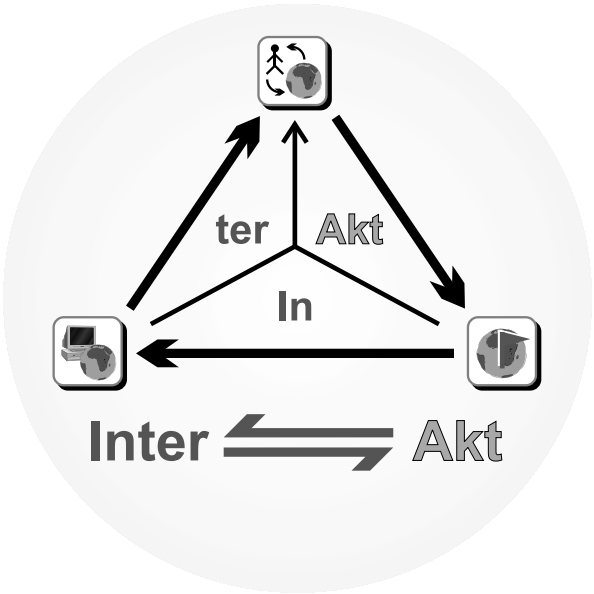
Abschlussbericht

PG 430
Interaktive Entwicklung
personalisierter Webapplikationen

Veranstaltet vom
Lehrstuhl für Programmiersysteme
und Übersetzerbau

1. April 2004



InterAkt sind:
Liang Dai, Steffen Dittrich, Dirk Jebing, Sven Jörges, Christian Kubczak, Nataliya Manusova, Sebastian Menge, Alexander Mitchkovski, Aidin Moumin, Guido Müller, Erick Gankam Tambo, Lufeng Zeng
Betreuer:
Dr. Oliver Niese, Dipl. Inform. Harald Raffelt, Dr. Tiziana Margaria, Prof. Dr. Bernhard Steffen
Kontakt:
interakt@yahoogroups.com

Inhaltsverzeichnis

Abbildungsverzeichnis

Code Beispiele

Abkürzungen

ABC Agent Building Center, Seite 1

CDB Constraint Database, Seite 64

DTD Document Type Definition, Seite 63

ESLTL Extended Semantic Linear Time Logic, Seite 61

EWIS Enhanced Web Information Service, Seite 1

HLL High Level Language, Seite 2

ITE Integrated Test Environment, Seite 1

LTL Linear Time Logic, Seite 60

SIBs Service Independent Building Blocks, Seite 2

SLG Service Logic Graph, Seite 5

Tcl/Tk Tool Command Language / Toolkit, Seite 64

XML Extensible Markup Language, Seite 63

1. Einleitung

1.1. Überblick

1.1.1. Motivation

Die Aufgabe der Projektgruppe InterAkt ist die Realisierung einer interaktiven Entwicklungs- und Wartungsumgebung für Web-Applikationen. Die konzeptuelle Komplexität moderner Web-Applikationen stellt extrem hohe Anforderungen an ihre (integrierten) Entwicklungsumgebungen in Bezug auf

- Aspektorientierte, verteilte Entwicklung,
- Validierung,
- Ausführungsumgebung,
- Tests, und
- das Monitoring in Produktion befindlicher Lösungen.

Damit Fehler frühstmöglich erkannt werden, ist es hierbei insbesondere wichtig, dass der jeweilige Nutzer (z.B. Entwickler oder Tester) in jeder Entwicklungsphase interaktiv mit der Entwicklungsumgebung arbeiten kann, also von der Konzeption bis zur Wartung nach der Inbetriebnahme durchgängig adäquate Ausführungs- und Validierungsmöglichkeiten mit direktem Feedback bekommt. Moderne internetbasierte Anwendungen sind mehrstufige, verteilte Applikationen, die typischerweise auf heterogenen Betriebssystemen ausgeführt werden. Ihre korrekte Ausführung hängt insbesondere von der komponentenübergreifenden Zusammenarbeit verschiedenster Teilsysteme ab.

Zur Verfügung stehen die Enhanced Web Information Service-Umgebung (EWIS) [2] und das Integrated Test Environment (ITE) [14]. Es handelt sich hierbei um Applikationen, die auf dem Agent Building Center (ABC) [19] basieren. Diese zwei Umgebungen decken bereits die flexible Entwicklung personalisierter Web-Applikationen sowie die Erstellung von Testfällen und deren Ausführung ab. Eine ganzheitliche interaktive Unterstützung wird aber derzeit nur ansatzweise geleistet.

1.1.2. Agent Building Center

Das ABC ist ein generisches Framework der Firma METAFrame zur intuitiven Softwareentwicklung und basiert auf formalen Methoden. Das ABC unterstützt bibliothekenbasierte Softwareentwicklung durch die Kombination von einzelnen Elementen auf einem abstrakten Niveau. Für verschiedene spezialisierte Entwicklungsaufgaben stellt das ABC

erweiterbare generische Schnittstellen zur Verfügung. Eine ausführliche Erklärung ist in [19] zu finden.

1.1.2.1. Grundlegende Funktionsweise

Ziel des ABC ist es, programmierungsfreie Softwareentwicklung zu ermöglichen. Hierzu werden verhaltensorientierte Funktionseinheiten eingesetzt, welche es ermöglichen, die Funktionsweise von Software auf einem abstraktem Niveau zu modellieren. Diese Funktionseinheiten werden innerhalb des ABC als Service Independent Building Blocks (SIBs) bezeichnet und bilden die Grundlage jeder Arbeit mit dem ABC. SIBs sind parametrisierbar, wodurch eine hohe Wiederverwendbarkeit geleistet wird. Die Funktionalität einer Applikation wird nun mit Hilfe von Flussgraphen, die SIBs als Knoten haben, grafisch dargestellt. Diese Grundstruktur ist dabei unabhängig von dem zugrunde liegenden Programmiermodell. Die Entwicklung einer Applikation ohne Programmierkenntnisse ist natürlich nur dann möglich, wenn die verfügbaren SIBs den Funktionalitätsbedarf abdecken.

1.1.2.2. Konsistenzprüfung

Durch die Definition von lose spezifizierten Regeln lässt sich die korrekte und konsistente Verwendung von Funktionseinheiten während der Entwicklung mit dem ABC überprüfen. Außerdem unterstützt das ABC das dynamische Wachstum eines hierarchisch organisierten Regelwerkes. Die Prüfung selbst wird hierbei durch Model-Checking Methoden vorgenommen.

1.1.2.3. Prototypen

Das ABC trennt ausdrücklich die applikationsspezifische Implementierung der SIBs von ihren Beschreibungen. Zusätzlich ist es möglich, für jeden SIB einen Tracer-Code in einer speziellen Skriptsprache, der sogenannten High Level Language (HLL), zu definieren. Die Beschreibung von der HLL ist in [10] zu finden. Der Tracer vom ABC kann Knoten für Knoten die Tracer-Codes der einzelnen SIBs ausführen. Mit diesen Prototypen lässt sich noch während der Entwicklung, vor der eigentlichen Implementierung der SIBs, ein Eindruck des modellierten Gesamtsystems gewinnen. Allerdings ist es aufwändig, den Tracer-Code zum Erstellen eines Prototypen zu verwenden, da das komplette Laufzeitverhalten eines SIB nachgebildet werden muss, um das Verhalten im realen Betrieb zu simulieren. Außerdem müssen Dialoge, die HTML-Formularen entsprechen, manuell erstellt werden.

1.1.2.4. Architektur

Einführung Mit Hilfe des ABC können verschiedene Module kombiniert werden. Durch die Kombination von Modulen entsteht eine vielseitige Entwicklungsumgebung für die unterschiedlichsten Anwendungsgebiete (z.B. Internet-Applikationen oder Telefon-Dienste). Als Steuerungssprache für die Kombination der Module wird die HLL als Program-

miersprache benutzt. Die Funktionalitäten und Datentypen der Module werden durch Funktionen und Typen der Programmiersprache dargestellt. Auf diese Weise können durch eine abstrakte, nicht technische Steuersprache die Funktionalitäten der Module genutzt werden. Die Flexibilität des Systems erlaubt es, die Funktionalitäten der Module durch Algorithmen, die in einer beliebigen Programmiersprache vorliegen können, zu implementieren. Eigenständige Programme, deren Sourcecode nicht verfügbar ist, können ebenfalls genutzt werden. Algorithmen und eigenständige Programme werden im Folgenden als Komponenten bezeichnet. Wegen der Heterogenität der Komponenten, die zum Beispiel als C oder C++ Algorithmen implementiert werden können, und der Komplexität des Systems, benötigt jede Komponente einen Adapter, der als Mediator zwischen der Komponentenschnittstelle und dem Interpreter des ABC fungiert. Die Adapter exportieren die Funktionalitäten und Datentypen der Komponenten als HLL-Module bestehend aus HLL-Funktionen und HLL-Typen. Die verschiedenen Bauteile des ABC werden also in Modulen (Komponenten + Adapter) gekapselt, die innerhalb des Gesamtsystems über einen Interpreter kommunizieren.

Bei der Entwicklung des ABC wurden wichtige Regeln der Software-Technik verfolgt, die ausführlich in [19] erklärt werden:

- Separation of Concern
- Parameterization, Exchangeability and Reuse
- Tailoring of the visible Complexity
- Evolutionary Application Programming
- Semantics-Based Control
- Incremental Formalization

Alle diese Eigenschaften sollten die Akzeptanz, die Benutzbarkeit und die Weiterentwicklung des Systems charakterisieren.

Komponenten und Schichten des ABC Das Design des ABC läßt sich auf unterschiedliche Sichtweisen betrachten. Die topologische Sichtweise ordnet das System in die Kategorie eines Drei-Schichten-Modells ein. Es werden dabei unterschieden:

- Darstellungsebene,
- Koordinationsebene und
- Modulebene.

Ein Blick ins Innere des Systems liefert die strukturelle Architektur. Sie wird, anders als die topologische Sicht, in vier Schichten unterteilt. Diese sind:

- Darstellungsebene,
- Interpreterebene,

- Adapterebene und
- Komponentenebene.

Dies sind die Hauptelemente des Systems, die in Abbildung 1.1 dargestellt werden, und in den zwei nächsten Abschnitten beschrieben werden.

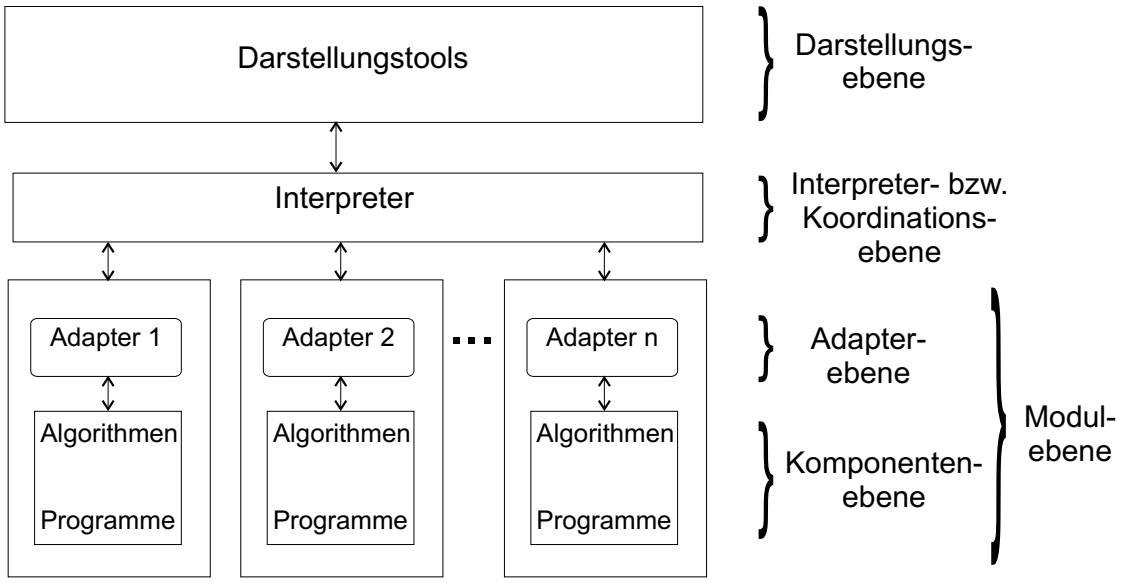


Abbildung 1.1.: Komponenten und Schichten des ABC

Topologische Sichtweise

Darstellungsebene Für die Modellierung oder die Beschreibung von Prozessen steht eine grafische Benutzeroberfläche zur Verfügung (SD-Modul). Auf dieser Benutzeroberfläche werden die Funktionen der Prozesse mit Hilfe der SIBs beschrieben. Die Funktionalität eines Prozesses wird durch einen Graphen, der den Kontrollfluss der Applikation modelliert, grafisch dargestellt.

Koordinationsebene Die Koordinationsebene, die zwischen der Darstellungsebene und der Modulebene liegt, fungiert als Kontrollplattform und bildet den Kern des Systems. Der Interpreter des ABC ist hier der wichtigste Bestandteil. Dieser arbeitet mit der Programmiersprache HLL und ermöglicht die Kommunikation zwischen den verschiedenen Modulen untereinander, sowie zwischen der Darstellungsebene und verschiedenen Modulen.

Modulebene Die Modulebene bildet die unterste Schicht in Hierarchie und ist eine Datenbank von Modulen, die mit der Koordinationsebene interagiert. Hier werden ver-

schiedene Funktionalitäten, die den Anforderungen von Prozessen entsprechen, implementiert. Eines der wichtigsten Module ist das SD-Modul, das es ermöglicht, Prozesse in einem Graphen zu modellieren und anschließend verschiedene Algorithmen darauf anzuwenden.

Strukturelle Sichtweise

Darstellungsebene Analog zur topologischen Sichtweise.

Interpreterebene Analog zur topologischen Sichtweise.

Adapterebene Die Adapterebene ist zuständig für die Integration der Komponenten in das System. Auf der Adapterebene werden die Schnittstellen der Module an den Interpreter angepasst.

Komponentenebene Die Komponentenebene wird durch die zu integrierenden Objekte gebildet. Dabei können die Komponenten zum Beispiel Algorithmen oder Datenstrukturen für die Analyse und Verifikation von Programmen sein.

Einen tiefergehenden Einblick in die Architektur des ABC liefert das Dokument [19].

1.1.2.5. Zusammenhang zwischen ABC, EWIS und ITE

EWIS und ITE sind zwei Module des ABC. Mit EWIS können Web-Applikationen grafisch mit Hilfe von Service Logic Graphen (SLGs) entworfen werden. Ein SLG besteht aus Kanten, in denen sich der Kontrollfluss der Applikation widerspiegelt, und Knoten, die aus SIBs mit ihren konkreten Parametern und Branches bestehen. Branches repräsentieren Ausgänge eines SIB und können Kanten zugewiesen werden. SIBs werden bei Web-Applikationen in Java implementiert und stellen die Programmfunktionen dar. Durch Übersetzen des SLG und seiner SIBs wird die entworfene Web-Applikation erstellt. Dieser Vorgang ist in Abbildung 1.2 auf der nächsten Seite dargestellt.

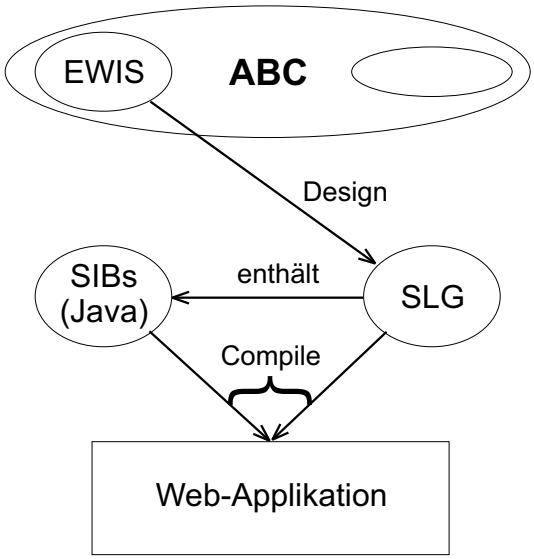


Abbildung 1.2.: Zusammenhang zwischen ABC und EWIS

Zum Testen von Applikationen wird das ITE verwendet. Das ITE ist eine Testumgebung zum Modellieren und Durchführen von Testfällen. Im Kontext vom ITE repräsentieren die SIBs der SLGs atomare Testblöcke, welche einfache Simulationen oder Verifikationen an dem zu testenden System vornehmen. SLGs werden im ITE Testmodelle genannt. Ein Testfall (z.B. Web-Applikation), der spezifisch zu testenden Merkmale der Applikation enthält, wird vom ITE im entsprechenden Testmodell dargestellt. Anschließend führt das ITE den Test mit Hilfe des Testmodells durch. Der Zusammenhang zwischen ABC und ITE ist in Abbildung 1.3 auf der nächsten Seite dargestellt. Um nicht zu jedem SLG manuell umfangreiche Testmodelle erstellen zu müssen, wurde von H. Raffelt [17] ein ABC Modul integriert, welches aus einem vorhandenen SLG ein Testmodell generieren kann.

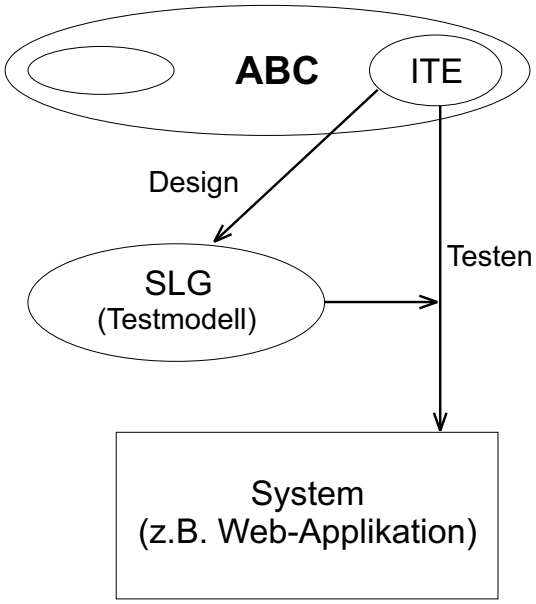


Abbildung 1.3.: Zusammenhang zwischen ABC und ITE

1.1.3. Enhanced Web Information Service

Beim EWIS handelt es sich um eine auf dem ABC basierende Entwicklungsumgebung zum Entwurf, Erstellung und Wartung von Internetapplikationen.

1.1.3.1. Grundlegende Funktionsweise

Die Entwicklung einer Web-Applikation mit Hilfe des EWIS wird in drei grobe Bereiche unterteilt: die Spezifikation des Workflows der Applikation, die Implementierung einzelner SIBs und die Gestaltung der Benutzeroberfläche (GUI). Da in jedem der drei Bereiche Kenntnisse nur in einem Wissensgebiet verlangt werden, ermöglicht dies die Spezialisierung der einzelnen Applikations-Entwickler .

Spezifikation Mit dem SLG wird der Kontrollfluss einer Applikation modelliert. Dieses Modellieren kann sehr einfach mit dem grafischen SLG Editor des ABC durchgeführt werden. Die eigentlichen Daten, die von der Applikation verarbeitet werden sollen, werden im SLG nicht modelliert. Details zu SLGs und ihren Bestandteilen werden im Abschnitt 1.1.3.2 auf der nächsten Seite beschrieben.

Implementierung Die einzelnen SIBs des SLG, welche die Funktionen im späteren Programm repräsentieren, werden implementiert. Sie bestehen dabei im Wesentlichen aus ihrer Schnittstelle und dem eigentlichen (Java-)Code. Hier werden Programmierer mit dem entsprechenden Know-How benötigt.

Benutzeroberfläche Es werden HTML-Templates erstellt, welche der späteren Interaktion mit dem Benutzer dienen sollen. Sie werden später mit dynamischen Inhalten aus den Interaktions-SIBs gefüllt und zu kompletten HTML-Seiten „aufgewertet“. Durch die Trennung von dynamischem Inhalt und statischem Design ist es möglich, komplexe und ansprechende Internetseiten zu erzeugen. Es lassen sich außerdem sehr einfach unterschiedliche Layouts für entsprechende Zielgruppen erstellen.

1.1.3.2. Service Logic Graph

Wie es schon im Abschnitt 1.1.2.5 auf Seite 5 erwähnt wurde, bestehen SLGs aus Kanten und SIBs. Es wird im Wesentlichen unter drei Kategorien von SIBs differenziert: Koordinations-SIBs, Interaktions-SIBs und Macros.

Koordinations-SIBs Koordinations-SIBs steuern den Kontrollfluss und können verändernd auf die Daten der Applikation zugreifen. Sie besitzen Eingabeparameter, welche von der jeweiligen Funktion abhängen, welche der SIB repräsentiert.

Interaktions-SIBs Interaktions-SIBs erzeugen HTML-Dokumente und dienen der Interaktion mit dem Benutzer. Sie besitzen einen Parameter, der das zur Erzeugung von HTML-Seiten benutzte Template festlegt.

Macros Es kommt oft vor, dass einige Teile eines SLG, unter Umständen mit unterschiedlichen SIB-Parametern, mehrmals verwendet werden. Solche SLG-Teile können in einem Macro eingekapselt werden, dann kann das Macro wie ein einfacher SIB mit entsprechenden Parametern in einem beliebigen SLG eingesetzt werden.

Andere SIBs Es gibt noch zwei besondere SIBs, die sich nicht in die bereits vorgestellten Kategorien einordnen lassen: den Jump-SIB und den ServiceEntry-SIB.

- Der Jump-SIB „teilt“ den SLG, indem er eine Weiterleitung vom URL so durchführt, dass der URL den auf den Jump-SIB folgenden Interaktions-SIB referenziert.
- Der ServiceEntry-SIB und seine Branches können auf einer HTML-Seite direkt referenziert werden. Mit Hilfe dieses SIB können Navigationsleisten erstellt werden, die beliebige SLG-Teile direkt referenzieren. Dabei muss der ServiceEntry-SIB vor dem SIB stehen, der ausgeführt werden soll.

Kanten Kanten haben unterschiedliche Funktionen. Sie leiten den Kontrollfluss von einem SIB zum nächsten oder repräsentieren Formulare und Verweise auf HTML-Seiten innerhalb der Applikation. In EWIS-Applikationen sind Verweise dabei entweder relativ zu dem aktuellen SIB oder sie verweisen auf einen entfernten SIB im SLG. Ausgehend von Koordinations-SIBs können nur Branches zur Steuerung des Kontrollflusses verwendet werden. Bei Interaktions-SIBs müssen die ausgehenden Kanten jedoch für die

Interaktionsfähigkeit der Kanten in einem SLG stehen und somit Formulare und Verweise auf HTML-Seiten modellieren. Neben Verweisen, die für Interaktionskanten stehen müssen, können in den modellierten HTML-Seiten noch weitere Verweise sein. Diese im SLG nicht modellierten Verweise haben im allgemeinen keine nennenswerte Funktion und werden häufig nicht modelliert, um die SLGs übersichtlicher zu gestalten. Somit kann eine Diskrepanz zwischen den in der realen Applikation vorhandenen und den im Graph modellierten Kanten auftreten.

1.1.4. Integrated Test Environment

Das ITE ist eine auf dem ABC basierende Testumgebung zum Modellieren und Durchführen von Testfällen. Da sich komplexe Systeme in unterschiedlichster Art und Weise untereinander beeinflussen, besteht für eine solche Testumgebung der Anspruch eines offenen und flexiblen Systems, um mit umfangreichen Testprozessen umgehen zu können. Dieses System schließt z.B. das Spezifizieren und Ausführen von Testfällen sowie eine Analyse der Testergebnisse ein.

1.1.4.1. Grundlegende Funktionsweise

Mit Hilfe eines Testkoordinators können im ITE verschiedene Testwerkzeuge ferngesteuert werden. Dies geschieht über eine CORBA/RMI-Schnittstelle, was zu einer Kompatibilität mit vielen am Markt verfügbaren Werkzeugen führt. Der Testprozess innerhalb des ITE besteht aus folgenden Schritten:

- Finden von generischen und wiederverwendbaren Testblöcken,
- Grafisches Design von Testfällen,
- Verifikation der Testfälle gegenüber vordefinierten Eigenschaften und
- Ausführen der Testfälle und Ausgabe eines Reports.

1.1.4.2. Finden geeigneter Testblöcke

Das Erste, was beim Testen eines komplexen Systems gemacht wird, ist das Auffinden von geeigneten Testblöcken. Für jede Aktion, die getestet werden soll, wird ein Testblock angelegt, welcher durch einen Namen und eine Klasse charakterisiert wird. Darüberhinaus bekommt jeder Block noch eine Menge von formalen Parametern, um eine spätere Wiederverwendbarkeit zu gewährleisten. Es wird somit inkrementell eine Bibliothek von Testblöcken angelegt, welche auch für spätere Tests benutzt werden kann, und welche darüberhinaus mit jeder neuen Spezifikation eines Testblocks dynamisch wächst. Eine solche Bibliothek enthält typischerweise verschiedene Testblöcke, welche zu diversen Klassen gruppiert werden können.

1.1.4.3. Grafisches Design von Testfällen

Das Design von Testfällen besteht aus der verhaltensorientierten Kombination von Testblöcken. Dieses Design wird im ITE grafisch vorgenommen. Dabei werden, wie schon beim SLG im EWIS, Testblöcke durch Symbole (vgl. SIB) zu einem Graphen verknüpft, welcher dann einen konkreten Testfall repräsentiert. Neben der Verwendung eines Knotens im Graphen pro Testblock besteht auch die Möglichkeit, Makros zu spezifizieren, die z.B. einen ganzen Initialisierungsprozess zu einem Knoten zusammenfassen. Diese Makros können dann als wiederverwendbare atomare Testausführungen eingesetzt werden. Zu beachten ist außerdem, dass am Anfang oder am Ende eines jeden Testfalles eine spezielle Reset-Operation stehen sollte, welche sicherstellt, dass die Testumgebung ordnungsgemäß zurückgesetzt wird.

1.1.4.4. Verifikation der Testfälle

Im ITE können die Testfälle während ihres Designs in Bezug auf ihren Kontrollfluss auf globale Korrektheit und Konsistenz geprüft werden. Wurde das zu testende System dabei mit dem EWIS entworfen, ergibt sich zudem der Vorteil, dass die dort bereits verwendeten Designspezifikationen wiederverwendet werden können, um eine Konsistenz mit dem Originalsystem zu garantieren. Diese Spezifikationen werden in einer benutzerfreundlichen Sprache, Extended Sematic Lineartime Temporal Logic (ESLTL), angegeben, die in einer Spezifikationsbibliothek abgelegt wird, auf welche zur Laufzeit von dem entsprechenden Model Checker zugegriffen werden kann. Wird während Verifikation eine Inkonsistenz entdeckt, wird eine textuelle Ausgabe des entsprechenden Regelverstoßes ausgegeben.

1.1.4.5. Ausführen und Analyse der Testfälle

Bei der Ausführung eines Testfalls, läuft der entsprechende Tracer des ITE den den Testfall repräsentierenden Graphen Testblock für Testblock durch. Die entsprechende Aktion eines Blocks wird dabei ausgeführt, evaluiert und führt entsprechend des Ergebnisses zum nächsten Testblock. Damit dieses Szenario diesbezüglich funktionieren kann, müssen die einzelnen Aktionen in Bezug auf das angesprochene externe Testwerkzeug implementiert werden. Dies wird in der Regel von erfahrenen Testern durchgeführt, die mit dem jeweiligen Werkzeug vertraut sind und die Instruktion spezifizieren, damit die entsprechende Aktionen ausgeführt werden können. Außerdem muss für jeden Block ein spezieller Tracer-Code in HLL implementiert werden, den der Tracer des ITE versteht und ausführen kann.

1.2. Ziele der Projektgruppe

1.2.1. Ziele

Im Rahmen der Projektgruppe soll eine ganzheitlich ausgerichtete, interaktive Entwicklungs- und Wartungsumgebung für Web-Applikationen erstellt werden, die es erlaubt,

Web-Services zu erstellen, zu validieren und dabei insbesondere zu animieren und anschließend zu testen. Hierbei steht immer ein abstraktes Modell der Web-Applikation im Hintergrund, welches aspektbezogen transformiert werden muss.

Web-Applikationen-Programmierung Es soll eine (visuelle) Programmierumgebung entstehen, die es erlaubt, personalisierte Web-Applikationen zu erstellen. Hier wird im Wesentlichen eine Erweiterung der bereits bestehenden EWIS-Lösung angestrebt.

Erstellung einer Animationsumgebung Es soll einfach möglich sein, eine Web-Applikation innerhalb der Entwicklungsumgebung zu animieren (Rapid Prototyping), um bestimmte Abläufe schon während der konzeptuellen Applikations-Modellierung animieren zu können. Hierdurch soll der Entwickler sich rein auf funktionale Aspekte, wie sie z.B. in UML durch Use Cases beschrieben werden, konzentrieren können, ohne von einer Laufzeitumgebung, wie z.B. einem (bestimmten) Webserver o.ä., abhängig zu sein.

Automatische Testerstellung Es wird eine Konkretisierung des Modells der Web-Applikation angestrebt, die es erlaubt, Testfälle für das ITE zu generieren, um sie dort ausführen zu können. Hier wird ggf. auch eine Integration zusätzlicher Testschnittstellen speziell für den Test von Web-Applikationen durchgeführt werden müssen.

Erstellung einer graphischen Benutzeroberfläche Es soll eine graphische Benutzeroberfläche entwickelt werden, um die einzelnen Phasen des Entwicklungsprozesses durch gezielte Animations-/Interaktionsmöglichkeiten zu unterstützen, und einem breiten Benutzerkreis zur Verfügung stellen zu können.

1.2.2. Konkretisierung der Ziele

Mit dem EWIS können Web-Applikationen erstellt werden, und diese dann mit Hilfe des ITE getestet werden. Die Applikationen werden als SLGs dargestellt. Um das Testen durchzuführen, wird ein Testmodell aus dem vorhandenen SLG generiert. Obwohl zur Zeit die Generierung mit dem von H. Raffelt [17] integrierten Modul realisiert werden kann, muss die Applikation auf dem abstrakten Niveau nach C++ nachprogrammiert werden. Die Möglichkeiten der Simulation und Animation der Web-Applikationen sind auch noch sehr begrenzt. Außerdem ist die Größe des zu Grunde liegenden Graphen ein großes Problem bei der interaktiven Entwicklung und Wartung von personalisierten Web-Applikationen mit Hilfe von SLGs. Um diese Probleme zu lösen, werden die folgenden zwei Ziele angestrebt:

1. Anlegen einer weiteren Spezifikationserweiterung des SLG, welche ein entsprechendes Simulationsmodul im ABC mit Informationen versorgt, damit nicht jeweils zum Testen und zum Simulieren Erweiterungen am SLG durchzuführen sind.

2. Erstellen von Views, um größere SLGs übersichtlicher machen zu können.

Die Abbildung 1.4 veranschaulicht die gesetzten Ziele.

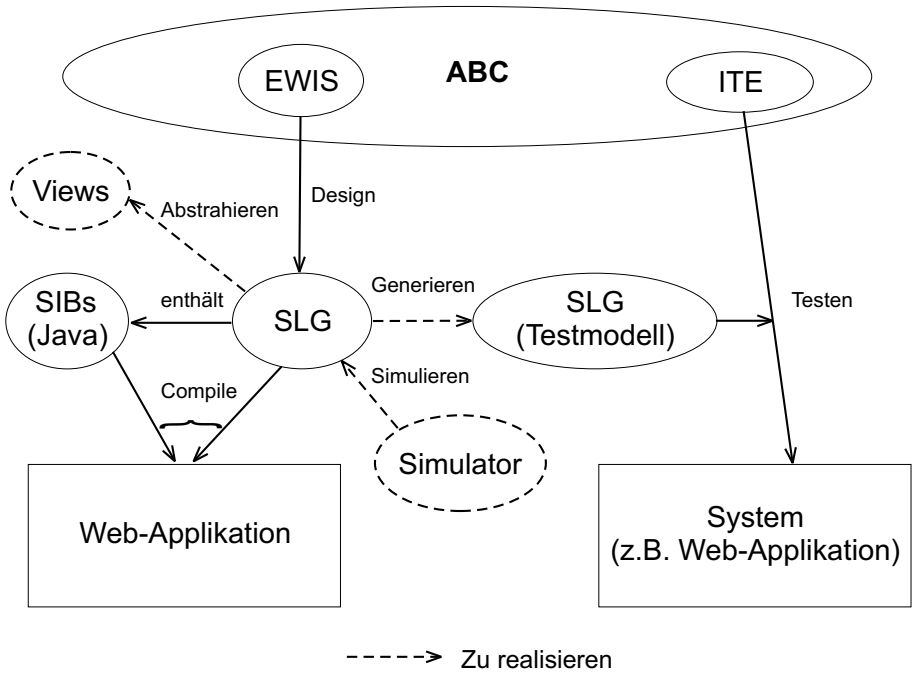


Abbildung 1.4.: Aufgabenbereich der Projektgruppe

In der Abbildung ist der Zusammenhang zwischen den schon in ABC bestehenden Komponenten und noch zu realisierenden Funktionalitäten dargestellt. Die noch konkret zu realisierenden Funktionalitäten (mit gestrichelten Linien dargestellt) sind:

- Generierung des Testmodells: Das automatische Generieren eines Testmodells aus einem SLG wird mit einem Modellgenerator realisiert.
- Simulation der Web-Applikation: Mit dem Simulator kann während der Entwicklung das finale Verhalten der Applikation zur Laufzeit schrittweise simuliert werden, ähnlich wie mit einem Debugger.
- Erstellen der Views: Benutzer oder Betrachter können sich eigene Sichtweisen auf die Graphen definieren, um die für sie irrelevanten Informationen auszublenden.

1.3. Aufgabenverteilung in den einzelnen Gruppen

Um die oben genannten Ziele zu erreichen, wurden die Aufgaben auf vier Gruppen verteilt:

1. Viewsgruppe

2. GUI-Gruppe
3. Testengruppe
4. Anwendungsgruppe.

Im Folgenden werden Aufgaben und Ergebnisse der einzelnen Gruppen kurz beschrieben.

1.3.1. Views

Ein Problem bei der interaktiven Entwicklung und Wartung von personalisierten Web-Applikationen mit Hilfe von SLGs liegt in der Größe der zu Grunde liegenden Graphen. Als Beispiel sei hier der Templus-Dienst genannt, der zur Zeit eine Größe von ca. 59.000 Knoten aufweist. Diesen Graphen vollständig zu überblicken und zu editieren, ist sehr schwierig, vor allen Dingen, wenn man den Graphen nicht selbst erstellt hat.

Genau bei dieser Problematik setzen die Views an. Views sind (geeignete) Abstraktionen von Graphen, die es erlauben, die Semantik des Graphen bzw. eines Teils des Graphen zu erkennen. Durch die Erzeugung von Views soll es leichter möglich sein, die Stellen des Graphen zu finden und zu editieren, die verändert werden sollen, außerdem wird dadurch das Verständnis der Applikation erleichtert.

Im Rahmen der Projektgruppe war es die Aufgabe der Views-Gruppe, einen Viewmanager zu entwickeln, der die verschiedensten Verfahren zur Generierung von Views (die teilweise schon implementiert wurden) anbietet, diese auf dem Bildschirm anzeigt, verwaltet und Änderungen in einem View an die anderen Views bzw. an den Ursprungsgraphen weiterleitet.

Mit dem entstandenen Viewmanager können Views auf PL-Graphen und SLGs generiert werden, wobei die Reihenfolge der Aufrufe von Algorithmen durch eine XML-Datei vorgegeben wird. Dabei können die folgenden Algorithmen für die Viewsgenerierung ausgewählt werden:

- SIBconvert - Konvertierung des SLG in für manche Algorithmen verständliche Form
- SIBdeconvert - dazu passender Rück-Konvertierungsalgorithmus
- Onlyedges - Konvertierung des SLG, indem alle Knoteninformationen in Kanten codiert werden, damit Algorithmen, die nur auf Kanten arbeiten, funktionieren
- Deonlyedges - dazu passender Rück-Konvertierungsalgorithmus
- Relabelling - Umbenennungsalgorithmus - arbeitet auf Kantennamen
- Tau-Elimination - Löschen der Kanten, die mit „tau“ bezeichnet sind
- Minimierung
- Weak Bisimulation
- Strong Bisimulation

- Basic Model Collapse

Die Algorithmen werden ausfürlicher im Kapitel 5 auf Seite 83 erläutert. Bei den Algorithmen Weak Bisimulation, Strong Bisimulation und Basic Model Collapse entstehen durch Verschmelzung mehrerer Knoten so genannte Meta-Knoten. Innerhalb des Meta-Knotens sind die verschmolzenen Knoten mit den dazu inzidenten Kanten und adjazenten Knoten in Form eines SLG gespeichert. Der Viewmanager macht diese Information verfügbar, indem der im Meta-Knoten hinterlegte SLG angezeigt wird. Zusätzlich ist der Viewmanager erweiterbar gehalten; es können weitere Algorithmen eingebunden werden.

1.3.2. Constraint Editor

Innerhalb der Projektgruppe war es die Aufgabe der GUI-Gruppe, für entstehende Lösungen grafische Benutzeroberflächen zu erstellen. Zusätzlich wurde ein Constraint Editor implementiert, der es dem Benutzer ermöglicht, auf Basis eines Pattern Systems auf bequeme Weise sogenannte Constraints zu generieren. Constraints sind prinzipiell nichts anderes als Eigenschaften eines endlichen Zustandsmodells. Im ABC ist ein solches Zustandsmodell der SLG, der als Knoten die SIBs hat. Hier finden Constraints zum Beispiel beim Model Checker eine Anwendung, wobei getestet wird, ob ein SLG gewisse Eigenschaften hat. Eine Motivation für den Editor war es innerhalb der PG, die Views mit Hilfe des Model Checkers zu generieren, wobei der Constraint Editor die nötigen Constraints geliefert hätte, um bestimmte Teile eines SLGs auszublenden. Da der Viewmanager jedoch nun anders vorgeht, wird der Constraint Editor nicht für die Erstellung von Views eingesetzt, sondern als eigenständiges Produkt der Projektgruppe vorgestellt.

Die Erstellung einer Constraint setzt Kenntnisse formaler Spezifikationssprachen wie z.B. LTL (Linear Time Logic) oder im Falle des ABC die erweiterte Variante ESLTL (Extended Semantic Linear Time Logic) [14] voraus. Um dem Benutzer das Erlernen solcher Formalismen zu ersparen, wird im Constraint Editor ein Pattern System verwendet, das einen weit intuitiveren Zugang zur Erstellung von Constraints bietet. Um größtmögliche Flexibilität zu erreichen, wird dieses Pattern System als XML-Datenbasis zur Verfügung gestellt.

Der Constraint Editor ist leicht erweiterbar und in vielen Bereichen einsetzbar. Implementiert wurde der Editor in fünf Schichten, wobei die konkrete Ausprägung der GUI austauschbar ist. Dies wurde mittels einer abstrakten GUI-Schicht erreicht, die eine uniforme Schnittstelle zur eigentlichen Logik des Editors darstellt, welche nochmal aufgeteilt ist in Controller-, Datenmodell- und eine globale Resources-Schicht. Damit ist die Modellierung des Editors von der konkreten Implementierung der grafischen Oberfläche komplett unabhängig, so dass er mit einer GUI in beliebiger Programmiersprache betrieben werden kann. Innerhalb der PG wurde die GUI mit Tcl/Tk implementiert. Da der Constraint Editor von konkreten Anwendungen möglichst losgelöst werden sollte, steht er als eigenständiges Modul im ABC zur Verfügung.

Der Vorteil eines solchen Editors ist, dass sich der Aufwand für die Erstellung einer Constraint auf die Angabe von Metadaten, die Wahl entsprechender Patterns sowie eine eventuelle Belegung einiger Parameter beschränkt. Das spart Zeit und minimiert die Anforderungen an den Kenntnisstand des Nutzers bzgl. formaler Spezifikationsformalismen.

1.3.3. Testen

Es wurde eine HLL-Spezifikation erarbeitet, welche es ermöglicht, Informationen zum Testen als auch zum Simulieren während der Entwicklungsphase einer Applikation bereitzustellen. Des Weiteren wurde ein Simulationsmodul entwickelt, welches auf den spezifizierten Daten arbeitet und das Laufzeitverhalten der zu entwickelnden Applikation im SLG nachbildet. Da ein HLL-Tracer für die einzelnen SIBs bereits in die METAFrame-Umgebung integriert war, wurde er um einige Funktionen zur komfortablen Simulation von Anwendungen erweitert und daraus ein eigenständiges Simulationsmodul erstellt. Das Modul unterstützt die Simulation von statischen HTML-Formularen, die entsprechenden Daten werden im EwisTest-Modul für jeden Interaktionsknoten spezifiziert. Des Weiteren ist die Undo-Funktion verfügbar, d.h. alle Simulationsschritte können rückgängig gemacht werden. Außerdem ist ein Zustandsraum-Debugger verfügbar, der eine Anzeige der Umgebungsvariablen der simulierten Web-Applikation ermöglicht und die Möglichkeit anbietet, diese manuell während der Simulation zu ändern.

Ein effizientes Testmodul ist ebenfalls bereits fester Bestandteil des ABC, wobei allerdings der Simulationscode in C++ erwartet wird. Um eine einheitliche Datenspezifikation zum Testen und Simulieren zu erhalten, das Testmodul aber unverändert zu belassen, war es demnach notwendig, den HLL-Code in vom Testmodul verwertbaren C++ Code zu transformieren. Zu diesem Zweck wurde ein HLL-Transformer erarbeitet, welcher Testskripte in HLL einliest, nach C++ transformiert und dem Testmodellgenerator übergibt.

1.3.4. Anwendung

Die Aufgabe der Anwendung-Gruppe war es, eine Internet-Applikation mit Hilfe des ABC zu erstellen, die im Rahmen der Projektgruppe zum Testen eingesetzt werden sollte. Es wurde entschieden, eine Applikation für die Verwaltung einer Literatur-Datenbank zu erstellen. Die entwickelte Applikation hat folgende Eigenschaften. Die bereits in der Datenbank vorhandenen Publikationen können angezeigt werden, wobei jede Publikation der Einfachheit halber nur mit zwei Attributen („title“ und „note“) versehen wurde. Neue Publikationen können erstellt und in der Datenbank angelegt werden, die bereits in der Datenbank vorhandenen Publikationen können modifiziert werden und Publikationen können aus der Datenbank entfernt werden.

Außerdem wurde eine um ein User- und Rollenmanagement erweiterte Applikation erstellt. Dem Benutzer wird eine oder mehrere Rollen zugewiesen. Die verfügbaren Rollen sind Student, Mitarbeiter und Administrator. Studenten können sich nur Informationen zu einer Publikation anzeigen lassen, Mitarbeiter können zusätzlich neue Publikationen ins System einfügen und Administratoren verwalten das ganze System und haben Zugriff auf alle Funktionalitäten. Des Weiteren wurden Dienste für die Verwaltung von Kategorien, Autoren, Editoren, Schlüsselwörten und Sprachen erstellt, weil es geplant war, die Teildienste mit der Verwaltung von Publikationen zu verknüpfen, und die Publikationen mit mehreren Attributen zu versehen, um sie kompatibel zu BibTeX-Einträgen zu machen.

2. Anwendung

2.1. Überblick

Alle von der Projektgruppe erstellten Module müssen getestet werden. Aus diesem Grund wurde eine Web-Applikation namens Literatur-Datenbank in zwei Versionen erstellt. Die erste Version von der Literatur-Datenbank, die kleine Literatur-Datenbank, ist quasi ein Prototyp mit einfacher Funktionalität (Kapitel 2.3 auf Seite 21). Der SLG zu der kleinen Literatur-Datenbank ist klein, übersichtlich und bequem zum Testen. Der große Nachteil der ersten Version ist, dass die Module damit nicht ausreichend getestet werden können. Die zweite Version, die große Literatur-Datenbank (Kapitel 2.4 auf Seite 29), ist dagegen sehr komplex. Der SLG dazu besteht aus 7300 SIBs, wenn alle Macros expandiert werden. Die große Literatur-Datenbank ermöglicht aber ein ausschöpfenderes Testen der Module, insbesondere des Viewmanagers.

Die allgemeine Vorgehensweise bei der Erstellung einer Web-Applikationen mit ABC wird im Kapitel 2.2 beschrieben.

2.2. Erstellung einer Applikation unter EWIS

Im Folgenden wird anhand eines Beispiels illustriert, wie eine Web-Applikation mit Java unter EWIS erstellt werden kann. Es wird eine Applikation mit dem Namen „Tutorial“ erstellt, die eine einfache Login-Seite anzeigt. Es werden folgende Programme verwendet, die zusammen mit dem EWIS die Entwicklungsumgebung bilden:

- Tomcat Web Server [3]
- Velocity Scripting Engine [4]
- PostgreSQL [6]
- Java Development Kit [5]

2.2.1. Verzeichnisstruktur

Der Source-Code einer EWIS-Applikation ist typischerweise in folgender Verzeichnisstruktur (Abbildung 2.1 auf der nächsten Seite) organisiert.

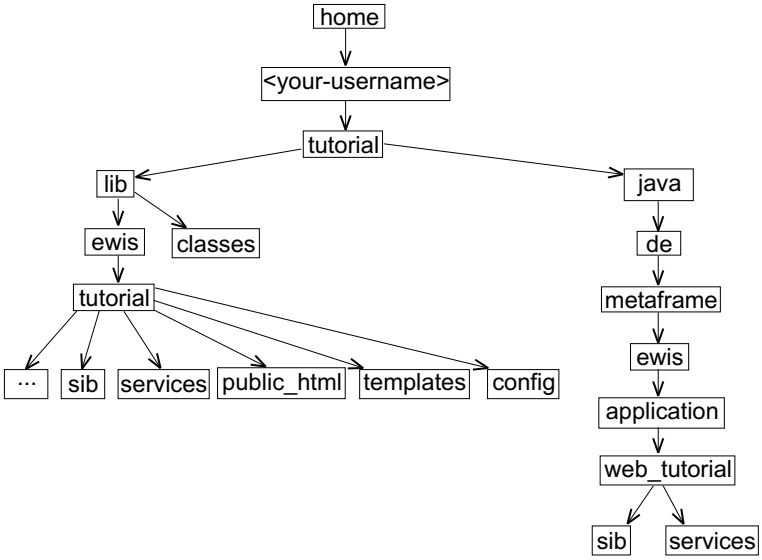


Abbildung 2.1.: Verzeichnisstruktur

Die Verzeichnisse „sib“ und „services“ enthalten jeweils einzelne SIB-Dateien und SLGs. Die statischen Seiten bzw. Templates zum Anzeigen im Browser werden in „public_html“ und „templates“ gespeichert. Die Konfigurationsdateien, z.B. „web.xml“, die die Umgebungsvariablen für die Applikation festlegen, liegen im „config“-Verzeichnis. Das Verzeichnis „java“ enthält Java-Quellen, die für die entsprechenden SIBs implementiert werden, oder vom Graph-Übersetzer generiert werden.

2.2.2. Web-Applikation

Nach dem Start des ABC wird im „SD Environment“ Fenster ein Umgebungstyp (in diesem Beispiel „in“) und ein Projekt (an dieser Stelle „tutorial“) ausgewählt. Mit dem „NewService“-Knopf kann dann ein neuer SLG erstellt werden. (Abbildung 2.2 auf der nächsten Seite)

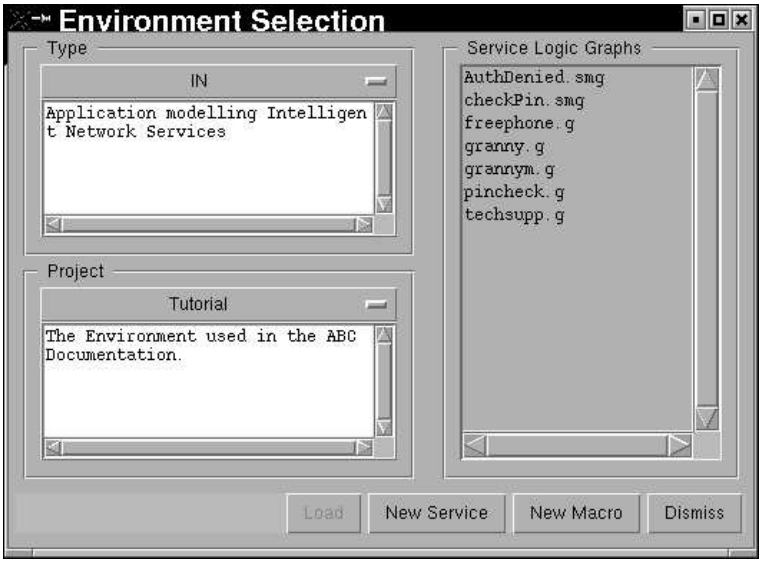


Abbildung 2.2.: SD Environment Fenster

Dann muss der Graph-Übersetzer konfiguriert werden. Im „SD Compilation Inspector Options“ Fenster wird der Paket-Name der Applikation festgelegt, die Bibliotheken, die bei der Erstellung der Applikation importiert werden müssen, wo die Java-Quellen und ihre kompilierten Klassen-Dateien gespeichert werden sollen, und schließlich das Verzeichnis, in das die benötigten Dateien für die Applikation kopiert werden. Da Tomcat als Web-Server verwendet wird, ist das Installationsverzeichis für die Applikation „tomcat-verzeichnis/webapps/Applikations-Name“. (Abbildung 2.3 auf der nächsten Seite)

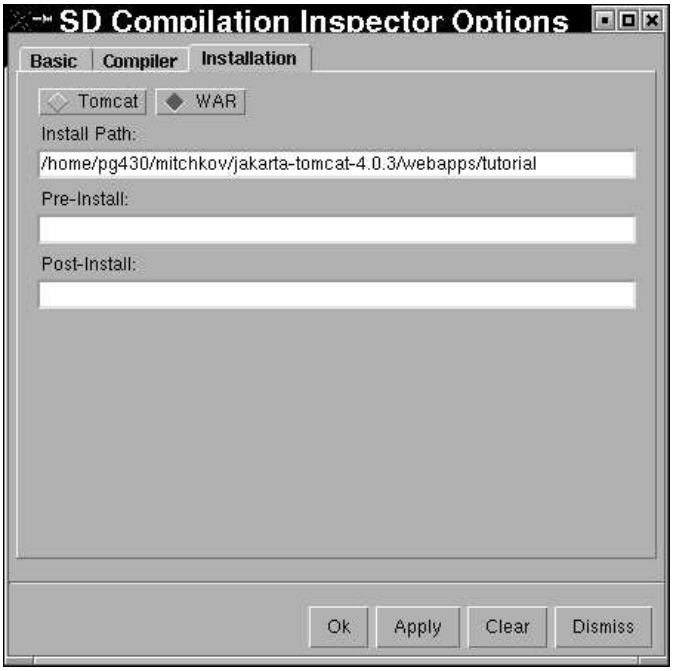


Abbildung 2.3.: Compiler Inspector Options Fenster

Außerdem muß die Datei „web.xml“ im „config“-Verzeichnis erstellt werden, damit die Applikation später auf dem Tomcat Web-Server initialisiert und gestartet werden kann. Ein Beispiel für die „web.xml“ ist im Programmtext 2.1 auf der nächsten Seite zu sehen. Zu beachten ist, dass der Parameter „ServiceName“ dem Namen des neuen SLG entsprechen muss.

```
<web-app>
    <servlet>
        <servlet-name>
            SimpleService
        </servlet-name>
        <servlet-class>
            de.metaframe.ewis.WebInfoService
        </servlet-class>

        <init-param>
            <param-name>ServiceName</param-name>
            <param-value>SimpleService</param-value>
        </init-param>

        <init-param>
            <param-name>Package</param-name>
            <param-value>de.metaframe.ewis.application.istest</param-value>
        </init-param>

    </servlet>

</web-app>
```

Programmtext 2.1: Tomcat-Konfiguration

Ein SLG für eine Applikation könnte wie in Abbildung 2.4 aussehen.

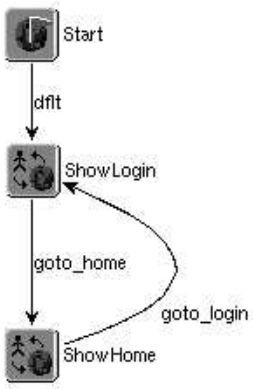


Abbildung 2.4.: Service Logic Graph

Die Interaktions-SIBs aus der „ShowFile“-Klasse (hier: „ShowLogin“ und „ShowHome“) haben den wichtigen Parameter „filename“, der die Template-Datei festlegt. Template-

-Dateien entsprechen HTML-Seiten, die der Benutzer im Browser sehen kann, nachdem sie vom Velocity Engine mit konkreten Inhalten gefüllt wurden.(Abbildung 2.5)

Abbildung 2.5.: ShowLogin SIB Parameter Fenster

Schließlich kann die Applikation installiert werden. In dieser Phase werden zuerst Java-Quellen kompiliert, danach werden alle für die Ausführung benötigten Dateien ins entsprechende Tomcat-Verzeichnis kopiert. Tomcat wird dann gestartet und die Applikation kann mit der Eingabe von der URL:
„http://localhost:8080/Applikations-Name/servlet/Service-Name“ gestartet werden. (Abbildung 2.6)

Abbildung 2.6.: Login-Seite

Genaueres zu der Erstellung von Web-Applikationen unter EWIS ist in [2] zu finden.

2.3. Kleine Literatur-Datenbank

Die kleine Literatur-Datenbank hat die folgende Funktionalität. Eine Publikation mit zwei Attributen „title“ und „note“ kann angelegt, aus einer Liste von Publikationen ausgewählt und angesehen, modifiziert oder gelöscht werden, dies passiert interaktiv in einem Browser. Die einzelnen Entwicklungsschritte sind in den nächsten Unterabschnitten beschrieben.

2.3.1. Datenbank und die Datenbankschnittstelle

In der Datenbank befindet sich eine Tabelle „genericentry„, die so angelegt wurde. (Programmtext 2.2)

```
1  CREATE TABLE genericentry (id INT4 PRIMARY KEY NOT NULL, title VARCHAR(200),
2  note VARCHAR(100));
3
4  INSERT INTO genericentry VALUES (1,
5  'Patterns in property specifications for finite-state verification',
6  'Software Engineering');
7
8  INSERT INTO genericentry VALUES (2,
9  'An integrated approach for testing complex systems',
10 'To be published');
11
12
```

Programmtext 2.2: Anlegen von GenericEntries

Die Tabelle sieht dann so aus:

id	title	note
1	Patterns in property specifications ...	Software Engineering
2	An integrated approach for testing complex systems	To be published

2.3.1.1. Datenbankschnittstelle

In der Datenbankschnittstelle sind für jede Tabelle in der Datenbank (in dem Fall für „genericentry“) drei Java-Klassen zu erstellen: „GenericEntry“, „JDBCGenericEntryAdministrator“, „SmartGenericEntryAdministrator“. (Abbildung 2.7 auf der nächsten Seite)

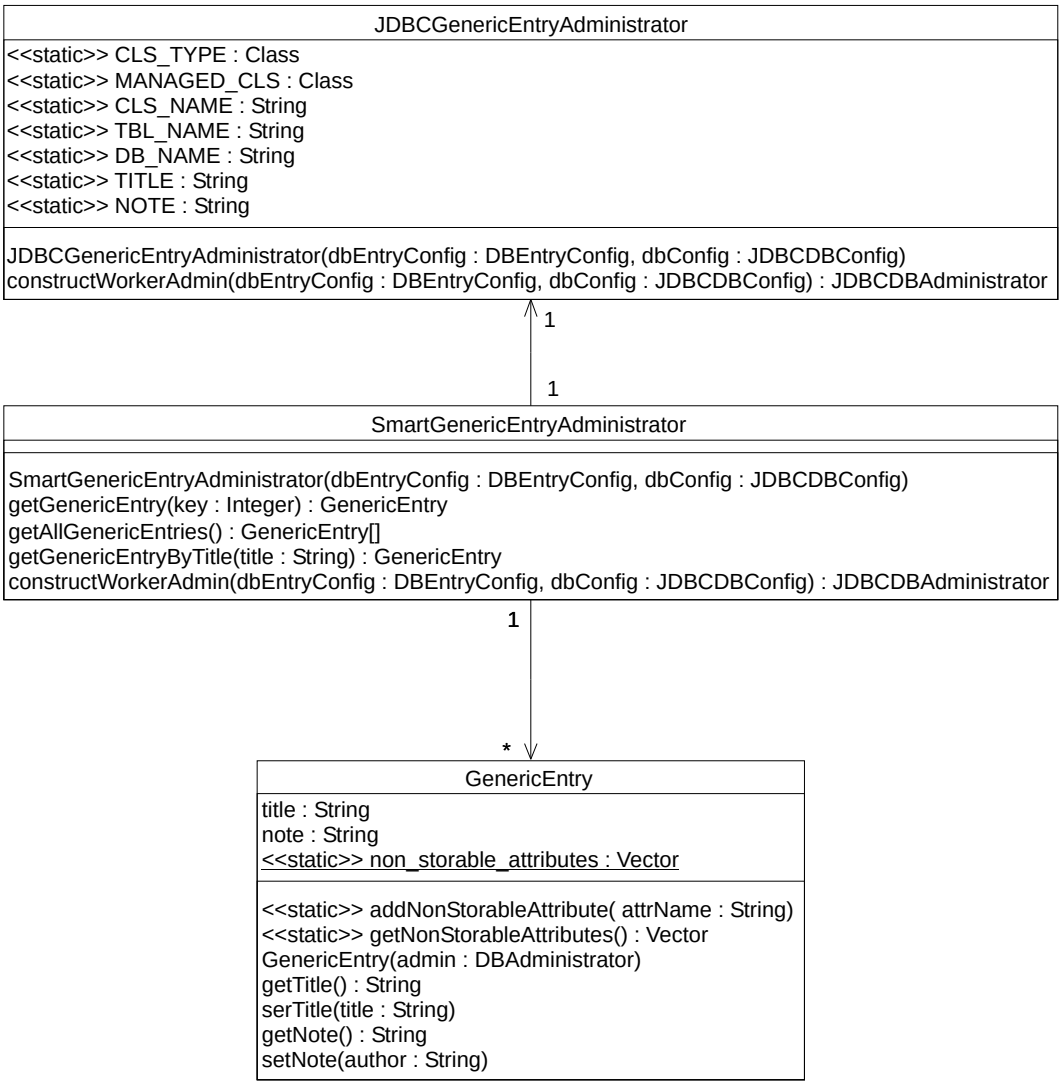


Abbildung 2.7.: Datenbankschnittstelle

2.3.1.2. GenericEntry

Die Klasse repräsentiert eine Publikation, deren Attribute „title“ und „note“ in der Datenbank gespeichert werden. Ein Objekt vom Typ „GenericEntry“ wird für jeden Datenbankeintrag vom „SmartGenericEntryAdministrator“ erzeugt. Wichtig ist die folgende statische Initialisierung: „static {addNonStorableAttribute (InDB); addNonStorableAttribute(Key); } “

Mit Hilfe der Initialisierung wird dafür gesorgt, dass die Spaltennamen in der Datenbank mit den entsprechenden Spaltennamen in der Schnittstelle übereinstimmen.

2.3.1.3. JDBCGenericEntryAdministrator

Diese Administrator-Klasse ist für den Zugriff auf die Datenbanktabelle verantwortlich, in der die Attribute des persistenten Objektes „GenericEntry“ gespeichert sind. Das Administratorobjekt agiert im Hintergrund und wird nie explizit in einer Applikation angesprochen, ist also in einer Applikation transparent. Es wird vom „SmartGenericEntryAdministrator“ erzeugt und jedes Mal angesprochen, wenn ein Zugriff auf die Datenbank erfolgen soll.

2.3.1.4. SmartGenericEntryAdministrator

„SmartGenericEntryAdministrator“ ist ein Application-Layer-Administrator, der persistente Objekte vom Typ „GenericEntry“ verwaltet. Es wird nur ein Objekt vom Typ „SmartAdministrator“ instantiiert, wobei gleichzeitig auch ein „JDBCGenericEntryAdministrator“ instantiiert wird. Der Administrator erzeugt Instanzen von der Klasse „GenericEntry“, vergibt den Instanzen die eindeutigen IDs, alle technischen Details werden vom „JDBCGenericEntryAdministrator“ verwaltet.

2.3.1.5. Datenbankzugriff

Der Zugriff auf die Datenbank läuft in der Literatur-Datenbank folgendermaßen ab. Der Benutzer startet eine Anfrage im Browser, die an einen Koordinations-SIB weitergeleitet wird, der die entsprechende Methode im „SmartGenericEntryAdministrator“ aufruft. Der „SmartGenericEntryAdministrator“ leitet die Anfrage an den „JDBCGenericEntryAdministrator“ weiter, der dann auf die Datenbank zugreift und entsprechende Objekte an den „SmartGenericEntryAdministrator“ zurückliefert, die abschließend an den anfragenden SIB weitergeleitet werden.

2.3.2. Service Logic Graph und SIBs

Ein Teil vom SLG für die Literatur-Datenbank ist in Abbildung 2.8 auf der nächsten Seite dargestellt.

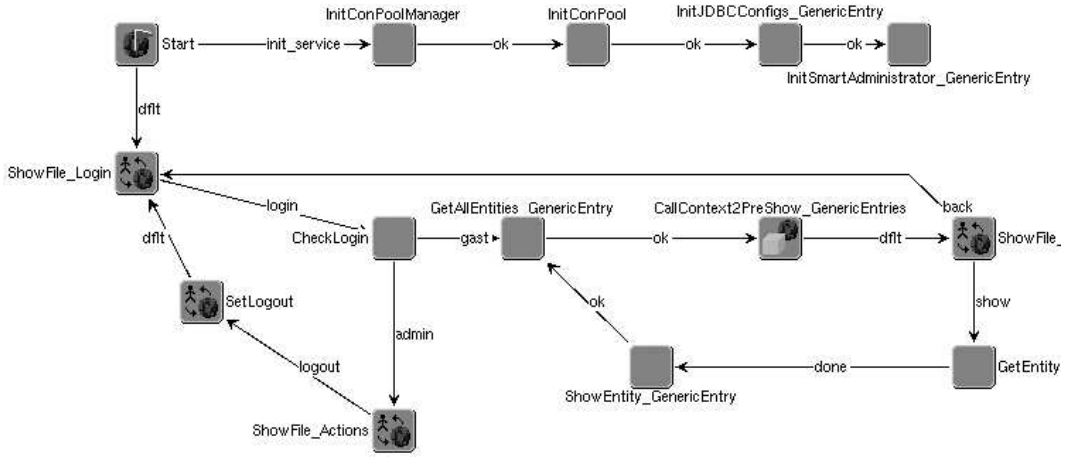


Abbildung 2.8.: Literatur-Datenbank-SLG

2.3.2.1. ModifyGenericEntry SIB

Fast alle SIBs, die im SLG sind, waren in EWIS schon vordefiniert. Es wurde nur ein neuer SIB „ModifyGenericEntry“ implementiert, weil kein passender SIB in EWIS vorhanden war. Der SIB überschreibt Attribute eines „GenericEntry“-Objektes. Zuerst musste der SIB in einer „.sib“-Datei definiert werden (Programmtext 2.3).

```
1 SIB     ModifyGenericEntry
2 CLS     Literatur
3 PAR     context SEL call_context session service container END 2
4 PAR     context_key STR 300 ""
5 BR      ok
```

Programmtext 2.3: Beispiel für eine sib.-Datei

Mit dem Eintrag „SIB ModifyGenericEntry“ wird der Name des SIBs definiert, der dann in EWIS-Umgebung zu sehen ist. Mit „CLS Literatur“ wird der Name der Klasse definiert, in der der SIB später zu finden ist. Dann werden 2 Parameter „context“ und „context_key“ mit „PAR“ definiert. „SEL“ bedeutet, dass für den Parameter „context“ einer von vier vordefinierten Werten ausgewählt werden kann. Und schließlich wird mit „BR ok“ eine Kante definiert. Außerdem musste eine Java-Datei implementiert werden, um die Funktionalität für den SIB festzulegen. Die Klasse implementiert das Interface „SIB“, das vorschreibt, dass eine Methode namens „exec“ implementiert werden soll (Programmtext 2.4 auf der nächsten Seite).

```
1  /**
2       * Exec method
3       *
4       * @param call_context the call context
5       * @return branch "ok" if the modification of the generic entry worked well
6       */
7      public String exec (CallContext call_context)
8          throws SIBExecException
9      {
10         String publication_title =
11             (String) call_context.getParameter ("publication_title");
12         String publication_note =
13             (String) call_context.getParameter ("publication_note");
14
15         GenericEntry newPublication =
16            (GenericEntry)SIBUtils.
17                getObjectFromContext(call_context, context_key, context);
18
19         if (newPublication == null) {
20             throw new SIBExecException
21                 ("Could not get the Entity: " +
22                  "context = \"" + context +
23                  "\", key = \"" + context_key + "\"");
24         }
25         newPublication.setTitle(publication_title);
26         newPublication.setNote(publication_note);
27         SIBUtils.
28             putObjectIntoContext(call_context, context_key,
29               context, newPublication);
30         return "ok";
31
32     }
```

Programmtext 2.4: Exec-Methode

Zuerst werden die Namen der Attribute „title“ und „note“ für die neue bzw. zu modifizierende Publikation aus dem „call_context“ gelesen. Zuvor wurden sie vom Benutzer im Browser eingegeben. Dann wird ein neues bzw. zu modifizierendes Objekt vom Typ „GenericEntry“ aus dem „call_context“ geholt, die Attribute „title“ und „note“ werden überschrieben, schließlich wird das Objekt zurück in den „call_context“ geschrieben.

2.3.2.2. Applikationsablauf

Nachdem ein Benutzer den Dienst im Browser gestartet hat, wird zuerst die Kante „init_service“ (Abbildung 2.8 auf der vorherigen Seite) bearbeitet, es wird also die Schnittstelle zur Datenbank initialisiert. Die SIBs „InitCoonPoolManager“ und „InitCoonPool“ initialisieren einen connections pool, über den dann mehrere Verbindungen zur Daten-

bank gleichzeitig erfolgen können. Mit Hilfe der SIBs „InitJDBCConfigs_GenericEntry“ und „InitSmartAdministrator_GenericEntry“ werden die Administratoren für die Tabelle „genericentry“ initialisiert.

Nach der Initialisierung kann sich der Benutzer im Login-Fenster entweder als Administrator oder Gast anmelden. Als Gast kann der Benutzer nur Publikationen ansehen. Als Administrator kann er eine der folgenden Aktionen auswählen: eine neue Publikation erstellen, Publikationen ansehen, bearbeiten oder löschen. Ein möglicher Interaktionsablauf ist in Abbildung 2.9 zu sehen.

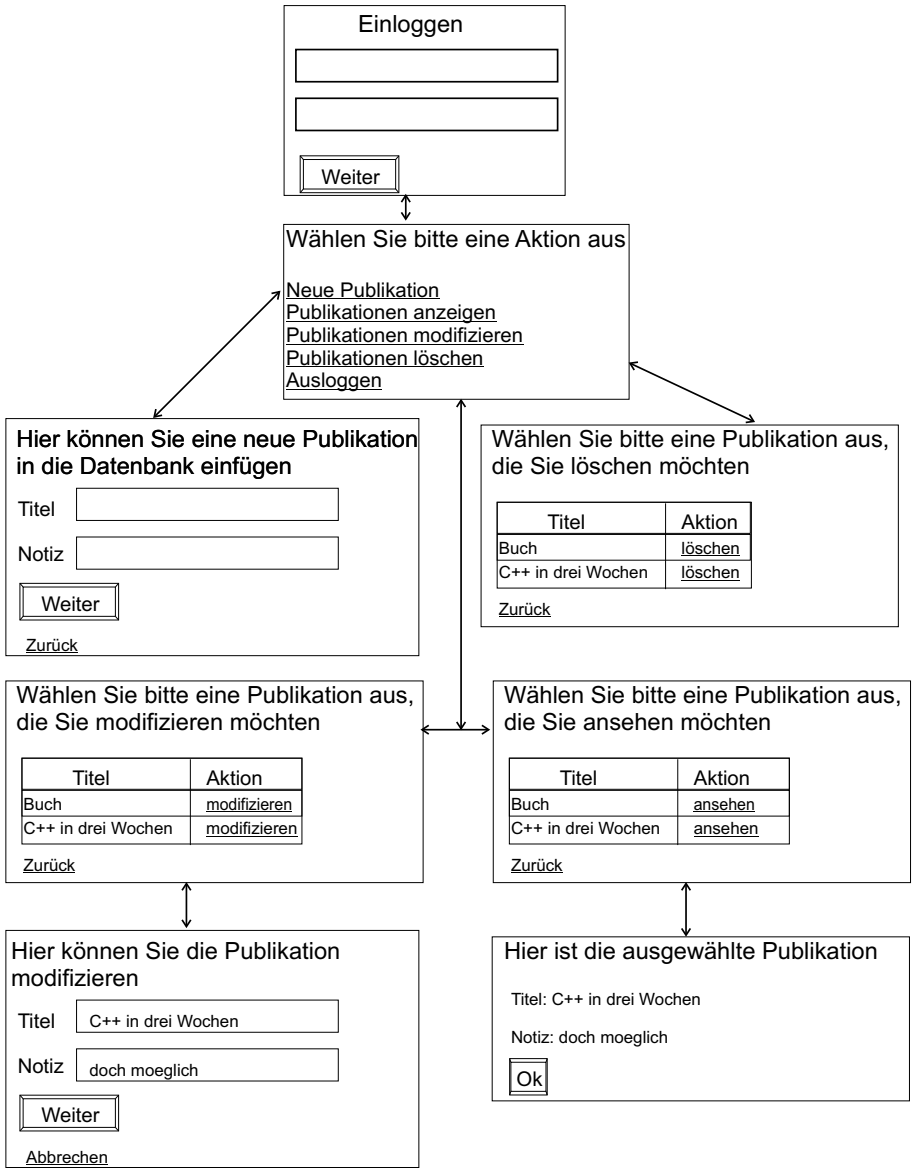


Abbildung 2.9.: GUI-LiteraturDatenbank

2.3.2.3. Template für die Anzeige aller Publikationen

```
<html>
<head>
<title>Show Publications</title>
<meta http-equiv="Content-Type" content="text/html; charset=iso-8859-1">

</head>

<body bgcolor="#ffffff">
<font face="Arial, Helvetica">

<h2>W&auml;hlen Sie bitte eine Publikation aus, die Sie ansehen
    m&ouml;chten</h2>
<p>
     <table border="4">
         <tr>
           <th>Titel</th>
           <th>Aktion</th>
         </tr>
     #if($!Publications)
     #foreach($pub in $!Publications)
     <tr>
       <td>$pub.Title</td>
           <td>
             <a href="$thisSIB.show.arg("PublicationID=$pub.ID")">ansehen</a>
     </tr>
     #end
     #end
   </table>
     <p>
     <a href="$thisSIB.back">Zur&uuml;ck</a>
</font>
</body>
</html>
```

Programmtext 2.5: Template

Mit Hilfe der Velocity-Engine [4] können dynamisch HTML-Seiten aus Templates erzeugt werden. Ein Template könnte wie Programmtext 2.5 aussehen.

An die Applikation wird ein Array „Publications“ mit Objekten vom Typ „GenericEntry“ übergeben, der Zugriff darauf erfolgt mit „$Publications“. Mit dem Konstrukt „#foreach($pub in $Publications)“ wird das Array durchlaufen. Mit „$pub.Title“ und „$pub.ID“ werden Attribute von einem „GenericEntry“-Objekt angesprochen. Die Anweisung „$thisSIB.show“ bewirkt, dass nach der Auswahl des entsprechenden Links derjenige SIB im Dienst ausgeführt wird, auf den die Kante „show“ zeigt, welche von dem gerade

ausgeführten SIB ausgeht.

2.4. Große Literatur-Datenbank

Aus dem TEMPLUS-Dienst, der erfolgreich beim Lehrstuhl 5 eingesetzt wird, wurde größtenteils das User- und Rollenmanagement übernommen. Es ist möglich, sich einzuloggen, nachdem ein Benutzer mit seinen persönlichen Daten manuell in der Datenbank angelegt wurde und ihm eine oder mehrere Rollen zugewiesen wurde. Dann darf der Benutzer mit der Rolle Mitarbeiter Publikationen anlegen, alle Publikationen ansehen oder modifizieren. Ein Benutzer mit der Rolle Student darf nur Publikationen ansehen. Ein Administrator hat Zugriff auf alle Funktionen des Systems, er kann also auch Publikationen löschen. Wenn ein Benutzer mehrere Rollen zugewiesen bekommt, kann er zwischen seinen Rollen umschalten. Es besteht auch die Möglichkeit, Rollen, Features und Featureklassen zu verwalten.

Außerdem wurde die kleine Literatur-Datenbank um mehrere Teildienste erweitert, solche wie Verwaltung der Kategorien, Autoren, Publisher, Editoren, Schlüsselwörtern, Sprachen. Ein Benutzer kann sie jeweils erzeugen, ansehen, modifizieren und löschen. Mit Hilfe der wurzelgerichteten Kategoriestruktur können auch Unterkategorien angelegt werden.

Bevor die Daten in der Literatur-Datenbank gespeichert werden, muss zunächst die Struktur der Datenbank festgelegt werden, anhand der dann die einzelnen Tabellen mit ihren Datenfeldern angelegt werden können. Diese Datenbankstruktur wird Datenmodell genannt. Das ist ein Modell, das die Daten/Tabellen in der Datenbank und ihre Beziehungen zueinander darstellt. Die grundsätzlichen Anforderungen sind:

- Die Tabelle in Bezug auf Publikationen soll an das BibTeX-Format angepasst werden.
- Der Benutzer verwaltet Favoritenlisten, die Verweise auf einzelne Publikationen beinhalten.
- Die Publikationen sind nach verschiedenen Themen klassifiziert. Somit sind Kategorie-Tabellen zu erstellen, die Publikationen mit demselben Thema beinhalten.

Insbesondere ist es wichtig, alle Daten, die in der Datenbank gespeichert werden sollen, nach ihren Beziehungen zu gruppieren. Die daraus entstandenen Tabellen werden noch nach ihrer Abhängigkeit miteinander verknüpft. Es gibt dafür verschiedene Entwurfsansätze. Für die „große Literatur-Datenbank“ wird ein objektorientiertes Design ausgewählt. Das sogenannte Objekt-Datenmodell beschreibt Daten als Objekte, die Instanzen von Klassen sind, die wiederum Eigenschaften und Methoden besitzen können. Die Klassen werden hier in Tabellen dargestellt und sind in eine Vererbungshierarchie eingeordnet.

Die zentrale Tabelle GenericEntry, die der Schablone für Publikationen entspricht, wurde an das BibTeX-Format angepasst, und zwar so, dass die Tabelleneinträge den

BibTeX-Attributen entsprechen. Sie steht mit fünf Haupttabellen über deren Assoziationstabellen in Verbindung. Sie sind jeweils Tabellen für Autor, Keywords, Publisher, Editor und Language. (siehe Abbildung 2.10)

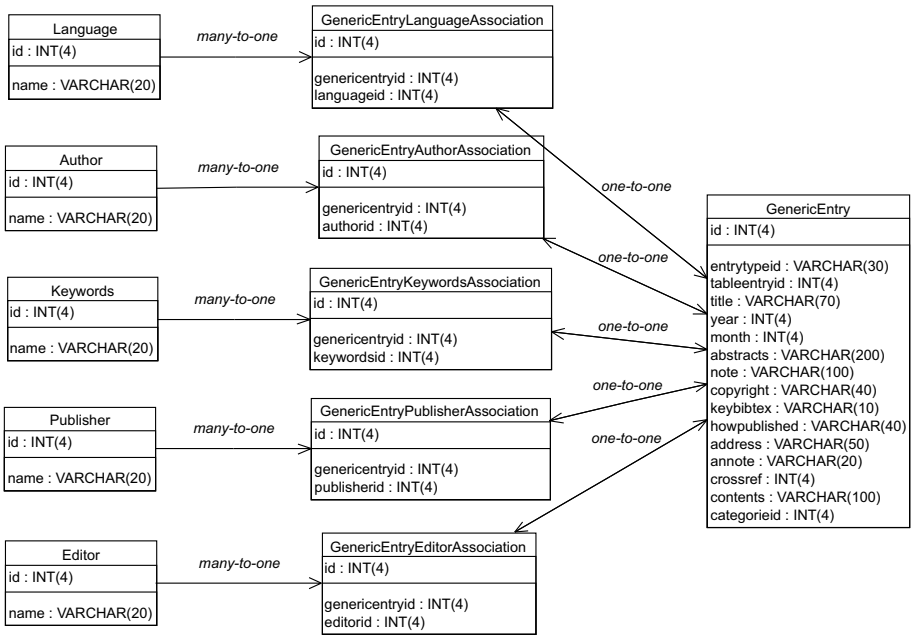


Abbildung 2.10.: Haupttabellen für die „große Literatur-Datenbank“

Die Tabellen InstituteEntry und MiscEntry erweitern GenericEntry und besitzen jeweils zusätzliche Attribute, wobei die Tabelle MiscEntry diejenigen Attribute enthält, die nicht zur Tabelle InstituteEntry gehören und selten gebraucht werden. (siehe Abbildung 2.11)

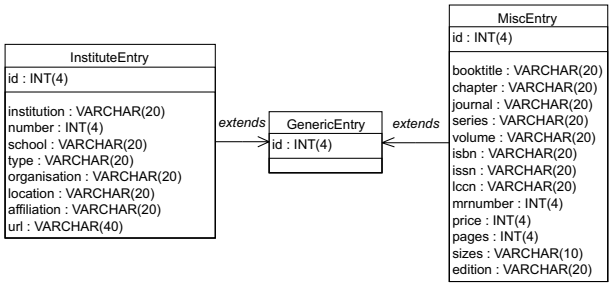


Abbildung 2.11.: Tabellen InstituteEntry und MiscEntry

Die Standard-Publikationstypen von BibTeX sind article, book, booklet, conference, inbook, incollection, inproceedings, manual, masterthesis, misc, phdthesis, proceedings,

techreport und unpublished. Die meisten Stardard-Typen werden durch die Tabelle GenericEntry erzeugt. Für masterthesis, phdthesis, techreport und conference ist die erweiterte Tabelle InstituteEnry eher geeignet, da sie zusätzliche Attribute für solche Publikationen besitzen. Das Attribut school gibt z.B. an, wo ein technischer Report geschrieben wurde.

Für die Tabelle User wird das User- und Rollenmanagement eingesetzt. Dadurch wird es ermöglicht, Benutzern verschiedene Rollen zuzuordnen und somit ihnen unterschiedliche Zugriffsrechte vergeben zu können. Es ist an dieser Stelle geplant, die Tabellen aus Templus zu übernehmen, die das User- und Rollenmanagement realisieren.

Die Tabelle Categorie steht mit der Tabelle GenericEntry in einer one-to-many-Beziehung, d.h., dass eine Kategorie mehrere Publikationseinträge mit dem gleichen Thema enthalten könnte. Sie hat noch die zusätzlichen Attribute hasparent (BOOL) und parentdirid (INT4), die eine Kategoriehierarchie ermöglichen.

Die Tabelle Favorites beinhaltet personalisierte Listen, die Verweise auf Publikationen (genericentryid) enthalten. (siehe Abbildung 2.12)

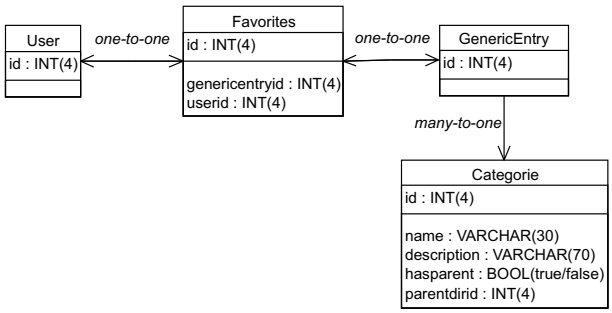


Abbildung 2.12.: Tabellen Categorie und Favorites

Nach dem objektorientierten Konzept liegt nun ein Datenmodell mit einer klaren Struktur vor. Es lässt somit leicht modifizieren und erweitern.

2.5. Fazit

Die große Literatur-Datenbank wurde leider nicht vollständig fertiggestellt, wobei sie trotzdem zum Testen geeignet ist. Die Datenbank kann initialisiert werden, und die Datenbankschnittstelle mit allen dazugehörigen Administratoren wurde vollständig implementiert. Außerdem können Publikationen, Kategorien, Autoren, Publisher, Editoren, Schlüsselwörter und Sprachen verwaltet werden. Bei der Publikationenverwaltung fehlt aber das Zusammenspiel aller Teildienste, das Erstellung von Publikationen mit allen modellierten Eigenschaften ermöglichen würde.

3. Testen

3.1. Überblick

Im ABC werden Web-Applikationen durch SDGraphen modelliert. Zu diesem Zweck liegt dem Entwickler mit dem EWIS eine entsprechende Programmierumgebung vor. Um die Modellierung einer Applikation zu testen oder zu veranschaulichen, existieren verschiedene Ansätze, von denen im Folgende zwei verfolgt werden.

Die Modellierung einer Web-Applikation wird durch den Graphen selber und das Verhalten seiner Knoten festgelegt. Dieses Verhalten wird mit Hilfe der HLL spezifiziert.

Da mit dem ITE bereits eine integrierte Testumgebung für das ABC existiert, liegt es nahe, sich diese zu Nutze zu machen. Als Eingabe werden Testmodelle benutzt, welche durch einen ebenfalls existierenden Testmodellgenerator erzeugt werden [17]. Die Spezifikation einzelner Tests wird durch C++-Skripte vorgenommen.

Da das Verhalten einer Applikation bereits durch die HLL spezifiziert ist und man redundante Informationen vermeiden möchte, besteht die erste Überlegung darin, eine Übersetzung des HLL-Codes in C++ vorzunehmen, um dann mit dem generierten Code als Eingabe automatisierte Tests im ITE durchzuführen. Diese Umwandlung soll vom Transformer übernommen werden. Der zweite Ansatz ist ein Simulator, welcher interaktiv während der Entwicklung des Graphen genutzt werden kann, um das Verhalten der Applikation zu simulieren und so zu verdeutlichen. Dieser Simulator arbeitet direkt mit dem HLL-Code unter Zuhilfenahme des im ABC integrierten HLL-Interpreters.

Beide Ansätze haben gemein, dass als Eingabe jeweils der HLL-Code zu Grunde gelegt wird. Dies ermöglicht ein automatisiertes Testen als auch eine Simulation der Anwendung mit nur einer Spezifikation.

Die Werkzeuge Transformer und Simulator werden im Folgenden beschrieben.

3.2. Transformer

3.2.1. Anforderungen

Der Ansatz, einen Transformer zur Umwandlung des HLL-Codes einzusetzen, entstand aus der Grundüberlegung, dass bereits eine Testumgebung existiert, welche als Eingabe Testskripte in Form von C++-Code erwartet [17]. Überdies muss zum automatisierten Testen die Menge von Zuständen einer Applikation geeignet definiert werden. Dies geschieht durch Angabe des sogenannten abstrakten Zustandsraums. Aus den Testskripten und dem Zustandsraum wird letztendlich ein Testmodell für das ITE generiert. Um dem Anwender eine redundante Implementierung eines Simulations- und Testcodes zu ersparen, liegt die Kernanforderung des als Brücke zwischen dem Simulator und der Testumge-

bung fungierenden Transformers darin, aus dem zur Simulation verwendeten HLL-Code entsprechende Codeblöcke für den Zustandsraum und die entsprechenden Testskripte zu generieren. Der Transformer läßt sich somit in zwei Teile gliedern: Einen zum Parsen des Zustandsraums und einen zum Transformieren der Testskripte. In dem von H. Raffelt entwickelten Testmodellgenerator wird der Zustandsraum durch eine Pascal-ähnliche Syntax angegeben. Dort existiert somit bereits ein Parser, welcher den Code dieser Spezifikationssprache übersetzt. Ziel ist es nun, diesen Parser so zu modifizieren, dass als Eingabe HLL-Code akzeptiert wird. Analog zur Existenz eines Zustandsraumparsers existiert im ABC der HLL-Interpreter [10]. Dieser bildet das Kernstück der Applikations-Suite und ist in der Lage, HLL-Code zeilenweise auszuführen. Es liegt somit nahe, sich als endgültiges Ziel den Interpreter zum Parsen des HLL-Codes nach C++ zu Nutze zu machen, da dieser bereits ausgereifte Funktionalitäten zum Umsetzen von HLL-Code beinhaltet.

3.2.2. Prototyp

Für das Parsen von Texten existieren bereits effiziente Werkzeuge auf der Basis von C/C++. Hier sind in diesem Zusammenhang Flex zur Konstrukterkennung (Scanner) und Bison zur Konstruktumwandlung (Parser) von besonderer Relevanz [9]. Da letztendlich die Implementierung jedoch ausnahmslos in C++ durchgeführt wird, greift man auf entsprechende objektorientierte Versionen (Flex++ und Bison++) zurück. Im Folgenden werden diese jedoch nicht weiter unterschieden und die Begriffe Flex und Bison statt Flex++ und Bison++ benutzt. Um die endgültige Funktionsweise des HLL-Transformers genauer zu umreissen, entsteht im ersten Schritt ein Prototyp auf der Basis von C++ und Flex [1]. Da es in diesem Entwicklungsstadium zunächst darum geht, triviale HLL-Konstrukte zu parsen und die Integration in die ABC-Umgebung zu testen, wird auf eine komplexe Grammatik zur Transformation mit Bison verzichtet. Ziel ist es triviale Datentypen und Ausgabe-Anweisungen von HLL nach C++ zu übersetzen. Hierzu ist es lediglich nötig, eine triviale Syntaxersetzung in Form eines Flex-Scanners zu implementieren. Einfache HLL-Konstrukte werden erkannt und direkt durch die entsprechenden C++-Pendants ersetzt. Da keine Grammatik mit Regeln zur Konstruktbildung eingesetzt wird, ist dieser Ansatz wenig flexibel und es muss penibel auf die Einhaltung von Konventionen bei der Eingabe von HLL-Code geachtet werden.

Die Integration in das ABC bzw. die Testumgebung gestaltet sich derart, dass zwischen das entsprechende Eingabefeld der GUI und der weiteren Verarbeitung durch den Testmodellgenerator der Transformer geschaltet wird. Es ist somit möglich, die (zunächst trivialen) Testskripte wie bisher in einer dafür vorgesehenen Eingabemaske zu spezifizieren, wobei allerdings nun HLL-Code statt C++ eingesetzt werden kann. Der abstrakte Zustandsraum wird in der Phase der Prototypisierung noch nicht berücksichtigt.

3.2.3. Entwurf

Anhand des Prototypen kann man deutlich sehen, dass man mit dem Ansatz der reinen Syntaxersetzung für spezielle Sprachkonstrukte relativ schnell an die Grenzen der Möglichkeiten eines solchen Transformers stößt. Lassen sich Deklarationen und Zuweisungen

von trivialen HLL-Datentypen und Ausgabeanweisungen noch recht einfach durch ihre entsprechenden Gegenstücke in C++ ersetzen, so treten spätestens bei komplexen HLL-Funktionsaufrufen und Schleifen Schwierigkeiten auf, die sich durch einfaches Austauschen der Sprachkonstrukte nicht mehr bewältigen lassen. Aus diesem Grund ist es nötig, den bisher implementierten prototypischen Flex-Scanner durch einen Bison-Parser zu erweitern. Dieser kann dann aufgrund von festgelegten Grammatikregeln erkannte Konstrukte entsprechend umwandeln.

Im ABC existieren bereits zwei solche Funktionale aus Flex-Scanner und Bison-Parser. Der HLL-Interpreter zur Interpretation der HLL-Kommandos im Tracer bzw. der Shell und der Parser für den abstrakten Zustandsraum im Testmodellgenerator. Da der Parser für den abstrakten Zustandsraum bereits eine Pascal-ähnliche Syntax zur Spezifikation der Zustandsvariablen anbietet, besteht hier das Ziel im Austausch dieser Sprache durch die HLL. Die Anpassungen beschränken sich also auf die Seite der Eingabe, die der Parser akzeptieren muss. Der Interpreter hingegen kann bereits mit der Eingabe von HLL-Code umgehen, ist aber lediglich in der Lage, diesen zu interpretieren. Er liefert somit keinen Ausgabecode und das Ziel der Projektarbeit besteht darin, ihn entsprechend zu erweitern, so dass C++-Code ausgegeben werden kann.

Abstakter Zustandsraum Für den Testmodellgenerator, der die Aufgabe hat, das Laufzeitverhalten der Applikation auf abstraktem Niveau nachzuahmen, oder den Simulator, der die gesamte Applikation simulieren soll, wird ein abstrakter Zustandsraum benötigt. Die wesentliche Abstraktion liegt darin, die in den meisten Graphen unendlich große Mengen von Zuständen, in denen sich die Applikation befinden kann, auf eine endliche und möglichst kleine Menge von abstrakten Zustände abzubilden. Diese abstrakten Zustände sollten sowohl vom Testmodellgenerator verarbeitet und zu einem abstrakten Zustandsmodell zusammengefasst, als auch vom Simulator für den Zugriff auf global definierte und benutzte Variablen für die Simulation der Applikation verwendet werden.

Abstakter Zustandsraum und Simulator Als ursprüngliche Interaktions-Schnittstelle zwischen dem Simulator und dem Testmodellgenerator wurde eine HLL-Spezifikation erarbeitet, welche es ermöglicht, Informationen zum Testen als auch zum Simulieren während der Entwicklungsphase einer Applikation bereitzustellen. Als Brücke zwischen beiden sollte ein Parser implementiert werden, der Namen und Typen von globalen Variablen an den Simulator weitergeben sollte. Eine Beispiel der Schnittstelle für die Variablen des abstrakten Zustandsraumes ist im folgenden dargestellt:

- var Bool:b := false;
- var Int:i := 0;
- var Real:r := 15.6;
- var Double:d := 12.3;
- var String:s := “String“;

- var String List:sl := [];
- var String List List:sll := [[“Test1“],[“Test2“]];

Bei der Spezifikation der Variablen im abstrakten Zustandsraum können Variablen nicht initialisiert werden, was zur Folge hat, dass für die Weitergabe der Variablen an den Testmodellgenerator der Parser für den abstrakten Zustandsraum nicht benutzt werden kann.

Parser des abstrakten Zustandsraums Damit sie schnelle verarbeitet und einfach implementiert werden können, müssen die abstrakten Zustände des Testmodellgenerators eine feste Struktur haben, die durch einen Verbundtypen festgelegt wird. Dieser Verbundtyp wird per HLL definiert und spannt den abstrakten Zustandsraum auf. In der alten Form der EWISTest-Module erfolgte die Definition des abstrakten Zustandsraums in einer an Pascal angelehnten Syntax. Eine spezielle C++-Klasse, deren Instanzen abstrakte Zustände repräsentiert, wird dann aus dieser Definition erzeugt. Die Spezifikation der abstrakten Zustände des Moduls ewistestpg430 läßt sich aus folgenden Basistypen aufbauen:

- Einfache Typen
  - Wahrheitswerte (Boolean) — var Bool:b;
  - Ganzzahlen (Integer) — var Int:i;
  - Gleitkommazahlen (Double) — var Double:d;
  - Gleitkommazahlen (Real) — var Real:r;
  - Zeichenketten (String) — var String:s;
- Zusammengesetzte Typen — var (Int * String * Bool List * Int Set * Bool):il;
- Parametrisierte Container Typen
  - Set — var Int Set:is;
  - Liste — var Bool List:bl;
- Kommentare — (* Das ist eine Kommentar *)

Die Syntax der Definitionssprache des abstrakten Zustandsraums ist in der Tabelle 3.1 angegeben. Die Beschreibung ist formal an die Backus Naur Form (BNF) angelehnt. Im folgenden wird kurz beschrieben, wie die Definition des abstrakten Zustandsraums aufgebaut ist. Um den Zusammenhang zur Syntax der Sprache darzustellen, sind zum Teil die zugehörigen Nonterminale in Klammern aufgeführt. Die Definition des abstrakten Zustandsraums besteht aus einer Liste von Typ-Vereinbarungen (TypBindingList), gefolgt von der Definition des Mainrecords. Die Liste der Typ-Vereinbarungen besteht aus keiner, einer oder mehreren Typ-Vereinbarungen (TypeBinding), die jeweils einen Bezeichner einem Typ zuordnen. Typen die zugeordnet werden können sind alle einfachen und zusammengesetzten Typen, sowie alle zuvor durch eine Typ-Vereinbarung deklarierte Typen.

Nonterminal		Ableitungsregel
start	::=	TypeBindingList BigComment MainRecList
TypeBindingList	::=	TypeBindingList TypeBinding | TypeBindingList
TypeBinding	::=	BigComment *'type'* Identifier *'='* Typedef ';'
Typedef	::=	RecordType | ParamType | DefinedType
RecordType	::=	*'('* RecordList *')'*
MainRecList	::=	*'var'* MainRecordBinding | MainRecList *'var'* MainRecordBinding
MainRecordBinding	::=	BigComment Typedef *':'* Identifier *';'*
RecordList	::=	RecordBinding | RecordList *'*'* RecordBinding
RecordBinding	::=	BigComment Typedef
ParamType	::=	SingleParamType
SingleParamType	::=	Typedef *'Set'* | Typedef *'List'*
DefinedType	::=	*'Bool'* | *'Int'* | *'String'* | *'Real'* | Identifier
Identifier	::=	Ein Identifier- Nonterminal besteht aus mindestens einem Buchstaben (A-Z oder a-z) gefolgt von Klein- und Großbuchstaben oder Ziffern (0-9)
BigComment	::=	Das Nonterminal BigComment stellt eine Zeichenkette dar, die mit (* beginnt, dann kein oder mehrere beliebige Zeichen einschließlich Zeilenumbrüchen und Sonderzeichen enthält und mit *) endet

Tabelle 3.1.: Syntax Definitionssprache des abstrakten Zustandsraumes

Eingabe des abstrakten Zustandsraumes Auf der *Abstract State Space Definition* Seite des Graph- Parameter Dialoges (siehe Abbildung 3.1 auf der nächsten Seite) kann die Definition des abstrakten Zustandsraums eingegeben werden. Mit Hilfe der Schaltfläche *Check* kann überprüft werden, ob die Definition korrekt ist. Syntaktische Fehler werden dabei im Definitionscode rot hervorgehoben und in speziellen Fehlerdialogen angezeigt. Ebenso wird überprüft ob die Namen der Attribute innerhalb eines Verbundes paarweise verschieden sind. Die Schaltfläche *Pretty Print* formatiert die Definition durch Einfügen und Entfernen von Leerzeichen und Zeilenumbrüchen, um eine bessere Lesbarkeit zu erzielen. Die Schaltfläche *Apply* bestätigt die Eingabe, die sich gerade im *Abstract State Space Definition* -Feld befindet.

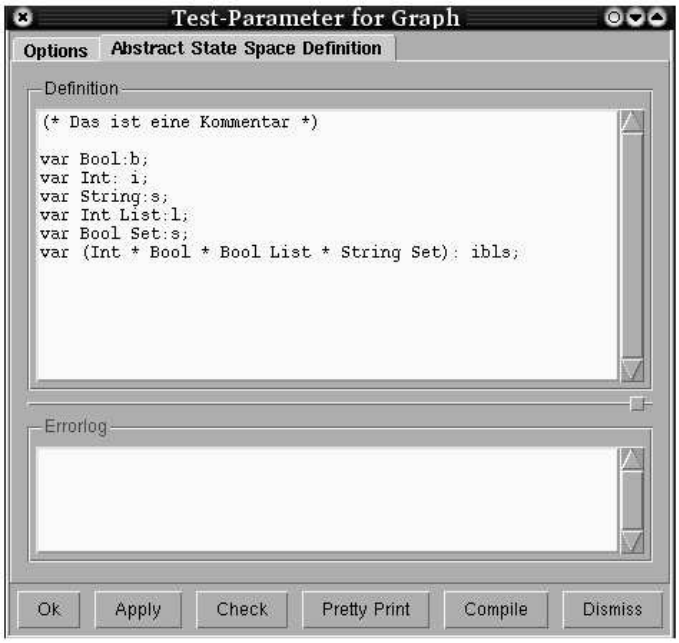


Abbildung 3.1.: Definition des abstrakten Zustandsraums

HLL-Code-Transformer Damit die HLL-Spezifikation, welche zum Simulieren einer Web-Applikation genutzt wird auch zur Definition der Testfälle herangezogen werden kann, muss sie in entsprechende C++-Skripte umgewandelt werden. Dieses soll mit Hilfe des bereits im ABC integrierten HLL-Interpreters umgesetzt werden.

Da kaum Dokumentation zu dem sehr komplexen HLL-Interpreter existiert und eine parametrisierte Erweiterung zum Ausgeben von C++-Code tiefergehende Kenntnis des Codes und der Funktionsweise erfordert wird zunächst eine redundante Version des Interpreters verwendet, welche statt einer Interpretation des HLL-Codes eine Ausgabe in C++ liefert.

Abbildung 3.2 auf der nächsten Seite zeigt die Funktion des HLL-Interpreters in seiner ursprünglichen Form.

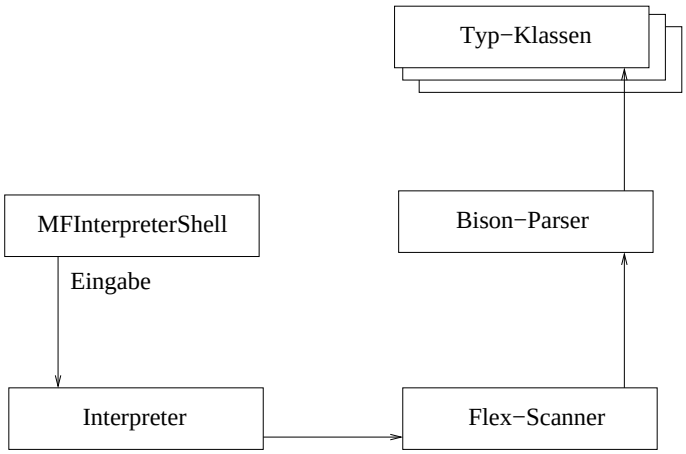


Abbildung 3.2.: Funktion des Interpreters

Von der beim Start des ABC sichtbaren Interpreter-Shell wird eingegebener HLL-Code an eine Interpreterinstanz weitergeleitet. Diese übergibt den kompletten Eingabecode initial an einen Flex-Scanner, welcher erkannte Konstrukte auf Anfrage Stück für Stück an einen Bison-Parser übermittelt. Da der Typ des Konstruktes anhand von grammatikalischen Regeln an dieser Stelle bekannt ist, baut der Parser nun einen Syntaxbaum aus ihnen auf und führt nach dem Ende einer Codezeile die entsprechende Code-Interpretation aus. Dazu bedient er sich einer Reihe von Konstrukt-Typ-Klassen, welche alle eine gemeinsame Menge von vordefinierten Funktionen implementieren. Hierzu zählen insbesondere eine „execute“- und eine „testPrintTypes“-Methode. Zur Interpretation von Code-Konstrukten werden die entsprechenden execute-Methoden aufgerufen, die je nach Typ die gewünschte Funktionalität ausführen und somit verändernd auf den Zustand des Interpreters zugreifen. Am Beispiel einer If-Bedingung soll dies im Folgenden demonstriert werden:

1. Einlesen des Codes
   Der HLL-Code wird von der Shell über den Interpreter zum Scanner gereicht, welcher ein HLL-If-Bedingungs-Konstrukt erkennt.

2. Übergabe an den Transformer
   Es wird der entsprechende Teil im Bison-Parser angesprochen, welcher die execute-Methode in der entsprechende Typ-Klasse (If-Bedingung) aufruft

3. Ausführen des Codes
   Die execute-Methode des If-Bedingungs-Typs führt auf C++ Ebene die entsprechenden Anweisungen aus.

Um nun die gewünschte Ausgabe des C++-Codes zu erhalten bedient man sich der eigentlich zu Debug-Zwecken implementierten testPrintTypes-Methoden. Diese werden

so modifiziert, dass sie statt Debug-Ausgaben, typspezifische C++-Code-Fragmente zurückliefern. Um diese Funktionalität handhabbar zu gestalten, wird zudem ein HLL-Transformer-Objekt eingeführt, welches nach einer Instanzierung eine „parse“-Methode zur Verfügung stellt. Abbildung 3.3 verdeutlicht die Funktionsweise.

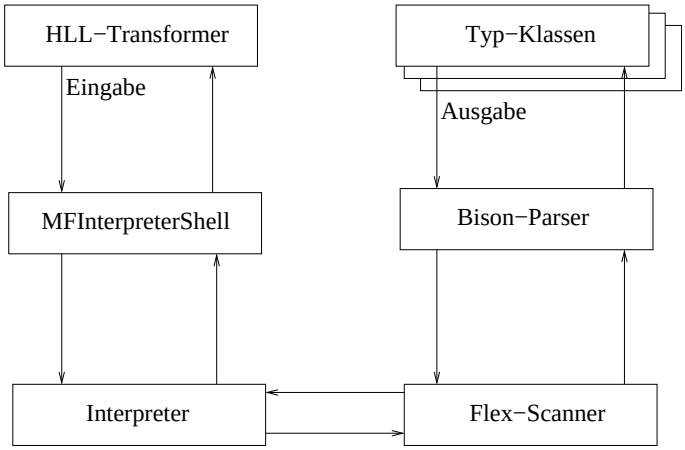


Abbildung 3.3.: Funktion des Transformers

Ein HLL-Transformer-Objekt reicht einen kompletten Codeblock über die Shell und den Interpreter an den Flex-Scanner weiter. Es folgt die Identifizierung des Code-Konstrukts analog zur Interpreterfunktion mit der Ausnahme, dass in den Typ-Klassen nun die testPrintTypes- statt der execute-Methode aufgerufen wird. Hinzu kommt die Rückgabe des geparsten C++-Codes durch alle Instanzen bis hin zum Ausgangspunkt, dem HLL-Transformer-Objekt.

Im Folgenden soll auf die genaue Funktionsweise des Parse-Vorgangs eingegangen werden. Dabei wird konkret Bezug auf einzelne Code-Beispiele genommen, um das Verständnis zu erleichtern. Die Schritte von der Instanzierung des HLL-Transformer-Objekts bis hin zur Übergabe des HLL-Eingabe-Codes an den Flex-Scanner werden dabei vernachlässigt, da es sich um triviale Schritte der objektorientierten Programmierung und Zeichenkettenverarbeitung handelt. Ebenso wird vorausgesetzt, dass die grundlegende Funktionsweise von Flex und Bison bekannt ist [9].

Die HLL-Eingabe wird durch den Code aus Abbildung 3.1 gegeben. Der Schwerpunkt der Erläuterungen zu diesem Beispiel liegt in der Bedingungsanweisung.

```
1 var Int: x := 10;
2 if (x == 10) then print("Hallo Welt!") fi;
```

Programmtext 3.1: HLL-EingabeCode

Die Eingabe wird als einfache Zeichenkette dem Flex-Scanner zugeführt. Anhand von definierten Schlüsselwörtern wird nun ein entsprechendes HLL-Konstrukt identifiziert (Anhang B auf Seite 109) , welches dem Bison-Parser zugänglich gemacht werden kann. Zu diesem Zweck existiert eine Kontroll-Klasse *iControl*, welche vom Flex-Scanner, als auch vom Bison-Parser abgeleitet ist und eine Parse-Methode bereitstellt, innerhalb derer die spezielle Bison-Methode *yyparse* aufgerufen wird (Beispiel 3.2). Diese ruft wiederum intern die Flex-Methode *yylex* auf, um das nächste Konstrukt übermittelt zu bekommen. Eine weiterführende, detaillierte Beschreibung der Kommunikation zwischen Flex-Scanner und Bison-Parser findet sich unter [9].

```
1  int iControl::Parse(const std::string& iString)
2  {
3    istrstream *input;
4    int return_value;
5
6    iIn = input = new istrstream (iString.c_str (), iString.length ());
7    iPos = 1;  iLine = 1;
8
9    return_value = yyparse ();
10   delete (input);
11   return (return_value);
12 }
```

Programmtext 3.2: iControl.Parse - yyParse-Aufruf

Da dem Bison-Parser nun ein entsprechender Konstrukt-Typ bekannt ist, können die jeweiligen Grammatikregeln, wie in Codebeispiel 3.3 auf der nächsten Seite verdeutlicht, zum Parsen angewendet werden. In diesem Fall werden entsprechende Typ-Klassen mit Parametern aus dem erkannten Konstrukt initialisiert. Ist die Identifikation einer kompletten Codezeile abgeschlossen, werden die jeweiligen execute-, bzw. testPrintTypes-Methoden in diesen Klassen aufgerufen.

```
1  if      :       expr THEN statements else
2                  { $$ = new EIfThen ($1, new ERunlist (*$3, @3, @3), $4, @1, @4);
3                  delete ($3); }
4          ;
5
6  else    :       FI { $$ = 0; }
7          |       ELSE statements FI
8                  { $$ = new ERunlist (*$2, @2, @2);
9                  delete ($2); }
10         |       ELSIF if
11                 { $$ = $2; }
12         ;
```

Programmtext 3.3: If-Then-Else Regel im Bison Parser

Das Ende einer Zeile wird dabei analog durch eine entsprechende Regel erkannt, innerhalb derer allerdings anstelle einer Typ-Klassen-Instanzierung, eben diese Klasse als Parameter übergeben und die execute-, bzw. testPrintTypes-Methode aufgerufen wird. Beipiel 3.4 zeigt den entsprechenden Ausschnitt, wobei in Zeile 8 die dem Typen zugehörige testPrintTypes-Methode der entsprechenden Klasse zu erkennen ist.

```
1  lines   :       line ';'
2                  { MFIValue* result;
3                    interpreter_parse_set.erase ($1);
4                    if ($1->completeType () == false) {
5
6                            theMachine.resetState ();
7
8                          $1->testPrintTypes(ostr);
9                          outStr = ostr.str();
10                         result = 0;
```

Programmtext 3.4: Zeilen-Regel im Bison-Parser

Die verwendete testPrintTypes-Methode für die If-Bedingung findet sich in Codebeispiel 3.5 auf der nächsten Seite.

```
1  void
2  EIfThen::testPrintTypes (std::ostream & out)
3  {
4      out << " if(";
5      conditional->testPrintTypes (out);
6      out << ") {\n";
7      if_true->testPrintTypes (out);
8      if (if_false != 0) {
9          out << "} else {\n";
10         if_false->testPrintTypes (out);
11     }
12     out << "}\n ";
13 }
```

Programmtext 3.5: TestPrintTypes-Aufruf des If-Then-ElseTypen

Es ist zu erkennen, das an dieser Stelle das endgültige Konstruieren des C++-Codes stattfindet. Es wird das Konstrukt linear auf Basis der Syntax der Zielsprache in eine Zeichenkette geschrieben. Diese kann dann im weiteren Verlauf durch alle Instanzen des Interpreters bzw. HLL-Transformers zurückgegeben werden. Eine vollständige Auflistung aller Typklassen mit Ausgabemethoden für C++ Ausgaben findet sich in Anhang A auf Seite 108. Nachdem nun die Funktionsweise des HLL-Transformers umschrieben wurde, sollen die aufgetretenen Probleme diskutiert werden. Im Rahmen der PG ist es nicht gelungen, die volle Funktionalität des Programms zu erreichen. Die zeilenweise Vorgehensweise des ursprünglichen Interpreters ist dabei eines der größten Probleme in Bezug auf den HLL-Transformer. So ist es beispielsweise nicht möglich, verschachtelte Strukturen oder Funktionsaufrufe ohne größeren Aufwand umzuwandeln. Da Funktionen nicht eins zu eins umgesetzt werden können, muss auf das Interpreter-Framework aufgesetzt werden. Der HLL-Funktionsname muss dabei dem Interpreter-Kern übergeben werden, so dass der tatsächliche Funktionsaufruf letztendlich von internen Interpreterroutinen geleistet wird. Hierzu ist es allerdings nötig, eine Reihe von Objektinstanziierungen und Methodenaufrufen im Zielcode vorzunehmen. Ein einzeiliger Funktionsaufruf in der HLL-Eingabe wird somit zu einem kompletten Codeabschnitt in der C++-Ausgabe. Probleme treten nun auf, wenn beispielsweise eine Funktion zur Auswertung eines Bool'schen Wertes benutzt werden soll. Hier kann nicht der ganze Block eingefügt werden, sondern es muss speziell die Zeile mit der Funktionsausführung und dem damit verbundenen Rückgabewert Verwendung finden. Der Code muss rückwärtig durchsucht und an entsprechender Stelle die Instanziierung und Deklarationen durchgeführt werden. Der eigentliche Funktionsaufruf muss dann wieder an der ursprünglichen Stelle stattfinden. Dies gelingt bisher nur teilweise, da durch die Verwendung von Zeichenketten-Streams und das sofortige Ausgeben dieser keine Pufferung vorhanden ist. Aus Zeitgründen wurde dieser Ansatz letztendlich nicht verfeinert, da als Umgehung des Problems ein eins zu eins Parsen der HLL-Funktionen eingefügt wurde. Allerdings ist es somit nicht mehr möglich, frei definierte Funktionen zu verwenden. Lediglich Methoden von standard C++ Elementen

können Verwendung finden (z.B. size() bei Listen). Da in diesem Zusammenhang allerdings die vorhandenen Dokumentationen der HLL nicht alle verfügbaren Methoden ihrer abstrakten Datentypen (List, Set, Tupel) abdecken, konnte auch dieser Ansatz nicht 100%ig umgesetzt werden. Nach den Problemen mit den Funktionsaufrufen folgt die Initialisierung von Listen, Sets und Tupeln. C++ sieht hier keine Mechanismen vor, diese abstrakten Datentypen direkt bei der Instanzierung mit Werten zufüllen, wie es in HLL möglich ist. Somit entsteht aus jeder Variablen-Initialisierung mit diesen Typen wiederum ein ganzer Codeblock in der Zielsprache. Sind all die beschriebenen Probleme noch mehr oder weniger vermeidbarer Natur, so liegt die wirkliche Problematik in der Verwendung des generierten C++-Codes im Testmodellgenerator. Zum Verständnis muss hier zunächst erwähnt werden, dass der ursprüngliche Interpreter und der Testmodellgenerator jeweils eine eigene Variablenumgebung und eigene Typ-Klassen besitzen, die sich gegenseitig nicht kennen. Durch die Spezifikation des abstrakten Zustandsraums werden zum Teil Variablen dem Testmodellgenerator bekannt gemacht, die dann in den Testskripten benutzt werden sollten. Da diese jedoch mit dem erweiterten Interpreter geparst werden müssen, dieser jedoch die Variablen nicht kennt, ist eine doppelte Deklaration unvermeidbar. Durch das Aufsetzen des HLL-Transformers auf den Interpreter erbt dieser natürlich auch die Fehlererkennung, die eine Verwendung von nicht deklarierten Variablen ausschließt. Mangels Dokumentation ist diese Abfrage nicht trivial zu entfernen. Da der Interpreter bei der Funktionsbestimmung von abstrakten Datentypen auf dem Typmodell aufsetzt und somit den Typ einer Variable vor der Verwendung kennen muss, ist ein entsprechendes Entfernen auch nicht wünschenswert oder ohne weitreichende Änderungen möglich. Die erbrachte Funktionalität bzw. nicht erreichte Funktionen befinden sich in Anhang C.2 auf Seite 112. Zusammenfassend soll festgehalten werden, dass sich aufgrund mangelnder Dokumentation die Einarbeitungsphase und Lösungsfindung im Interpreter als überdurchschnittlich zeitaufwendig herausstellte. Teilprobleme konnten somit nicht gelöst werden und für entsprechende Umgehungen fehlte letztendlich die Zeit, zumal es nicht abzusehen war, dass eine reibungslose Funktion zu erreichen wäre.

3.3. Simulator

3.3.1. Anforderungen

Im Rahmen der PG wurde der Simulator als ein neues Modul für das ABC entwickelt. Der Simulator soll den Benutzer bei der Entwicklung von Service Logic Graphen für Web-Applikationen unterstützen.

Ziel ist es dem Entwickler das Durchlaufen seines Graphen visuell und interaktiv zu ermöglichen und dabei möglichst weitreichende Eingriffsmöglichkeiten zu gestatten. Der Simulator ist eine Erweiterung des SDTracers, welcher Teile der Funktionalität schon enthält. Der Kern des SDTracers ist ein GraphLabel, implementiert durch die Klasse SDTracerGraphLabel. Von dieser Klasse erbt das SimulatorGraphLabel. In Service Logic Graphen gibt es (etwas vereinfacht dargestellt) zwei verschiedene Arten von Knoten. Zum einen die Interaktionsknoten. Sie repräsentieren die Interaktion mit dem Benutzer, stellen also HTML-Seiten dar. Alle anderen Knoten (z.B. Koordinationsknoten) spiegeln

die interne Funktionalität der Web-Applikation wieder.

Um die Anforderungen zu verdeutlichen werden nun die grundsätzlichen Unterschiede zum SDTracer erläutert:

Im SDTracer muss es für jede SIB eine .rtc-Datei geben, welche HLL-Code enthält und den nächsten Branch festlegen muss. Der SDTracer ruft den HLL-Code für einen Knoten auf, dieser entscheidet den nächsten Branch. Der SDTracer folgt diesem und gelangt so zum nächsten Knoten, woraufhin der Prozess von vorne beginnt. Der Simulator soll dies dynamischer und ähnlicher einer echten Web-Applikation gestalten. Zum einen sollten Interaktionsknoten wirklich eine Interaktion bieten: Der Benutzer des Simulators wird aufgefordert, die Interaktion auszuführen. Es müssen also die HTML-Navigationselemente simuliert werden. Für alle anderen Knoten (z.B. Koordinationsknoten) soll es weiterhin HLL-Code geben, welcher das Verhalten spezifiziert. Im Gegensatz zum SDTracer soll der Code nicht nur per SIB sondern auch per Knoten spezifizierbar sein. Beim SDTracer ist der HLL-Code der verschiedenen Knoten, welcher nacheinander ausgeführt wird in einem gemeinsamen Scope. Dies führt dazu, dass Variablen, wenn sie einmal deklariert wurden in späteren Knoten immer noch zur Verfügung stehen. Der Simulator trennt den Code der einzelnen Knoten, sie können also nicht willkürlich Informationen austauschen. Um dies dennoch möglich zu machen (z.B. um Session-Variablen oder Datenbanken zu simulieren) soll eine konstante Menge von globalen Variablen spezifiziert werden können. Diese sollen zusammen mit ihrem aktuellen Wert stets angezeigt werden und es sollen immer Eingriffe von Seiten des Benutzers möglich sein. Eine weitere Erweiterung ist eine sogenannte Undo-Funktion. Simulationsschritte sollen also rückgängig gemacht werden können.

Zusammengefasst ergeben sich folgende Anforderungen:

- HTML-Formular Simulation
  Eine simulierte Anzeige von Navigationselementen der Webseite.
- Undo-Funktion
  Die Möglichkeit Simulationsschritte rückgängig zu machen.
- StateSpace Debugger
  Eine Anzeige der Zustandsvariablen der simulierten Web-Applikation und die Möglichkeit diese manuell während der Simulation zu ändern.

3.3.2. Entwurf

Der Simulator ersetzt den SDTracer und wird (wenn installiert) auf dieselbe Art gestartet, wie sonst der SDTracer gestartet wurde. Um die folgende Beschreibung besser zu verstehen, ist es nützlich die Abbildung 3.4 auf der nächsten Seite parallel vor Augen zu haben. Alle Klassen, deren Namen in Fettschrift gedruckt sind wurden im Rahmen des Simulators neu entwickelt.

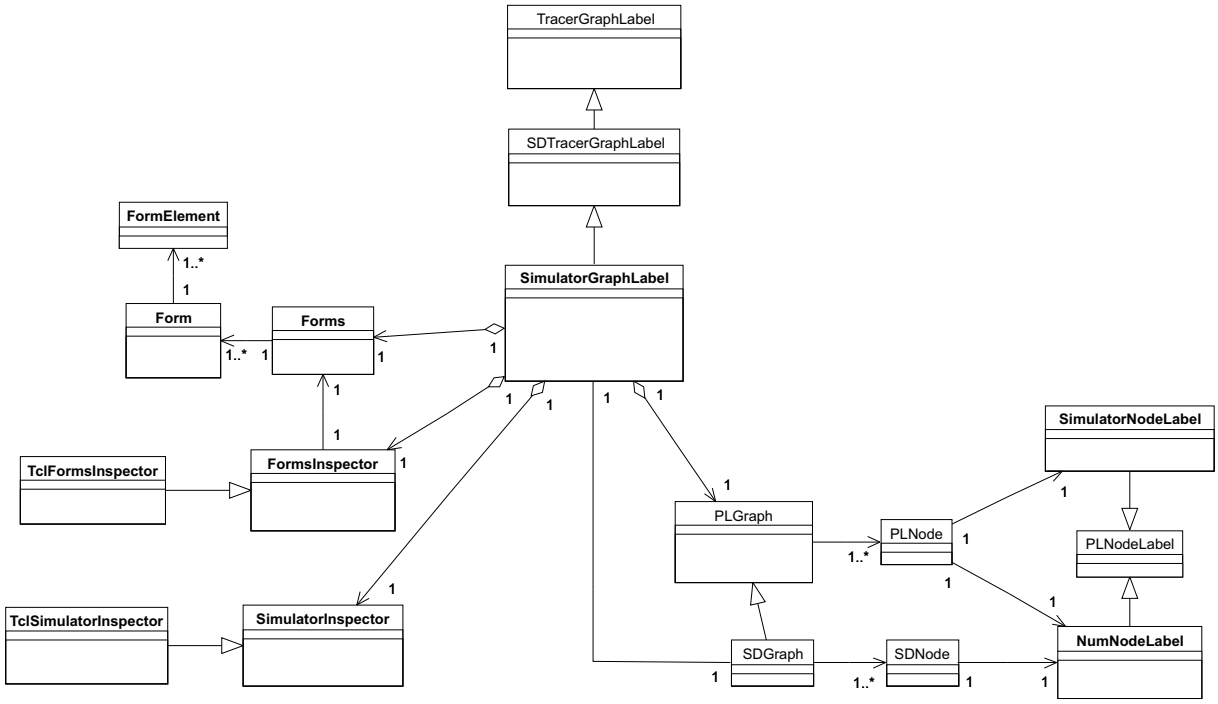


Abbildung 3.4.: Klassendiagramm

Der Kern des Simulators ist die Klasse SimulatorGraphLabel. Diese erbt von SDTracerGraphLabel, welche wiederum von TracerGraphLabel abgeleitet ist. Es gibt genau eine Instanz dieses GraphLabels, welche dem zu simulierenden SDGraphen als Attribut hinzugefügt wird.

Das SimulatorGraphLabel legt einen PLGraphen als Undo-Datenstruktur an und fügt diesem das SimulatorNodeLabel hinzu. Sowohl dem SDGraphen als auch dem PLGraphen wird das NumNodeLabel ergänzt, welches zur eindeutigen Nummerierung der Knoten dient. Siehe hierzu auch die Beschreibung zur Umsetzung der Undo-Funktion in Kapitel 3.3.3.3 auf Seite 48.

Zur GUI-Anbindung benutzt der Simulator die Klassen SimulatorInspector und FormsInspector. Ersterer dient zur primären Bedienung des Tracings, der Undo-Funktion und des StateSpace-Debuggers und steuert somit das Fenster wie es in der Abbildung 3.6 auf Seite 49 zu sehen ist. Die Klasse FormsInspector dient zur Darstellung der HTML-Formular-Simulation wie in Kapitel 3.3.3.2 auf Seite 47 beschrieben (Fenster wie in Abbildung 3.5 auf Seite 48).

3.3.3. Funktionalität

3.3.3.1. Voraussetzungen

Der Simulator differenziert beim Durchlaufen des Graphen zwischen Interaktionsknoten und allen anderen Knoten. Die anderen Knoten werden weitestgehend wie im SDTracer behandelt: Es muss für jeden Knoten eine .rtc-Datei existieren, welche HLL-Code enthält. Dies dient dazu das Verhalten der Knoten zu spezifizieren. Der Code kann beliebige Operationen ausführen, muss jedoch den nächsten Branch bestimmen, um dem Simulator die Information zu geben, welcher Kante er folgen muss. Der HLL-Code 3.6 enthält ein Beispiel, wie abhängig von der Richtigkeit zweier Eingaben der Branch festgelegt wird. Wenn kein Branch festgelegt wird kann die Simulation an dieser Stelle nicht fortgesetzt werden, es sei denn der Benutzer gibt manuell einen Branch ein.

```
1  if(user_name == name) then
2
3      if(user_password == password) then
4          loggedIn := true;
5          Tracer.setBranch ("ok");
6      fi;
7
8      else
9          loggedIn := false;
10         Tracer.setBranch ("fail");
11 fi;
```

Programmtext 3.6: HLL Code Beispiel zum Setzen des nächsten Branches (nextBranch)

Die .rtc-Dateien müssen im SIB-Verzeichnis des Projektes liegen und können auf zwei unterschiedliche Arten benannt werden:

1. ‹„name des sib“›.rtc
   Dieser Code wird für alle Knoten, die von diesem SIB abstammen, benutzt. Mit der zweiten Möglichkeit wird dieses für bestimmte Knoten überschrieben:

2. ‹„sib-id“_„name des knotens“›.rtc
   Dieser Code wird genau für einen speziellen Knoten benutzt, welcher über seine Sib-ID und seinen Knotennamen identifiziert wird.

Interaktionsknoten werden grundlegend anders und bedeutend komplexer abgehandelt als im SDTracer. Interaktionsknoten stellen die Seiten der Web-Applikation dar, repräsentieren also HTML-Seiten. Auf solchen Seiten kann es mehrere Navigationselemente geben; dies sind zum einen normale Links und parametrisierte Links, zum anderen auch Formulare mit all ihren Formularelementen.

Von Datei-Upload Formularelementen, sowie Javascript und Plugins, wie z.B. Flash oder Java-Applets wird an dieser Stelle abstrahiert.

Sowohl Links als auch Formulare haben eine Ziel-URL, welche im Graphen dem Branch entspricht.

Im EWISTest-Modul ist eine Möglichkeit zur Spezifikation von Links, parametrisierten Links und Formularen für Interaktionsknoten bereits vorgesehen. Diese Informationen werden in einer Datenbank gespeichert. Hierauf greift der Simulator zurück und daher sind diese Angaben zwingend notwendig für alle Interaktionsknoten.

Zustandsvariablen der Web-Applikation werden vom Simulator ebenfalls berücksichtigt. Sie werden mit HLL-Code in der Datei ‹„StateSpace“› deklariert und initialisiert. Diese Datei muss sich im selben Ordner wie die SIBs und die .rtc-Dateien befinden. In dieser Datei muss es zwei Kommentare geben: „(*!- -!*)“ und „(*- -*)“ . Die Datei muss mit dem ersten Kommentar beginnen und alle Variablen, die hiernach deklariert werden, werden vom Simulator berücksichtigt. Danach kommt der zweite Kommentar; nach diesem können weitere Variablen deklariert werden, die der Simulator allerdings ignoriert. Siehe hierzu auch das Beispiel im Programmtext 3.7. Dort werden drei Variablen deklariert, aber nur zwei im ersten Bereich.

```
1 (*!--!*)
2 var Bool: loggedIn := false;
3 var String List: publikation := [];
4
5 (*--*)
6 var String List List: genericEntries := [["Test1"],["Test2"]];
```

Programmtext 3.7: Beispiel einer StateSpace Datei

Der Simulationscode in den .rtc-Dateien kann auf diese Variablen zugreifen, sie auslesen und auch ändern.

Durch diesen StateSpace können wesentliche Teile einer echten Web-Applikation simuliert werden. Ein Beispiel wäre eine Datenbank (auf die jeder Koordinationsknoten Zugriff hätte); oder auch Session-Variablen können hier nachgestellt werden.

3.3.3.2. HTML-Formulare

Bedienung Beim Durchlaufen eines Interaktionsknoten wird ein Fenster dargestellt, welches die HTML-Seite des Knoten simuliert (siehe Abbildung 3.5 auf der nächsten Seite). Es werden die Formulare der Seite dargestellt; alle Links werden als Buttons wiedergegeben. Der Benutzer kann Formularfelder ausfüllen oder die Ausführung eines Links durch Klick auf den entsprechenden Button simulieren. Genau wie im HTTP-Protokoll verschwindet dann die Seite (bzw. das Fenster) und eine Neue öffnet sich (die Simulation folgt dem gewählten Branch).

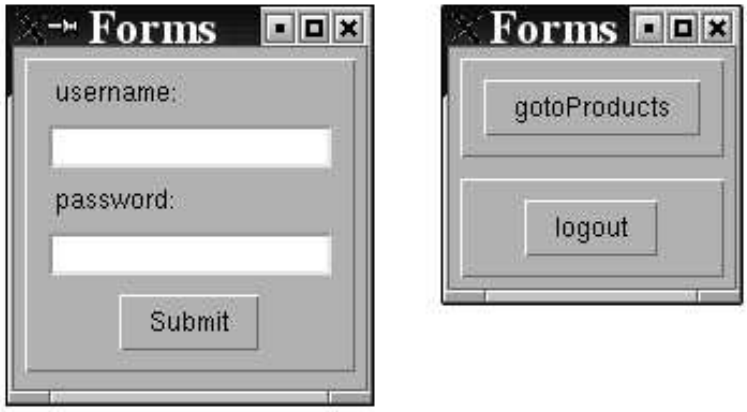


Abbildung 3.5.: Beispiel einer Darstellung von Formularelementen

Umsetzung Navigationselemente werden aus den EdgeLabels des EWISTest-Moduls ausgelesen. Aus diesen Informationen wird eine einfache Datenstruktur erzeugt, welche zur Darstellung des Formularfensters an die GUI weitergereicht wird (siehe auch das Klassendiagramm in Abbildung 3.4 auf Seite 45). Folgende Elemente sind möglich:

- Texteingabefelder
- DropDown-Boxen
- Radiobuttons
- Check-Boxen
- Select-Boxen
- Buttons

Buttons werden zum einen zur Darstellung aller Links benutzt. Zum anderen wird jedem Formular ein Button mit dem Namen „submit“ hinzugefügt, welcher zum Bestätigen (Abschicken) des Formulars dient.

Das Ergebnis eines abgeschickten Formulars oder eines parametrisierten Links ist eine Liste aus [name/value]-Paaren. Dies wird im CallContext gespeichert (Variable im State-Space) und beim Eintritt in den nächsten Interaktionsknoten gelöscht. Eine Auswertung des CallContext und somit aller Formulareingaben kann also nicht in einem Interaktionsknoten erfolgen, sondern muss durch einen vorgelagerten Koordinationsknoten erledigt werden.

3.3.3.3. Undo-Funktion

Bedienung Während eines Simulationsvorgangs können jederzeit Schritte rückgängig gemacht werden. Dies kann der Benutzer durch Betätigung des Buttons mit dem Pfeil

nach links auslösen. Dabei wird sowohl der aktuelle Knoten (mit entsprechender Aktualisierung der Graphanzeige), als auch der StateSpace zurückgesetzt. Es ist immer möglich mehrere Undo-Schritte in Folge auszuführen, bis der Start der Simulation wieder erreicht ist.

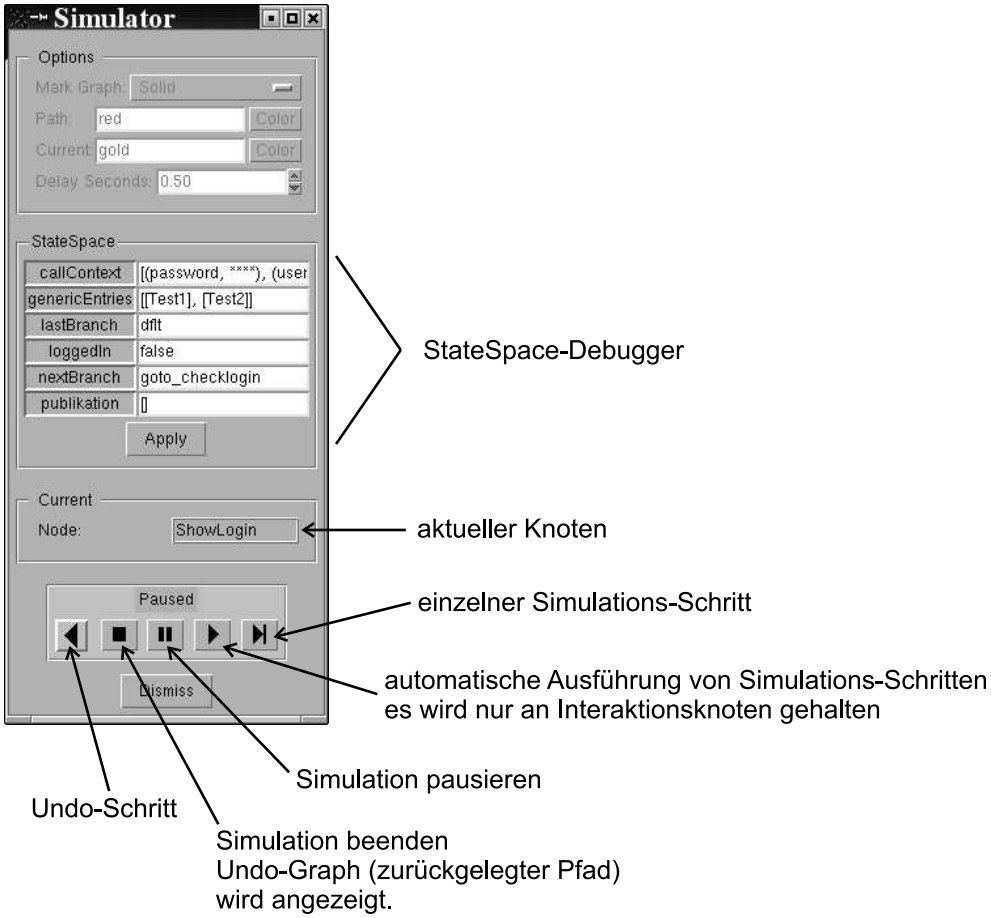


Abbildung 3.6.: Bedienungsfenster

Beim normalen schrittweisen Simulieren gibt es zwei verschiedene Haltepunkte. Zum einen wird am Ende jedes Knotens gehalten. Bei Interaktionsknoten wird zusätzlich im Knoten gehalten, um auf die Formulareingaben des Benutzers zu warten (siehe Kapitel 3.3.3.2 auf Seite 47). An dieser Stelle ist der Next-Knopf (Pfeil nach rechts) des Simulators deaktiviert, denn nur durch die Betätigung eines Formularelementes wird die Simulation fortgesetzt.

Genau zu diesen Haltepunkten wird beim „Undo“ zurückgesprungen. Der StateSpace wird wiederhergestellt, wie er an diesem Punkt existierte. Es können über den Debugger Veränderungen im StateSpace erfolgen und danach kann die Simulation wieder fortgesetzt werden.

Umsetzung Um die Undo-Funktion zu realisieren, war es notwendig während der Simulation den zurückgelegten Pfad zu speichern. Dies wird mit Hilfe eines zusätzlichen PLGraphen realisiert.

Jeder Knoten und jede Kante, die beim Simulieren passiert werden, wird daraufhin dem PLGraphen (also der Undo-Datenstruktur) hinzugefügt.

Die Knoten im Undo-Graphen sind mit zwei NodeLabels ausgestattet. Zum einen ein NumNodeLabel, welches nur eine Nummer speichert. Dieses NodeLabel wird vom Simulator auch dem zu simulierenden Graphen hinzugefügt. Im Service Logic Graphen ist die Nummerierung der Knoten eindeutig und diese Nummer wird in die NodeLabels des Undo-Graphen übertragen. Dies dient zur eindeutigen Zuordnung der Knoten in den beiden Datenstrukturen.

Das zweite NodeLabel im Undo-Graphen ist das SimulatorNodeLabel. Dies enthält den StateSpace, wie er am Ende des Knotens vorhanden war. Der beschriebene Aufbau wird schematisch in der Zeichnung 3.7 veranschaulicht.

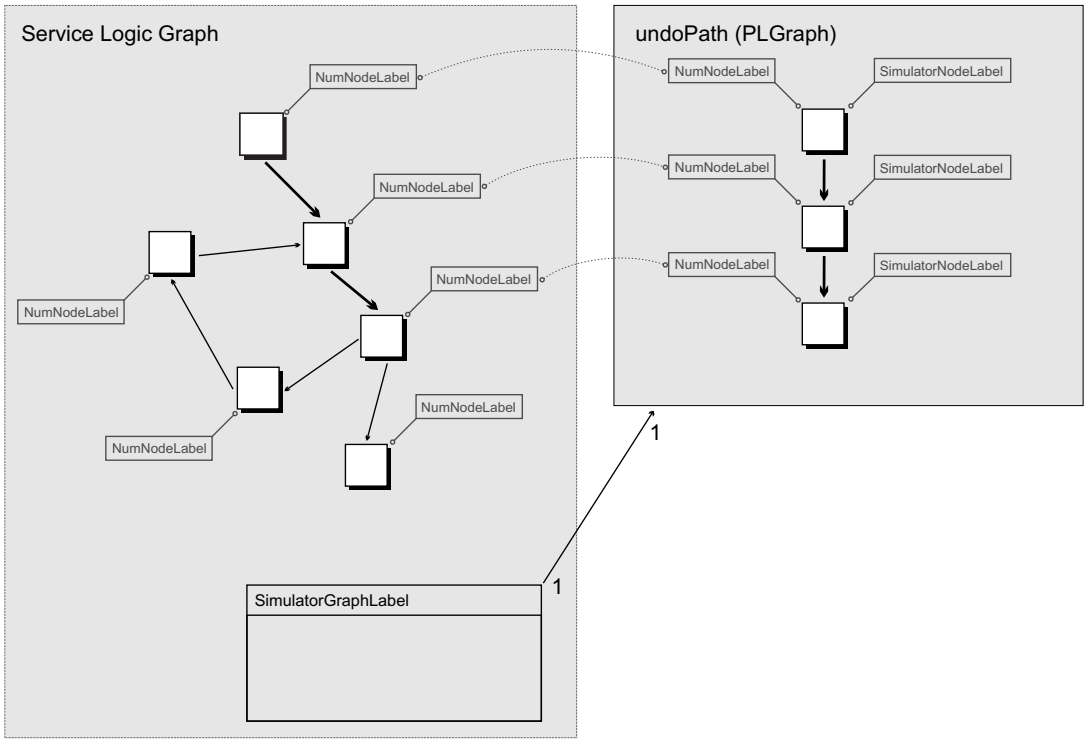


Abbildung 3.7.: Schematische Darstellung der Undo-Datenstruktur

Die so gespeicherten Informationen werden dann zur Umsetzung eines Undo-Schrittes benutzt. Es gibt genau drei Fälle, die betrachtet werden (siehe Abbildung 3.8 auf der nächsten Seite). Wie oben schon erwähnt wurde, unterscheidet man zwischen Interaktionsknoten und Koordinationsknoten:

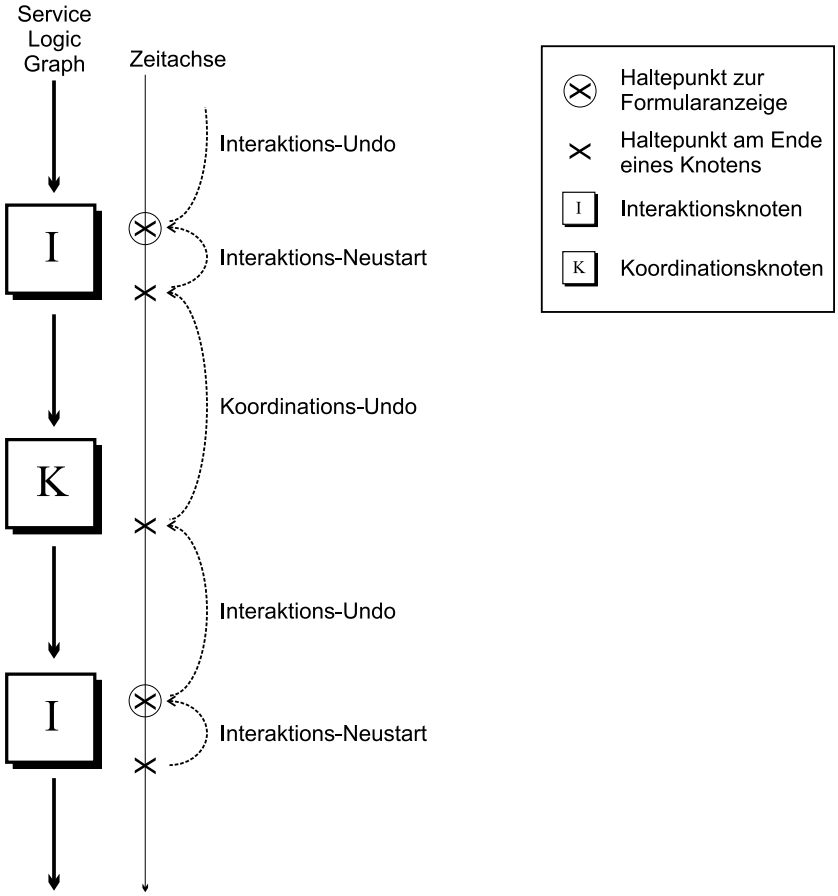


Abbildung 3.8.: Unterschiedliche Undo-Schritte

1. Interaktions-Undo
   Wenn die Simulation in einem Interaktionsknoten ist, der noch nicht abgearbeitet (also weder ein Formular abgeschickt, noch ein Link betätigt) wurde, und ein „Undo“ ausgeführt werden soll, dann wird zum Ende des vorherigen Knoten zurückgesprungen. Das Formular-Fenster wird einfach geschlossen. Der Zustandsraum zum Ende des vorherigen Knotens wird wieder hergestellt. Der Undo-Graph muss nicht verändert werden, denn der aktuelle Knoten wurde dem Undo-Graph noch nicht hinzugefügt. (Dies geschieht erst beim Erreichen des Endes eines Knotens.)

2. Interaktions-Neustart
   Der zweite Fall tritt ein, wenn am Ende eines Interaktionsknotens der Undo-Button gedrückt wird. Es erfolgt ein Sprung zurück zum Formular des Knotens. Der Undo-Graph (welcher einen zurückgelegten Pfad enthält) wird am Ende um einen Knoten gekürzt, denn sonst würde der Knoten beim erneuten Erreichen seines Endes zum zweiten Mal eingefügt. Ein erneuter Klick auf den Undo-Button führt zum Ende des Vorgängerknotens. Dies entspricht dem ersten Fall.

3. Koordinations-Undo
   Der letzte Fall tritt ein, wenn die Undo-Funktion in einem Koordinationsknoten aufgerufen wird. In diesem Fall springt der Simulator direkt zum Ende des vorhergehenden Knotens. Der Undo-Graph wird am Ende um einen Knoten gekürzt.

3.3.3.4. StateSpace-Debugger

Bedienung StateSpace-Variablen können z.B. zur Simulation von Datenbanken oder Session-Attributen einer Web-Applikation genutzt werden. Sie können auch direkten Einfluss auf den Ablauf der Simulation haben, denn ein Koordinationsknoten kann abhängig vom Inhalt solcher Variablen den nächsten Branch festlegen.

Im Simulatorfenster wird eine Tabelle mit den Werten dieser Variablen angezeigt. Auf der linken Seite stehen die Namen der Variablen, auf der rechten Seite ihre aktuellen Werte. Hier können jederzeit manuelle Eingriffe erfolgen, indem man Werte ändert und dann auf „Apply“ klickt.

Dies ist eine Funktion, die Kenntnisse des Benutzers voraussetzt, da keinerlei Kontrolle der Eingaben erfolgt. Diese müssen syntaktisch und semantisch korrekt erfolgen. So führt z.B. die Änderung des nextBranch auf einen Wert, der im Graph nicht existiert, zu einem Fehler beim HLL-Interpreter.

Umsetzung An jedem Haltepunkt fragt der Simulator die aktuellen Werte der Variablen vom Interpreter ab und speichert diese intern. Diese Informationen werden an die GUI weitergegeben. Welche Variablen vorhanden sind, weiß der Simulator zum einen durch die StateSpace-Datei, welche er parst. Zum anderen gibt es drei Variablen, die immer vorhanden sind: Den CallContext, den lastBranch und den nextBranch; diese drei Variablen sind fester Bestandteil des Simulators.

3.3.3.5. Bekannte Probleme

StateSpace Variablentypen Um alle Variablentypen als String in der GUI anzeigen zu können wird die im Interpreter vorgesehene unparse() Methode benutzt. Diese liefert auch für komplexe Datentypen eine String-Repräsentation. Eine Liste aus Listen von Strings wird z.B. folgendermaßen kodiert:

```
[[abc,def,xxx],[fgh,opi]]
```

Wenn jedoch eine manuelle Änderung im StateSpace-Debugger erfolgt, muss genau dieser String durch den Interpreter neu interpretiert werden. Hier fehlt jedoch das Quotieren der Strings. Der String müsste folgendermassen aussehen:

```
[["abc","def","xxx"],["fgh","opi"]]
```

Dies wird vom Simulator durchgeführt; er ist jedoch auf bestimmte, dem Simulator bekannte Datentypen begrenzt. Derzeit werden Folgende unterstützt:

- Liste von Strings
- Liste von Listen von Strings
- Liste von Tupeln von Strings

Dynamische Formulare In der gegenwärtigen Version des Simulators erfasst dieser nur statische Formulare; dynamische werden nicht unterstützt. Dynamische Formulare sind Formulare, deren Elemente oder deren Anzahl von dynamischen Faktoren abhängen (z.B. von Einträgen in Datenbanken). Ein Beispiel: Es soll eine Liste dargestellt werden und jeder einzelne Eintrag soll gelöscht werden können. Also wäre es naheliegend hinter jedem Eintrage einen Delete-Button anzuzeigen. Die Anzahl dieser Buttons wäre aber abhängig von der Anzahl der Einträge in der Liste.

3.3.4. Beispiel

Die Web-Applikation „kleine Literatur-Datenbank“, die in Kapitel 2.3 auf Seite 21 beschrieben ist, lässt sich durch den Simulator modellieren.

Kontrollfluss Der grundsätzliche Kontrollfluss kann durch die folgende Abbildung 3.9 veranschaulicht werden.

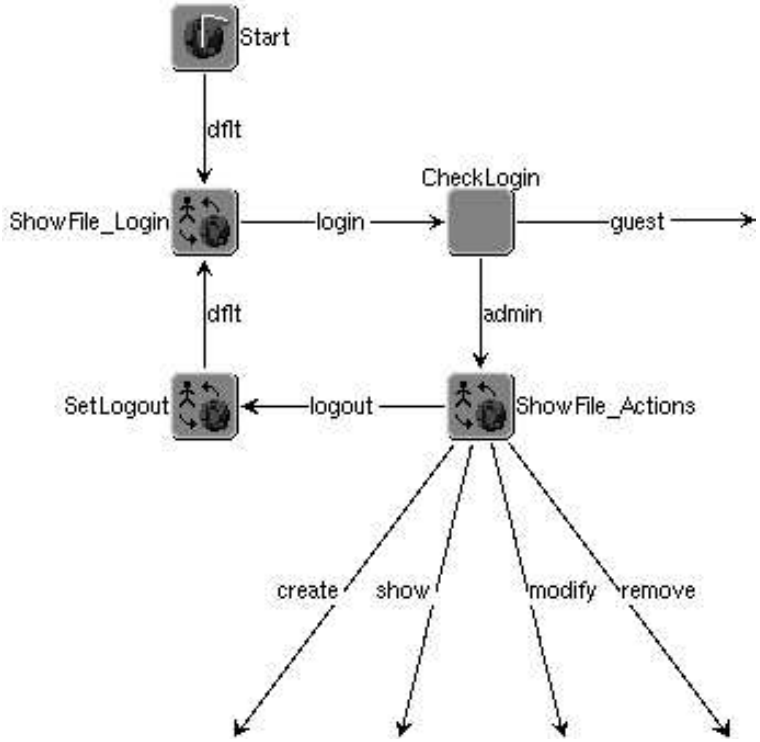


Abbildung 3.9.: Kontrollfluss der Beispiel-Anwendung

Die Simulation beginnt mit dem Start-Knoten. Der Benutzer bekommt zunächst das Login-Formular angezeigt (siehe Abbildung 3.10 auf der nächsten Seite). Gemäß der eingegebenen Daten wird es unterschieden, ob es sich um den Administrator oder den Gast

handelt. Wenn der Benutzer als Administrator identifiziert ist, kommt er zu dem Interaktionsknoten „ShowFile_Actions“(siehe Abbildung 3.13 auf Seite 56), bei dem er eine der vier verfügbaren Funktionen „show-“, „create-“, „modify-“ oder „remove-publication“ auswählen kann. Dagegen steht dem Gast nur die einzige Funktion „show-publication“ zur Verfügung.

Abbildung 3.10.: Login-Formular der Beispiel-Anwendung

Zustandsraum Die globalen Variablen der Applikation sind im Zustandsraum gespeichert und bestehen aus folgenden Einträgen:

- „CallContext“ besteht aus einer Liste aus [key/value]-Paaren, in denen das Ergebnis eines abgeschickten Formulars gespeichert wird.
- „GenericEntries“ besteht aus einer Liste von Listen von Strings und modelliert die Datenbank der reellen Applikation. Jede Publikation in der Datenbank wird durch eine Liste von Strings dargestellt und besitzt der Einfachheit halber nur zwei Attribute (title, note).
- „lastBranch“ und „nextBranch“ geben an, welcher Kante der Simulator gerade folgt bzw. folgen wird.
- „LoggedIn“ gibt an, ob der Benutzer als Administrator identifiziert ist.
- „PublicationIndex“ gibt an, welche Publikation gewählt worden ist. Wenn der Publikationsindex größer als die Anzahl der bereits vorhandenen Publikationen ist (in diesem Fall um 1 größer), dann bedeutet dies, dass eine neue Publikation mit dem Index in die Datenbank „GenericEntries“ hinzugefügt wird.

Der oben beschriebene Zustandsraum ist in der folgenden Abbildung 3.11 auf der nächsten Seite einzusehen. Die Zustandsraum-Definition ist im Programmtext 3.8 auf der nächsten Seite anzugeben.

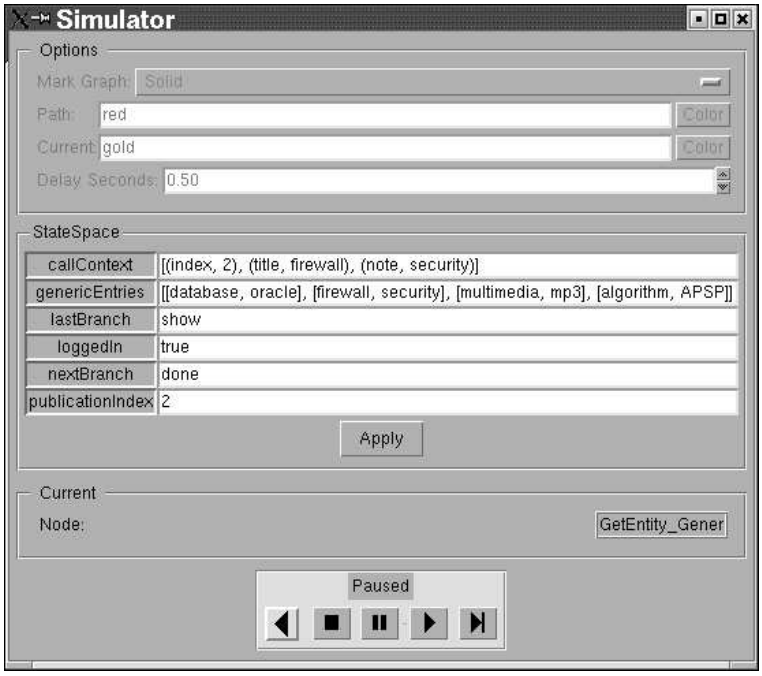


Abbildung 3.11.: Simulationsfenster der Beispiel-Anwendung

```
(*!--!*)
var Bool: loggedIn := false;
var String: publicationIndex := "";
var String List List: genericEntries := [["database", "oracle"],
                                          ["firewall", "security"],
                                          ["multimedia", "mp3"],
                                          ["algorithm", "APSP"]];
```

Programmtext 3.8: Quell-Code der Zustandsraum-Definition

Simulationscode Der Simulator arbeitet direkt mit dem HLL-Code für die Beispiel-Anwendung. Es wird im folgenden exemplarisch die Funktion „show-publication“ anhand des Simulationscodes im Detail beschrieben. Die folgende Abbildung 3.12 auf der nächsten Seite illustriert den „show-publication“-Zweig.

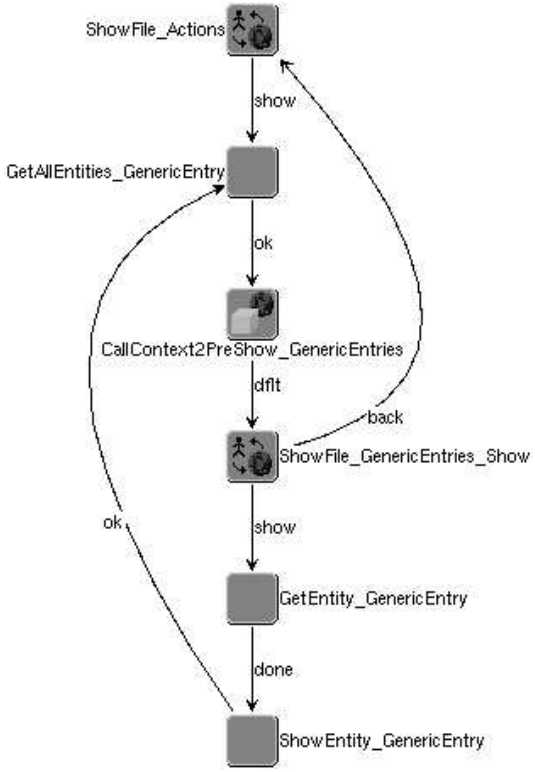


Abbildung 3.12.: „show-publication“-Zweig

Nachdem der Benutzer als „Admin“ identifiziert ist, kommt er zu dem Interaktionsknoten „ShowFile_Actions“, der die folgende Abbildung 3.13 darstellt.

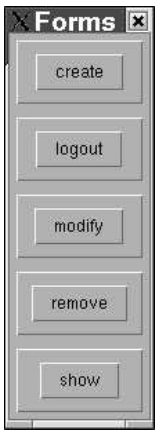


Abbildung 3.13.: Interaktionsknoten „ShowFile_Actions“

Falls der Benutzer nun den Zweig „show“ auswählt, kommt er von dem Interaktionsknoten „ShowFile_Actions“ zu dem Koordinationsknoten „GetAllEntities_GenericEntry“, dessen Funktionalitäten in „GetAllEntites.rtc“ spezifiziert sind. Siehe hierzu den Programmtext 3.9.

```
1 publicationIndex := "";
2 Tracer.setBranch("ok");
```

Programmtext 3.9: GetAllEntites.rtc

Hier wird der Publikationsindex initialisiert und der nächste Branch auf „ok“ gesetzt.

Dann kommt der Benutzer zu dem Knoten „CallContext2PreShow_GenericEntries“. In der reellen Applikation wird bei diesem Knoten der Inhalt vom CallContext in eine Map geschrieben, dessen Inhalt dann im nächsten Interaktionsknoten angezeigt werden kann. In der Simulation wird dabei lediglich der nächste Branch auf „dflt“ gesetzt, da der CallContext bei diesem Knoten nicht verwendet wird.

Beim nächsten Knoten „ShowFile_GenericEntries_Show“ handelt es sich wieder um einen Interaktionsknoten. Der Benutzer bekommt ein statisches Formular 3.14 zum Auswählen einer Publikation angezeigt.

Abbildung 3.14.: Formular zur Auswahl einer Publikation

Auf diesem Formular gibt es vier parametrisierte Links, die jeweils auf eine Publikation verweisen. Wenn der Benutzer „publication2“ auswählt, dann wird der Index dieser Publikation automatisch im CallContext gespeichert.

Die eigentliche Funktion „show-publication“ ist in dem folgenden Koordinationsknoten „GetEntitiy_GenericEntry“ implementiert (siehe Programmtext 3.10 auf der nächsten Seite).

```
1  procedure setCallContext(String: key, String: value)
2          var (String * String): oneObjectPair := ("", "");
3  begin
4          oneObjectPair := (key, value);
5          append(callContext, oneObjectPair);
6  end;
7
8  procedure getPublication(String: index)
9          var String List: currentPublication := ["", ""];
10         var String: stringTitle := "";
11         var String: stringNote := "";
12 begin
13         currentPublication:= fetch(genericEntries, index:Int);
14         stringTitle := fetch(currentPublication, 1);
15         stringNote := fetch(currentPublication, 2);
16         callContext := [];
17         setCallContext("index", index);
18         setCallContext("title", stringTitle);
19         setCallContext("note", stringNote);
20 end;
21
22 var Int: lengthCallContext := size(callContext);
23 var Int: counterCallContext := 1;
24 var String: attributeKey := "";
25 while counterCallContext <= lengthCallContext do
26         attributeKey := #1.(fetch(callContext, counterCallContext));
27         if(attributeKey == "index") then
28                 publicationIndex:=#2.(fetch(callContext, counterCallContext));
29                 getPublication(publicationIndex);
30         fi;
31         counterCallContext := counterCallContext + 1;
32 od;
33
34 Tracer.setBranch("done");
```

Programmtext 3.10: GetEntity.rtc

Der Simulationscode besteht aus zwei Prozeduren und dem Teil, der die Prozeduren aufruft. Mit der Prozedur „setCallContext“ wird manuell ein [name/value]-Paar im CallContext gespeichert. Die Prozedur „getPublication“ benötigt als Parameter den Index einer Publikation, die der Benutzer über GUI selektiert hat. Es wird dann die entsprechende Publikation anhand des Indexes aus der simulierten Datenbank „GenericEntries“ geholt. Die geholten Daten (title, note) werden anschließend im CallContext gespeichert. Bei der Ausführung des Simulationscodes wird zuerst der Publikationsindex aus dem CallContext durch eine While-Schleife ermittelt. Danach kommt die Prozedur „getPublication“ ins Spiel, die die Funktion „show-publication“ realisiert.

Nach der Ausführung dieses Knotens sind die Werte des Zustandsraums im Simulati-

onsfenster einzusehen (siehe Abbildung 3.11 auf Seite 55). Man kommt zuletzt zu dem Interaktionsknoten „ShowEntity_GenericEntry“, der das Formular mit dem Button „ok“ darstellt, mit dem man zurückkehren und somit einen erneuten Durchlauf starten kann.

3.4. Fazit

Abschließend kann man sagen, dass die gestellten Anforderungen erfüllt wurden. Obgleich nicht alle Funktionalitäten zu 100% umgesetzt werden konnten, sind doch brauchbare Lösungsansätze und Ergebnisse entstanden. Hier ist in vorderster Front der Simulator zu nennen, welcher es nunmehr möglich macht, das Verhalten von Anwendungen während der Entwicklung mit dem ABC zu veranschaulichen. Hier wurde der grösste Teil der Anforderungen umgesetzt. Auch das Benutzen von nur einer Spezifikation des abstrakten Zustandsraum zum Testen und zum Simulieren ist vollständig implementiert.

Der HLL-Transformer funktioniert eingeschränkt. Triviale Konstrukte und Datentypen lassen sich umwandeln, lediglich verschachtelte Strukturen, allgemeine Funktionsaufrufe und das Übergeben der Variablen an die Umgebung des Testmodellgenerators funktionieren noch nicht. Erste Schritte in diese Richtung sind allerdings gemacht worden. So könnten Schachtelungen durch ein Zwischenspeichern der Ausgabe mit Einfügung von Codeblöcken realisiert werden (siehe Kapitel 3.2.3 auf Seite 33). Durch die doppelte Übersetzung des abstrakten Zustandsraums im HLL-Transformers kann das Problem der Variablendeklaration gelöst werden. Zur Verwendung von allgemeinen Funktionen kann dynamisches Laden im Testmodellgenerator zum Erfolg führen. Letztendlich bleibt die redundante Implementierung des HLL-Transformers auf Basis des Interpreters. Hier kann über eine entsprechende Parametrisierung dafür gesorgt werden, dass der Interpreter je nach Aufruf, entweder den übergebenen Code interpretiert oder umgewandelt zurückgibt.

4. Constraint Editor

4.1. Überblick

Techniken der Verifikation von endlichen Zustandsmodellen (Finite-State Verification, z.B. Model Checking), haben sich trotz ihrer Effizienz noch nicht in der Praxis durchgesetzt. Eine große Hürde für den Benutzer von Tools zu diesen Techniken ist das Erlernen formaler Spezifikationssprachen wie z.B. LTL (Linear Time Logic), welche zur Spezifikation von Eigenschaften eines Systems benötigt werden. Solche Spezifikationen können ebenfalls enorm komplex werden, was die Arbeit mit den zugehörigen Formalismen zusätzlich erschwert. Dwyer, Avrunin und Corbett haben deshalb ein System vorgeschlagen [12], das den Weg von der natürlichsprachlichen Beschreibung einer Eigenschaft zu ihrer formalen Spezifikation erleichtern soll. Elementare Bestandteile des Systems sind die Patterns. Ein Pattern kann man als eine verallgemeinerte Beschreibung einer häufig auftretenden Anforderung für zulässige Sequenzen von Zuständen/Ereignissen in einem endlichen Zustandsmodell verstehen. Jedes Pattern hat einen Geltungsbereich (Scope) , wobei fünf Geltungsbereiche zu unterscheiden sind:

Global Die Eigenschaft gilt global.

Before Die Eigenschaft gilt vor einem bestimmten Zustand/Ereignis.

After Die Eigenschaft gilt nach einem bestimmten Zustand/Ereignis.

Between Die Eigenschaft gilt zwischen zwei bestimmten Zuständen/Ereignissen.

After-Until Die Eigenschaft gilt nach einem bestimmten Zustand/Ereignis, und zwar so lange, bis ein zweiter Zustand erreicht wird/ein zweites Ereignis eintrifft.

Essentiell für die Beschreibung eines Patterns ist die Angabe eines Namens, des Geltungsbereiches sowie der Abbildung in entsprechende Spezifikationsformalismen. Dwyer, Avrunin und Corbett haben herausgefunden, dass sich fast alle Spezifikationen in acht wesentliche Patterns einordnen lassen:

Absence Ein bestimmter Zustand/ein bestimmtes Ereignis kommt nicht in einem gegebenen Geltungsbereich vor.

Existence Ein bestimmter Zustand/ein bestimmtes Ereignis muss in einem gegebenen Geltungsbereich vorkommen.

Bounded Existence Ein bestimmter Zustand/ein bestimmtes Ereignis muss in einem gegebenen Geltungsbereich k-mal vorkommen.

Universality Ein bestimmter Zustand/ein bestimmtes Ereignis kommt in einem gegebenen Geltungsbereich durchweg vor.

Precedence Einem Zustand/Ereignis P muss in einem gegebenen Geltungsbereich immer ein Zustand/Ereignis Q vorausgehen.

Response Ein Zustand/Ereignis P muss in einem gegebenen Geltungsbereich immer von einem Zustand/Ereignis Q gefolgt werden.

Chain Precedence Einer Sequenz von Zuständen/Ereignissen $P_1, ..., P_n$ muss immer eine Sequenz von Zuständen/Ereignissen $Q_1, ..., Q_m$ vorausgehen.

Chain Response Eine Sequenz von Zuständen/Ereignissen $P_1, ..., P_n$ muss immer von einer Sequenz von Zuständen/Ereignissen $Q_1, ..., Q_m$ gefolgt werden.

Diese Patterns lassen sich gemäß ihrer Semantik in eine Hierarchie einordnen, die dann ein sogenanntes Pattern System bildet. Mit Hilfe dieses Systems soll der Benutzer auf weit intuitivere Weise eine Spezifikation erstellen können.

Beispiel. Es soll folgende Eigenschaft eines Aufzugsystems spezifiziert werden: Drückt man den Knopf auf Stockwerk i ($button.i$), so wird der Aufzug in jedem Fall dort ankommen ($floor.i$). Diese Eigenschaft lässt sich exakt auf das oben beschriebene Response Pattern mit dem Geltungsbereich Global abbilden. Mit dieser Zuweisung zu einem Pattern und der dabei mitgelieferten Abbildung auf Spezifikationsformalismen ist die formale Spezifikation nun trivial, es ergibt sich z.B. die LTL-Formel: $G(button.i \Rightarrow F(floor.i))$, wobei $button.i$ und $floor.i$ atomare Propositionen sind. Soll die oben beschriebene Eigenschaft so geändert werden, dass sie erst nach der Inbetriebnahme des Aufzuges ($start$) gelten soll, so ändert sich der Geltungsbereich, und statt Response global wird nun Response after verwendet. Dies führt natürlich auch zu einer Änderung der Formel, auf die das Pattern abbildet: $G(start \Rightarrow G(button.i \Rightarrow F(floor.i)))$.

Ergo führt die Änderung des Geltungsbereiches zwangsweise auch zu einem unterschiedlichen Pattern, da der Geltungsbereich die Semantik einer spezifizierten Eigenschaft entscheidend beeinflusst. Das Pattern System verfügt somit über 45 essentielle Patterns (acht Patterns mit je fünf Scopes). Im ABC [19] wird zur Spezifikation von Eigenschaften eines endlichen Zustandsmodells ESLTL [14] verwendet. Eine detaillierte Auflistung, wie jedes einzelne Pattern in eine ESLTL-Formel abgebildet wird, ist zu finden in Anhang D auf Seite 117. Das endliche Zustandsmodell, zu dem Eigenschaften spezifiziert werden, entspricht im ABC genau dem SLG, dessen Knoten die SIBs sind. Um in diesem Kontext zum Einsatz kommen zu können, muss das oben beschriebene Pattern System erweitert werden.

1. Atomare Propositionen reichen nicht mehr aus, da im ABC auch SIBs parametrisierbar sein können - es sind also Prädikate nötig. Außerdem ist der Allquantor sowie der Existenzquantor hinzuzufügen.

2. Das gesamte Pattern System ist zu erweitern, um die Behandlung von Gruppen von Patterns, sogenannten Composite Patterns, zu ermöglichen. Solche Composite

Patterns sind Konjunktionen von Patterns, welche zur besseren Begriffsabgrenzung nun Logic Patterns genannt werden. Composite Patterns werden nach semantischen Gesichtspunkten in Klassen organisiert.

Beispiel. Gegeben sei ein einfacher SLG, der zwei SIBs *login* und *logout* sowie eine Kante von *login* zu *logout* enthält. Beide SIBs lassen sich für einen bestimmten Typ von Benutzer spezialisieren. Dies geschieht durch den Parameter *usertype*, der angibt, ob der Login-SIB ein Login für den Administrator oder für einen anderen Benutzer durchführt (Logout-SIB analog). Es sollen nun folgende zwei Eigenschaften spezifiziert werden:

1. Jedem Login mit $usertype = X$ muss irgendwann ein Logout mit $usertype = X$ folgen (Response, Global).
2. Jedem Logout mit $usertype = X$ muss ein Login mit $usertype = X$ vorausgehen (Precedence, Global).

Um beide Eigenschaften zusammen spezifizieren zu können, muss man sie zu einem Composite Pattern kombinieren. Wenn man sich nun z.B. nicht auf einen speziellen Benutzertypen festlegen will, sondern die Eigenschaften für alle Typen X gelten sollen, wird auch die Notwendigkeit von Quantoren deutlich. Die Formel für ein solches Composite Pattern sieht in ESLTL folgendermaßen aus (in passender Syntax für den Model Checker des ABC): $Forall\ X\ in\ model\ (AG\_F\ ('login[usertype == X] \Rightarrow AF\_F\ ('logout[usertype == X])) \&\ ('logout[usertype == X] \Rightarrow AF\_B(login[usertype == X])))$.

Nun kann das erweiterte Pattern System verwendet werden, um im ABC Eigenschaften (Constraints) von SLGs zu spezifizieren. Eine solche Constraint kann einem SLG zugewiesen werden, um z.B. dann mit dem Model Checker zu verifizieren, ob das modellierte System die Constraint erfüllt. Auf Basis des erweiterten Pattern Systems sollte nun ein Constraint Editor realisiert werden, der dem Benutzer eine bequeme Erstellung, Editierung und Generierung von Constraints ermöglicht. Vorteil eines solchen Editors ist, dass sich der Aufwand für die Erstellung einer Constraint auf die Angabe von Metadaten, die Wahl des entsprechenden Composite Patterns sowie eine eventuelle Belegung der Parameter der SIBs beschränkt. Der Editor verfügt im Idealfall über eine sehr große Datenbank von anwendungsnahen Composite Patterns, die dann für die Erstellung von Constraints verwendet werden können. Das spart Zeit und minimiert die Anforderungen an den Kenntnisstand des Nutzers bzgl. formaler Spezifikationsformalismen. Im Rahmen der PG InterAkt entsprang die Realisierung des Constraint Editors aus anfänglichen Vorüberlegung, möglicherweise die Views (siehe 5 auf Seite 83) mit Hilfe des Model Checkers (also auch mit Hilfe von Constraints) zu generieren. Dies wurde letztlich verworfen, dennoch wurde der Constraint Editor trotzdem fertiggestellt, und wird im Folgenden als Teil der Projektgruppenarbeit detailliert vorgestellt.

4.2. Beschreibung

4.2.1. Anforderungen

Im Folgenden sollen die Anforderungen an den Constraint Editor beschrieben werden.

Aus funktionaler Sicht soll zunächst das Pattern System (d.h. die Hierarchie sowie bestehende Logic und Composite Patterns) mit XML beschrieben werden (wie spezifiziert in [14]) und für den Editor verfügbar sein. Die dazu benötigten XML-Dateien sollen folgende Informationen bereitstellen:

1. die zur Verfügung stehenden Logic Patterns mit ihren Abbildungen in die formale Spezifikationssprache (ESLTL) in einer entsprechenden Syntax (z.B. die Syntax des Model Checkers),

2. die Composite Patterns mit der Definition der verwendeten Parameter, der Angabe eines Eingabedialoges zur Spezifikation und der Festlegung, wie das entsprechende Logic Pattern zu generieren ist (d.h. wie die globalen Parameter des Composite Patterns in die lokalen Parameter jedes Logic Patterns abzubilden sind),

3. die Constraints selbst, d.h. Referenzen zu den entsprechenden Composite Patterns mit konkreten Werten für die Parameter.

Weiterhin sollen die XML-Dateien wichtige Metadaten zur Verfügung stellen, wie z.B. Name und Klassenzugehörigkeit von Logic bzw. Composite Patterns oder eine natürlichsprachliche Beschreibung der Constraints. Die korrekte Syntax der XML-Dateien soll mit Hilfe von DTDs beschrieben werden. Damit die Parameter des Composite Patterns belegt werden können, müssen zusätzlich die im aktuellen ABC-Projekt verfügbaren SIBs eingelesen und dem Benutzer zur Auswahl gestellt werden. Dazu wird eine Anbindung des Constraint Editors an das ABC-Projektmanagement notwendig. Hat der Benutzer alle erforderlichen Eingaben getätigt, so soll die erstellte Constraint generiert und gespeichert werden. Sie wird (in entsprechender Syntax) im aktuellen ABC-Projekt abgelegt, so dass sie auch für spätere Verwendung zur Verfügung steht. Dabei soll die Constraint sowohl im XML-Format als auch im ABC-konformen CDB-Format gespeichert werden, um eine spätere Editierung zu ermöglichen.

Die Erstellung bzw. Editierung einer Constraint durch den Benutzer soll nun folgende Schritte umfassen. Zunächst soll die Wahl des aktuellen ABC-Projekts sowie der entsprechenden Constraint Database in diesem Projekt erfolgen. Der Benutzer soll dann aus den bestehenden Klassen von Composite Patterns eine Klasse auswählen können. Aus den zugehörigen Patterns wird ebenfalls ein Pattern ausgewählt. Schließlich kann der Benutzer die Parameter des gewählten Composite Patterns belegen. Dazu wählt er aus den im Projekt zur Verfügung stehenden SIBs entsprechend aus. Ist ein SIB parametrisierbar, so sollen auch diese Parameter spezifiziert werden können. Dem Parameter eines SIBs kann entweder direkt ein konkreter Wert zugewiesen werden, oder er wird dem Existenz- bzw. Allquantor untergeordnet (siehe 4.1 auf Seite 60).

Aus nicht-funktionaler Sicht ist eine „lose" Integration des Constraint Editors in das ABC wünschenswert. Der Editor soll ein eigenständiges Modul des ABC sein, um ihn

für alle denkbaren Anwendungsfelder von Constraints verfügbar zu machen. Weiterhin sollte der Editor leicht erweiterbar, und z.B. durch eine zentrale Resources-Schicht an verschiedenste Anforderungen anpassbar sein. Z.B. ist es möglich, für die Parameter eines Composite Patterns komplette Teilformeln einzusetzen: hier wäre eine bloße Festlegung auf SIBs zu statisch.

Zur Implementierung der grafischen Benutzeroberfläche wird Tcl/Tk verwendet. Es sei aber betont, dass das Design des Editors generell von der Implementierung der GUI unabhängig sein soll. Dies folgt zum einen dem Model-View-Controller-Prinzip, zum anderen wird auch hier eine leichte Erweiterbarkeit des Editors angestrebt.

4.2.2. Prototyp

Im ersten Semester wurde ein Prototyp des Constraint Editors entwickelt. Das Ziel der Erstellung des Prototyps war es, sich mit den Techniken vertraut zu machen, die für die Realisierung des Constraint Editors benötigt werden. Dazu gehört neben Tcl/Tk [7], C++ und deren Schnittstelle auch die Bibliothek libxml2 [15], die für das Parsen und Schreiben von XML-Dateien zuständig ist.

Der Prototyp besteht aus einem Datenmodell, einem prototypischen Controller und einer Tcl/Tk-GUI. Das Datenmodell kann die Daten aus einer Test-XML-Datenbasis holen (also die entsprechenden XML-Dateien parsen), und diese in Objekte kapseln. Die Daten, die auf der GUI gezeigt werden sollen, werden durch den prototypischen Controller in geeigneter Form umgewandelt und an die GUI übergeben. Die GUI zeigt schließlich die Daten an, und die Widgets werden nach Bedarf mit Events verbunden. Außerdem kann der Prototyp per einfacher Konsoleneingaben spezifizierte Test-Constraints als CDB-Datei generieren. Solche CDB-Dateien werden im ABC zur Speicherung von Constraints verwendet.

Damit waren alle technischen Schwierigkeiten abgedeckt und die Realisierbarkeit der wichtigsten Anforderungen aus der Anforderungsanalyse validiert.

4.2.3. Entwurf

4.2.3.1. Architektur

Die Modellierung des Constraint Editors erfolgte nach einem Schichtenansatz, der das Programm in einzelne Komponenten unterteilen soll, welche strikt voneinander getrennt sind. Dabei wird der klassische Model-View-Controller-Ansatz verfolgt, also eine Trennung von Model (Datenmodell), View (Benutzer-Oberfläche) und Controller (Steuerungsteil). Zusätzlich wird die View-Schicht in eine AbstractGUI und eine TclGUI getrennt. Dies hat den Vorteil, dass die GUI prinzipiell austauschbar ist und somit die Tcl/Tk-GUI auch durch eine Andere (z.B. implementiert in JAVA, GTK, etc.) ersetzt werden könnte, ohne die anderen Schichten des Constraint Editors anpassen zu müssen. Die AbstractGUI soll zum einen von der Implementierung der GUI und vom Design der Fenster-Oberfläche völlig unabhängig sein, zum anderen muss sie konkret genug sein, um eine vollständige Schnittstelle für den Controller darzustellen. Dies wird natürlich erschwert durch die Vielfalt und Unterschiedlichkeit der Mechanismen in verschiedenen

GUI-Systemen. Eine Lösung stellt folgende Idee dar: eine GUI wird hier im Wesentlichen als zwei Mengen von Eingaben und Ausgaben verstanden. Völlig unabhängig von Aussehen und Implementierung der GUI gibt der Controller anzuzeigende Daten an die GUI und erwartet dann wiederum bestimmte Daten als Antwort. Diese „Kommunikation" zwischen Controller und GUI wird durch eine uniforme Schnittstelle (eben die AbstractGUI) reglementiert (weiteres siehe 4.2.3.2 auf Seite 70). Die Tcl/Tk-GUI implementiert und nutzt die Klassen der AbstractGUI, um eine GUI in Tcl/Tk umzusetzen. Dieser Teil der GUI ist austauschbar, d.h. es wäre ebenso gut z.B. eine Java-GUI oder eine GTK-GUI denkbar.

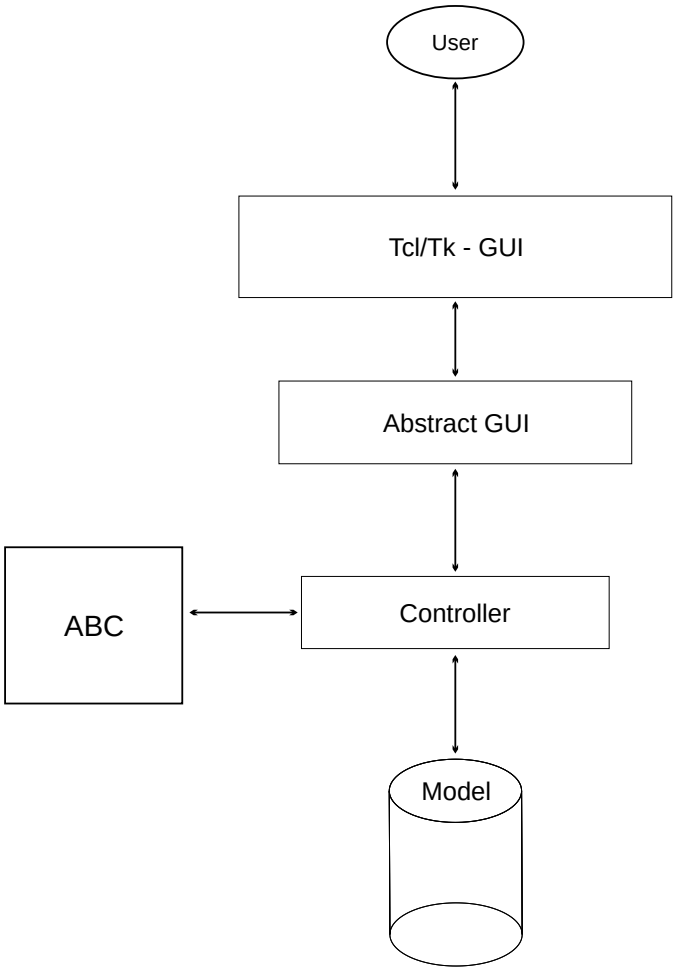


Abbildung 4.1.: Aufbau des Constraint Editors

Der Constraint Editor soll aus Sicht des Benutzers einen Assistenten darstellen, der aus

sechs Fenstern besteht. Diese sechs Fenster stellen modale Dialoge dar. Zwischen ihnen kann der Benutzer mit den Buttons „Next" und „Back" navigieren. Die sechs Fenster sind:

1. Projektauswahl (Abbildung 4.2)
   Zunächst muss der Benutzer das ABC-Projekt wählen, auf das sich der Editor beziehen soll.

2. Auswahl der Constraint Database
   Abhängig vom gewählten Projekt muss zusätzlich die Constraint Database gewählt werden, in der die Constraints gespeichert werden können.

3. Constraint Overview (Abbildung 4.3 auf der nächsten Seite)
   Dies ist das Fenster mit der Ansicht aller aktuellen Constraints. Hier kann ausgewählt werden, ob man eine Constraint editieren, löschen oder neu erzeugen möchte.

4. Constraint Meta Data (Abbildung 4.4 auf der nächsten Seite)
   Hier wird der Name und die Beschreibung eines Constraints angezeigt bzw. editiert.

5. Composite Pattern Selector (Abbildung 4.5 auf Seite 68)
   Die Auswahl eines Composite Patterns zu einem Constraint erfolgt in diesem Dialog.

6. Parameter Dialog (Abbildung 4.6 auf Seite 68)
   Dieser Dialog dient der Spezifikation aller Parameter eines Composite Patterns für die Constraint, also im Falle des ABC der Auswahl der SIBs sowie der Belegung ihrer Parameter.

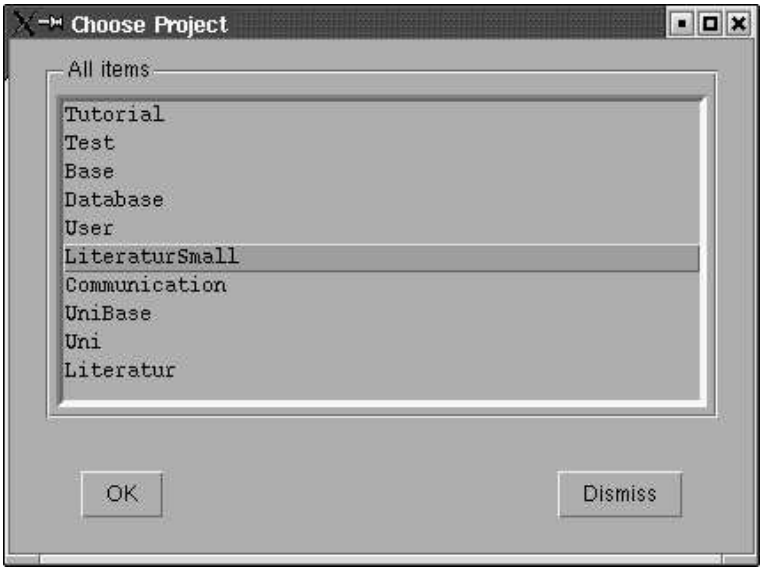


Abbildung 4.2.: Projektauswahl

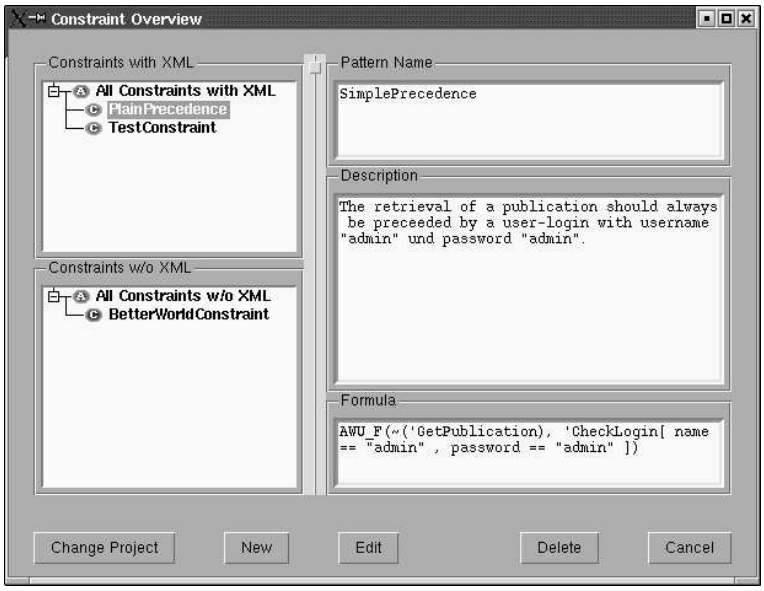


Abbildung 4.3.: Constraint Overview

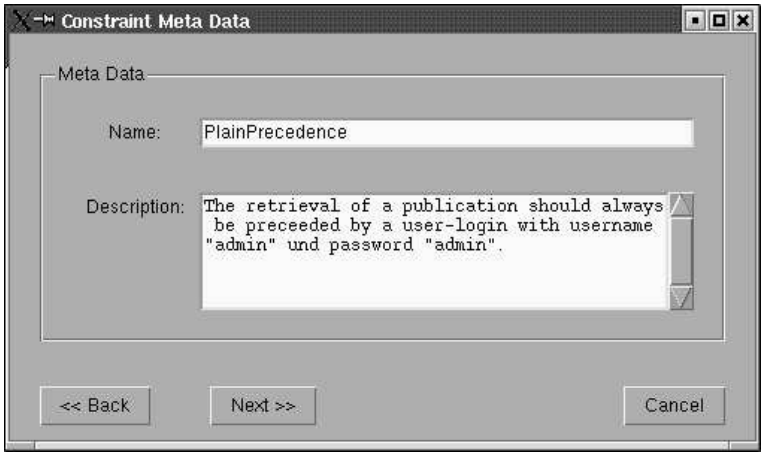


Abbildung 4.4.: Constraint Meta Data

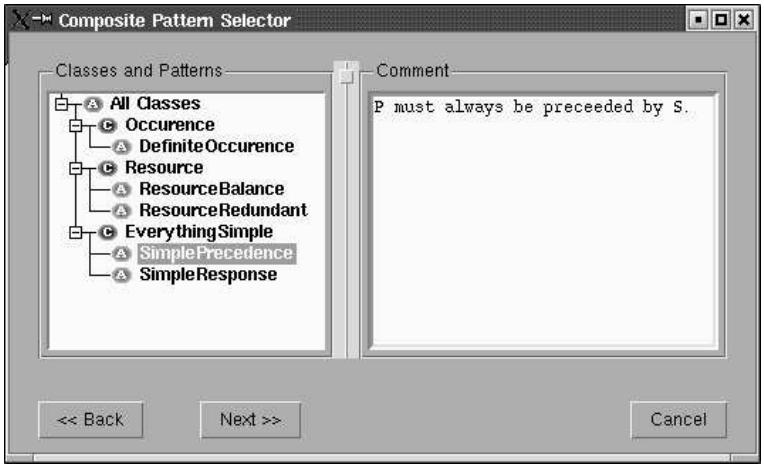


Abbildung 4.5.: Composite Pattern Selector

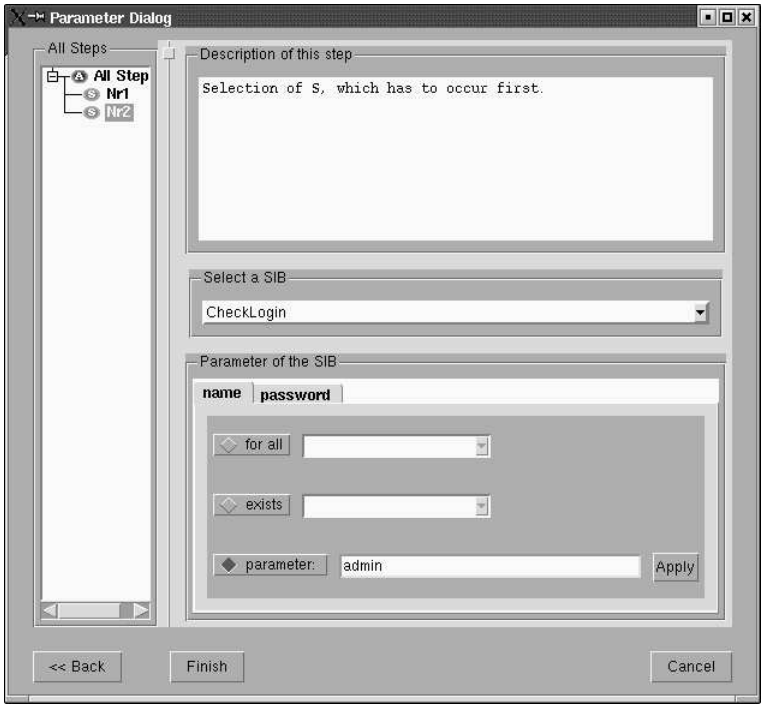


Abbildung 4.6.: Parameter Dialog

Abbildung 4.7 auf der nächsten Seite veranschaulicht den Workflow des Constraint Editors. Die eckigen Kästen repräsentieren die Dialoge, die Pfeile dazwischen markieren die möglichen Übergänge zwischen den einzelnen Fenstern. Ein solcher Übergang wird durch

den Benutzer, der auf die im Editor angebotenen Buttons klickt, angestoßen.

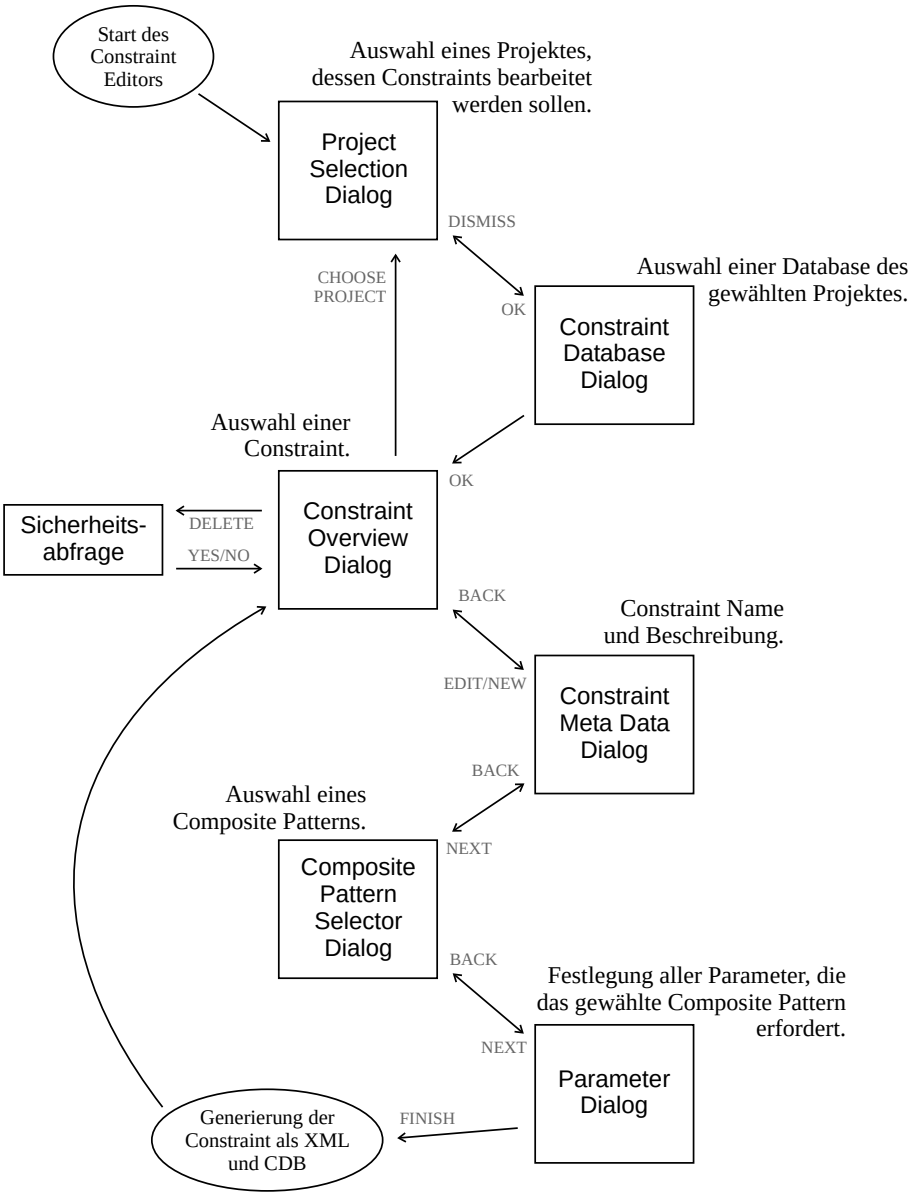


Abbildung 4.7.: Workflow des Constraint Editors

Im Folgenden sollen die einzelnen Schichten des Editors (Abstrakte GUI, Tcl/Tk-GUI, Controller, Datenmodell, Resources) detailliert erläutert werden, wobei für das tiefer gehende Verständnis des Datenmodells ebenfalls die dem Constraint Editor zu Grunde liegende XML-Datenbasis beschrieben wird.

4.2.3.2. Abstrakte GUI-Schicht

Diese Schicht abstrahiert von der konkreten Implementierung der GUI und stellt eine allgemeine Schnittstelle für den Controller dar. Die AbstractGUI definiert Folgendes:

1. die Methoden zum Anzeigen von Dialogen in einer abstrakten Klasse (AbstractGUI),
2. die Datenstrukturen, die alle nötigen Informationen enthalten, um die Dialoge anzuzeigen (im Folgenden DATA-Objekte genannt), und
3. die Objekte, welche die GUI an den Controller zurückgibt, wenn ein Dialog durch den Benutzer abgearbeitet ist, und welche alle Daten über die Benutzereingabe enthalten (im Folgenden EVENT-Objekte genannt).

AbstractGUI Dies ist eine abstrakte Klasse, die genau für jedes Fenster der GUI eine Methode enthält. Diese Methode wird vom Controller aufgerufen und weist die GUI an, das Fenster anzuzeigen. Die Parameter dieser Methoden sind die DATA-Objekte, und als Ergebnis werden die EVENT-Objekte zurückgeliefert. Abbildung 4.8 zeigt die Klasse „AbstractGUI".

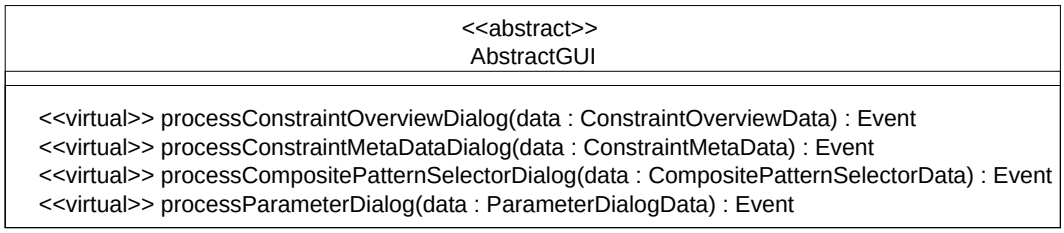


Abbildung 4.8.: Die Klasse „AbstractGUI"

DATA-Objekte Diese Objekte kapseln Daten, die für die Anzeige der Fenster benötigt werden. Zum Großteil stellen die DATA-Objekte eine Datenstruktur dar, die als Attribute vornehmlich Strings enthält. Diese DATA-Objekte werden vom Controller mit den Daten aus dem Datenmodell gefüllt und sind die Informationsquelle für die GUI.

EVENT-Objekte Diese Objekte enthalten die Informationen, die der Benutzer im Fenster angegeben hat. Das Event kann auch wiederum ein DATA-Objekt enthalten. Dies wurde dann passend mit Informationen durch die GUI gefüllt. Zur Vereinheitlichung von Events gibt es eine abstrakte Event-Klasse, die verschiedene Action Types in Form einer Enumeration definiert. Der Action Type stellt den Button dar, mit dem der Benutzer das Fenster bestätigt hat, also z.B. OK, Abbruch, Zurück, Neu, etc. Für jedes der sechs Fenster (wobei die Dialoge zur Wahl des Projektes und der Constraint Database von ein und derselben Klasse implementiert werden) erbt dann eine konkrete Event-Klasse, die neben dem Action Type noch weitere Informationen enthält:

1. ChooseProjectDialogEvent
   Enthält einen String, der den Namen des gewählten Projektes bzw. der gewählten Constraint Database enthält.

2. ConstraintOverviewEvent
   Beinhaltet einen String mit dem Namen der gewählten Constraint.

3. ConstraintMetaDataEvent
   Enthält ein DATA-Objekt (ConstraintMetaData).

4. CompositePatternSelectorEvent
   Enthält einen String, der den Namen des gewählten Composite Patterns enthält.

5. ParameterDialogEvent
   Enthält ebenfalls ein DATA-Objekt (ParameterDialogData).

Abbildung 4.9 zeigt die Modellierung der EVENT-Objekte, wobei die Darstellung der Übersicht halber auf Essentielles reduziert wurde.

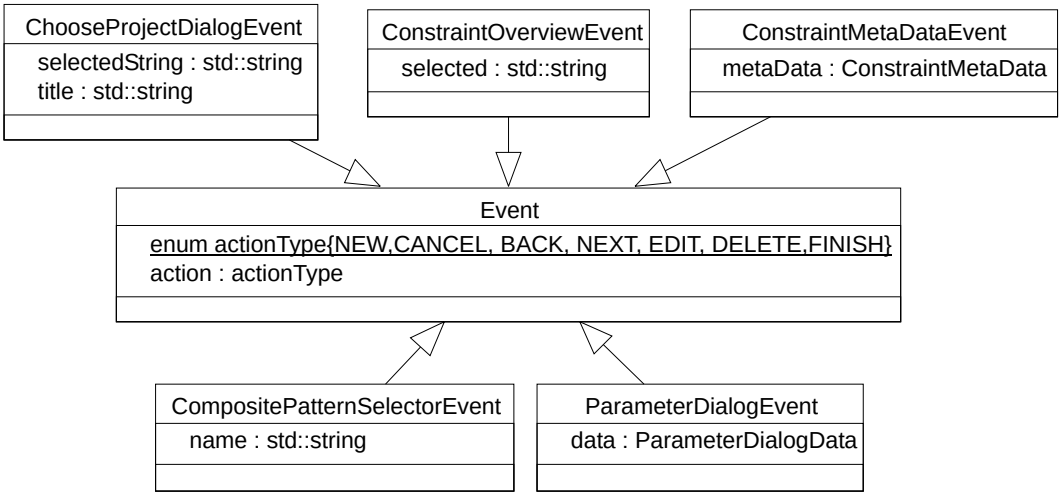


Abbildung 4.9.: Modellierung der EVENT-Objekte

4.2.3.3. Tcl/Tk GUI-Schicht

Die konkrete GUI wird mit Hilfe von Tcl/Tk implementiert und besteht aus den sechs Hauptfenstern (siehe oben), einer Schnittstelle zwischen Tcl/Tk und C++ sowie der Implementierung der AbstractGUI. Die vier Inspector-Klassen ConstraintOverviewInspector, ConstraintMetaDataInspector, CompositePatternSelectorInspector und ParameterDialogInspector stellen die Schnittstelle zwischen Tcl/Tk und C++ dar. Sie werden von der abstrakten Klasse CEInspector abgeleitet, die wiederum von MFTclWindow erbt, das die Schnittstelle zwischen Tcl/Tk und C++ im ABC ist und viele nützliche Methoden

zur Verfügung stellt. Die vier Inspector-Klassen sind zuständig für die Erzeugung der Dialoge und die Rückgabe der Events. Sie verfügen zu diesem Zweck über einen Konstruktor und die drei Hauptmethoden „update", „recall" und „getEvent". Die Tcl/Tk-GUI erbt von der AbstractGUI und implementiert sämtliche Methoden. Sie benutzt die vier Inspector-Klassen, um die Befehle aus dem Controller über die abstrakte GUI durchzuführen. Schließlich gibt sie ein Event zum Controller über die abstrakte GUI zurück, damit der Controller weiß, was er im nächsten Schritt weiter zu tun hat. Abbildung 4.10 zeigt einen Ausschnitt des Klassendiagrammes der konkreten GUI.

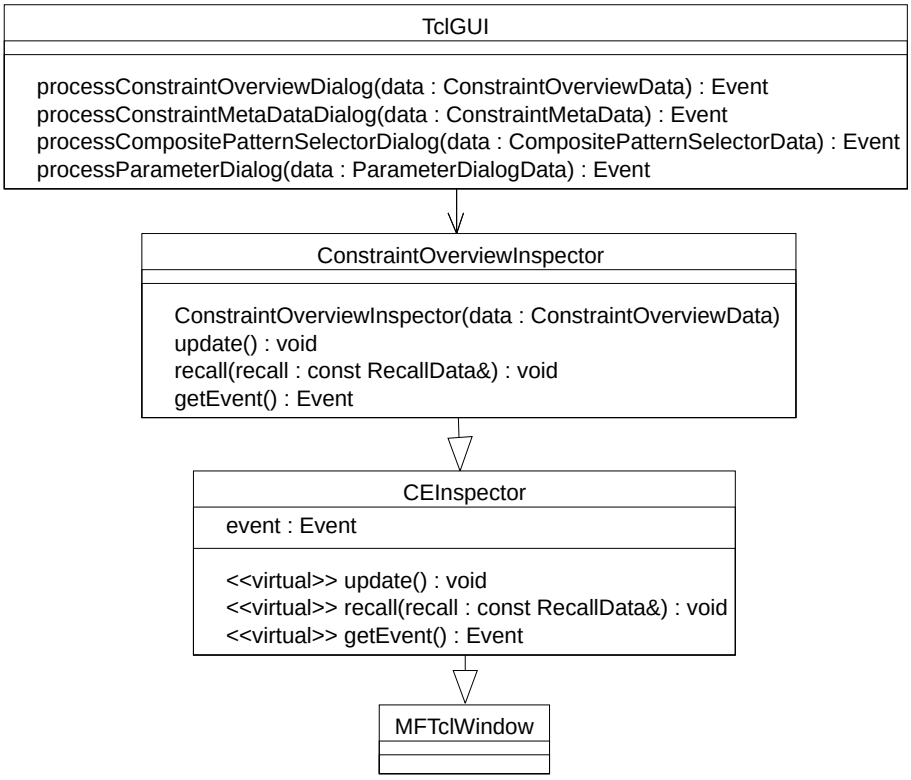


Abbildung 4.10.: Ein Ausschnitt der Tcl/Tk-GUI

4.2.3.4. Controller-Schicht

Das Herz der Controller-Schicht ist die Klasse ControllerKernel. Diese wird vom ABC instanziiert, wodurch der Constraint Editor gestartet wird. Da diese Klasse das Zentrum ist und alles steuert, gibt es auch keine andere Klasse die den ControllerKernel kennt. Die GUIFactory instanziiert die konkrete GUI (in unserem Fall also die TclGUI) und gibt eine Referenz auf die GUI zum ControllerKernel zurück. Diese Klasse ist auch die einzige Stelle, an der ein Eingriff in die „Nicht-GUI-Komponenten" nötig ist, falls eine andere Implementierung der GUI als Frontend für den Constraint Editor genutzt werden

soll. Abbildung 4.11 veranschaulicht die Anbindung der GUI an den Controller.

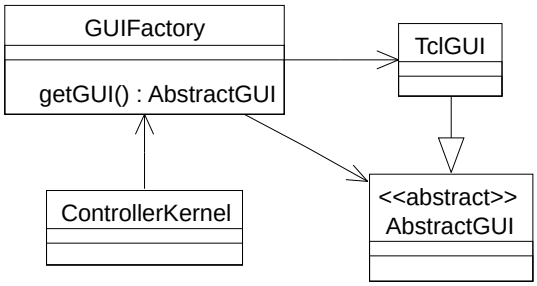


Abbildung 4.11.: Anbindung der GUI an den Controller

Die Klassen DataWriter und DataFetcher (siehe Abbildung 4.12) werden ebenfalls vom ControllerKernel instanziiert. Sie dienen zum Zugriff auf die Datenmodell-Schicht vom Kernel aus. Der DataWriter kann schreiben und mit Hilfe der Klasse CDBGenerator eine Constraint in der aktuellen Constraint Database generieren. Der DataFetcher holt die Daten aus dem Datenmodell und bereitet die DATA-Objekte für die GUI vor. Außerdem kann er die aus der GUI gewonnenen Daten so aufbereiten, dass daraus eine Constraint generiert werden kann. Die Speicherung und Generierung der Constraint wird von der Klasse CDBGenerator übernommen. Die Speicherung der Constraint im XML-Format (zur Ermöglichung einer späteren Editierung im Editor) wird durch den DataWriter angestoßen und vom Datenmodell selbst übernommen.

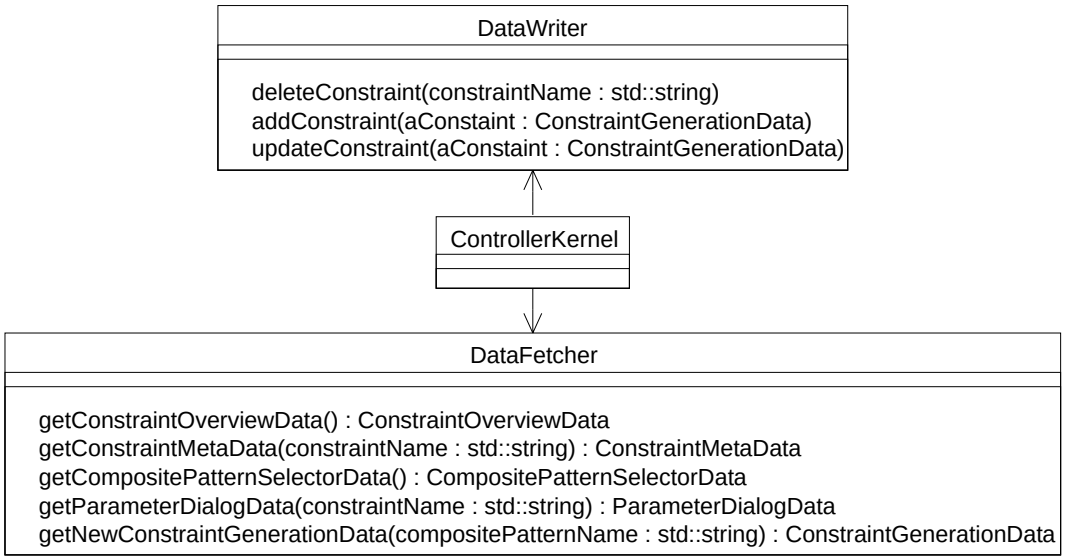


Abbildung 4.12.: ControllerKernel und DataWriter/Fetcher

DataWriter und DataFetcher kennen die Store-Objekte aus der Datenmodell-Schicht.

Die Store-Objekte stellen die Schnittstelle zum Datenmodell dar. Für genauere Informationen siehe 4.2.3.6 auf Seite 77.

4.2.3.5. XML-Datenbasis

Zum näheren Verständnis der Datenmodell-Schicht wird im Folgenden die dem Constraint Editor zu Grunde liegende XML-Datenbasis erläutert.

Logic Patterns, Composite Patterns und Constraints liegen für den Constraint Editor als XML-Dateien vor, deren Syntax durch DTDs beschrieben werden (siehe Anhang E auf Seite 121 für eine detaillierte Darstellung der verwendeten DTDs). Die vorhandenen Logic Patterns werden dabei in einer XML-Datei gespeichert, in der für jedes Logic Pattern die Formel spezifiziert wird, auf die das entsprechende Pattern abbilden soll. Codebeispiel 4.1 zeigt, wie ein Logic Pattern mit Hilfe von XML beschrieben wird:

```
1 <pattern name="response" scope="global">
2     <raw><![CDATA[(AG_F(']]></raw>
3     <p/>
4     <raw><![CDATA[ => AF_F(']]></raw>
5     <s/>
6     <raw><![CDATA[)))]]></raw>
7 </pattern>
```

Programmtext 4.1: Beispiel für eine Beschreibung eines Logic Patterns in XML

Neben den Metadaten des Logic Patterns wie Name und Geltungsbereich (im Beispiel: Response global) wird hier die entsprechende Formel mit ihren Platzhaltern angegeben. In Codebeispiel 4.1 ist dies die ESLTL-Formel $AG\_F('P \Rightarrow AF\_F('S))$, wobei P und S die Platzhalter der Formel sind. Beim eigentlichen Generierungsprozess ersetzt der Constraint Editor diese Platzhalter durch die SIBs, die der Benutzer ausgewählt hat.

Codebeispiel 4.2 auf der nächsten Seite zeigt, wie ein Composite Pattern mit Hilfe von XML spezifiziert wird:

```
<CompositePattern name="ResourceBalance" class="Resource">
  <patternDescription>...</patternDescription>

  <input>
    <step no="1">
      <stepDescription>
        Now please select P, which has to occur first.
      </stepDescription>
    </step>
    <step no="2">
       <stepDescription>
         Now please select S, which has to occur after P.
       </stepDescription>
    </step>
  </input>

  <generateLogicPattern>
    <logicPattern pattern="response" scope="global" type="single">
      <logicPatternDescription>
        P must always be followed by S.
      </logicPatternDescription>
      <sibs>
        <p>
          <name><advancedLink step="1">nameSib</advancedLink></name>
          <args><simpleLink step="1"/></args>
        </p>
        <s>
          <name><advancedLink step="2">nameSib</advancedLink></name>
          <args><simpleLink step="2"/></args>
        </s>
      </sibs>
    </logicPattern>
    <logicPattern pattern="precedence" scope="global" type="single">
      ...
    </logicPattern>
  </generateLogicPattern>

</CompositePattern>
```

Programmtext 4.2: Beispiel für eine Beschreibung eines Composite Patterns in XML

Neben der Angabe von Metadaten wie Name, Klasse und Beschreibung gliedert sich eine solche XML-Datei für ein Composite Pattern in zwei Teile:

1. Der erste Teil („input") beschreibt, welche Schritte notwendig sind, um alle Informationen darüber zu bekommen, wie die Platzhalter der entsprechenden Logic Patterns zu belegen sind. Für jeden Platzhalter muss ein Schritt („step") definiert werden, der eine eindeutige Nummer und eine natürlichsprachliche Beschreibung

besitzt. Letztere wird dem Benutzer angezeigt, um ihn zu informieren, was er im aktuellen Schritt tun soll.

2. Der zweite Teil („generateLogicPattern") beschreibt, wie die Eingaben des Benutzers auf die Platzhalter der Logic Patterns abgebildet werden sollen, was besonders für den Generierungsprozess von Bedeutung ist. Für jedes im Composite Pattern enthaltene Logic Pattern wird ein Eintrag vorgenommen („logicPattern"), der Metadaten wie Name, Geltungsbereich und natürlichsprachliche Beschreibung des Logic Patterns enthält. Außerdem muss an dieser Stelle für jeden Platzhalter des Logic Patterns angegeben werden, in welchem Schritt die für den entsprechenden Platzhalter relevanten Informationen gefunden werden können. Die Namen der Platzhalter müssen denen aus der XML-Beschreibung des Logic Patterns entsprechen, da diese dann während der Generierung verglichen werden, um die durch den Benutzer eingegebenen Informationen an die richtige Stelle in der Formel einsetzen zu können. Um die Verbindung zwischen den Schritten und den konkreten Platzhaltern herzustellen, werden zwei Konstrukte (sogenannte Links) verwendet:

   Simple Link Dieser Link bewirkt, dass alle Parameter des in dem entsprechenden Schritt ausgewählten SIBs durch den Benutzer spezifiziert werden können und somit für die Generierung herangezogen werden.

   Advance Link Dieser Link ermöglicht eine Vorbelegung gewisser Informationen, die dann durch den Benutzer nicht mehr editierbar sind. In den Zeilen 24 und 28 des Codebeispiels 4.2 auf der vorherigen Seite wird dieser Link z.B. verwendet, um den Namen des SIBs festzulegen, der während der Generierung verwendet werden soll.

Codebeispiel 4.3 auf der nächsten Seite zeigt, wie eine Constraint in XML beschrieben wird. Eine solche XML-Datei wird durch den Constraint Editor automatisch erstellt, wenn der Benutzer eine neue Constraint erzeugt. An dieser Stelle befinden sich alle Informationen, die ein späteres erneutes Editieren und Generieren der Constraint ermöglichen. Da eine Constraint im CDB-Format im Wesentlichen nur Name, Beschreibung und die zugehörige Formel enthält, würde ohne die zusätzliche Speicherung im XML-Format z.B. die Information über das verwendete Composite Pattern verloren gehen.

```
<constraint pattern="ResourceBalance" name="BalancedResourceConstraint">
  <description>...</description>
  <step no="1">
    <nameSib>CheckLogin</nameSib>
    <parameter>
      <name>name</name>
      <content>admin</content>
    </parameter>
    <parameter>
      <name>password</name>
      <content>admin</content>
    </parameter>
  </step>
  <step no="2">
    <nameSib>CreateStorableEntity</nameSib>
  </step>
</constraint>
```

Programmtext 4.3: Beispiel für eine Beschreibung einer Constraint in XML

Neben wichtigen Metadaten wie dem verwendeten Composite Pattern, Name und Beschreibung der Constraint enthält eine solche XML-Datei zusätzlich Informationen, welche SIBs in den einzelnen Schritten ausgewählt wurden, und wie deren Parameter durch den Benutzer spezifiziert wurden. In Codebeispiel 4.3 wurde in Schritt 1 z.B. der SIB „CheckLogin" ausgewählt, dessen zwei Parameter „name" und „password" beide mit dem Wert „admin" belegt wurden.

4.2.3.6. Datenmodell-Schicht

Das Datenmodell dient dem Einlesen und der Aufbereitung der im Constraint Editor benötigten Daten. Unter diesen Daten sind fünf verschiedene Entitäten zu unterscheiden:

- die Composite Patterns,
- die XML-Constraints,
- die CDB-Constraints,
- die Logic Patterns und
- die SIBs.

Nach außen bilden fünf Klassen (pro Entität eine Klasse) mit dem Suffix „Store" die Schnittstelle des Datenmodells. Sämtliche Kommunikation anderer Komponenten (z.B. des Controllers) erfolgt über diese fünf Klassen. Im Folgenden werden die Entitäten des Datenmodells beschrieben, wobei exemplarisch auf den CompositePatternStore im Detail eingegangen wird.

CompositePatternStore Der CompositePatternStore dient dem Zugriff auf die zur Verfügung stehenden Composite Patterns. Wie in 4.2.3.5 auf Seite 74 beschrieben liegen diese im XML-Format vor und werden mit der Bibliothek Libxml2 [15] entsprechend eingelesen. Der CompositePatternStore hält dann die Menge aller verfügbaren Klassen von Composite Patterns, die durch ihren Namen eindeutig bestimmt sind. Jedes Objekt vom Typ CompositePatternClass bietet verschiedene Zugriffsmethoden auf die zugehörigen Composite Patterns. Abbildung 4.13 auf der nächsten Seite zeigt die Modellierung der Composite Patterns, wobei zu Gunsten der Übersicht auf Konstruktoren sowie get- und set-Methoden verzichtet wurde.

Ein Objekt vom Typ CompositePattern hält neben einigen Metadaten (Name, Beschreibung) eine Menge von Eingabeschritten sowie eine Menge von Regeln zur Generierung der Logic Patterns, die durch Konjunktion das Composite Pattern bilden. Die Eingabeschritte (Klasse InputStep) sind die Schritte, die in der GUI nötig sind, um die Parameter eines Composite Patterns zu spezifizieren, d.h. nach Durchlauf dieser Schritte sind alle nötigen Daten durch den Benutzer angegeben worden. Jeder Schritt besteht aus einer Nummer, einer Beschreibung sowie der Angabe eines Dialoges, der in der GUI für diesen Schritt angezeigt werden soll (im Falle der SIBs: werden die Parameter der SIBs angezeigt oder nicht?).

Eine Regel zur Generierung (Klasse LogicPatternRule) eines Logic Patterns enthält zunächst einige Metadaten, vor allem aber den Namen und den Geltungsbereich, wodurch das Logic Pattern eindeutig bestimmt ist. Weiterhin enthält eine LogicPatternRule eine Menge von Angaben, wie die einzelnen Parameter des entsprechenden Logic Patterns zu belegen sind. Pro Parameter werden diese Angaben durch ein Objekt vom Typ LPParameterTemplate modelliert.

Im Falle der SIBs gibt es nun verschiedene Möglichkeiten, z.B. kann der für einen Parameter einzusetzende SIB schon in der XML-Datenbasis fest angegeben sein. Ist dies nicht der Fall, wird diese Angabe von der GUI erwartet. Auch für die Argumente eines SIBs gibt es ähnliche Varianten (pro Argument noch einmal modelliert durch die Klasse ArgumentTemplate). Um diese verschiedenen Möglichkeiten nun leicht änder- und erweiterbar zu halten, wurden diese in Form von enum-Types in der globalen Resources-Schicht (siehe 4.2.3.7 auf Seite 80) abgelegt, so dass zentrale Parameter schnell angepasst werden können.

Diese Modellierung der Composite Patterns spiegelt recht „maßstabsgetreu" die Konzepte der XML-Datenbasis wider, wie sie in [14], S.70/71 beschrieben ist. Für tiefergehendes Verständnis hier nicht erläuterter Details, die an dieser Stelle den Rahmen sprengen würden, sei dorthin verwiesen.

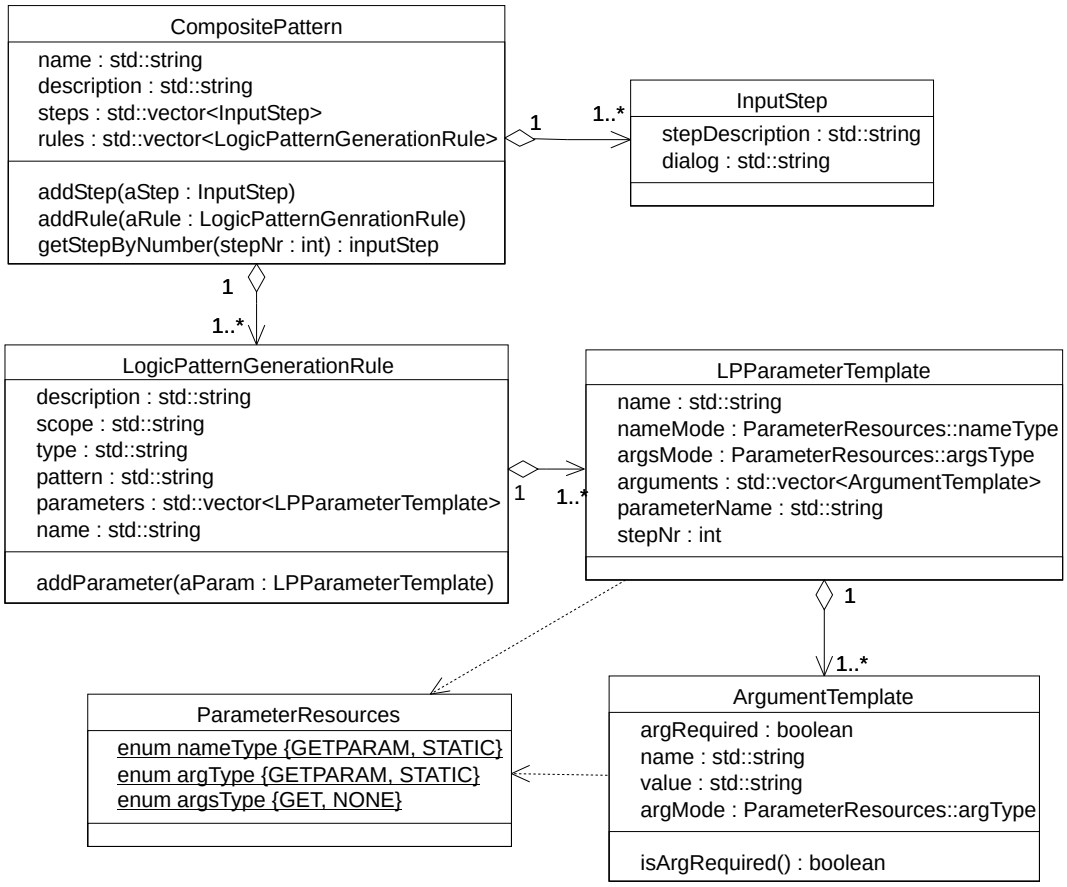


Abbildung 4.13.: Modellierung der Composite Patterns

ConstraintStore Der ConstraintStore ist die Schnittstelle zu den Constraints, die in XML-Dateien zur Verfügung stehen. Da diese XML-Constraints im eigentlichen Sinne ja noch keine fertigen Constraints sind (sie dienen lediglich der Speicherung in der Editor-eigenen Datenbasis, um eine spätere Editierung zu ermöglichen), wurde für die repräsentierende Klasse der Name *ConstraintGenerationData* gewählt. In einem solchen Objekt befinden sich alle Daten, die neben den Vorlagen für die entsprechenden Patterns zur Generierung der Constraint notwendig sind. Dies sind z.B. Metadaten wie Name und Beschreibung der Constraint, aber auch die Angabe des verwendeten Composite Patterns samt Belegung der Parameter.

Die Parameter sind wiederum in Form von Steps festgehalten, deren Nummern und Reihenfolge mit denen der Steps im Composite Pattern übereinstimmen (Klasse ConstraintStepData). Jeder Step enthält den Namen des entsprechenden Parameters sowie eine Menge von Name-Content-Paaren als Parameterwert. Ein solches Name-Content-Paar ist in der Klasse ConstraintParameterData modelliert, deren verschiedene Modi erneut in Form von enum-Types in die Resources-Schicht ausgelagert sind, um die Flexibilität

zu erhalten. Im Falle der SIBs kann ein solcher Parameter z.B. mit einem konkreten Wert belegt werden oder dem All- bzw. Existenzquantor untergeordnet werden. Abbildung 4.14 zeigt dieses Konzept.

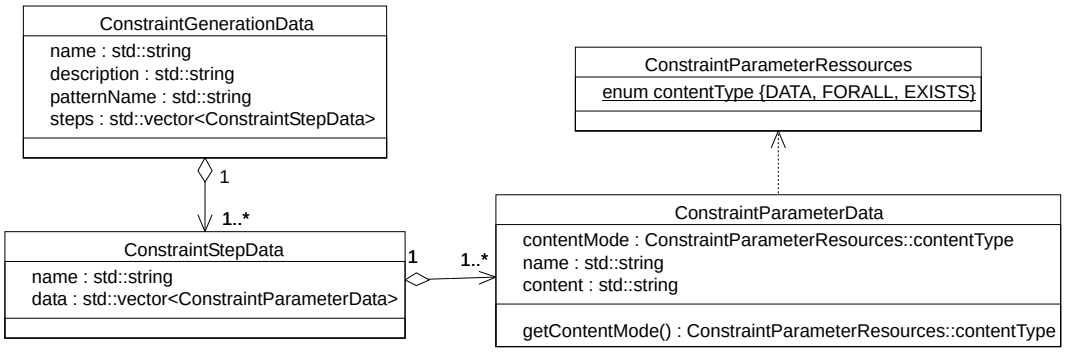


Abbildung 4.14.: Modellierung der XML-Constraints

CDB-Store und SibStore Der CDB-Store und der SibStore bieten Zugriffsmethoden für die Constraints im CDB-Format und für die SIBs. Beide Entitäten werden aus dem aktuellen ABC-Projekt gewonnen und sind durch je ein einzelnes Objekt modelliert.

LogicPatternStore Der LogicPatternStore stellt die Schnittstelle zu den Logic Patterns dar, die im XML-Repository gespeichert sind. Hier befinden sich die konkreten Formeln, die zur Generierung benötigt werden. Die Parameternamen werden mit den Angaben in den Objekten vom Typ ConstraintStepData verglichen, um festzustellen, an welcher Stelle die von dem Benutzer spezifizierten Werte in die Formel eingesetzt werden müssen.

4.2.3.7. Resources-Schicht

Die Resources-Schicht ist eine global (d.h. von allen anderen Komponenten) zugreifbare Schicht. In den enthaltenen Klassen ParameterResources und ConstraintParameterResources wurden verschiedene Typisierungen ausgelagert. Beispiele:

- Sind Namen und Parameter der SIBs schon im XML angegeben oder müssen sie durch den Benutzer spezifiziert werden?
- Welche Art Inhalt hat der Parameter einer Constraint (hat der Parameter schon einen konkreten Wert oder wird er dem Existenz- oder Allquantor zugeordnet)?

Derlei Modi sind somit an zentraler Stelle änderbar.

4.2.4. Beispiel

Um die Funktionalität des Constraint Editors zu validieren, wurde dieser mit der Literatur-Datenbank (siehe 2 auf Seite 16) getestet. Das Bindeglied dieser Tests war der im

ABC integrierte Model Checker, der die vom Constraint Editor generierten Constraints verwendet, um den SLG der Literatur-Datenbank zu prüfen.

In diesem Zusammenhang förderten die Tests Probleme mit dem Model Checker zu Tage. Zum einen kommt dieser nicht mit bestimmten Benutzereingaben für den konkreten Wert eines SIB-Parameters zurecht, nämlich Strings mit Leerzeichen und/oder doppelten Unterstrichen. An dieser Stelle ist das volle Potential des Constraint Editors also noch nicht ausschöpfbar. Zum anderen liefert der Model Checker oft unlogische Ergebnisse: auch augenscheinlich erfüllte, sehr einfache Constraints werden oft als nicht erfüllt zurückgewiesen. Selbst nach Negierung der zu einer solchen Constraint gehörigen Formel bleibt die Antwort des Model Checkers gleich. Dies lässt sich auch mit von Hand eingegebenen, nicht mit dem Constraint Editor erstellten Constraints rekonstruieren.

Dennoch soll folgender kurzer Beispielablauf den Hergang eines erfolgreichen Tests skizzieren:

1. Zunächst wird der Constraint Editor gestartet. Als Projekt wird eine kleine Version der Literatur-Datenbank („LiteraturSmall") gewählt, (siehe Abbildung 4.2 auf Seite 66) als Constraint Database „literatur", welche zu Beginn der Tests im ABC manuell angelegt wurde.

2. Im Constraint Overview-Dialog (siehe Abbildung 4.3 auf Seite 67) folgt ein Klick auf „New", um eine neue Constraint anzulegen.

3. Im folgenden Dialog (siehe Abbildung 4.4 auf Seite 67) erhält die Constraint den Namen „PlainPrecedence" sowie die Beschreibung „P should always be preceeded by S.".

4. Als assoziiertes Composite Pattern wird im nächsten Schritt „SimplePrecedence" aus der Klasse „EverythingSimple" gewählt (siehe Abbildung 4.5 auf Seite 68).

5. Im letzten Dialog (siehe Abbildung 4.6 auf Seite 68) werden nun die Parameter der Constraint spezifiziert. Für Step Nummer 1 wird der SIB „GetPublication" gewählt, der keine weiteren Parameter hat. In Step Nummer 2 wird dann „CheckLogin", deren beide Parameter „name" und „password" beide mit dem konkreten Wert „admin" belegt werden.

6. Nach Klick auf „Finish" wird eine Constraint generiert, die natürlichsprachlich ausgedrückt Folgendes von der Literatur-Datenbank verlangt: Bevor eine Publikation eingelesen werden kann, muss man sich immer eingeloggt haben, und zwar mit „admin" als Benutzername und Kennwort.

7. Die neue Constraint wird nun im Model Check-Fenster des ABC dem SLG der Literatur-Datenbank-Applikation hinzugefügt. Wenn nun die Validierung mittels Model Checker gestartet wird, so meldet dieser, dass die angegebene Constraint erfüllt ist.

Dies war ein sehr einfaches Beispiel für einen Test. Im produktiven Betrieb soll der Constraint Editor natürlich die Generierung wesentlich komplexerer Constraints unterstützen.

4.3. Fazit

Abschließend betrachtet konnte der Constraint Editor den Anforderungen (siehe 4.2.1 auf Seite 63) entsprechend realisiert werden. Er ist als eigenständiges Modul im ABC einsetzbar und somit verfügbar für viele mögliche Anwendungsbereiche, in denen Constraints benötigt werden. Das Design des Editors sieht eine hohe Flexibilität des Moduls vor. So ist z.B. die Implementierung der graphischen Benutzeroberfläche dank einer abstrakten Schnittstelle von den restlichen Komponenten des Constraint Editors gänzlich entkoppelt, so dass generell eine GUI in jeder beliebigen Programmiersprache denkbar wäre. Die Funktionalität des Editors ist in anderen Worten unabhängig davon, welches Frontend für die Benutzereingaben bei der Constraintgenerierung verwendet wird.

Speziell im ABC wird der Constraint Editor die Verwendung des Model Checkers erleichtern. Ohne den Constraint Editor müssen Constraints hier noch „zu Fuß" erstellt werden, d.h. die zugehörigen temporal-logischen Formeln müssen durch den Benutzer selbst in Gänze spezifiziert werden. Der Constraint Editor vereinfacht diesen Ablauf und verlangt dem Benutzer weniger ab: dem Erlernen von ESLTL (oder einer anderen Spezifikationssprache) steht ein ungleich kürzeres Vertrautmachen mit den verschiedenen Patterns gegenüber.

Wichtig ist auch, dass sich besonders anfangs viele Benutzer in der Erstellung von Composite Patterns betätigen. Solche ergeben sich meistens aus der konkreten Anwendung, für die Constraints benötigt werden. Composite Patterns werden durch direktes Editieren der XML-Datenbasis des Editors erstellt - auch an dieser Stelle wäre sicherlich in Zukunft eine Editierung über eine GUI denkbar. Eine gute Datenbasis von sinnvollen und anwendungsnahen Composite Patterns, die im Optimalfall bedarfsgesteuert anwächst, erleichtert die Verwendung des Constraint Editors.

5. Views

5.1. Überblick

In diesem Kapitel wird die Entwicklung eines Moduls zur Erzeugung und Verwaltung von Views (Abstraktionen von SLGs) beschrieben. Nach diesem kurzen Überblick folgt die detaillierte Beschreibung der Entwicklung des Moduls mit Anforderungsanalyse, Entwurf und Implementierung. Abschließend wird ein kurzes Fazit zum Viewmanager gegeben.

5.1.1. Motivation

Die Web-Applikationen, die mit dem ABC und EWIS erstellt werden, können sehr groß werden. Service-Logic-Graphen (SLG) mit über 50000 Knoten sind dabei keine Seltenheit. Die Modellierung solcher Applikationen als dynamische Systeme macht die Benutzung eines Modul-Konzepts zur Wiederverwendung von Code-Teilen schwierig. In allen Phasen des Entwicklungsprozesses, also Konzeptions-, Implementierungs- und Wartungsphase, stellt sich das Problem, dass der einzelne Entwickler die Komplexität nicht bewältigen kann. Um sich also auf einzelne Aspekte einer Applikation zu konzentrieren, wäre eine Abstraktion auf ein intellektuell beherrschbares Maß wünschenswert. Ein Entwickler könnte zum Beispiel nur an der Kommunikation der Applikation mit einer Datenbank interessiert sein, und möchte alle dabei unwesentlichen Knoten ausblenden. Das Problem, das innerhalb der Projektgruppe gelöst werden sollte, war, aus einem SLG einen „abstrahierten“ Graphen zu erstellen, der aber weiterhin ein gültiger SLG sein sollte. Diesen abstrahierten Graphen sollte man dann beispielsweise testen oder animieren können (siehe dazu Abschnitte 3.2 auf Seite 32 und 3.3 auf Seite 43).

5.1.2. Konzept

Unter einem View wird also hier die Abstraktion eines SLGs (oder allgemeiner eines reaktiven Systems) verstanden. Dieses Kapitel beschäftigt sich mit der Erstellung eines Moduls für das ABC, welches die Erstellung, Anzeige und Verwaltung von verschiedenen Views realisiert. Zur Berechnung solcher Views werden im folgenden verschiedene Algorithmen vorgestellt, die aus der Automatentheorie oder der Theorie verteilter Systeme stammen. Dann wird das ABC-Modul „Viewmanager“ mit seinen Komponenten beschrieben.

5.1.3. Ziele

Das grobe Ziel war die Entwicklung von Algorithmen zur Erstellung von Views und einem Modul, mit dem man die Views erzeugen und betrachten kann. Mit dem „Viewmanager“

ist genau dies möglich. Zur Erstellung eines Views wird ein vorhandener SLG geladen, aus den entwickelten Algorithmen eine Konstruktions-Vorschrift für den View erstellt, und diese dann auf den SLG angewendet.

Da ein View ein eigenständiger SLG ist, bedeutet dies, dass Änderungen an einem View, z.B. das Hinzufügen von Knoten oder Kanten, keine Änderungen am Ursprungsgraphen nach sich ziehen. Diese wünschenswerte Anforderung wäre aber auch um einiges schwieriger zu realisieren, da die Korrespondenzen zwischen View und Applikation verwaltet werden müssten. Daher wurde diese Anforderung nicht in den Katalog der zu realisierenden Anforderungen aufgenommen.

5.2. Beschreibung

5.2.1. Formale Grundlagen

Dieser Abschnitt beschreibt die formalen Grundlagen, die zum Verständnis der folgenden Abschnitte notwendig sind.

5.2.1.1. Die Problemstellung

Wie schon festgestellt wurde, ist die Größe eines Graphen, der eine Internet-Applikation repräsentiert, ein großes Problem. Um auf einem solchen Graphen arbeiten zu können, bedarf es geeigneter Abstraktionen, die *Views* genannt werden. Wie aber berechnet man Views und welche Verfahren bzw. Algorithmen können dafür verwendet werden? Dieses soll in den folgenden Abschnitten diskutiert werden, wobei einige Vorschläge vorgestellt werden, wie man Views berechnen kann. Vorher werden noch drei Strukturen vorgestellt, mit deren Hilfe man den Graphen modellieren kann und die als Grundlage für verschiedene Algorithmen dienen.

5.2.1.2. Das Modell

Eine Internet-Applikation, die mit Hilfe eines SLGs erstellt wurde, ist ein reaktives System, also ein System, was nicht notwendigerweise terminiert bzw. ein abschließendes Ergebnis liefert, sondern im Prinzip endlos läuft und von internen Zuständen abhängt. Solch ein System muss normalerweise in ein Modell umgewandelt werden, um für diverse Algorithmen (wie z.B. der Model-Checking-Algorithmus) handhabbar zu sein. Dies ist hier jedoch nicht notwendig, da es dieses Modell in der Form des SLGs schon gibt. Dieses Modell kann als Kripke-Transitionssystem (KTS) , Kripke-Struktur (KS) oder gelabeltes Transitionssystem (LTS) gesehen werden.

Definition *Kripke-Strukturen*: Eine Kripke-Struktur $KS = (S, s_0, AP, R, L)$ ist ein Fünftupel mit folgenden Eigenschaften:

- S ist eine Menge von Zuständen
- s_0 ist der Start-Zustand

- AP ist die Menge der atomaren Propositionen
- $R \subseteq S \times S$ ist eine Menge von Übergängen (Transitionen)
- $L : S \to 2^{AP}$ ist die Benennungsfunktion, die jedem Zustand eine Menge von Atomaren Propositionen zuordnet.

Oft möchte man aber auch Beschriftungen an den Übergängen haben und nicht bzw. nicht nur an den Zuständen. Deswegen hat man die gelabelten Transitionssysteme und die Kripke-Transitionssysteme eingeführt.

Definition *gelabelte Transitionssysteme*: Ein gelabeltes Transitionssystem $LTS = (S, s_0, Act, R)$ ist ein Viertupel mit folgenden Eigenschaften:

- S ist eine Menge von Zuständen
- s_0 der Start-Zustand
- Act ist eine Menge von Aktionen (also Labels für die Transitionen)
- $R \subseteq S \times Act \times S$ ist die Menge von gelabelten Transitionen

Eine Konsolidierung von Kripke-Strukturen und gelabelten Transitionssystemen ergibt Kripke-Transitionssysteme:

Definition *Kripke-Transitionssysteme*: Ein Kripke-Transitionssystem $KTS = (S, s_0, AP, Act, R, L)$ ist ein Sechstupel mit den folgenden Eigenschaften:

- S ist eine Menge an Zuständen
- s_0 ist der Start-Zustand
- Act ist eine Menge von Aktionen (also Labels für die Transitionen)
- $R \subseteq S \times Act \times S$ ist die Menge von gelabelten Transitionen
- AP ist die Menge der atomaren Propositionen
- $L : S \to 2^{AP}$ ist die Benennungsfunktion, die jedem Zustand eine Menge von Atomaren Propositionen zuordnet.

5.2.1.3. Die Spezifikation

Die Spezifikation wird in temporaler Logik angegeben. Im Gegensatz zu der klassischen Logik, in der Aussagen global (zeit- und ortsunabhängig) gültig sind, wird bei der temporalen Logik die Zeit als zusätzlicher Parameter spezifiziert. Dabei sollen Konzepte wie z.B. „irgendwann“ „nie“ oder „danach“ spezifizierbar sein. Dafür gibt es Operatoren, die im folgenden vorgestellt werden:

G f	„globally“-Operator	„f gilt ab jetzt immer“
X f	„next“-Operator	„f gilt im nächsten Schritt“
F f	„finally“-Operator	„f wird irgendwann auf jeden Fall gelten“
f U g	„until“-Operator	„g wird gelten, davor gilt f“
f W g	„weak until“-Operator	„g wird gelten, davor gilt f, oder f gilt immer“
f R g	„release“-Operator	„g wird gelten, davor gilt f, oder g gilt immer“

Es gibt im wesentlichen zwei Logiken: Linear Time Logic (LTL) und Branching Time Logic (CTL), von denen nur LTL hier behandelt werden soll.
Die Grammatik in BNF für Linear Time Logic (LTL) sieht folgendermaßen aus:
$\phi ::= p \mid -\phi \mid \phi_1 \vee \phi_2 \mid X\phi \mid \phi_1 U \phi_2$

Beispiele:

1. F rain: es wird regnen
2. sun U rain: es scheint die Sonne, bis es regnet
3. G rain: es wird immer regnen
4. sun W rain: es scheint die Sonne bis es regnet (aber es muss nicht unbedingt regnen)

Die Grammatik beinhaltet nicht die Operatoren G,R,W und F, da diese mit den restlichen Operatoren gebildet werden können:

$$Gf \Leftrightarrow \neg F \neg f$$
$$Ff \Leftrightarrow trueUf$$
$$fWg \Leftrightarrow (fUg) \vee f$$
$$fRg \Leftrightarrow \neg(\neg fU \neg g)$$

Durch das geeignete „Ineinanderschachteln“ von Modalitäten lassen sich gewünschte Eigenschaften formulieren.

Wenn man allerdings nicht in temporaler Logik bewandert ist, kann es recht schwierig bzw. kompliziert sein, geeignete Formeln für informelle Sachverhalte zu finden. Deswegen gibt es z.B. Patterns, die einem bei der Formulierung helfen, und die man z.B. mit Hilfe des Constraint Editors (s. Kapitel 4 auf Seite 60) erstellen kann. Patterns decken viele Fälle ab, die oft bei der Arbeit mit temporal-logischen Formeln benötigt werden (z.B. dass ein Zustand immer vor einem anderen sein muss).

5.2.1.4. Der Model-Checking-Algorithmus

Hier soll nun nicht der eigentliche Algorithmus beschrieben werden, sondern nur noch die allgemeine Vorgehensweise. Man übergibt dem Algorithmus als Parameter ein Modell und eine Spezifikation in temporaler Logik. Die Ausgabe ist *true*, wenn das Modell die Spezifikation erfüllt. Im anderen Fall gibt der Algorithmus einen (oder alle) Pfad(e) im Graphen zurück, die die Spezifikation verletzt/verletzen. Das Ganze geschieht in einer

Laufzeit von O(|K|*|Teilformeln|), sie hängt also von der Größe des Modells (|K|, definiert als die Summe von Knoten und Kanten) und der Anzahl der Teilformeln der Spezifikation ab. Es ist also wünschenswert, wenn das Modell nicht zu groß ist und die Anzahl der Teilformeln eher gering.

5.2.1.5. View-Generierung mit Model-Checking

Wie nun hilft einem das Model-Checking-Verfahren bei der Generierung von Views? Model Checking ist, wie schon erwähnt, eine Verifikationsmethode und hat mit Views generell gesehen nicht viel zu tun. Zur Erinnerung: Views sind Abstraktionen von Graphen. Vorteilhaft für die Erstellung und das Warten von solchen Graphen ist es, wenn die Abstraktionen gewissen Semantiken folgen, die sich aus der Anwendung des Graphen ergeben. Zur Verdeutlichung ein kleines Beispiel: Bei einer Internet-Applikation, die mit Hilfe eines Graphen gebildet wird, interessiert sich der Entwickler nur für den Login-Teil. Also wäre es vorteilhaft, wenn der View nur diesen Teil anzeigen würde und die anderen Teile ausblendet. Für dieses Szenario könnte man sich eine temporal-logische Formel spezifizieren, die nur für den Login-Bereich gilt. Wenn man nun den Model-Checking-Algorithmus laufen lässt, gibt er als Fehler nun den Rest des Graphen (also den Graphen minus der Login-Komponente) als sog. Error-View aus. Also muss man den entgegengesetzten Weg gehen: Man negiert die oben genannte Formel und startet den Model-Checking-Algorithmus. Anschließend wird nur die Login-Komponente als Error-View angezeigt.

5.2.2. Der Prototyp

Im ersten Semester wurde zunächst ein einfacher Prototyp entworfen und implementiert. Dies diente einerseits zur Einarbeitung in die komplexe METAFrame-Entwicklungsumgebung, andererseits als Machbarkeitsstudie für ein einfaches Konzept zur Viewgenerierung. Der Prototyp wurde als Modul für das Agent-Building-Center erstellt. Er basiert auf der PLGraph-Bibliothek, die generische Datenstrukturen für gelabelte Transitionssysteme zu Verfügung stellt. Die einzelnen Schritte der Viewgenerierung sind separat durchführbar und können so leichter getestet werden. Der Prototyp sah die Generierung eines Views nach folgendem Schema vor:

Transformation Um Standard-Algorithmen wie z.B. die tau-Elimination benutzen zu können, die keine Kanten- oder Knotenlabels bzw. Namen von Knoten/Kanten berücksichtigen, ist es nötig, eine Transformation vorzunehmen, die die Informationen an den Knoten auf zusätzlich vom Algorithmus eingebaute Kanten überträgt.

Relabelling Relabelling bedeutet, die Labels/Namen von Kanten/etc. umzubenennen. So fallen dann äquivalente Knoten des Graphen zu Äquivalenzklassen zusammen, wenn man den Minimierungsalgorithmus aufruft. Ebenso können statt der Knoten die Kanten auf diese Weise zusammengefasst werden. Das Relabelling geschieht entweder manuell oder syntaktisch. Mögliche syntaktische Eigenschaften sind Klasse,

Name, evtl. IDs. Beim syntaktischen Verfahren wird anhand syntaktischer Eigenschaften eine Menge an Knoten ausgewählt und zu „tau“ umbenannt. Zum Beispiel könnte man alle internen Knoten auswählen und umbenennen. Anschließend muss die tau-Elimination aufgerufen werden. Dadurch lässt sich ein View generieren, der lediglich diejenigen Knoten darstellt, die den logischen Ablauf beschreiben, ohne sich in Details der Implementierung zu verlieren.

tau-Elimination Die tau-Elimination bezeichnet ein Verfahren, bei dem alle Kanten im Graph, die das Label bzw. den Namen „tau“ besitzen, gelöscht werden, und das die eingehenden und ausgehenden Kanten entsprechend anpasst. In manchen LTS gibt es so genannte tau-Kanten, welche interne Aktionen darstellen, die nicht beobachtet werden können.

Minimierung Wenn aus einem Modell zunächst die Zustände, die vom Anfangs-Zustand aus nicht erreichbar sind, entfernt werden und dann daraus der Äquivalenzklassenautomat $\tilde{A}$ konstruiert wird (Hopcraft [11]), ist $\tilde{A}$ ein minimaler zu A äquivalenter endlicher Automat. Somit werden die redundanten Zustände reduziert. Zum einen spart man Platz und zum anderen kann das bei einem großen Graphen von der Übersichtlichkeit her enorme Auswirkungen haben. Der reduzierte Automat bewahrt wiederum seine vorherigen Eigenschaften. Mögliche Minimierungsalgorithmen sind z.B. Namensminimierung (Knoten mit gleichen Namen werden zu Äquivalenzklassen zusammengefasst.), Sprachminimierung (siehe Algorithmus 4.2.6 zur DFA-Zustandsminimierung in [20]) oder Algorithmen zur Branching Time Minimierung (siehe dazu die Artikel von Paige/Tarjan [16] und Fernandez [8]). Die Minimierungs-Algorithmen nehmen als Eingabe einen Graphen und fassen eine Knotenmenge (also z.B. Knoten mit gleichem Namen) in einen Knoten, der *Meta-Knoten* genannt wird, zusammen.

Die PLGraph-Bibliothek unterscheidet zwischen Namen und Label der Graph-Objekte (Knoten, Kanten, Graphen). Label sind viel allgemeiner als Namen definiert und können beliebige Objekte enthalten. Beispielsweise werden Graphalgorithmen üblicherweise als Graphlabel implementiert.

Die Erzeugung von Views auf PL-Graphen wurde somit rudimentär als Modul für das ABC realisiert. Die Erweiterung auf SLGs hat sich später als sehr viel problematischer herausgestellt als zunächst angenommen (siehe dazu Abschnitt 5.2.4.8 auf Seite 104). Mit dem Prototypen wurde sichergestellt, dass die Erstellung von Views nach diesem Schema grundsätzlich funktioniert. Dabei wurden Views auf einfachen PLGraphen erstellt. Im Endprodukt sollten beliebige Algorithmen zur Erstellung von Views unterstützt werden, außerdem sollten SLGs als Ursprungsgraphen benutzt werden. Um dies zu gewährleisten, ist eine Konvertierung notwendig, die innerhalb des Viewmanagers ein eigener Algorithmus (SIBConvert) übernimmt (dazu später mehr: Abschnitt 5.2.4.3 auf Seite 91).

5.2.3. Anforderungen

Aus den Erfahrungen, die mit dem Prototyp gesammelt wurden, ergaben sich einige Änderungen an den ursprünglich aufgestellten Anforderungen: Zunächst sollte das Modul

auf SLGs arbeiten, damit die Entwicklung und Wartung verschiedener Applikationen, die mit dem ABC realisiert wurden, konkret unterstützt wird. Die Algorithmen des Prototyps haben sich als tragfähig erwiesen. Zusätzlich gibt es aber einige Algorithmen, die außerhalb der PG entwickelt wurden, und sich ebenfalls zur Berechnung von Views eignen. Dies führte dazu, dass der Schwerpunkt von der Entwicklung verschiedener Algorithmen zur Entwicklung eines Rahmenwerks zur Verwaltung von Views verschoben wurde. Es sollten die existierenden Algorithmen aus dem Prototypen und die oben genannten Externen leicht in das Modul zu integrieren sein. Außerdem sollte ein einheitlicher Viewmanager--Dialog für die Auswahl und Zusammenstellung von neuen Views realisiert werden, die es auch möglich macht, diese Zusammenstellungen persistent zu machen, um sie nachträglich auch auf unterschiedlichen SLGs anzuwenden. Dem Benutzer sollte weiterhin die Möglichkeit geboten werden, einen View einfach nach verschiedenen Kriterien (z.B. SIB-Id, SIB-Klasse) zu erstellen.

Andere Anforderungen haben sich dagegen als nicht realisierbar herausgestellt: Die Verwaltung der Information, die verloren geht, wenn ein View erstellt wird, wäre wünschenswert gewesen, um beispielsweise aus einem vorhandenen View wieder den Ursprungsgraphen zu erstellen. Ein anderer Verwendungszweck dieser Information ist auch die Realisierung von Editieroperationen auf dem View, die dann auch auf dem Ursprungsgraphen wirksam sind. Ansatzweise wurde dies zwar realisiert, aber es ist bislang nur möglich, sich zu einem vorhandenen Meta-Knoten (im View) die Ursprungsknoten anzeigen zu lassen. Die Verwertung dieser Information zur Editierung oder der Realisierung einer „Undo-Funktionalität“ hat sich leider als zu komplex für unseren Rahmen dargestellt, weshalb sie zu Beginn des zweiten Semesters gestrichen wurden. Das Hauptproblem bestand darin, dass der View je nach angewendetem Algorithmus eine andere Semantik haben kann; möglicherweise kann der SLG auch nicht mehr in gewohnter Weise durchlaufen werden, wenn beispielsweise durch eine Tau-Elimination Knoten bzw. Kanten verschwunden sind. Außerdem liefern nicht alle externen Algorithmen die benötigten Informationen, die gebraucht werden, um die Korrespondenz herzustellen.

5.2.4. Entwurf & Realisierung des Viewmanagers

Dieser Abschnitt beschreibt detailliert, nach welchen Gesichtspunkten der Viewmanager entworfen wurde, wie die grobe Architektur des Moduls aussieht und wie die einzelnen Teile realisiert wurden. Dazu gehört die Integration in das ABC, das Datenmodell für Views, die Umsetzung der verschiedenen Algorithmen und die Spezifikation der Schnittstelle zur Erweiterung des Moduls um weitere Algorithmen. Außerdem wird am Ende auf die Benutzungsschnittstelle eingegangen und ein einfaches Beispiel vorgeführt.

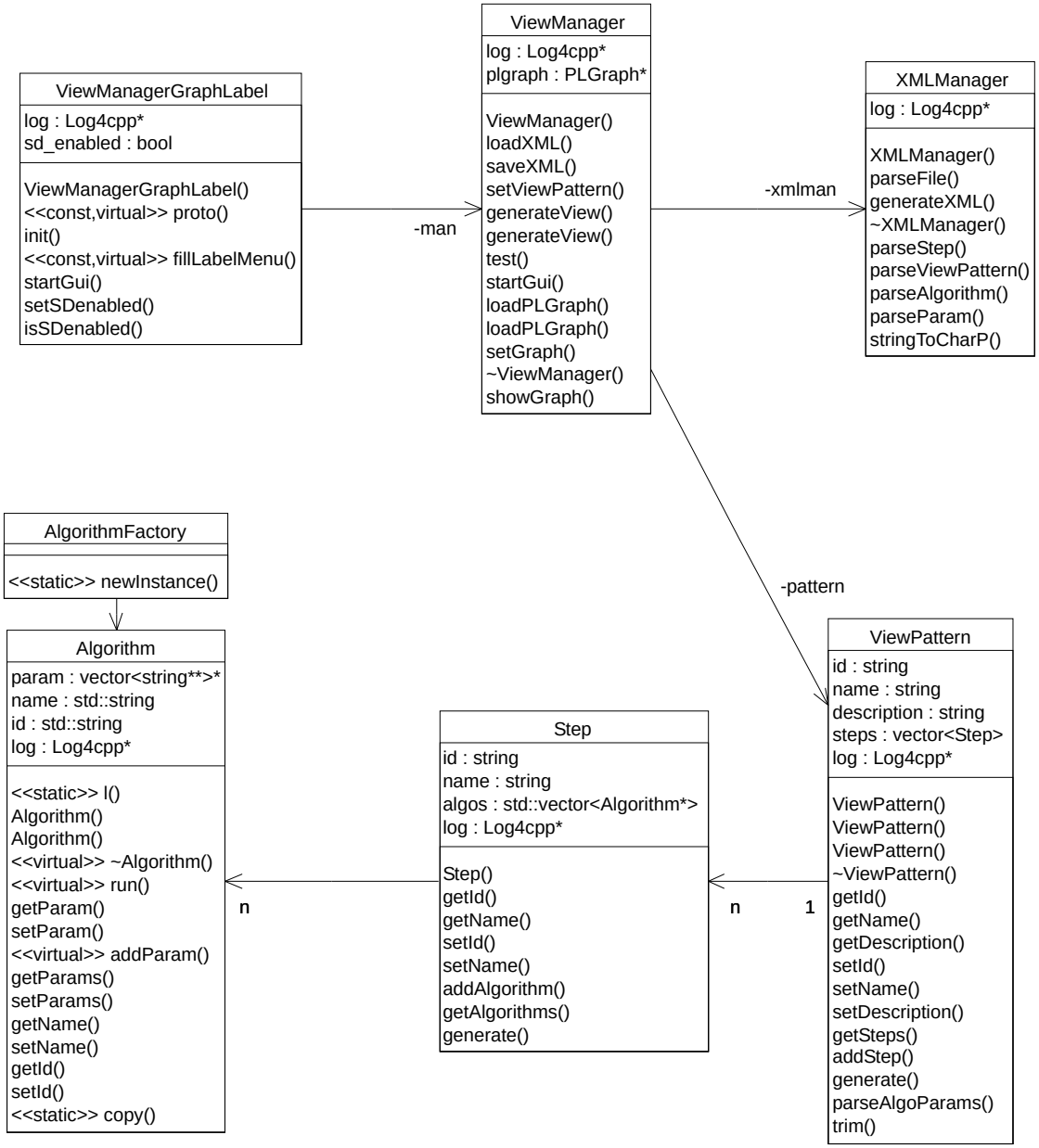


Abbildung 5.1.: logischer Aufbau des Viewmanagers

5.2.4.1. Architektur

Das entstandene Modul kapselt die verschiedenen Arten der Views (Kapitel 5.1 auf Seite 83) und deren Operationen. Der Viewmanager steht in Wechselwirkung mit dem assoziierten Graphen. Beim Entwurf wurde das Model-View-Controller-Konzept [18] eingesetzt: Die Klasse `ViewManager` verkörpert die Kontrollinstanz. Der View (im Sinne des

MVC) wird durch die Tcl-GUI beziehungsweise ihre Adapter-C++-Klassen dargestellt. Das Model ist in unserem Fall ein `ViewPattern`, das gleichzeitig auch das Datenmodell für die XML-Dateien darstellt. Dieses Datenmodell wird über den `XMLManager` persistent gemacht. Die einzelnen Algorithmen finden sich in diesem Modell und können mit Hilfe der Klasse `AlgorithmFactory` (siehe Abbildung 5.6 auf Seite 96) instanziiert werden. Die einzelnen Algorithmen werden im Detail in Abschnitt 5.2.4.3 betrachtet.

5.2.4.2. Realisierung

Integration in das ABC Die Integration in das ABC ist durch das Erstellen eines Moduls realisiert worden. Das ABC fügt den Viewmanager mit Hilfe eines GraphLabels dem jeweiligen SLGs hinzu, wenn das Viewmanager-Modul mit METAFrame gestartet wird. Lädt man dann einen SLGs, kann man im Kontextmenü den Punkt „start view manager“ aufrufen. Es gibt zwei Möglichkeiten, einen View zu generieren: Man kann die zu verwendenden Algorithmen aus der Viewmanager-Dialog wählen und ausführen oder alternativ eine vorhandene Auswahl in Form einer XML-Datei laden und diese ausführen. Mit der Viewmanager-Option „Generate View“ wird im ViewPattern die Methode `generate()` aufgerufen; daraufhin werden die ausgewählten Algorithmen abgearbeitet. Wie genau der Viewmanager zu handhaben ist, lässt sich aus dem Beispiel in Abschnitt 5.2.4.4 auf Seite 96 entnehmen.

Datenmodell und Persistenz In einer ähnlichen Vorgehensweise wie beim Constraint-Editor (Kapitel 4 auf Seite 60) wird ein View durch ein sogenanntes ViewPattern beschrieben, das eine Konstruktionsvorschrift zur Erstellung eines Views darstellt. Ein ViewPattern besteht aus mehreren Schritten (`Step`), welche wiederum aus mehreren Algorithmen bestehen. Jedes dieser Elemente (ViewPattern, Step und Algorithm) besitzt in unserem Klassenmodell eine korrespondierende Klasse (Abbildung 5.1 auf der vorherigen Seite unten). Diese hierarchische Struktur lässt sich mit Hilfe der `libxml2`-Bibliothek einfach in XML-Dateien schreiben. Beim Einlesen einer solchen XML-Datei wird dann vom `XMLManager` in einem Top-Down-Ansatz für jedes Element ein Objekt der entsprechenden Klasse erzeugt, und mit den angegebenen Parametern versorgt. Das so erzeugte Datenmodell wird dann an den Viewmanager zur Berechnung der Views weitergegeben. Der Vorteil der Aufteilung des ViewPatterns in mehrere Schritte und Algorithmen liegt in der hohen Wiederverwendbarkeit der Algorithmen. Man könnte beispielsweise verschiedene Minimierungsalgorithmen anbieten, unter denen der Benutzer wählen kann.

5.2.4.3. Algorithmen

In diesem Abschnitt werden die einzelnen Funktionsweisen der Algorithmen, die man im Viewmanager auswählen kann, beschrieben. Zusätzlich wird auf die Probleme eingegangen, die die Algorithmen „Weak Bisimulation“, „Strong Bisimulation“ und „Basic Model Collapse“ verursachen, und ein Beispiel für eine sinnvolle Zusammensetzung eines Views gegeben.

SIB-Convert Um den SLG für die Algorithmen handhabbar zu machen, müssen erst noch einige Veränderungen am Graph selber vorgenommen werden. Der wesentliche Grund dafür ist, dass Algorithmen wie z.B. „Relabelling“ oder „Minimize“ (um nur zwei Beispiele zu nennen) nur auf Kantennamen arbeiten. Jegliche Information an den Knoten werden von ihnen nicht berücksichtigt. Speziell die SIB-Informationen, die für den Anwender interessant sind, müssen zuerst in den Namen des jeweiligen Knoten codiert werden. Genau diese Aufgabe hat der Algorithmus „SIBConvert“. Merkmale wie z.B. der Name, die ID, die Klasse oder eine beliebige Kombination dieser können vom Anwender angegeben werden. Grundsätzlich codiert „SIBConvert“ alle Informationen in den Namen hinein, die vom Anwender spezifizierten Merkmale werden durch ein „_active“ zusätzlich markiert, so dass die Algorithmen, die danach den Graph bearbeiten, erkennen können, welche verändert werden dürfen. Um die Veränderungen im Graphen durch „SIBConvert“ wieder rückgängig zu machen, muss später „SIBDeConvert“ aufgerufen werden. Da die meisten Algorithmen nur auf Kantennamen arbeiten, ist es empfehlenswert, „OnlyEdges“ nach „SIBConvert“ aufzurufen. Ein Knotenname nach dem Aufruf von „SIBConvert“ kann beispielsweise so aussehen:
SIB_CLASS_active{Interactive}SIB_NAME{ShowDialog}SIB_ID_active{ShowFile}

OnlyEdges Da einige Algorithmen nur auf Kantennamen arbeiten, muss in diesen Fällen zunächst eine Transformation des Graphen vorgenommen werden, die die Knotennamen in Kantennamen überführt. In der Abbildung 5.2 und im Code-Beispiel 5.1 auf der nächsten Seite wird gezeigt, wie diese Transformation funktioniert.

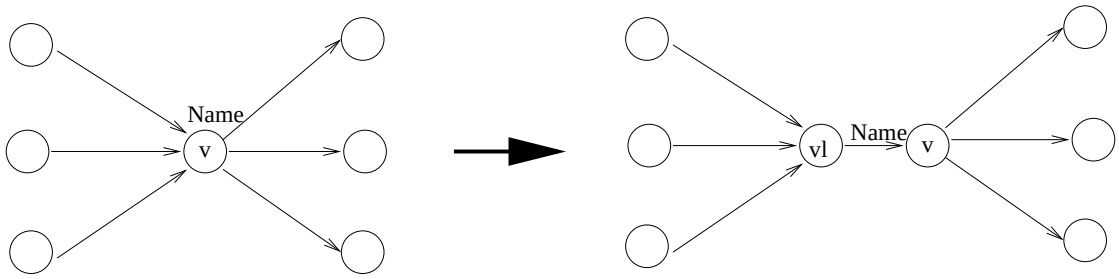


Abbildung 5.2.: Transformation des Graphen von Knoten- auf Kantennamen

```
1  foreach (s in S) {
2    name = s.name;
3    if name != leer {
4      sl := new Node();
5      sl.incomingEdges := s.incomingEdges;
6      s.incomingEdges := null;
7      e := new Edge(sl, s);
8      e.name := name;
9    }
10 }
```

Programmtext 5.1: Der Algorithmus „OnlyEdges“ in Pseudo-Code

DeOnlyEdges Durch „OnlyEdges“ werden PL-Knoten (und -Kanten) dem SLGs hinzugefügt, so dass kein gültiger SLG mehr vorliegt. Um diesen SLGs wieder gültig zu machen, schreibt „DeOnlyEdges“ den Kantennamen wieder an den ursprünglichen Knoten zurück und löscht die PL-Knoten und -Kanten, die nötig waren, um die Knotennamen an Kanten dranzuhängen.

SIBDeConvert Dieser Algorithmus hat die Aufgabe, die von „SIBConvert“ gemachten Änderungen, also das Codieren von SIB-Informationen in den Knotennamen wieder rückgängig zu machen. Aus dem Knotenname werden die drei Teilinformationen, also Klasse, ID und Name, entnommen und die drei Werte neu gesetzt.

Minimize Minimize ist ein Algorithmus zur Minimierung endlicher Zustandsautomaten. Dabei werden mit Hilfe einer Zeugenliste äquivalente Zustände (bzgl. der Nachfolger) zu Äquivalenzklassen-Zuständen zusammengefasst. Eine genauere Beschreibung findet man unter Wegener [20] oder unter Hopcroft [11]. Ein Beispiel dieser Minimierungsart zeigt Abbildung 5.3 auf der nächsten Seite.

Relabelling Zur Durchführung des Relabelling wird ein einfaches syntaktisches Verfahren verwandt. Als Kriterium dient der Name der Kante. Der Anwender spezifiziert die Parameter, die zur Umbenennung notwendig sind. Der Algorithmus geht die Kanten des Graphs durch und ersetzt alle Vorkommnisse des ersten Parameters durch den zweiten. Die Abbildung 5.4 auf der nächsten Seite zeigt eine einfache Art des Relabellings.

Tau-Elimination Bei der Tau-Elimination werden die Kanten, in deren Namen „tau“ vorkommt, aus dem Graphen gelöscht. Der Graph wird entsprechend angepasst, indem die Knoten, die mit der Kante inzident waren, verschmolzen werden. Durch ein vorheriges Relabelling kann somit der Graph wirkungsvoll verkleinert werden. Die Abbildung 5.5 auf Seite 94 und das Code-Beispiel 5.2 auf Seite 95 entsprechen der gegebenen Beschreibung. Zur Analyse vergleiche Kapitel 4.4.4 aus Wegener [20].

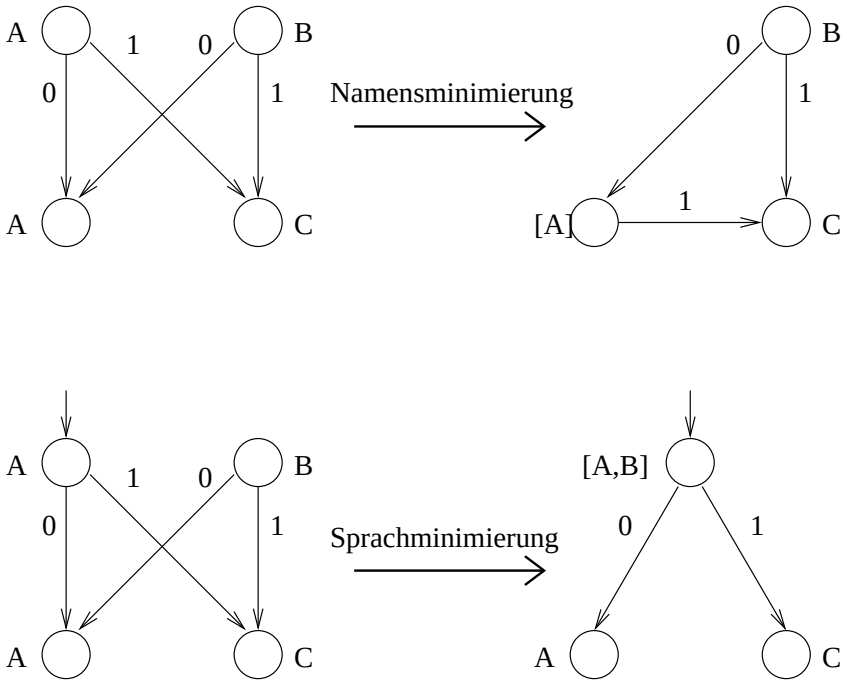


Abbildung 5.3.: Ein Beispiel für eine Minimierung eines Graphen

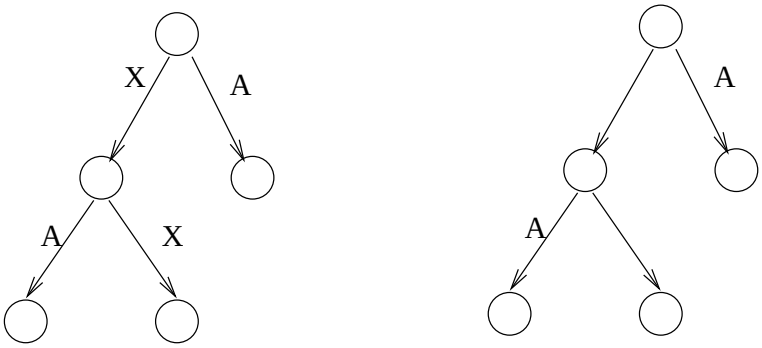


Abbildung 5.4.: Ein Beispiel für eine Umbenennung eines Kantennamen

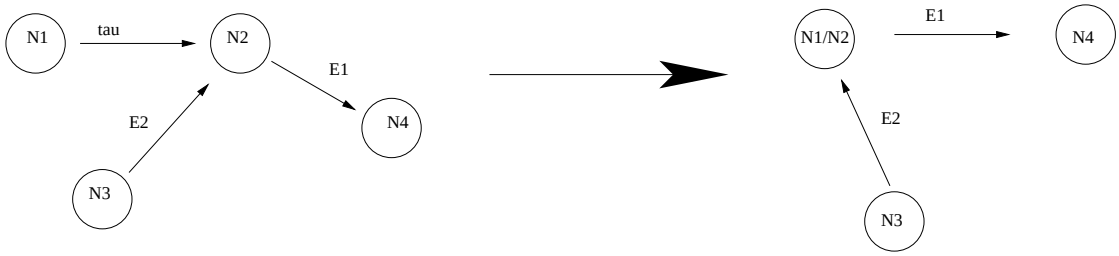


Abbildung 5.5.: Ein Beispiel für die Tau-Elimination

```
1  foreach (s in S) {
2    foreach (e in s.outgoingEdges) {
3      if (e.findInName(name, "tau")) {
4        Node tmp := m.targetNode;
5        foreach (f in tmp.outgoingEdges) {
6          if (f.targetNode == s) removeEdge(f);
7          else f.sourceNode := s;
8        }
9        foreach (f in tmp.incomingEdges) {
10         f.targetNode:=s;
11       }
12     }
13   }
14 }
```

Programmtext 5.2: Der Algorithmus „Tau-Elimination“ in Pseudo-Code

Weak Bisimulation Eine ausführliche Beschreibung der schwachen Bisimulation findet man in Milner [13]. Dieser Algorithmus ist im METAFrame-Modul mc_game verfügbar.

Strong Bisimulation Eine ausführliche Beschreibung der starken Bisimulation findet man in Milner [13]. Dieser Algorithmus ist im METAFrame-Modul mc_game verfügbar.

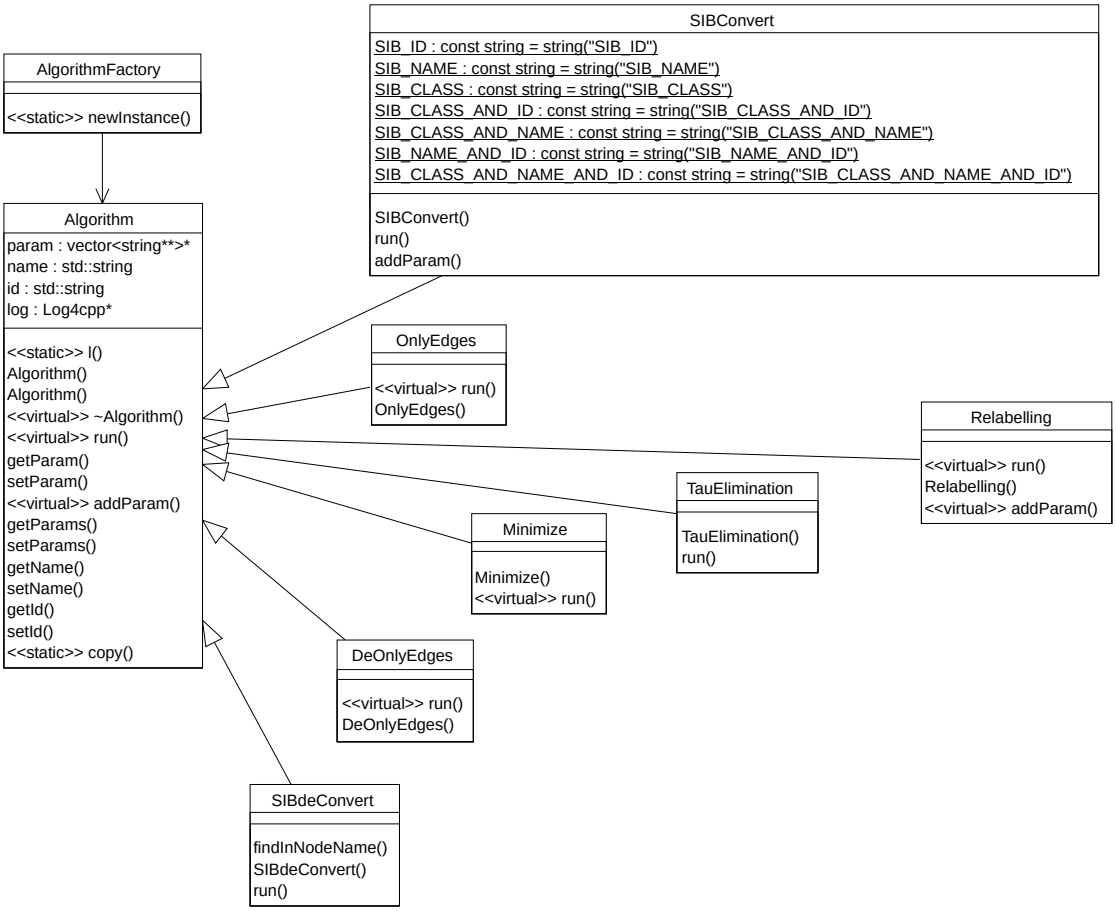


Abbildung 5.6.: Die `AlgorithmFactory` und die Algorithmen

Basic Model Collapse Dieser Algorithmus ist im METAFrame-Modul mc_game verfügbar.

5.2.4.4. Ein Beispiel-View

Eine sinnvolle Zusammenstellung für einen View könnte beispielsweise so aussehen:

1. SIBConvert(SIB_CLASS)
2. OnlyEdges
3. Relabel(Interaction,tau)
4. Tau-Elimination
5. DeOnlyEdges

6. SIBDeConvert

Ziel dieses Views ist es, die Interaktionsknoten aus dem Graphen zu entfernen. Deswegen wählt man als Parameter für SIBConvert die SIB-Klasse aus. Anschließend werden die Knotennamen auf Kantennamen abgebildet und alle Kantennamen, die „Interaction“ in sich tragen, zu „tau“ umbenannt. Die Tau-Elimination entfernt diese Kanten und passt die Knoten entsprechend an. Danach wird der Graph mit den letzten beiden Algorithmen wieder zu einem SLG konvertiert. Beispielhaft kann man dies an den Abbildungen 5.7 und 5.8 auf der nächsten Seite sehen: Dort wurde an einem Graphen (Abbildung 5.7), der die kleine Literatur-Applikation (LiteraturService_small.slg) darstellt, mit der oben genannten Vorschrift ein View erstellt (Abbildung 5.8 auf der nächsten Seite).

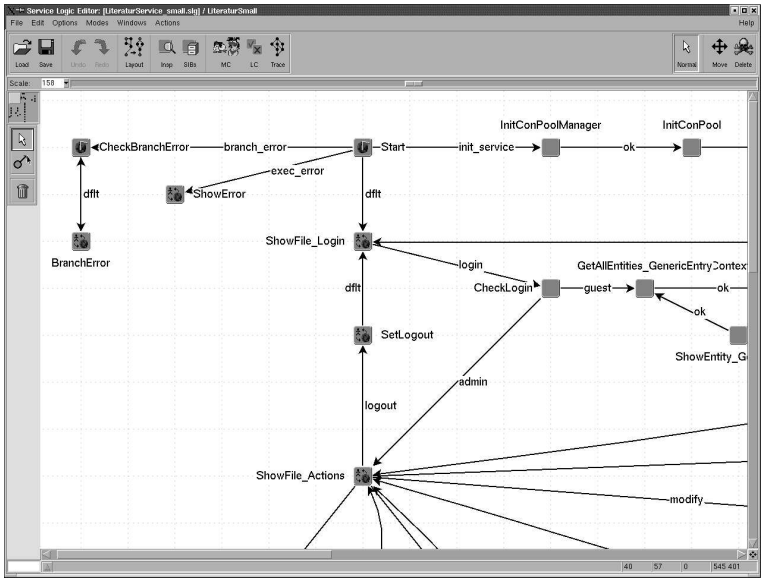


Abbildung 5.7.: Ein Beispiel-View: Der Ursprungsgraph

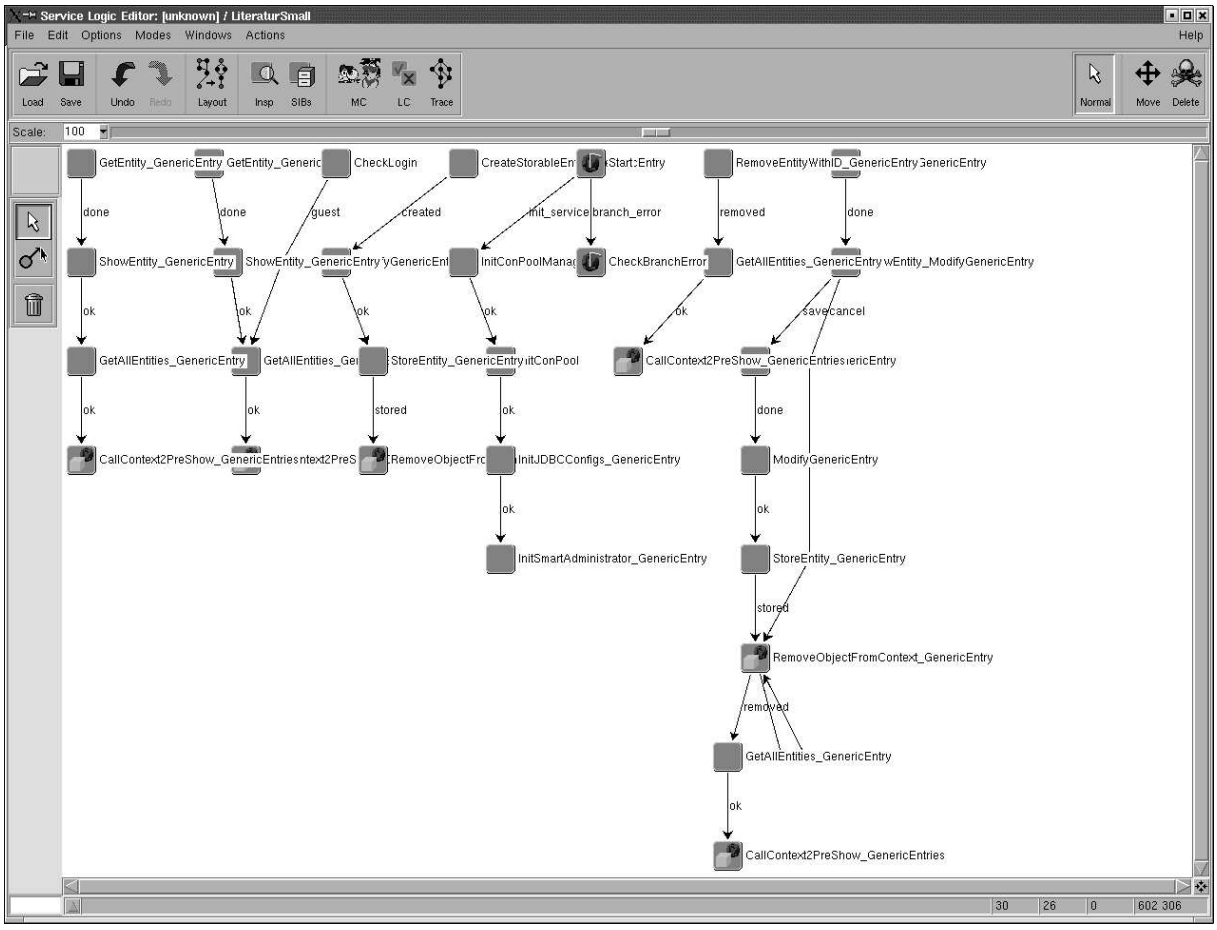


Abbildung 5.8.: Ein Beispiel-View: Der View

5.2.4.5. Erweiterbarkeit des Viewmanagers

```
Algorithm*
AlgorithmFactory::newInstance(string _name, string _id, string type)
{
  Algorithm* algo;
  if(type.compare("relable")==0) {
    algo = (Algorithm*) new Relabelling(_name,_id);
  }
[...]
    else if (type.compare("minimize")==0) {
    algo = (Algorithm*) new Minimize(_name,_id);
  } else if ((type.compare("strongbi")==0) ||
             (type.compare("strongbisimulation")==0)) {
    algo = (Algorithm*) new StrongBisimulation (_name,_id);
  } else
  {
    //if no algorithm was found use dummy/copy-algorithm
    algo = new Algorithm(_name,_id);
  }
  return algo;
}
```

Programmtext 5.3: Die Klasse `AlgorithmFactory`

Die Erweiterung des Viewmanagers um einen weiteren Algorithmus wird im Folgenden beschrieben:

Algorithmus-Interface implementieren Die neue Algorithmus-Klasse, die erstellt werden muss, erbt von der Klasse `Algorithm` und implementiert deren `run()`-Methode neu; hier wird der eigentliche Algorithmus aufgerufen. Als Eingabe muss ein vorhandener PL-Graph (der Ursprungsgraph) durch den Algorithmus in einen neuen PL-Graphen (View-Graph) überführt werden (`PLGraph* run(PLGraph*)`), der zurückgegeben werden muss.

Einbinden in die AlgorithmFactory In der Klasse `AlgorithmFactory` (Abbildung 5.3) muss eine neue Zeile eingefügt werden, um die Erkennung eines entsprechenden XML-Tags zu ermöglichen. So müsste beispielsweise für einen neuen Algorithmus mit der Klasse `NewOne` der Eintrag in der `AlgorithmFactory` eingefügt werden, wie im Codebeispiel (Abbildung 5.4) zu sehen ist.

```
  } else if((type.compare("newone")==0) ||
            (type.compare("NewOne")==0)) {
    algo = (Algorithm*) new NewOne(_name,_id);
```

Programmtext 5.4: Ein neuer Algorithmus wird hinzugefügt

Includes im Makefile anpassen In der `Makefile.in`-Datei müssen die Verzeichnisse zum Kompilieren des tatsächlichen Algorithmus-Moduls in den Include-Path hinzukommen. Das Modul des Algorithmus' selber muss dann für die Erstellung des View-Manager-Moduls ebenfalls hinzugefügt werden.

Anpassung des Viewmanager-Dialogs (optional) Der Dialog muss angepasst werden, wenn man den Algorithmus per Drag& Drop auswählen möchte. Das Einbinden in eine XML-Datei hingegen funktioniert auch ohne weitere Anpassung.

5.2.4.6. Viewmanager-Dialog

Der Viewmanager beinhaltet einen Dialog, um Views zu erstellen und zu verwalten. Der Viewmanager-Dialog wurde mit Tcl/Tk implementiert und beinhaltet zwei Inspector-Klassen. Die zwei Inspector-Klassen `TclViewInspector` und `TclHelpInspector` stellen die Schnittstelle zwischen Tcl/Tk und C++ dar. Für jedes Fenster gibt es somit eine Inspector-Klasse, um die Fenster zu steuern. Ersteres dient zu primären Bedienung des Viewmanagers, zur Generierung von Views und zum Laden und Speichern der Viewvorschriften in Form von XML-Dateien. Der `TclHelpInspector` beinhaltet ein Hilfe-Fenster.

TclViewInspector In diesem Abschnitt wird auf die Funktionalität des TclViewInspector eingegangen. Wie es in Abbildung 5.10 auf Seite 102 zu sehen ist, gibt es drei Menü Einträge: File, GenerateView und Help. Das File-Menü enthält den Menü-Button LoadXML und SaveXML, um eine XML-Datei zu laden und zu speichern, wobei über das Menü GenerateView die eigentliche View-Generierung angestoßen wird. Auf der linken Seite befinden sich die zur Auswahl stehenden Algorithmen und auf der rechten Seite befindet sich eine Liste, in der alle ausgewählten Algorithmen aufgeführt werden. Eine Auswahl kann man rückgängig machen, indem man auf sie doppelklickt. Danach wird der Eintrag entfernt. Das Fenster ist in Abbildung 5.10 auf Seite 102 zu sehen.

Für die Algorithmen Relabelling und SIBConvert müssen Parameter angegeben werden. Nach der Auswahl des Relabelling-Algorithmus, wird überprüft, ob beide Parameter angegeben wurden. Falls dies nicht der Fall sein sollte, wird das Dialog-Fenster, wie es in Abbildung 5.9 zu sehen ist, angezeigt.

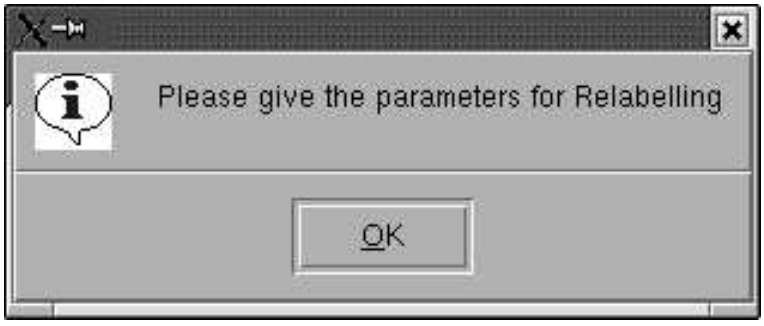


Abbildung 5.9.: Fenster GenerateXML: Parameterkontrolle für Relabelling

Die Algorithmen können in beliebiger Reihenfolge und auch mehrmals ausgewählt werden. Um die Auswahl in eine XML-Datei zu speichern, wählt man „SAVE“ aus. Danach erscheint das Dialog-Fenster, wie es in Abbildung 5.11 auf Seite 103 zu sehen ist. Um die Auswahl zu speichern, muss der Dateiname und das Verzeichnis angegeben werden.

5.2.4.7. Beispiel für eine XML-Datei

Das Beispiel aus Abschnitt 5.2.4.4 auf Seite 96 wird audführlich behandelt, um zu sehen, wie eine XML-Datei erzeugt werden kann: Zunächst wird der Viewmanager gestartet. Nach dem Start des Viewmanagers erscheint der Dialog (siehe Abbildung 5.10 auf der nächsten Seite) zur Generierung von Views und zum Laden und Speichern der Viewvorschriften in Form von XML-Dateien.

Der Algorithmus „SIBconvert“ benötigt als Parameter „SIB_CLASS“ um in folgenden Schritten die Interaktionsknoten aus dem Graphen zu entfernen. Nach der Eingabe kann der Algorithmus mit Hilfe der „SIBconvert“-Schaltfläche in die Viewgenerierungsvorschrift auf der linken Seite übernommen werden.

Der Algorithmus „OnlyEdges“ führt eine Transformation des Graphen durch, in dem die Knotennamen in Kantennamen überführt werden. Dies ist unbedingt notwendig, da einige Algorithmen nur auf Kantennamen arbeiten können. Der Algorithmus wird nach Betätigen der „OnlyEdges“-Schaltfläche in die Viewgenerierungsvorschrift übernommen.

Der Algorithmus „Relabelling“ benötigt zwei Parameter für die Umbenennung der Kantennamen. Alle Kantennamen, die „Interaction“ in sich tragen, werden zu „tau“ umbenannt. Nach der Eingabe des Wertes „Interaction“ für „Relabelling von“ und des Wertes „tau“ für „Relabelling nach“ kann der Algorithmus mit Hilfe der „Relabelling“-Schaltfläche in die Viewgenerierungsvorschrift übernommen werden.

Der Algorithmus „Tau-Elimination“ entfernt die Kanten, in deren Namen „tau“ vorkommt. Der Graph wird entsprechend angepasst, indem die Knoten, die mit der Kante inzident waren, verschmolzen werden. Durch ein vorheriges „Relabelling“ kann somit der Graph wirkungsvoll verkleinert werden, was in diesem Fall schon geschehen ist. Nach der Eingabe kann der Algorithmus mit Hilfe der „Relabelling“-Schaltfläche in die Viewgenerierungsvorschrift übernommen werden.

Der Algorithmus „DeOnlyEdges“ macht die Änderungen des Algorithmus „OnlyEdges“ rückgängig, indem die Kantennamen wieder an den ursprünglichen Knoten zurück geschrieben werden. Der Algorithmus wird nach Betätigen der „DeOnlyEdges“-Schaltfläche in die Viewgenerierungsvorschrift übernommen. Um die Veränderungen im Graphen durch den Algorithmus „SIBConvert“ wieder rückgängig zu machen, muss später der Algorithmus „SIBDeConvert“ angewandt werden. Der Algorithmus wird nach Betätigen der „SIBDeConvert“-Schaltfläche in die Viewgenerierungsvorschrift übernommen.

Mit den beiden letzten Algorithmen wurde der Graph wieder zu einem SLG konvertiert. Nach Betätigen der „SAVE“-Schaltfläche werden die ausgewählten Patterns an den Viewmanager übergeben und anschließend erscheint der folgende Dialog (siehe Abbildung 5.11 auf Seite 103). Nun muss noch ein Dateiname und ein Verzeichnis angegeben werden. Nach Betätigen der „Apply & Generate“-Schaltfläche wird die Viewgenerierungsvorschrift ohne vorher zu speichern auf den Graphen angewendet und die View-

Generierung erfolgt.

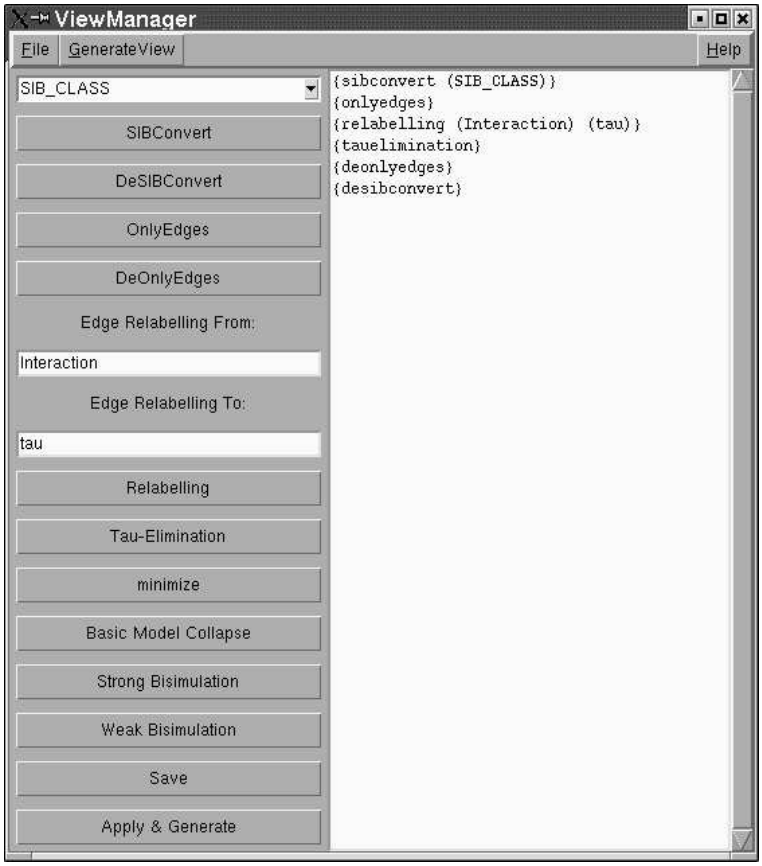


Abbildung 5.10.: Fenster Viewmanager für Beispiel Anwendung

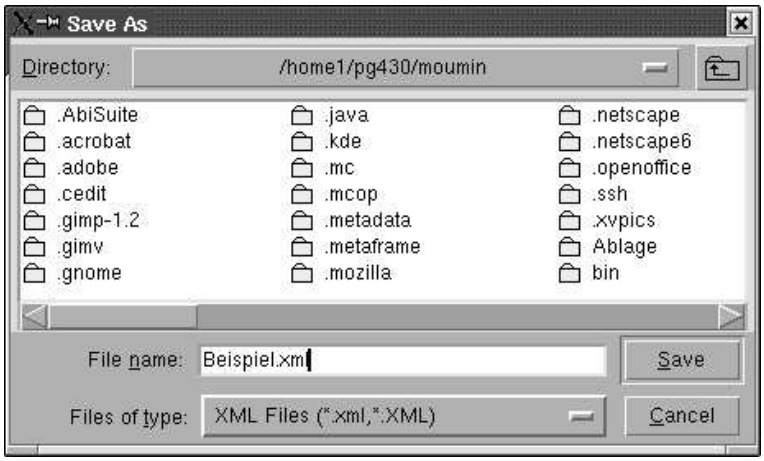


Abbildung 5.11.: Fenster Viewmanager für Beispiel Anwendung: Save File

Wenn nun die XML-Datei erzeugt worden ist, kann diese Viewgenerierungsvorschrift bei der View-Generierung verwendet werden. Die gespeicherte XML-Datei ist in der Abbildung 5.5 zu sehen.

```
1  <?xml version="1.0" encoding="UTF-8"?>
2  <viewpattern id="1" name="userEditedView" description="an user edited View">
3      <step name="userEditedView" id="1.1">
4
5          <algorithm name="sibconvert" id="1.1.1">
6              <param param="sibparam" value="SIB_CLASS"/>
7          </algorithm>
8          <algorithm name="onlyedges" id="1.1.2"/>
9
10         <algorithm name="relabelling" id="1.1.3">
11             <param param="old" value="Interaction"/>
12                 <param param="new" value="tau"/>
13         </algorithm>
14         <algorithm name="tauelimination" id="1.1.4"/>
15
16         <algorithm name="deonlyedges" id="1.1.5"/>
17         <algorithm name="desibconvert" id="1.1.6"/>
18
19     </step>
20 </viewpattern>
```

Programmtext 5.5: erzeugtes XML-File

5.2.4.8. Probleme bei der Implementierung

Die Mehrzahl der Algorithmen arbeiten lediglich auf PL-Graphen. Damit man aber auch Views auf SLGs erzeugen kann, soll vorher ein Algorithmus (genannt „SIBConvert“) aufgerufen werden, der den SLGs entsprechend umkonvertiert, so dass die Informationen der SIBs weiterverwendet werden können.

Dieser einfache Ansatz musste modifiziert werden, da das Gegenstück zu „SIBConvert“, genannt „SIBDeConvert“, so einfach nicht funktionieren konnte. Dies liegt daran, dass „SIBDeConvert“ sich nicht das „SDNodeLabel“ holen konnte, um Methoden aufzurufen, die man für „SIBDeConvert“braucht. Daraufhin wurde ein neues Graph-Label, genannt ViewManagerGraphLabel, erstellt, welches einem SLGs über einem Graph-Handler automatisch angehängt wird. Mit Hilfe dieses Labels kann der Viewmanager gestartet werden und „SIBDeConvert“ kann auf „ SDNodeLabel“ zugreifen . Allerdings tauchte in „SIBDeConvert“ noch das Problem auf, das alle Kanten ihre Branches (zusätzliche Informationen an den Kanten, die für die Internet-Anwendung benötigt wird) verloren hatten. Dieses Problem konnte allerdings behoben werden.

Die Bisimulationsalgorithmen, also „Strong Bisimulation“, „Weak Bisimulation“ und „Basic Model Collapse“, funktionieren nicht auf SLGs. Deshalb wurde versucht, aus der Rückgabe der Algorithmen wieder einen SIB-Graphen zu erstellen. Dort ist allerdings ein Problem aufgetaucht. Die ausgehenden Kanten der Metaknoten, welche bei den Algorithmen entstehen, verlieren ihre Branches, so dass der Graph keine gültige Anwendung mehr darstellt, wenn man einen der drei Algorithmen benutzt. Obwohl ein ähnliches Problem bei „SIBDeConvert“ (s.o.) gelöst werden konnte, ist dies an dieser Stelle nicht gelungen. Woran der Fehler liegen könnte, ist unbekannt; die Vermutungen liegen darin, dass eine Methode im SD-Modul nicht richtig funktioniert, dies ist allerdings auch nur Spekulation. Der Ansatz sah vor, dass bei der Verschmelzung von Knoten zu einem Metaknoten, bevor die Kanten umgesetzt werden, die Branches gespeichert und nach den Kantenumsetzungen wieder an die Kanten hinzuaddiert werden. Leider besitzen die Kanten die Branches danach nicht, obwohl dieses eigentlich nicht der Fall sein sollte.

5.3. Fazit

Insgesamt kann man festhalten, dass die Anforderungen an den Viewmanager fast alle erfüllt wurden. Die Views selber sind allerdings in dem Sinne nicht editierbar, dass keine Korrespondenz zum Ursprungsgraphen besteht; das heißt, Änderungen an einem View wirken sich nicht auf den Ursprungsgraphen aus. Für die Editierbarkeit muss man viele Informationen speichern, damit Änderungen an einem View auch zu dem Ursprungsgraphen durchgereicht werden können; der Aufwand, um die Korrespondenz zwischen abstrakter und konkreter Ebene herzustellen, wurde als zu hoch eingeschätzt. Dadurch machte eine „Undo“-/„Redo“-Funktion keinen Sinn mehr und wurde daher nicht realisiert.

Es ist möglich, einen View auf einem View zu erstellen. Man hat als Anwender prinzipiell zwei Möglichkeiten dies zu tun. Entweder man startet eine neue Viewmanager-Instanz per Kontextmenü aus dem View heraus oder man schmilzt die zwei XML-Dateien ma-

nuell im XML-Editor zusammen und erzeugt daraus eine neue XML-Datei, um diesen View permanent zur Verfügung zu stellen.

Zusätzliche Algorithmen wie die Bisimulationsalgorithmen wurden in die Anforderungen aufgenommen, die View-Generierung mit Hilfe einen Model Checkers konnte aus zeitlichen Gründen nicht realisiert werden, somit ging die Verbindung zum Constraint Editor verloren. Allerdings ist die Algorithmenseite des Viewmanagers erweiterbar (siehe Kapitel 5.2.4.5 auf Seite 99), so dass eine Einbindung des Model Checkers außerhalb der PG denkbar ist; ebenso die Einbindung anderer Algorithmen.

Die Metaknoten, die entstehen, wenn einer der Bisimulationsalgorithmen benutzt wurde, sind einsehbar. Per Kontextmenü kann man sich die Knoten, die zu dem Metaknoten verschmolzen wurden, zusammen mit den Vorgänger- und Nachfolger-Knoten in einem separaten Fenster ansehen, so dass der Anwender nachvollziehen kann, wie die Verschmelzung vonstatten ging.

Wenn man von den Problemen absieht, erhält man mit dem Viewmanager ein Werkzeug, mit dem man bequem Views zusammenstellen und abspeichern kann, so dass man sich dadurch leicht eine Bibliothek an Viewgenerierungsvorschriften erstellen kann. Dadurch ist es möglich, brauchbare Views auf Internetanwendungen, die in Form eines SLGs dargestellt werden, zu erstellen, zu bearbeiten und zu verwalten.

6. Zusammenfassung

Im Folgenden soll abschließend ein globaler Überblick über die im Rahmen der Projektgruppe InterAkt entstandenen Lösungen vermittelt werden. Außerdem soll ebenfalls auf die Arbeitsweise und -bedingungen in der Projektgruppe näher eingegangen werden.

6.1. Ergebnisse

Ziel der Projektgruppe war die Erstellung einer Entwicklungs- und Wartungsumgebung für Web-Applikationen, die es ermöglichen soll, Web-Services auf leichte Weise zu erstellen, zu validieren und zu testen. Solche Web-Applikationen werden im ABC mit Hilfe des EWIS-Moduls erstellt.

Im Bereich der Erstellung von Web-Applikationen ist oft schon die Größe des SLG, der die Applikation modelliert, ein großes Problem, da zum einen leicht die Übersicht verloren geht, zum anderen aber auch oft die gezielte Konzentration auf bestimmte Teilbereiche des SLG gewünscht ist. In diesem Zusammenhang vereint der entstandene Viewmanager (siehe 5 auf Seite 83) verschiedenste Techniken zur Generierung von Views, die dem Benutzer ermöglichen, schneller gewünschte Bereiche des SLG zu finden und den Überblick bei der Erstellung einer großen Web-Applikation behalten zu können.

Soll im SLG modelliertes Verhalten validiert werden, so ist der im ABC integrierte Model Checker ein leistungsstarkes Hilfsmittel. Dieser benötigt jedoch sogenannte Constraints, welche nichts Anderes sind als Eigenschaften eines SLG. Solche lassen sich mit Hilfe des entstandenen Constraint Editors (siehe 4 auf Seite 60) auf bequeme Weise erstellen, und dank der Verwendung eines Pattern Systems stellt dieser Vorgang minimale Anforderungen an den Benutzer, was die Kenntnis formaler Spezifikationsmechanismen betrifft. Dies erleichtert den Zugang zur Validierung von Web-Applikationen und verkürzt den Validierungsprozess.

Neben der Erstellung und der Validierung einer Web-Applikation ist es hilfreich, einen Eindruck der Funktionsweise des fertigen Gesamtsystems zu bekommen, d.h. die modellierte Applikation zu simulieren. Hier sollte zunächst ein HLL-Compiler entstehen (siehe 3.2 auf Seite 32), der den HLL-Code, mit Hilfe dessen das Verhalten des Web-Services spezifiziert wurde, in C++-Code übersetzen sollte. Der Bedarf an einem solchen Compiler entstand durch eine bereits existierende Testumgebung (siehe [17]), welche als Eingabe Testskripte in Form von C++-Code erwartet. Mit Hilfe dieser Testskripte können schließlich automatisierte Tests durchgeführt werden. Die Implementierung dieses Compilers ist zwar erfolgt, die schlussendliche Integration in das System schlug jedoch aufgrund enormer technischer Probleme fehl. Ein weiterer Ansatz, der erfolgreich umgesetzt wurde, ist der Simulator (siehe 3.3 auf Seite 43), der es schon während des Entwicklungsprozes-

ses ermöglicht, das Verhalten der erstellten Web-Applikation zu simulieren. Dazu wird der SLG interaktiv durchlaufen und dem Benutzer eine Reihe von Eingriffsmöglichkeiten geboten (ähnlich einem Debugger). So kann schon während der Entwicklungsphase ein Eindruck vom Laufzeitverhalten des modellierten Systems gewonnen werden.

Für eine detaillierte Auflistung sämtlicher Funktionalitäten der entstandenen Lösungen siehe auch Anhang C auf Seite 111.

Insgesamt bieten die im Rahmen der Projektgruppenarbeit entstandenen Lösungen hilfreiche Möglichkeiten, den Entwicklungsprozess einer Web-Applikation von ihrer Erstellung über ihre Validierung bis hin zum Testen einfacher und effizienter zu gestalten.

6.2. Arbeitsweise

Die Arbeit innerhalb der Projektgruppe war geprägt von einer Unterteilung in mehrere Arbeitsgruppen, die anfangs recht „isoliert" voneinander arbeiteten. Zum Bindeglied zwischen den einzelnen Arbeiten wurden die Betreuer, die dafür sorgten, dass wir den roten Faden behielten und besonders technisch eine wichtige Hilfe darstellten. Aufgrund der extrem mangelhaften Dokumentation des METAFrame-Projektes, die die Einarbeitung und das Verständnis enorm erschwerte, war die Arbeit oft sehr langwierig. Ebenfalls wurde hier die Hilfe der Betreuer unverzichtbar. Umso ärgerlicher, dass Oliver Niese praktisch während der laufenden Arbeit als Betreuer ausschied.

Bei zunehmender Reife der einzelnen Arbeiten wurde letztlich auch eine Kommunikation zwischen den einzelnen Gruppen notwendig, um das einwandfreie Zusammenspiel der entstandenen Module zu sichern. Hier gewann besonders die Rolle des Koordinators an Bedeutung, der in der Projektgruppe für Progress sowie gut koordinierte Arbeitsabläufe und Kommunikation sorgte. Diese Rolle wurde zwischen den Teilnehmern der Projektgruppe im monatlichen Rhythmus weitergegeben. So konnten alle Teilnehmer in aller Deutlichkeit erfahren, wie wichtig Kommunikation und Koordination für den Erfolg von Software-Projekten sind.

Zu Beginn der Projektgruppe kostete die verspätete Verfügbarkeit der Arbeitsumgebungen viel Zeit. Die Rechner sowie die Installation und Einrichtung der benötigten Software hätte schneller erfolgen müssen, um einen unmittelbareren Arbeitsbeginn zu gewährleisten. Ansonsten waren die zur Verfügung gestellten Räumlichkeiten und Mittel im Großen und Ganzen jedoch zufriedenstellend. Ebenfalls als positiv festzuhalten ist die gelungene Seminarfahrt zu Beginn der Projektgruppe, die große Hilfsbereitschaft der Betreuer ebenso wie das Erlernen neuer Techniken wie z.B. C++, XML, die Verwendung von Makefiles usw., die sich die Teilnehmer der Projektgruppe durch kleine Vorträge größtenteils selbst näher gebracht haben. Besonders auch die in punkto effizienter Kommunikation gewonnenen Erfahrungen werden den Teilnehmern bei späteren Projekten von Nutzen sein.

A. Liste der vom Interpreter verwendeten Typ-Klassen

- EList - Listen rechtsseitig
- ESet - Sets rechtsseitig
- ETuple - Tupel rechtsseitig
- ELiSeTu - Listen, Sets und Tupel linksseitig
- EPrint - Ausgabeanweisungen
- EIfThen - If-Then-Else Bedingungen
- EWhile - While-Schleifen
- EBind - Wertzuweisungen
- EDecl - Deklarationen (hier auch triviale Datentypen Int, Boolean, String, da diese in C++ identische Namen, allerdings klein geschrieben)
- EInfix - Infix-Notationen
- ECast - Typ-Umwandlung
- EFCall - Funktionsaufrufe

B. Flex-Code des HLL-Transformers mit Schlüsselwörtern

```
 /* keywords */

<INITIAL>"quit"|"QUIT"|"exit"|"EXIT"    { iPos += 4; return EParser::QUIT;}
<INITIAL>"begin"|"BEGIN"                { iPos+=5;returnEParser::_BEGIN;}
<INITIAL>"end"|"END"                    { iPos +=3;returnEParser::END; }
<INITIAL>"beginenv"|"BEGINENV"          { iPos += 8;returnEParser::BEGINENV; }
<INITIAL>"endenv"|"ENDENV"              { iPos += 6;returnEParser::ENDENV; }
<INITIAL>"type"|"TYPE"                  { iPos += 4;returnEParser::TYPE; }
<INITIAL>"procedure"|"PROCEDURE"        { iPos += 9;returnEParser::PROCEDURE; }
<INITIAL>"if"|"IF"                      { iPos += 2;returnEParser::IF; }
<INITIAL>"then"|"THEN"                  { iPos += 4;returnEParser::THEN; }
<INITIAL>"else"|"ELSE"                  { iPos += 4;returnEParser::ELSE; }
<INITIAL>"fi"|"FI"                      { iPos += 2;returnEParser::FI; }
<INITIAL>"elsif"|"ELSIF"                { iPos += 5;returnEParser::ELSIF; }
<INITIAL>"while"|"WHILE"                { iPos += 5;returnEParser::WHILE; }
<INITIAL>"do"|"DO"                      { iPos += 2;returnEParser::DO; }
<INITIAL>"od"|"OD"                      { iPos += 2;returnEParser::OD; }
<INITIAL>"function"|"FUNCTION"          { iPos += 8;returnEParser::FUNCTION; }
<INITIAL>"print"|"PRINT"                { iPos += 5;returnEParser::PRINT; }
<INITIAL>"return"|"RETURN"              { iPos += 6;returnEParser::TRETURN; }
<INITIAL>"var"|"VAR"                    { iPos += 3;returnEParser::VAR; }
<INITIAL>"const"|"CONST"                { iPos += 5;returnEParser::_CONST; }
<INITIAL>"ref"|"REF"                    { iPos += 3;returnEParser::REF; }
<INITIAL>"require"|"REQUIRE"            { iPos += 7;returnEParser::REQUIRE; }


/* anything else */

<INITIAL>{BLANKS}                { iPos += std::strlen(yytext);}
<INITIAL>"."|"#"|":"|";"|","|"("|")"|"["|"]"|"{"|"}"|"="|"*"|"@"|"^"|"+"|"-"|"/"
|"%"|"~"|"?" { iPos += 1; return yytext[0]; }
<INITIAL>":="                      {iPos += 2;return EParser::ASS; }
<INITIAL>"->"                        {iPos += 2;returnEParser::ARROW; }
<INITIAL>"=="                        {iPos +=2; returnEParser::EQ;}
<INITIAL>"!="                        {iPos += 2;return EParser::NEQ;}
<INITIAL>">="                        {iPos += 2;returnEParser::GEQ;}
<INITIAL>"<="                    {iPos += 2; returnEParser::LEQ;}
<INITIAL>">"                         {iPos += 1; return EParser::GT; }
<INITIAL>"<"                           {iPos += 1;returnEParser::LT; }
<INITIAL>"and"|"AND"                 {iPos +=3; returnEParser::ANDALSO; }
```

```
41 <INITIAL>"andalso"|"ANDALSO"        {iPos +=7;return EParser::ANDALSO; }
42 <INITIAL>"or"|"OR"              {iPos += 2;returnEParser::ORELSE; }
43 <INITIAL>"orelse"|"ORELSE"      {iPos += 6;returnEParser::ORELSE; }
44 <INITIAL>"::"                   {iPos += 2;returnEParser::ADDELEM; }
```

C. Features & Known Issues der erstellen Module

C.1. Anwendung

Features: kleine Anwendung

- Die Verwaltung der Publikationen ist möglich. Dabei können bereits in der Datenbank vorhandene Publikationen angezeigt, modifiziert und entfernt werden. Ebenso können neue Publikationen erstellt und in der Datenbank entfernt werden. Jede Publikation ist der Einfachheit halber nur mit zwei Attributen (title, note) versehen.
- Zwei Rollen sind verfügbar: „guest" und „admin". Unregistrierte Benutzer werden automatisch als „guest" angesehen und können sich nur Publikationen ansehen. Der registrierte Benutzer „admin" darf Publikationen verwalten.
- Die Simulation der Anwendung mit dem Tracer und dem Simulator ist realisiert. Mit dem Tracer lassen sich die Funktionalitäten der reellen Applikation eins zu eins simulieren, d.h. der tatsächliche Ablauf, der bei der reellen Applikation erfolgt, kann mit dem Tracer modelliert werden. Insbesondere werden dynamische Formulare unterstützt. Mit dem Simulator lässt sich die Anwendung ebenfalls simulieren. Interaktionsknoten sind nun durch eine Standard-GUI darstellbar, so dass die Implementierung der Formulare mit HLL-Codes nicht mehr nötig ist.

Known Issues: kleine Anwendung

In der gegenwärtigen Version des Simulators werden dynamische Formulare nicht unterstützt. Dynamische Formulare sind Formulare, deren Elemente von dynamischen Faktoren abhängen (z.B. von Publikationen in der Datenbank).

Features: große Anwendung

- Verschiedene Teildienste sind verfügbar: analog zur kleinen Anwendung ist die Verwaltung von Publikationen möglich, ebenso die Verwaltung von Kategorien. Auch hier kann man ähnliche Aktionen (anzeigen, erstellen, modifizieren, entfernen) ausführen. Außerdem kann man mit Hilfe der wurzelgerichteten Kategoriestruktur Unterkategorien anlegen. Darüber hinaus ist eine Verwaltung der Autoren / Publisher / Editoren / Schlüsselwörter / Sprachen möglich.

- User- und Rollenmanagement sind verfügbar: Aus dem TEMPLUS-Dienst, der erfolgreich beim Lehrstuhl 5 eingesetzt wurde, ist größtenteils das User- und Rollenmanagement übernommen. Es ist möglich, sich einzuloggen, nachdem ein User mit seinen persönlichen Daten manuell in der Datenbank angelegt wurde und ihm eine oder mehrere Rollen zugewiesen wurde. Dann darf der User mit der Rolle „Mitarbeiter" Publikationen anlegen sowie alle Publikationen ansehen oder modifizieren. Ein User mit der Rolle „Student" darf nur Publikationen ansehen. Ein Administrator hat Zugriff auf alle Funktionen des Systems, er kann also auch Publikationen löschen. Wenn ein User mehrere Rollen zugewiesen bekommt, kann er zwischen seinen Rollen umschalten. Es besteht auch die Möglichkeit, Rollen, Features und Featureklassen zu verwalten.

- Die Simulation der Anwendung für einen festgelegten Pfad ist realisiert. Der festgelegte Pfad „editFeatureClass" von der großen Anwendung lässt sich durch den Simulator simulieren. Alle Funktionalitäten des Simulators können sich in der Simulation widerspiegeln.

Known Issues: große Anwendung

- Da Templus-Komponenten (z.B. Rollenmanagement) in der Anwendung integriert sind, ist ein riesiger Graph mit tausenden SIBs entstanden. Das führt zu Problemen beim „generate" des SLG. Es wird für „generate" über eine halbe Stunde benötigt. Die generierten Java-Methoden sind zu groß. Es wurden zwei Skripte implementiert, um große Methoden in kleinere kompilierbare Methoden umzuwandeln.

- Das User-Management kann nur manuell durchgeführt werden, d.h. man kann nur manuell neue User in der Datenbank anlegen und ihnen anschließend Rollen zuweisen.

- Aus organisatorischen Gründen ist die große Anwendung nur zum Teil fertiggestellt. Jede Publikation ist im Moment der Einfachheit halber nur mit zwei Attributen (title, note) versehen. Weitere Attribute, die in dem Entwurf spezifiziert sind, sollten noch hinzugefügt werden. Im Moment sind außerdem verschiedene Teildienste getrennt und unabhängig voneinander. Es sollte realisiert werden, dass beim Anlegen einer neuen Publikation andere Teildienste mit berücksichtigt werden können, z.B. soll eine bestimmte Kategorie einer Publikation zugewiesen werden.

C.2. Transformer

Features

Geparst werden können:

- primitive Datentypen (int, boolean, string), dabei Deklarationen, Initialisierungen und Zuweisungen,

- Listen, Sets und Tupel, dabei Deklaration, sowie die size-Funktionen
- Boole'sche und arithmetische Ausdrücke, dabei +, -, *, /, %, <, >, <=, >=, and, or,
- Funktionsaufrufe,
- if- und while-Schleifen und
- Print-Anweisungen.
- Eingabe-Strings werden verarbeitet und ein Ausgabe-String wird zurückgeliefert, darüber hinaus ist Kommandozeilen-Parsing möglich.

Known Issues

Folgendes kann der Transformer nicht:

- Variablen ohne vorherige Deklaration benutzen,
- Listen, Sets und Tupel mit Werten versehen (initial und allgemein),
- Type-Casting,
- Funktionsdeklarationen (dies allerdings nur aus Zeitgründen, ist aber einfach zu implementieren),
- mit dem Environment des Testgenerators umgehen.

C.3. Simulator

Features

- HTML-Formulare: Beim Durchlaufen eines Interaktionsknoten wird ein Fenster dargestellt, welches die HTML-Seite des Knotens simuliert. Es werden die Formulare der Seite dargestellt; alle Links werden als Buttons widergegeben. Der Benutzer kann Formularfelder ausfüllen oder die Ausführung eines Links durch Klick auf den entsprechenden Button simulieren. Folgende Elemente sind möglich: Texteingabefelder, Drop-Down-Boxen, Radiobuttons, Check-Boxen, Select-Boxen sowie Buttons.
- Undo-Funktion: Es können Simulationsschritte rückgängig gemacht werden. Dabei wird sowohl der aktuelle Knoten (mit entsprechender Aktualisierung der Graphanzeige), als auch der Zustandsraum zurückgesetzt. Es ist immer möglich, mehrere Undo-Schritte in Folge auszuführen, solange, bis der Start der Simulation wieder erreicht ist. Es können über den Debugger Eingriffe erfolgen und danach kann die Simulation wieder fortgesetzt werden.

- State-Space-Debugger: Zustandsraum-Variablen können z.B. zur Simulation von Datenbanken oder Session-Attributen einer Web-Applikation genutzt werden. Sie können auch direkten Einfluss auf den Ablauf der Simulation haben, denn ein Koordinationsknoten kann abhängig von Inhalt solcher Variablen den nächsten Branch festlegen. Im Simulatorfenster wird eine Tabelle mit den Werten dieser Variablen angezeigt. Hier können jederzeit manuelle Eingriffe erfolgen, indem man Werte ändert. Dies ist eine Funktion, die Kenntnisse des Benutzers voraussetzt, da keinerlei Kontrolle der Eingaben erfolgt.

Known Issues

- StateSpace-Variablentypen: Der Simulator ist auf bestimmte Datentypen begrenzt. Derzeit werden neben den einfachen Datentypen Folgende unterstützt: Liste von Strings, Liste von Listen von Strings, Liste von Tupeln von Strings.
- Dynamische Formulare: In der gegenwärtigen Version des Simulators erfasst dieser nur konstante Formulare, Dynamische werden nicht unterstützt. Dynamische Formulare sind Formulare, deren Elemente oder deren Anzahl von dynamischen Faktoren abhängen (z.B. von Einträgen in Datenbanken).
- Speichern und Laden von Simulationen: Diese Funktion wurde wegen Zeitmangels nicht umgesetzt.

C.4. Constraint Editor

Features

- Constraints können anhand einer Datenbasis von Composite und Logic Patterns (siehe [12]) generiert werden.
- Existierende Constraints können verwaltet, also editiert (nur, wenn die Constraint auch mit dem Editor generiert wurde, d.h. auch im XML-Format vorliegt) und gelöscht werden.
- Die konkrete Implementierung der GUI ist austauschbar.
- Die Implementierung folgt den Entwürfen in Oliver Nieses Dissertation (siehe [14]), ist also nicht auf SIBS festgelegt. Mit geringfügigen Änderungen in der Implementierung sind z.B. auch ganze Teilformeln als Werte für die Parameter einer Constraint denkbar.
- Ein englischer User Guide existiert im Modul unter `METAFrame/constraint_editor/doc/userguide`.

Known Issues

- Der Constraint Editor bietet keine Möglichkeit zur Editierung des Pattern Systems. Composite und Logic Patterns müssen von Hand in XML geschrieben werden, entsprechende DTDs sind jedoch vorhanden.
- Constraints, die nur in einer CBD-Version vorliegen, können mit dem Editor nicht editiert werden, da sich die Formeln nicht auf die verwendeten Patterns abbilden lassen.

C.5. Viewmanager

Features

- Mit dem Viewmanager ist es möglich, einen View zu erstellen, anzuzeigen und zu speichern.
- Für die View-Generierung benötigte Algorithmenreihenfolge werden in einer XML-Datei gespeichert, welche über die GUI erstellt werden kann. Die Algorithmen können auch mehrmals ausgewählt werden.
- Der XML-Manager kann aus einer XML-Datei ein View-Pattern (Kapitel 5.2.4.2 auf Seite 91) erstellen. Auch die Rückrichtung ist implementiert, so dass durch Speichern einer XML-Datei eine persistente View-Erstellungsvorschrift hinterlegt wird. In gewisser Weise ist der View dann sogar dynamisch, denn es wird nur die Vorschrift zur Erstellung in der XML-Datei gespeichert. Wenn sich also der Ursprungsgraph ändert, so gilt dies auch für den bei der Anwendung der View-Vorschrift entstehenden View.
- Wenn der Benutzer den View statisch machen möchte, d.h. den View unabhängig von dem zugrunde liegenden Graphen speichern möchte, so kann er dies wie gewohnt tun, indem er den resultierenden Graphen als SLG abspeichert. In diesem Fall muss man allerdings auf die Korrespondenz zwischen Ursprungsgraph und View verzichten.
- Die Views sind editierbar, allerdings spiegeln sich Änderungen im View nicht im Ursprungsgraphen wider.
- Ein View kann auf einem View erstellt werden, indem im Kontextmenü des entstandenen Graphen eine neue Instanz des Viewmanager gestartet wird. Eine permanente Verbindung mehrerer Viewvorschriften kann durch Zusammenfügung der einzelnen XML-Dateien erfolgen.
- Metaknoten sind einsehbar, d.h. man kann sich die Knoten ansehen, die zu dem Metaknoten verschmolzen wurden. Zusätzlich werden deren Vorgänger- und Nachfolgerknoten angezeigt und lila eingefärbt, so dass sie leicht erkennbar sind.

Known Issues

- In den Algorithmen Strong Bisimulation, Weak Bisimulation und Basic Model Collapse verlieren die Kanten, die von den entstandenen Meta-Knoten ausgehen, ihre „Branches“ (zusätzliche Informationen an den Kanten, die für die Internet-Anwendung benötigt wird).

D. Abbildungen der Patterns in ESLTL

Im Folgenden wird aufgelistet, wie die einzelnen Patterns in ESLTL-Formeln abgebildet werden (Quelle: [14]). Dabei werden die Quantoren notiert mit $\mathcal{Q} = \{\exists, \forall\}$.

D.1. Absence

P ist nicht erfüllt:

Globally	$\mathcal{Q}x(\mathrm{G}(\neg P(x_p)))$
Before R	$\mathcal{Q}x(\mathrm{F}(R(x_r)) \Rightarrow (\neg P(x_p)\ \mathrm{U}\ R(x_r)))$
After Q	$\mathcal{Q}x(\mathrm{G}(Q(x_q) \Rightarrow G(\neg P(x_p))))$
Between Q and R	$\mathcal{Q}x(\mathrm{G}((Q(x_q) \wedge \neg R(x_r) \wedge \mathrm{F}(R(x_r))) \Rightarrow (\neg P(x_p)\ \mathrm{U}\ R(x_r))))$
After Q until R	$\mathcal{Q}x(\mathrm{G}(Q(x_q) \wedge \neg R(x_r) \Rightarrow (\neg P(x_p)\ \mathrm{WU}\ R(x_r))))$

D.2. Universality

P ist erfüllt:

Globally	$\mathcal{Q}x(\mathrm{G}(\neg P(x_p)))$
Before R	$\mathcal{Q}x(\mathrm{F}(R(x_r)) \Rightarrow (P(x_p)\ \mathrm{U}\ R(x_r)))$
After Q	$\mathcal{Q}x(\mathrm{Q}(Q(x_q) \Rightarrow \mathrm{G}(P(x_p)))$
Between Q and R	$\mathcal{Q}x(((Q(x_q) \wedge \neg R(x_r) \wedge \mathrm{F}(R(x_r))) \Rightarrow (P(x_p)\ \mathrm{U}\ R(x_r))))$
After Q until R	$\mathcal{Q}x(\mathrm{G}(Q(x_q) \wedge \neg R(x_r) \Rightarrow (P(x_p)\ \mathrm{WU}\ R(x_r))))$

D.3. Existence

P ist erfüllt:

Globally	$\mathcal{Q}x(\mathrm{F}(P(x_p)))$
Before R	$\mathcal{Q}x(\neg R(x_r)\ \mathrm{WU}\ (P(x_p) \wedge \neg R(x_r)))$
After Q	$\mathcal{Q}x(\mathrm{G}(\neg Q(x_q) \vee \mathrm{F}(Q(x_q) \wedge \mathrm{F}(P(x_p)))))$
Between Q and R	$\mathcal{Q}x(\mathrm{G}(Q(x_q) \wedge \neg R(x_r) \Rightarrow (\neg R(x_r)\ \mathrm{WU}\ (P(x_p) \wedge \neg R(x_r)))))$
After Q until R	$\mathcal{Q}x(\mathrm{G}(Q(x_q) \wedge \neg R(x_r) \Rightarrow (\neg R(x_r)\ \mathrm{U}\ (P(x_p) \wedge \neg R(x_r)))))$

D.4. Precedence

S kommt vor P:

Globally	$\mathcal{Q}x(\neg P(x_p)\ \mathrm{WU}\ S(x_s))$
Before R	$\mathcal{Q}x(\mathrm{F}(R(x_r)) \Rightarrow (\neg P(x_p)\ \mathrm{U}\ (S(x_s) \vee R(x_r))))$
After Q	$\mathcal{Q}x(\mathrm{G}(\neg Q(x_q)) \vee \mathrm{F}(Q(x_q) \wedge (\neg P(x_p)\ \mathrm{WU}\ S(x_s))))$
Between Q and R	$\mathcal{Q}x(\mathrm{G}((Q(x_q) \wedge \neg R(x_r) \wedge \mathrm{F}(R(x_r))) \Rightarrow (\neg P(x_p)\ \mathrm{U}\ (S(x_s) \vee R(x_r)))))$
After Q until R	$\mathcal{Q}x(\mathrm{G}(Q(x_q) \wedge \neg R(x_r) \Rightarrow (\neg P(x_p)\ \mathrm{WU}\ (S(x_s) \vee R(x_r)))))$

D.5. Response

S folgt P:

Globally	$\mathcal{Q}x(\mathrm{G}(P(x_p) \Rightarrow \mathrm{F}(S(x_s))))$
Before R	$\mathcal{Q}x(\mathrm{F}(R(x_r)) \Rightarrow (P(x_p) \Rightarrow (\neg R(x_r)\ \mathrm{U}\ (S(x_s) \wedge \neg R(x_r)))\ \mathrm{U}\ R(x_r)))$
After Q	$\mathcal{Q}x(\mathrm{G}(Q(x_q) \Rightarrow \mathrm{G}(P(x_p) \Rightarrow \mathrm{F}(S(x_s)))))$
Between Q and R	$\mathcal{Q}x(\mathrm{G}((Q(x_q) \wedge \neg R(x_r) \wedge \mathrm{F}(R(x_r))) \Rightarrow (P(x_p) \Rightarrow (\neg R(x_r)\ \mathrm{U}\ (S(x_s) \wedge \neg R(x_r)))\ \mathrm{U}\ R(x_r))))$
After Q until R	$\mathcal{Q}x(\mathrm{G}(Q(x_q) \wedge \neg R(x_r) \Rightarrow ((P(x_p) \Rightarrow (\neg R(x_r)\ \mathrm{U}\ (S(x_s) \wedge \neg R(x_r))))\ \mathrm{WU}\ R(x_r))))$

D.6. Bounded Existence

Instanz des Patterns mit k=2, d.h. Übergänge zu P-Zuständen kommen maximal zweimal vor:

Globally	$\mathcal{Q}x((\neg P(x_p)\ \mathrm{WU}\ (P(x_p)\ \mathrm{WU}\ (\neg(P(x_p)\ \mathrm{WU}\ (P(x_p)\ \mathrm{WU}\ \mathrm{G}(\neg P(x_p)))))))$
Before R	$\mathcal{Q}x(\mathrm{F}(R(x_r)) \Rightarrow ((\neg P(x_p) \wedge \neg R(x_r))\ \mathrm{U}\ (R(x_r) \vee ((P(x_p) \wedge \neg R(x_r))\ \mathrm{U}\ (R(x_r) \vee ((\neg P(x_p) \wedge \neg R(x_r))\ \mathrm{U}\ (R(x_r) \vee ((P(x_p) \wedge \neg R(x_r))\ \mathrm{U}\ (R(x_r) \vee (\neg P(x_p)\ \mathrm{U}\ R(x_r))))))))))$
After Q	$\mathcal{Q}x(\mathrm{F}(Q(x_q)) \Rightarrow (\neg Q(x_q)\ \mathrm{U}\ (Q(x_q) \wedge (\neg P(x_p)\ \mathrm{WU}\ (P(x_p)\ \mathrm{WU}\ (\neg P(x_p)\ \mathrm{WU}\ (P(x_p)\ \mathrm{WU}\ \mathrm{G}(\neg P(x_p))))))))$
Between Q and R	$\mathcal{Q}x(\mathrm{G}((Q(x_q) \wedge \mathrm{F}(R(x_r))) \Rightarrow ((\neg P(x_p) \wedge \neg R(x_r))\ \mathrm{U}\ (R(x_r) \vee ((P(x_p) \wedge \neg R(x_r))\ \mathrm{U}\ (R(x_r) \vee ((\neg P(x_p) \wedge \neg R(x_r))\ \mathrm{U}\ (R(x_r) \vee ((P(x_p) \wedge \neg R(x_r))\ \mathrm{U}\ (R(x_r) \vee (\neg P(x_p)\ \mathrm{U}\ R(x_r))))))))))$
After Q until R	$\mathcal{Q}x(\mathrm{G}(Q(x_q) \Rightarrow ((\neg P(x_p) \wedge \neg R(x_r))\ \mathrm{U}\ (R(x_r) \vee ((P(x_p) \wedge \neg R(x_r))\ \mathrm{U}\ (R(x_r) \vee ((\neg P(x_p) \wedge \neg R(x_r))\ \mathrm{U}\ (R(x_r) \vee ((P(x_p) \wedge \neg R(x_r))\ \mathrm{U}\ (R(x_r) \vee (\neg P(x_p)\ \mathrm{WU}\ R(x_r)) \vee \mathrm{G}(P(x_p))))))))))$

D.7. Precedence Chain

S,T kommen vor P („2 Ursachen-1 Wirkung Precedence Chain"):

Globally	$\mathcal{Q}x(\mathrm{F}(P(x_p)) \Rightarrow (\neg P(x_p)\ \mathrm{U}\ (S(x_s) \wedge \neg P(x_p) \wedge \mathrm{X}(\neg P(x_p)\ \mathrm{U}\ T(x_t)))))$
Before R	$\mathcal{Q}x(\mathrm{F}(R(x_r)) \Rightarrow (\neg P(x_p)\ \mathrm{U}\ (R(x_r) \vee (S(x_s) \wedge \neg P(x_p) \wedge \mathrm{X}(\neg P(x_p)\ \mathrm{U}\ T(x_t))))))$
After Q	$\mathcal{Q}x((\mathrm{G}(\neg Q(x_q))) \vee (\neg Q(x_q)\ \mathrm{U}\ (Q(x_q) \wedge \mathrm{F}(P(x_p)) \Rightarrow (\neg P(x_p)\ \mathrm{U}\ (S(x_s) \wedge \neg P(x_p) \wedge \mathrm{X}(\neg P(x_p)\ \mathrm{U}\ \mathrm{T}(x_t)))))))$
Between Q and R	$\mathcal{Q}x(\mathrm{G}((Q(x_q) \wedge \mathrm{F}(R(x_r))) \Rightarrow (\neg P(x_p)\ \mathrm{U}\ (R(x_r) \vee (S(x_s) \wedge \neg P(x_p) \wedge \mathrm{X}(\neg P(x_p)\ \mathrm{U}\ T(x_t)))))))$
After Q until R	$\mathcal{Q}x(\mathrm{G}(Q(x_q) \Rightarrow (\mathrm{F}(P(x_p)) \Rightarrow (\neg P(x_p)\ \mathrm{U}\ (R(x_r) \vee (S(x_s) \wedge \neg P(x_p) \wedge \mathrm{X}(\neg P(x_p)\ \mathrm{U}\ T(x_t))))))))$

P kommt vor S, T („1 Ursache-2 Wirkungen Precedence Chain"):

Globally	$\mathcal{Q}x((\mathrm{F}(S(x_s) \wedge \mathrm{X}(\mathrm{F}(T(x_t)))) \Rightarrow (\neg S(x_s)\ \mathrm{U}\ P(x_p))))$
Before R	$\mathcal{Q}x(\mathrm{F}(R(x_r)) \Rightarrow ((\neg(S(x_s) \wedge \neg R(x_r) \wedge \mathrm{X}(\neg R(x_r)\ \mathrm{U}\ (T(x_t) \wedge \neg R(x_r))))\ \mathrm{U}\ (R(x_r) \vee P(x_p))))$
After Q	$\mathcal{Q}x((\mathrm{G}(\neg Q(x_q))) \vee (\neg Q(x_q)\ \mathrm{U}\ (Q(x_q) \wedge ((\mathrm{F}(S(x_s) \wedge \mathrm{X}(\mathrm{F}(T(x_t))))) \Rightarrow (\neg S(x_s)\ \mathrm{U}\ P(x_p))))))$
Between Q and R	$\mathcal{Q}x(\mathrm{G}((Q(x_q) \wedge \mathrm{F}(R(x_r))) \Rightarrow ((\neg(S(x_s) \wedge \neg R(x_r) \wedge \mathrm{X}(\neg R(x_r)\ \mathrm{U}\ (T(x_t) \wedge \neg R(x_r))))\ \mathrm{U}\ (R(x_r) \vee P(x_p)))))$
After Q until R	$\mathcal{Q}x(\mathrm{G}(Q(x_q) \Rightarrow (\neg(S(x_s) \wedge \neg R(x_r) \wedge \mathrm{X}(\neg R(x_r)\ \mathrm{U}\ (T(x_t) \wedge \neg R(x_r))))\ \mathrm{U}\ (R(x_r) \vee P(x_p)) \vee \mathrm{G}(\neg(S(x_s) \wedge \mathrm{X}(\mathrm{F}(T(x_t))))))))$

D.8. Response Chain

P folgt S, T („2 Stimuli-1 Wirkung Response Chain"):

Globally	$\mathcal{Q}x(\mathrm{G}(S(x_s) \wedge \mathrm{X}(\mathrm{F}(T(x_t))) \Rightarrow \mathrm{X}(\mathrm{F}(T(x_t) \wedge \mathrm{F}(P(x_p))))))$
Before R	$\mathcal{Q}x(\mathrm{F}(R(x_r)) \Rightarrow (S(x_s) \wedge \mathrm{X}(\neg R(x_r)\ \mathrm{U}\ T(x_t)) \Rightarrow \mathrm{X}(\neg R(x_r)\ \mathrm{U}\ (T(x_t) \wedge \mathrm{F}(P(x_p)))))\ \mathrm{U}\ R(x_r))$
After Q	$\mathcal{Q}x(\mathrm{G}(Q(x_q) \Rightarrow \mathrm{G}(S(x_s) \wedge \mathrm{X}(\mathrm{F}(T(x_t))) \Rightarrow \mathrm{X}(\neg T(x_t)\ \mathrm{U}\ (T(x_t) \vee \mathrm{F}(P(x_p))))))))$
Between Q and R	$\mathcal{Q}x(\mathrm{G}((Q(x_q) \wedge \mathrm{F}(R(x_r))) \Rightarrow (S(x_s) \wedge \mathrm{X}(\neg R(x_r)\ \mathrm{U}\ T(x_t)) \Rightarrow \mathrm{X}(\neg R(x_r)\ \mathrm{U}\ (T(x_t) \wedge \mathrm{F}(P(x_p)))))\ \mathrm{U}\ R(x_r)))$
After Q until R	$\mathcal{Q}x(\mathrm{G}(Q(x_q) \Rightarrow (S(x_s) \wedge \mathrm{X}(\neg R(x_r)\ \mathrm{U}\ T(x_t)) \Rightarrow \mathrm{X}(\neg R(x_r\ \mathrm{U}\ (T(x_t) \wedge \mathrm{F}(P(x_p)))))\ \mathrm{U}\ (R(x_r) \vee \mathrm{G}(S(x_s) \wedge \mathrm{X}(\neg R(x_r)\ \mathrm{U}\ (T(x_t) \wedge \mathrm{F}(P(x_p))))))))))$

S, T folgen auf P („1 Stimulus-2 Wirkungen Response Chain"):

Globally	$\mathcal{Q}x(\mathrm{G}(P(x_p) \Rightarrow \mathrm{F}(S(x_s) \wedge \mathrm{X}(\mathrm{F}(T(x_t))))))$
Before R	$\mathcal{Q}x(\mathrm{F}(R(x_r)) \Rightarrow (P(x_p) \Rightarrow (\neg R(x_r)\ \mathrm{U}\ (S(x_s) \wedge \neg R(x_r) \wedge \mathrm{X}(\neg R(x_r)\ \mathrm{U}\ T(x_t)))))\ \mathrm{U}\ R(x_r))$
After Q	$\mathcal{Q}x(\mathrm{G}(Q(x_q) \Rightarrow \mathrm{G}(P(x_p) \Rightarrow (S(x_s) \wedge \mathrm{X}(\mathrm{F}(T(x_t)))))))$
Between Q and R	$\mathcal{Q}x(\mathrm{G}((Q(x_q) \wedge \mathrm{F}(R(x_r))) \Rightarrow (P(x_p) \Rightarrow (\neg R(x_r)\ \mathrm{U}\ (S(x_s) \wedge \neg R(x_r) \wedge \mathrm{X}(\neg R(x_r)\ \mathrm{U}\ T(x_t)))))\ \mathrm{U}\ R(x_r)))$
After Q until R	$\mathcal{Q}x(\mathrm{G}(Q(x_q) \Rightarrow (P(x_p) \Rightarrow (\neg R(x_r)\ \mathrm{U}\ (S(x_s) \wedge \neg R(x_r) \wedge \mathrm{X}(\neg R(x_r)\ \mathrm{U}\ T(x_t)))))\ \mathrm{U}\ (R(x_r) \vee \mathrm{G}(P(x_p) \Rightarrow (S(x_s) \wedge \mathrm{X}(\mathrm{F}(T(x_t))))))))$

E. DTDs für die Datenbasis des Constraint Editors

E.1. DTD für die Logic Patterns

```
<!-- The collection of logic patterns contains at least -->
<!-- one pattern                                        -->
<!ELEMENT patternSystem (pattern+)>

<!-- raw-tags and placeholders toggles                  -->
<!ELEMENT pattern (raw|(p|q|r|s|t|u|v|w))*>
<!ATTLIST pattern name CDATA #REQUIRED
                  scope CDATA #REQUIRED>

<!ELEMENT raw (#PCDATA)>

<!-- The placeholders                                   -->
<ELEMENT p EMPTY>
<ELEMENT q EMPTY>
<ELEMENT r EMPTY>
<ELEMENT s EMPTY>
<ELEMENT t EMPTY>
<ELEMENT u EMPTY>
<ELEMENT v EMPTY>
<ELEMENT w EMPTY>
```

Programmtext E.1: DTD für die Logic Patterns

E.2. DTD für ein Composite Pattern

```
<!-- Composite Pattern consists of a description,       -->
<!-- the input description and the rules how to map it  -->
<!-- to the logic patterns                              -->
<!ELEMENT CompositePattern (patternDescription, input,
                            generateLogicPattern?)>
<!ATTLIST CompositePattern name CDATA #REQUIRED
                           class CDATA #REQUIRED>

<!ELEMENT patternDescription (#PCDATA)>
```

```
10
11  <!-- The input description consists of several steps    -->
12  <!ELEMENT input (step+)>
13
14  <!-- An input step consists of its description          -->
15  <!-- and the corresponding input dialog.                -->
16  <!ELEMENT step (stepDescription, dialog)>
17  <!ATTLIST step no CDATA #REQUIRED>
18
19  <!ELEMENT stepDescription (#PCDATA)>
20  <!ELEMENT dialog (#PCDATA)>
21
22  <!-- Two types of link are needed                       -->
23  <!ELEMENT simpleLink EMPTY>
24  <!ATTLIST simpleLink step CDATA #REQUIRED>
25
26  <!ELEMENT advancedLink (#PCDATA)>
27  <!ATTLIST advancedLink step CDATA #REQUIRED>
28
29  <!-- The generation of constraints                      -->
30  <!ELEMENT generateLogicPattern (logicPattern+)>
31
32  <!-- For a constraint the "name" and the corresponding  -->
33  <!-- pattern are needed. It consists of a description   -->
34  <!-- and a declaration how to generate the placeholders -->
35  <!-- for the logic pattern                              -->
36  <!ELEMENT logicPattern (logicPatternDescription, sibs)>
37  <!ATTLIST logicPattern  type (single|multiple) #REQUIRED
38                          pattern CDATA #REQUIRED
39                          scope CDATA #REQUIRED>
40
41  <!ELEMENT logicPatternDescription (#PCDATA)>
42
43  <!-- The placeholders for the test blocks in the pattern -->
44  <!-- system                                               -->
45  <!ELEMENT sibs (p?, q?, r?, s?, t?, u?, v?, w?)>
46
47  <!ENTITY % sib "(name, (arg* | args))">
48
49  <!ELEMENT p %sib; >
50  <!ELEMENT q %sib; >
51  <!ELEMENT r %sib; >
52  <!ELEMENT s %sib; >
53  <!ELEMENT t %sib; >
54  <!ELEMENT u %sib; >
55  <!ELEMENT v %sib; >
56  <!ELEMENT w %sib; >
57
58  <!-- The name of a test block can be given directly     -->
```

```
59 <!-- or through a link or through a composition, needed -->
60 <!-- for GUI tests or actions                            -->
61 <!ELEMENT name (#PCDATA | advancedLink)*>
62
63 <!-- <arg> generates the test block parameter "name"    -->
64 <!-- The Value of the parameter can be given directly   -->
65 <!-- or can be specified through a link                 -->
66 <!ELEMENT arg (#PCDATA | advancedLink)*>
67 <!ATTLIST arg name CDATA #REQUIRED
68               required (yes|no) "yes">
69
70 <!-- <args> takes all <name>/<content>-pairs out of     -->
71 <!-- a certain step and generates parameters for a      -->
72 <!-- test block                                         -->
73 <!ELEMENT args (simpleLink)>
74
```

DTD für ein Composite Pattern

E.3. DTD für eine Constraint

```
<!-- A concrete Constraint consists of a description        -->
<!-- and several steps.                                     -->
<!ELEMENT constraint (description, step*)>

<!-- The Composite Pattern and the name of the constraint   -->
<!-- will be given as arguments                             -->
<!ATTLIST constraint pattern CDATA #REQUIRED
                     name CDATA #REQUIRED>

<!ELEMENT description (#PCDATA)>

<!ELEMENT step nameSib, parameter*)>

<!ATTLIST step no CDATA #REQUIRED>

<!ELEMENT nameSib (#PCDATA)>

<!-- Parameters of the test blocks                          -->
<!ELEMENT parameter (name,content)>
<!ELEMENT name (#PCDATA)>
<!ELEMENT content (#PCDATA | forAll | exists)*>
<!ELEMENT forAll EMPTY>
<!ATTLIST forAll placeholder CDATA #REQUIRED>
<!ELEMENT exists EMPTY>
<!ATTLIST exists placeholder CDATA #REQUIRED>
```

Programmtext E.2: DTD für eine Constraint

Autoren

Literaturverzeichnis

[1] Lehrstuhl V am Fachbereich Informatik an der Universität Dortmund. Zwischenbericht der pg 430 interakt, 2003.

[2] The Agent Building Center: *Bringing a Web Application into Operation.*

[3] Tomcat *http://jakarta.apache.org/tomcat/.*

[4] Velocity Scripting Engine *http://jakarta.apache.org/velocity/.*

[5] Java Development Kit *http://java.sun.com/.*

[6] PostgreSQL *http://www.postgresql.com/.*

[7] Tcl Developer Site *http://www.tcl.tk/.*

[8] Jean-Claude Fernandez. An implementation of an efficient algorithm for bisimulation equivalence. *Science of Computer Programming*, 13(1):219–236, 1989.

[9] Flex and Bison *http://www.gnu.org/.*

[10] A. Holzmann. Der metaframe-interpreter: Entwicklung und implementierung eines dynamischen modulkonzeptes. Master's thesis, Universität Passau, 1997.

[11] John E. Hopcroft, Jeffrey D. Ullman, and Rajeev Motwani. *Einfhrung in die Automatentheorie, Formale Sprachen und Komplexitï¿½stheorie.* Pearson Studium, zweite edition, 2002.

[12] J.C. Corbett M.B. Dwyer, G.S. Avrunin. In *Patterns in property specifications for finite-state verification*, pages 411–421. May 1999. In Proc. of Int. Conf. on Software Engineering.

[13] Robin Milner. *Communication and Concurrency.* Prentice-Hall, 1989. MIL r 89:1 1.Ex.

[14] O. Niese. *An integrated approach for testing complex systems.* PhD thesis, University of Dortmund, 2003. To be published.

[15] The XML C parser and toolkit of Gnome *http://www.xmlsoft.org/.*

[16] R. Tarjan R. Paige. Three partition refinement algorithms. *SIAM J. Comput.*, 6(16), 1987.

[17] H. Raffelt. Automatisiertes testen von internetapplikationen. Master's thesis, Universität Dortmund, 2002.

[18] S.S.-Adams. Metamethods: The mvc paradigm. *HOOPLA!*, 1(4), July 1988.

[19] B. Steffen and T. Margaria. Metaframe in practice: Design of intelligent network services. In B. Steffen E.-R. Olderog, editor, *Correct System Design - Recent Insights and Advances*, volume 1710 of *Lecture Notes in Computer Science (LNCS) State-of-the-Art Survey*, pages 390–415. Springer-Verlag, Heidelberg, Germany, October 1999. Dedicated to Hans Langmaack on the occasion of his retirement from his professorship.

[20] Ingo Wegener. *Grundlagen der theoretischen Informatik*. Teubner Verlag, 1999.

Index