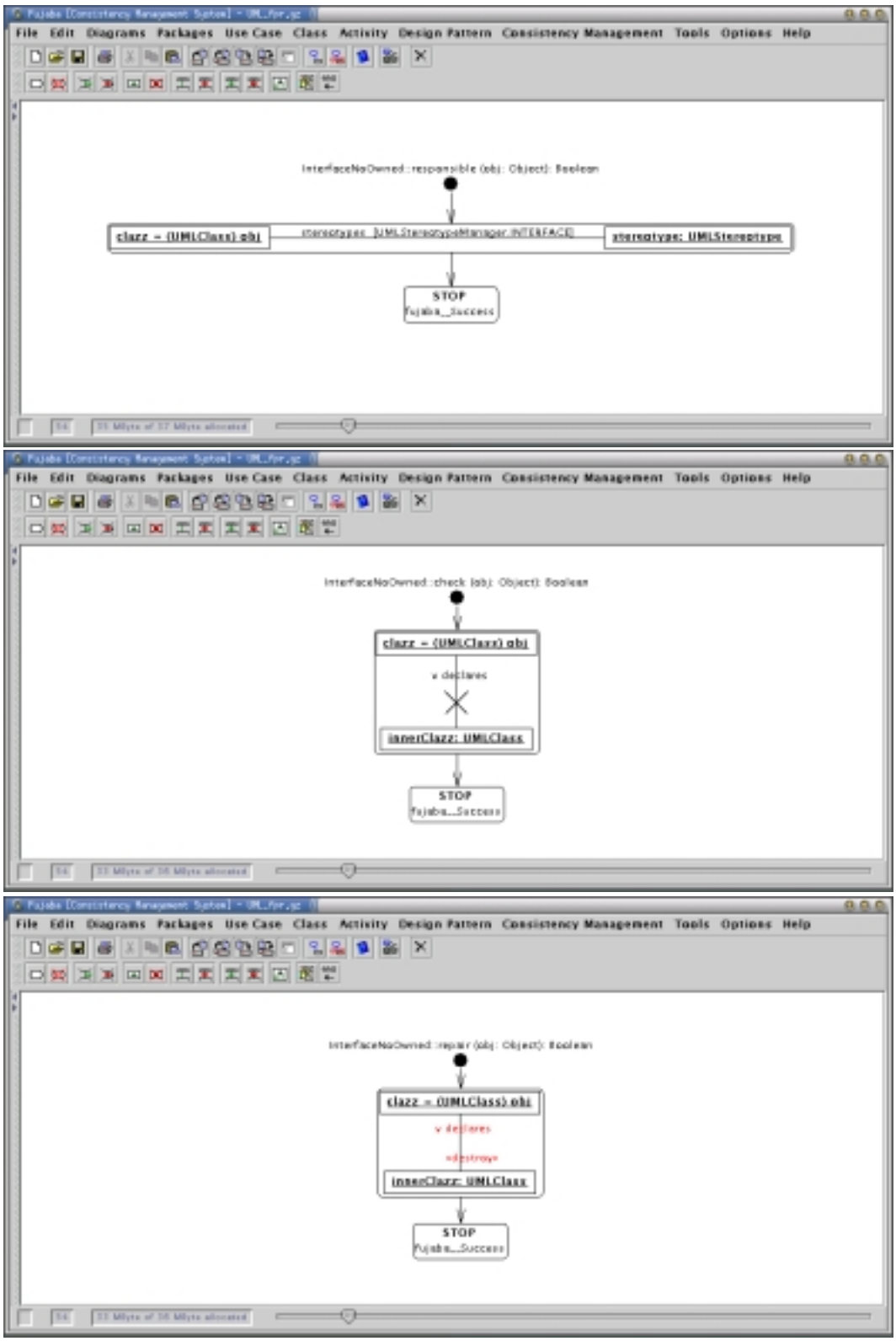


Abbildung A.13: InterfaceNoOwned

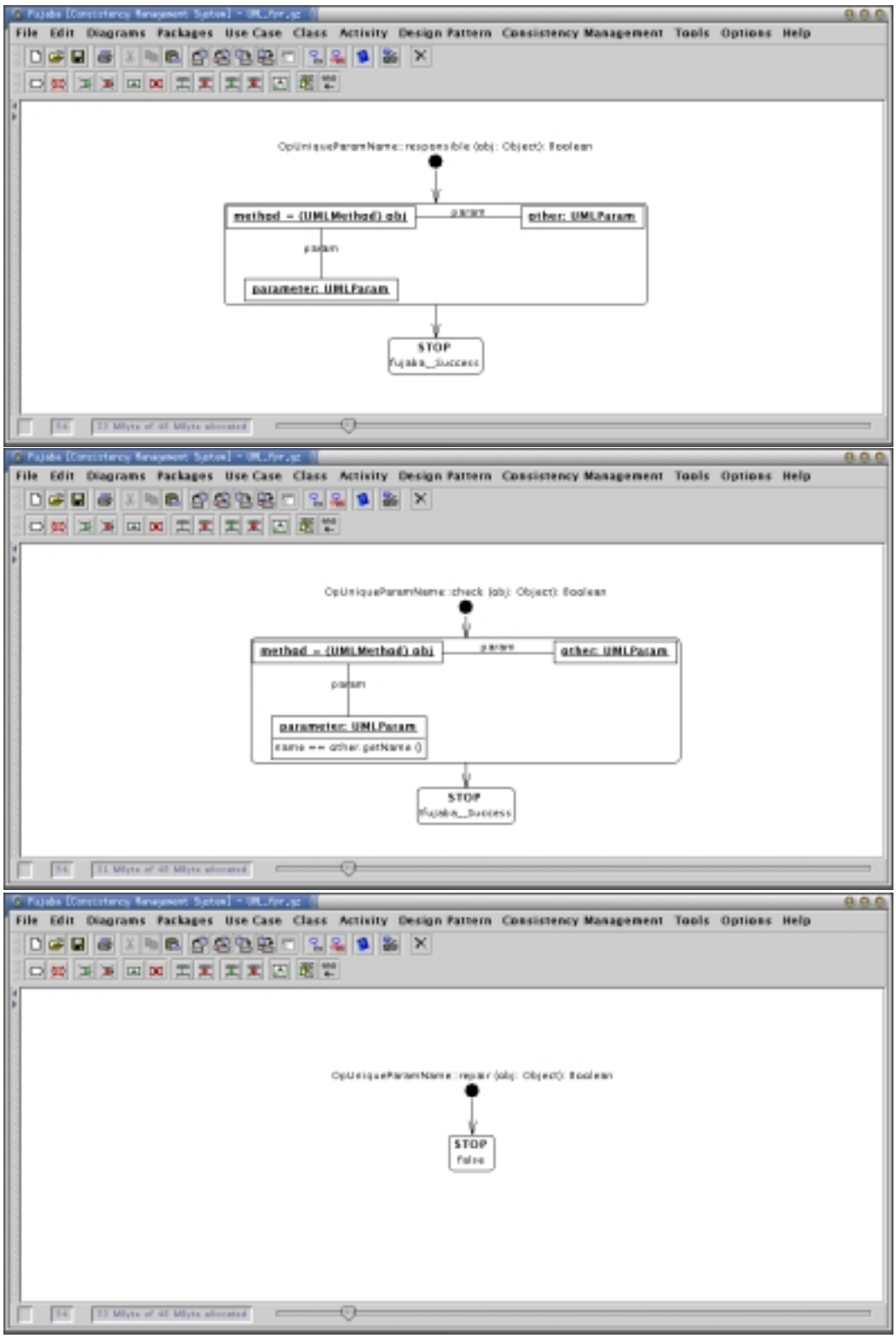


Abbildung A.14: OpUniqueParamName

Literatur

[Bal91] R. Balzer. *Tolerating Inconsistency*. In Proc. of the 13th International Conference on Software Engineering, Austin, Texas USA, pages 158-165. IEEE Computer Society Press, May 1991.

[BKPT00] P. Bottoni, M. Koch, F. Parisi-Presicce, and G. Taentzer. *Consistency Checking and Visualization of OCL Constraints*. Technical Report, University of Rome, SI-2000-03, 2000.

[EGHK01] G. Engels, L. Groenewegen, R. Heckel, and J.M. Küster. *A Methodology for Specifying and Analyzing Consistency of Object-Oriented Behavioral Models*. Joint 8th European Software Engineering Conference (ESEC) and 9th ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE-9), Vienna University of Technology, Austria, ACM Press, September 2001.

[Egy00] A. Egyed. *Automatically Validating Model Consistency during Refinement*. Technical Report, Center for Software Engineering, University of Southern California, Los Angeles, October 2000, http://sunset.usc.edu/publications/TECHRPTS/2000/index.html

[FNT98] T. Fischer, J. Niere, and L. Torunski. *Konzeption und Realisierung einer integrierten Entwicklungsumgebung für UML, Java und Story-Driven-Modeling*. University of Paderborn, Department of Mathematics and Computer Science, Paderborn, Germany, Diplomarbeit, July 1998.

[FNTZ98] T. Fischer, J. Niere, L. Torunski, and A. Zündorf. *Story Diagrams: A new Graph Rewrite Language based on the Unified Modeling Language*. In: Engels, G. (Hrsg.) ; G.Rozenberg

(Hrsg.): *Proc. of the 6th International Workshop on Theory and Application of Graph Transformation (TAGT)*, Paderborn, Germany, Springer Verlag, 1998 (LNCS1764).

[GHJV95] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object Oriented Software*. Reading, MA : Addison-Wesley, 1995.

[Gli00] M. Glintz. *A Lightweight Approach to Consistency of Scenarios and Class Models*. In: Proceedings of the 4th IEEE Conference on Requirements Engineering, Schaumburg, Illinois, 2000.

[GMT99] M. Goedicke, T. Meyer, and G. Taentzer. *ViewPoint-oriented Software Development by Distributed Graph Transformation: Towards a Basis for Living with Inconsistencies.* In Proc. 4th IEEE International Symposium on Requirements Engineering (RE'99), University of Limerick, Ireland. IEEE Computer Society, June 1999.

[GN98] C. Ghezzi and B. Nuseibeh, *Guest Editorial: Introduction to the Special Section on Managing Inconsistency in Software Development* (1). IEEE Transactions on Software Engineering, Vol. 24, No. 11, pages 906-907, November 1998.

[GN99] C. Ghezzi and B. Nuseibeh, *Guest Editorial: Introduction to the Special Section on Managing Inconsistency in Software Development (2)*. IEEE Transactions on Software Engineering, Vol. 25, No. 6, pages 782-783, November/December 1999.

[Har87] D. Harel. *Statecharts: A Visual Formalism for Complex Systems*. Science of Computer Programming, Vol. 8, pages 231-274, 1987.

[KNNZ00] H.-J. Köhler, U.Nickel, J.Niere, and A. Zündorf. *Integrating UML Diagrams for Production Control Systems*. In Proc. of the 22th International Conference on Software Engineering, Limerick, Ireland. ACM Press, 2000.

[Lea97] D. Lea. *Concurrent Programming in Java, Design Principles and Patterns*. Addison Wesley Longman, Bonn, 1997.

[Lef95] M. Lefring. *Integrationswerkzeuge in einer Softwareentwicklungsumgebung*. Verlag Shaker, Aachen, 1995. Dissertation am Lehrstuhl für Informatik III, RWTH Aachen.

[Müh00] J.H. Mühlenhoff. *Integration und Konsistenzprüfung von XML Datenbeständen*. University of Paderborn, Department of Mathematics and Computer Science, Paderborn, Germany, Diplomarbeit, Oktober 2000.

[NER00] B. Nuseibeh, E. Easterbrook, and A. Russo. *Leveraging Inconsistency in Software Development*. IEEE Computer, Vol,. 33, No. 4, pages 24-29, IEEE Computer Society Press, April 2000.

[NKF84] B. Nuseibeh, J. Kramer, and A. Finkelstein. *A Framework for Expressing the Relationships between Multiple Views in Requirements Specifikation*. IEEE Transactions on Software Engineering, pages 760-773, 1984.

[NW01] U.A. Nickel, and R. Wagner. *Graph-Grammar Based Completion and Transformation of SDL/UML-Diagrams.* In: Workshop on Transformations in UML (WTUML), Genova, Italy, April 2001.

[Pra71] T.W. Pratt. *Pair Grammars, Graph Lanuages and String-to-Graph Translations*. In: Journal of Computer and System Siences, Vol. 5, San Diego: Academic Press (1971), pages 560-595.

[Roz97] G. Rozenberg (Hrsg.). *Handbook of Graph Grammars and Computing by Graph Transformations*. World Scientific, Singapore, 1997.

[Sch94] A. Schürr. *Specification of Graph Translators with Tripel Graph Grammars*. In G. Tinhofer (ed.): Proc. WG 94 20th Int. Workshop on Graph-Theoretic Concepts in Computer Sci-

ence, Herrsching, Germany, Juni 1994. LNCS 903, Berlin, Springer Verlag (1994), pages 151-163.

[SDL96] International Telecommunication Union (ITU), Geneva. *ITU-T Recommendation Z.100: Specification and Description Language (SDL)*, 1994 + Addendum 1996.

[TE00] A. Tsiolakis and H. Ehrig. *Consistency Analysis between UML Class and Sequence Diagrams using Attributed Graph Grammars*. In H. Ehrig and G. Taentzer, editors, Proc. GraTra 2000 - Joint APPLIGRAPH and GETGRATS Workshop on Graph Transformation Systems, Technical Report 2000/2, pages 77-86. Technische Universität Berlin, March 25-27 2000.

[UML99] Rational Software Corporation: *UML documentation version 1.3 (1999). Online at http://www.rational.com*

[Zün96] A. Zündorf. *PROgramierte GRaphErsetzungsSysteme*. Deutscher Universitätsverlag, Wiesbaden, 1996.

Index