Universität Koblenz–Landau
Fachbereich Informatik
Institut für Softwaretechnik

Studienarbeit

GXL-Validator

Validierung von GXL-Dokumenten auf Instanz-, Schema- und Metaschemaebene

Alexander Kaczmarek
In der Weglänge 9
56072 Koblenz

betreut von Prof. Jürgen Ebert, Dr. Andreas Winter,
Dipl. Inf. Volker Riediger

Dokumentenversion: 1.6, 3. September 2003

Inhaltsverzeichnis

Kapitel 1

Einleitung

Graphen sind ein weit verbreitetes Medium zur Darstellung beliebiger Daten und werden von einer Vielzahl verschiedenster Programme zur Datenhaltung verwendet. Ein Anwendungsbereich ist das Reverse Engineering. Viele Werkzeuge aus diesem Bereich arbeiten mit graphbasierten Daten. Um diese Daten zwischen verschiedenen Programmen austauschen zu können, ist die Entwicklung einer standardisierten Sprache ein wichtiger Schritt.
GXL (*Graph eXchange Language* [1]) ist ein Austauschformat für graphbasierte Daten. Ziel der Entwicklung und Standardisierung dieser XML-Subsprache (*eXtensible Markup Language*, vgl. `http://www.w3.org/XML`) ist es, den Datenaustausch zwischen möglichst vielen Tools aus dem Bereich des Reverse Engineerings zu ermöglichen. Um dieses Ziel zu erreichen, müssen mit GXL möglichst alle Arten von Graphen repräsentiert werden können. GXL ist daher eine sehr mächtige und umfangreiche Sprache, die typisierte, attributierte, gerichtete und geordnete Graphen, sowie Hypergraphen und hierarchische Graphen darstellen kann. Die verschiedenen Sprachelemente und deren Verwendung führen zu einer Vielzahl von Einschränkungen an GXL-Graphen, die zum einen in der GXL-DTD (vgl. Anhang A.5) und auch in dem GXL-Graphmodel und GXL-Metaschema (vgl. Anhang A.6) erfasst worden sind. Diese Liste von Einschränkungen ist allerdings nicht vollständig.
Teilweise liegen weitere Einschränkungen in textuellen Beschreibungen vor (vgl. [3], Kommentare zur DTD), allerdings genügen auch diese nicht. Die erste Aufgabe dieser Arbeit ist, alle Einschränkungen zu sammeln und bisher noch nicht erfasste aufzustellen, um eine möglichst komplette Liste von Einschränkungen an GXL-Graphen zu besitzen.
Die so gewonnenen Anforderungen an GXL-Graphen bilden die Grundlage für den zweiten Teil dieser Arbeit, die Entwicklung des sogenannten GXL-Validators. Der GXLValidator ist ein Programm, das die aufgestellten An-

forderungen für ein gegebenes GXL-Dokument überprüft und gegebenenfalls Abweichungen oder Verletzungen erfasst. Mit Hilfe des Programmes sollen also Fehler in GXL-Dokumenten gefunden werden und es dem Benutzer durch eine möglichst genaue Angabe der Fehlerquelle ermöglichen, diese Fehler zu beheben.
Kapitel 2 befasst sich mit den nötigen Grundlagen für diese Arbeit. Dabei wird näher betrachtet, wann ein GXL-Graph korrekt und wann er konform zu seinem Schema ist. Des Weiteren wird gezeigt, wie ein vollständiger Testablauf aussieht, um Korrektheit und Schemakonformität eines GXL-Graphen sicherzustellen.
Nachfolgend geht Kapitel 3 auf die Anforderungen an Instanzgraphen, an Schemakonformität und an Schemagraphen ein. Die Bedingungen werden einzeln natürlichsprachlich beschrieben und ihre Überprüfung anhand von Pseudocode dargestellt. Zu den jeweiligen Anforderungen werden außerdem Beispiele gegeben.
Kapitel 4 beschäftigt sich mit der Umsetzung der aufgestellten Anforderungen für den GXLValidator. Hierbei wird ein Überblick über die eingesetzten Mittel und die Struktur des Systems gegeben und auf die Kernelemente des Systems näher eingegangen. Außerdem werden Möglichkeiten der Erweiterung des GXLValidators und Wiederverwendung einzelner Komponenten vorgestellt. Danach folgt in Kapitel 5 eine Kurzanleitung, die in die Benutzung des GXLValidators einführt.
Abschließend werden in Kapitel 6 bisherige Erfahrungen bei Entwicklung und Einsatz des GXLValidators aufgeführt, sowie ein Ausblick auf die Weiterentwicklung des Systems bzw. des Konzepts in Hinblick auf GXL 1.1 gegeben.

Kapitel 2

Grundlagen

Dieses Grundlagenkapitel beschäftigt sich mit der Art und Weise, wie GXL Graphen und Schemata repräsentiert und welche Auswirkungen das auf die Validierung von GXL-Dokumenten hat. Außerdem werden wichtige Begriffe besprochen und es wird gezeigt, wie ein vollständiger Testablauf zur Überprüfung eines GXL-Graphen auszusehen hat.

2.1 Graphrepräsentation mittels GXL

Anhand eines Beispiels (vgl. [3], *complexExample*) wird in diesem Abschnitt gezeigt, wie GXL Graphen darstellt. Gegeben ist in Abbildung 2.1 ein Graph, der einen Auszug aus einem Programm repräsentiert.

Dargestellt werden die Funktionen *main*, *max* und *min*, jeweils repräsentiert durch einen Knoten, der über ein Attribut *name* verfügt, das den Funktionsnamen enthält. Beziehungen zwischen den Funktionen werden über Kanten realisiert. In diesem Beispiel enthält die Funktion *main* zwei Funktionsaufrufe (*FunctionCall*). Funktionsaufrufe werden ebenfalls durch Knoten dargestellt. Ob eine Funktion eine andere aufruft oder aufgerufen wird, wird durch Kanten festgehalten. So ruft die Funktion *main* in Zeile 8 die Funktion *max* auf. Beschrieben wird dieser Zustand durch zwei Kanten. Die Kante *e1* sagt aus, dass die Funktion *main* den Funktionsaufruf durchführt und zwar in Zeile 8, wie es das Attribut *line* angibt. Welche Funktion bei diesem Aufruf das Ziel ist, wird durch die Kante *e3* dargestellt. Wie bereits erwähnt wird die Funktion *max* aufgerufen.

Über weitere Kanten werden dem Funktionsaufruf Variablen zugeordnet, sowohl Eingangs- als auch Ausgangsvariablen. Im Fall der Eingangsvariablen sind in diesem Beispiel die Kanten geordnet, um die Reihenfolge der Variablen festzulegen. Die Variablen selbst werden wie Funktionen und Funkti-

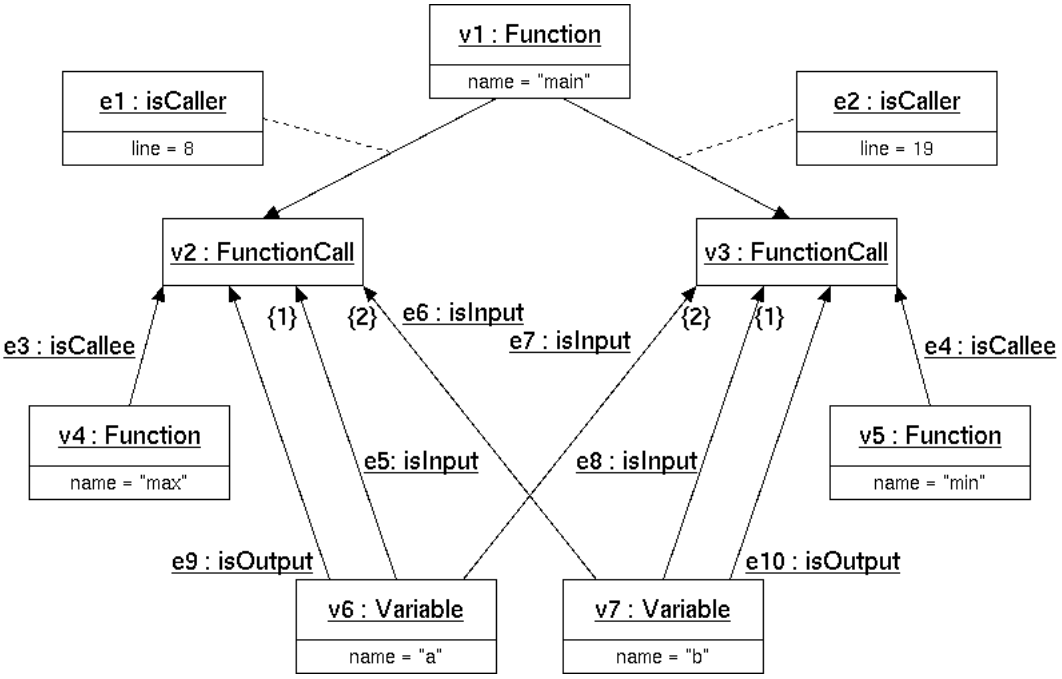

Abbildung 2.1: Typisierter, attributierter, gerichteter, geordneter Graph (Quelle: [3])

onsaufrufe durch Knoten repräsentiert.
Dieser Graph kann durch GXL dargestellt werden. GXL ist eine Subsprache von XML. Daher erfolgt die Darstellung in textueller Form. Abbildung 2.2 zeigt das GXL-Dokument, das diesen Graphen enthält. Knoten werden durch `<node>`-Tags beschrieben, Kanten analog durch `<edge>`-Elemente. Diese Tags enthalten eine eindeutige Id, so zum Beispiel `<node id="v1">`. Eine Id identifiziert ein einzelnes Element und wird dazu verwendet, um Kanten zwischen Elementen darzustellen. So verläuft z.B. die Kante `<edge id="e1" from="v1" to="v2">` vom Knoten *v1* zum Knoten *v2*. Attribute eines Graphelementes werden durch eingeschachtelte Elemente beschrieben. Das Tag `<attr name="name">` des Knotens *v1* repräsentiert so zum Beispiel das Attribut *name*.
Wie in diesem Beispiel-Graphen zu sehen, können sowohl Knoten als auch Kanten einen Typ besitzen. Diese Typen werden in Schemata zusammen mit den Beziehungen zwischen den einzelnen Typen definiert. Abbildung 2.3 zeigt das Schema, das dem Graph aus Abbildung 2.1 zugrunde liegt.
In GXL werden Schemata in Form von UML-Klassendiagrammen angegeben (vgl. `http://www.uml.org`). Knotenklassen werden mit Klassen, Kantenklassen mit Assoziationen definiert. Enthält ein Schema Relationsklassen, so werden diese durch n-äre Assoziationen beschrieben. In dem hier vorliegenden Schema wird festgelegt, dass Funktionen (*Function*) andere Funktionen aufrufen können (*FunctionCall*). Zu jedem Funktionsaufruf gehört ein Aufrufer (*isCaller*) und ein Aufgerufener (*isCallee*), sowie beliebig viele Eingangs- (*isInput*) und genau eine Ausgangsvariable (*isOutput*).
GXL-Schemata selbst können ebenfalls als Graphen aufgefasst werden, da UML-Klassendiagramme ihrerseits strukturierte Informationen sind. Daher ist es möglich, GXL-Schemata in GXL-Graphen zu transformieren. Abbildung 2.4 zeigt das Schema aus Abbildung 2.3 als Graph.
Für GXL-Schemagraphen existiert ein Schema, das die Menge aller gültigen GXL-Schemata definiert. Dieses Schema ist das GXL-Metaschema (vgl. Anhang A.6). Da es sich bei diesem Schema ebenfalls um ein GXL-Schema handelt, kann auch das Metaschema als GXL-Graph dargestellt werden. Die Besonderheit hierbei ist, dass das Schema des Metaschemagraphen das Metaschema selbst ist.
Eine Einführung und ein kompletter Überblick über GXL findet sich in [1]. Die Verwendung von XML, einer DTD und Schemata in Hinblick auf GXL wird außerdem ausführlich in [2] behandelt.

```
<?xml version="1.0"?>
<!DOCTYPE gxl SYSTEM
   "http://www.gupro.de/GXL/gxl-1.0.dtd">
<gxl xmlns:xlink="http://www.w3.org/1999/xlink">
   <graph id=complexExample">
      <type xlink:href=complexExampleSchema.gxl
         #complexExampleSchema"/>
      <node id="v1">
         <type xlink:href=complexExampleSchema.gxl
            #Function"/>
         <attr name=name">
               <string>main</string>
         </attr>
      </node>
      <node id="v2">
         <type xlink:href=complexExampleSchema.gxl
            #FunctionCall"/>
      </node>
      <node id="v3">
         <type xlink:href=complexExampleSchema.gxl
            #FunctionCall"/>
      </node>
      <node id="v4">
         <type xlink:href=complexExampleSchema.gxl
            #Function"/>
         <attr name=name">
            <string>max</string>
         </attr>
      </node>
      <node id="v5">
         <type xlink:href=complexExampleSchema.gxl
            #Function"/>
         <attr name=name">
            <string>min</string>
         </attr>
      </node>
      <node id="v6">
         <type xlink:href=complexExampleSchema.gxl
            #Variable"/>
         <attr name=name">
            <string>a</string>
         </attr>
      </node>
      <node id="v7">
         <type xlink:href=complexExampleSchema.gxl
            #Variable"/>
         <attr name=name">
            <string>b</string>
         </attr>
      </node>
      <edge id="e1from="v1to="v2">
         <type xlink:href=complexExampleSchema.gxl
            #isCaller"/>
         <attr name=line">
            <int>8</int>
         </attr>
      </edge>
      <edge id="e2from="v1to="v3">
         <type xlink:href=complexExampleSchema.gxl
            #isCaller"/>
         <attr name=line">
            <int>19</int>
         </attr>
      </edge>
      <edge id="e3from="v4to="v2">
         <type xlink:href=complexExampleSchema.gxl
            #isCallee"/>
      </edge>
      <edge id="e4from="v5to="v3">
         <type xlink:href=complexExampleSchema.gxl
            #isCallee"/>
      </edge>
      <edge id="e5from="v6to="v2toorder="1">
         <type xlink:href=complexExampleSchema.gxl
            #isInput"/>
      </edge>
      <edge id="e6from="v7to="v2toorder="2">
         <type xlink:href=complexExampleSchema.gxl
            #isInput"/>
      </edge>
      <edge id="e7from="v6to="v3toorder="2">
         <type xlink:href=complexExampleSchema.gxl
            #isInput"/>
      </edge>
      <edge id="e8from="v7to="v3toorder="1">
         <type xlink:href=complexExampleSchema.gxl
            #isInput"/>
      </edge>
      <edge id="e9from="v6to="v2">
         <type xlink:href=complexExampleSchema.gxl
            #isOutput"/>
      </edge>
      <edge id="e10from="v7to="v3">
         <type xlink:href=complexExampleSchema.gxl
            #isOutput"/>
      </edge>
   </graph>
</gxl>
```

Abbildung 2.2: GXL-Dokument

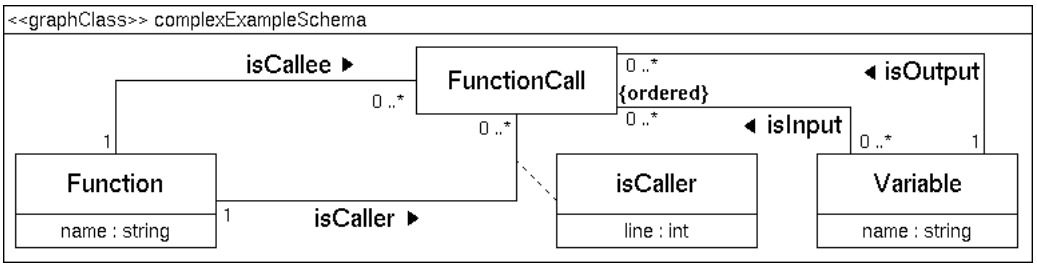

Abbildung 2.3: Schema als UML-Klassendiagramm (Quelle: [3])

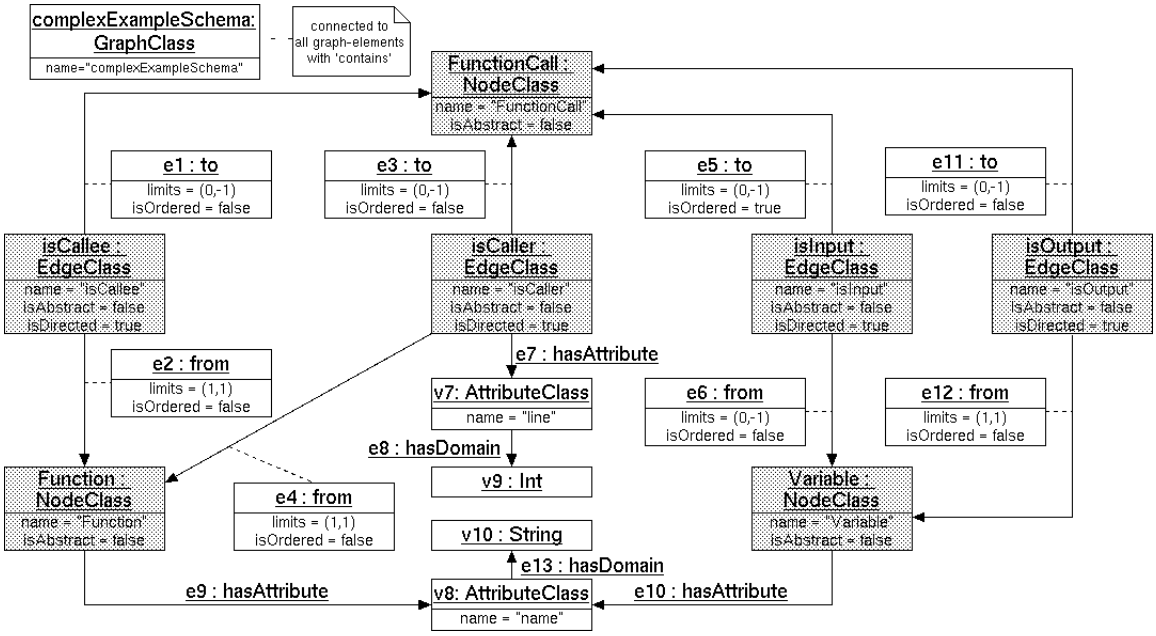

Abbildung 2.4: Schemagraph (Quelle: [3])

2.2 Validierung von GXL-Graphen und GXL-Schemata

GXL ist eine Subsprache von XML (*eXtensible Markup Language*), die durch eine DTD (*Document Type Definition*) definiert ist. Die GXL-DTD (vgl. Anhang A.5) bestimmt den syntaktischen Aufbau aller gültigen GXL-Dokumente. Zusätzlich beschreibt das Graphmodell (vgl. Anhang A.6), wie die in der GXL-DTD definierten Elemente zur Darstellung von Graphen verwendet werden sollen. DTD-Konformität eines GXL-Dokumentes wird im Rahmen dieser Arbeit vorausgesetzt. Aus dem Graphmodell sowie textuellen Beschreibungen zur Verwendung des Graphmodells resultieren verschiedene Anforderungen, die GXL-Graphen erfüllen müssen, aber durch die DTD bzw. XML-Schema nicht formuliert werden können. Die Überprüfung dieser **Instanzkorrektheit** ist die erste Aufgabe des GXLValidators.
Um weitere Eigenschaften von Graphen aufzustellen zu können, verwendet GXL Schemata. Ein Schema definiert eine Menge von GXL-Graphen, die jeweils die durch das Schema aufgestellten Bedingungen erfüllen. Ähnlich einem Klassendiagramm kann ein Schema Knoten-, Kanten- und Relationsklassen und deren Beziehungen untereinander definieren. Ein Graph ist nur dann konform zu seinem Schema, wenn er diese Klassendefinitionen und Beziehungen erfüllt. Da GXL-Schemata in GXL beschrieben werden, sind auch sie GXL-Instanzgraphen. Für Schemagraphen existiert daher ebenfalls ein Schema, das GXL-Metaschema (vgl. Anhang A.6). Dieses Metaschema bestimmt die Menge aller gültigen GXL-Schemata.
Für diese Arbeit ergibt sich aus der Verwendung von Schemata ein weiterer Aspekt. Neben der Überprüfung der Instanzkorrektheit von Graphen muss deren **Schemakonformität** getestet werden. Dabei ist allerdings zu berücksichtigen, dass ein GXL-Graph nicht direkt zu einem ganzen Schema passen muss, sondern zu seiner Graphklasse. Ein GXL-Schema ist ein GXL-Graph, dessen Schema das GXL-Metaschema ist. Eine Graphklasse subsumiert Knoten-, Kanten und Relationsklassen mitsamt deren Beziehungen untereinander. Ein Schema wiederum kann mehrere Graphklassen enthalten. Schemakonformität impliziert also die Konformität eines Graphen zu einer Graphklasse.
Soll ein Graph auf Schemakonformität untersucht werden, so muss zunächst sichergestellt sein, dass das Schema selbst korrekt ist. Neben Instanzkorrektheit und Schemakonformität müssen dazu weitere Anforderungen speziell an Schemagraphen betrachtet werden. Diese Anforderungen ergeben sich zum einen direkt aus dem GXL-Metaschema und dessen Beschreibung. Im Rahmen dieser Arbeit wurde diese Anforderungsliste um zusätzliche Punkte er-

weitert, mit dem Ziel der vollständigen Erfassung. Die Sicherstellung dieser **Schemakorrektheit** ist die dritte Aufgabe des GXLValidators.

2.3 Begriffe

Sowohl in der Einleitung wie auch im vorangegangenen Abschnitt wurde häufiger von Korrektheit und Konformität gesprochen. Im Folgenden sollen diese Begriffe nochmals zusammenfassend aufgegriffen werden, um sie voneinander abzugrenzen.

Instanzkorrektheit

Ein GXL-Graph ist **instanzkorrekt**, gdw. seine XML-Repräsentation konform ist zur GXL-DTD und alle Anforderungen an Instanzgraphen erfüllt sind (vgl. dazu 3.4).

Schemakonformität von GXL-Graphen

Ein GXL-Graph ist **schemakonform**, gdw. er instanzkorrekt ist und die durch das Schema aufgestellten Anforderungen erfüllt sind (vgl. dazu 3.5).

Schemakorrektheit

Ein Schema ist **schemakorrekt**, gdw. es schemakonform zu dem GXL-Metaschema ist und die Anforderungen an GXL-Schemata erfüllt (vgl. dazu 3.6).

Der folgende Abschnitt betrachtet nun die Reihenfolge, in der die oben angesprochenen Eigenschaften Instanzkorrektheit, Schemakonformität und Schemakorrektheit untersucht werden müssen.

2.4 Testablauf

Soll ein GXL-Graph auf Korrektheit hin überprüft werden, so reicht es nicht, nur den Graphen selbst zu betrachten. Um eine vollständige Korrektheit sicherzustellen, ist es vielmehr notwendig, auch das Schema des Graphen zu untersuchen. Da ein Schema selbst aber wiederum ein GXL-Graph ist, muss auch hier prinzipiell das zugrunde liegende Schema untersucht werden, also das Metaschema. Der Testvorgang bezogen auf einen Graphen stellt sich im Idealfall also dar wie es in Abbildung 2.5 zu sehen ist.

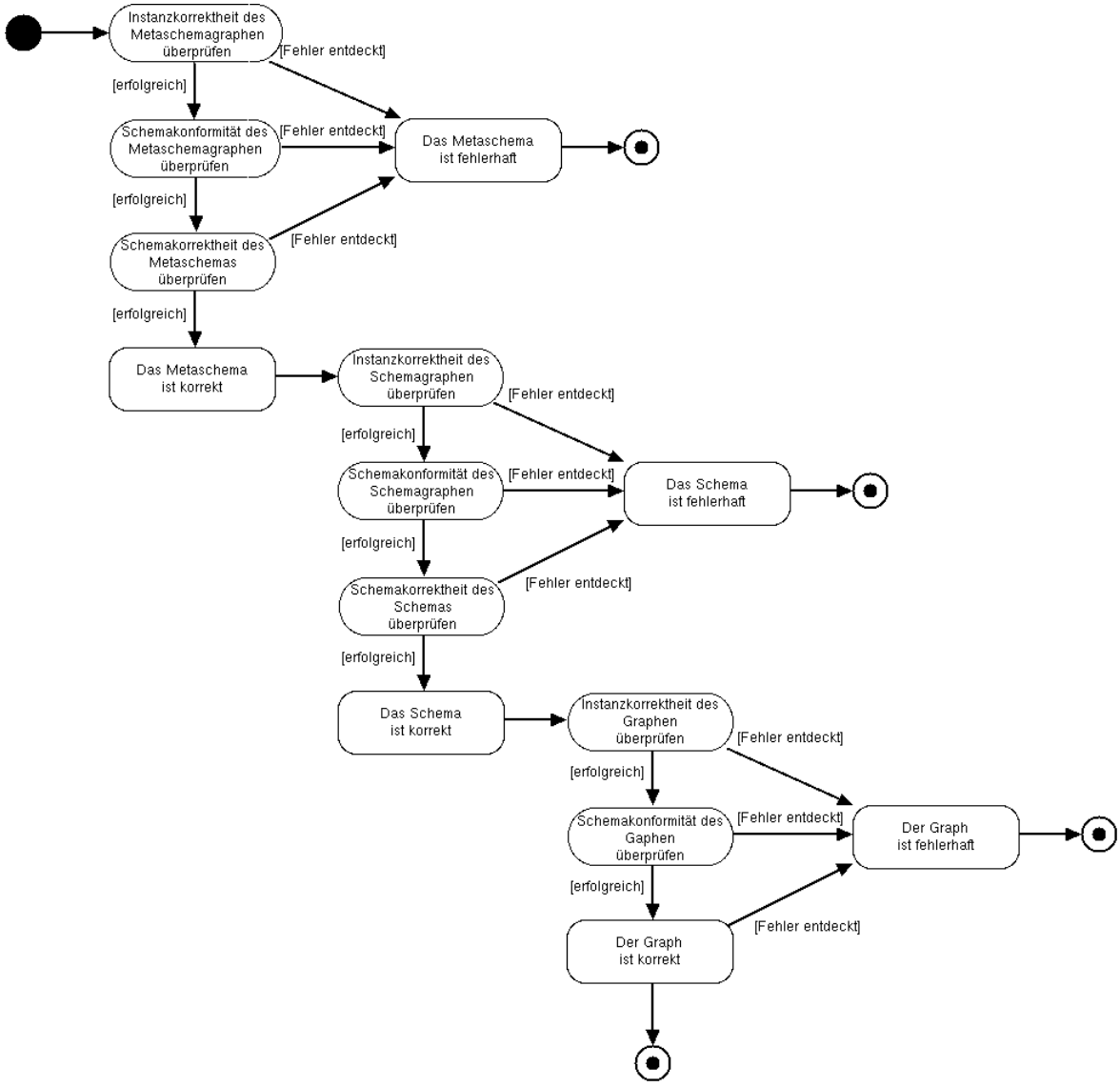

Abbildung 2.5: Vollständiger Testablauf

Bevor das Schema des Graphen untersucht werden kann, muss sichergestellt sein, dass das Metaschema selbst korrekt und schemakonform zu sich selbst ist. Ist dies der Fall, so kann anschließend überprüft werden, ob das Schema des zu untersuchenden Graphen eine gültige Instanz, schemakonform zu dem GXL-Metaschema und ein gültiges Schema ist. Erst wenn dies gewährleistet ist, kann der Graph selbst betrachtet werden.
Wie erwähnt stellt dieses Vorgehen den Idealfall dar. In der Praxis kann davon ausgegangen werden, dass das Metaschema korrekt ist. Das Schema eines Graphen sollte allerdings auf Korrektheit untersucht werden, bevor der Graph auf Korrektheit und Schemakonformität getestet wird.
Nachdem dieses Kapitel die nötigen Grundlagen betrachtet hat, widmet sich das folgende Kapitel den Anforderungen an GXL-Instanz- und -Schemagraphen.

Kapitel 3

Anforderungen an GXL-Instanz- und -Schemagraphen

Dieses Kapitel beschäftigt sich mit den Anforderungen an GXL-Instanz und -schemagraphen, die erfüllt sein müssen, um Korrektheit und Schemakonformität sicherzustellen. Jede dieser Anforderungen wird einzeln natürlichsprachlich beschrieben und mit geeigneten Beispielen veranschaulicht. Außerdem wird für jede Anforderung eine in Pseudocode formulierte Funktion angegeben, die die jeweilige Anforderung für einen gegebenen Graphen oder ein gegebenes Schema untersucht.

3.1 Pseudocode

Zu jeder einzelnen Anforderung wird eine Funktion angegeben, die diese Anforderung bezogen auf einen Graphen testen und mögliche Abweichungen feststellen kann. Diese Funktionen werden mittels eines Pseudocodes formuliert, der sich an der Sprache *ObjectPascal* orientiert. Anzumerken ist in diesem Zusammenhang, dass in Pascal bei Ausführung einer Funktion der Wert von *result* bei Erreichen des Methodenendes zurückgegeben wird. Durch Zuweisung eines Wertes an *result* wird eine Funktion nicht beendet. Zusätzlich zu den bekannten Pascal-Sprachelementen werden weitere Konstrukte verwendet, die separat definiert werden (vgl. Anhang A.1).
Den Umgang mit erkannten Fehlern und sonstigen Abweichungen von den geforderten Eigenschaften zeigt der folgende Abschnitt.

3.2 Fehlerbehandlung

Für jede Anforderung wird im Folgenden eine in Pseudocode formulierte Funktion angegeben, die die Anforderung für einen gegebenen Graph oder Schema überprüfen kann. Wird innerhalb einer solchen Funktion ein Fehler entdeckt, so wird dieser protokolliert. Diese Aufgabe übernimmt ein *Errorhandler*.
Abbildung 3.1 zeigt den Aufbau des Errorhandlers, den die Pseudocode-Funktionen verwenden. Dieser ist in der Lage, zwischen zwei Fehlerklassen, *Error* und *Warning*, zu unterscheiden. Außerdem kann die Anzahl der bereits gefundenen Fehler abgefragt und das Fehlerprotokoll angezeigt werden. Bei der späteren Implementation (vgl. Kapitel 4) des GXLValidators wurde dieses Konzept eines Errorhandlers ebenfalls aufgegriffen.
Bevor die eigentlichen Anforderungen thematisiert werden, geht der nächste Abschnitt auf die Darstellung der Beispiele ein.

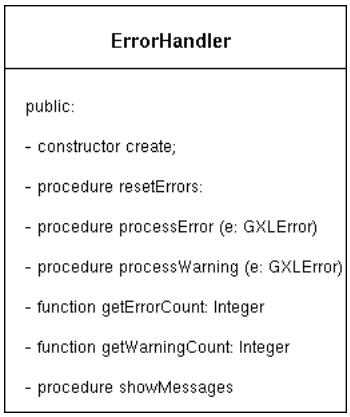

Abbildung 3.1: Klasse ErrorHandler

3.3 Beispiele

Zu jeder in den folgenden Abschnitten aufgeführten Anforderung werden zur Veranschaulichung Beispiele gegeben. Die Beispiele erfolgen in Form von UML-Objektdiagrammen für Graphen und UML-Klassendiagrammen für Schemata (vgl. `http://www.uml.org`). Dabei wird jeweils ein Positiv- und ein Negativbeispiel angeführt.
Das Positivbeispiel erfüllt die jeweilige Anforderung, während das Negativbeispiel sie verletzt. Um redundante Beispiele zu vermeiden, werden neue

Positivbeispiele nur dann angegeben, wenn bereits vorhandene für die Veranschaulichung der Anforderung nicht geeignet sind. Die Graphen aus den einzelnen Beispielen beziehen sich dabei auf die Schemata aus den Abbildungen 3.2, 3.3, 3.4 und 3.5.
Die nun folgenden Abschnitte beschäftigen sich mit den Anforderungen an GXL-Instanz- und -schemagraphen.

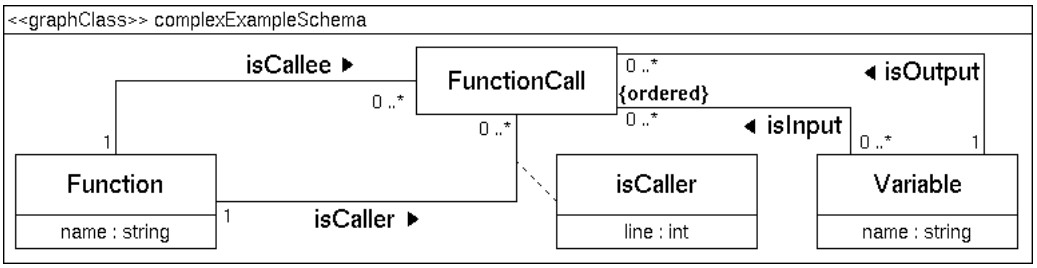

Abbildung 3.2: Beispielschema 1 (Quelle: [3])

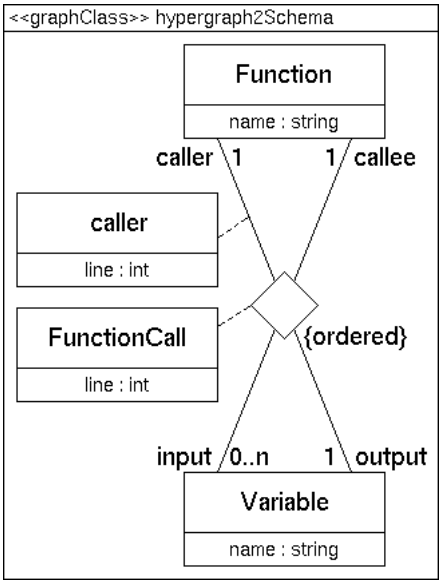

Abbildung 3.3: Beispielschema 2 (Quelle: [3])

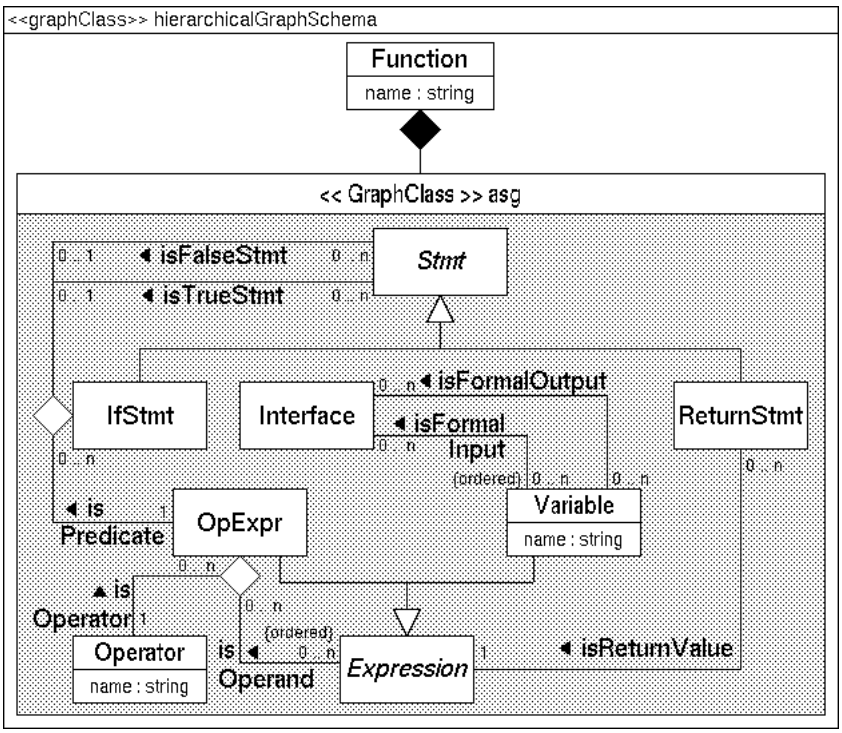

Abbildung 3.4: Beispielschema 3 (Quelle: [3])

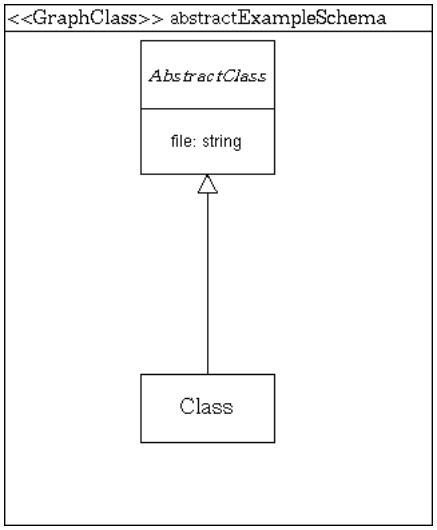

Abbildung 3.5: Beispielschema 4

3.4 Anforderungen an GXL-Graphen

Dieser Abschnitt beschäftigt sich mit den Anforderungen an GXL-Graphen ohne Berücksichtigung von Schemata. Aufbauend auf der geforderten DTD-Konformität werden weitere Bedingungen betrachtet, die erfüllt sein müssen, damit ein GXL-Graph instanzkorrekt ist.

3.4.1 Korrektheit der Graphattribute

Beschreibung

Ein GXL-Graph besitzt Attribute, die Informationen über den Graphen enthalten. Diese Attribute sind *edgeids (bool)*, *edgemode (directed, undirected, defaultdirected, defaultundirected)* und *hypergraph (bool)*. Das Attribut *edgeids* gibt an, ob Kanten eine Id besitzen dürfen, *edgemode* definiert die zulässigen Richtungsangaben für Kanten und Relationsenden und *hypergraph* sagt aus, ob es sich um einen Graphen mit Relationen handelt oder nicht.
Damit ein Graph instanzkorrekt ist, müssen die Graphattribute mit den tatsächlichen Gegebenheiten des Graphen übereinstimmen.

Pseudocode

Die Funktion `testInstanceGraphAttributes` überprüft, ob ein Graph G die durch die Graphattribute aufgestellten Anforderungen erfüllt. Dazu wird untersucht, ob alle Kanten gemäß dem Graphattribut *edgeids* eine Id besitzen oder nicht. Es wird des Weiteren überprüft, ob alle Kanten und Relationsenden den Richtungsmodus einhalten und ob keine Relationen existieren, wenn das Attribut *hypergraph* dies angibt. Sollte es sich laut *hypergraph* um einen Hypergraphen handeln, so ist umgekehrt die Existenz einer Relation optional und muss nicht überprüft werden.

```
function testInstanceGraphAttributes (
             G: Graph;
             eh: ErrorHandler): Boolean;
var
  isDirected,
  isUndirected,
  hasId: Boolean;
  r: Relation;
```

```
begin
  result := true;

  isDirected := (G.getEdgeMode=directedEdges);
  isUndirected := (G.getEdgeMode=unDirectedEdges);

  foreach Edge e of G do
    begin
      hasId := (e.getId<>'');
      if (hasId<>G.hasEdgeIds) then
        begin
          eh.processError (GXLEdgeIdError.create(e));
          result := false;
        end;

      if (e.getIsDirected and isUndirected) or
         ((not e.getIsDirected) and isDirected) then
        begin
          eh.processError (GXLEdgeDirectionError.create(e));
          result := false;
        end;
    end;

  if (not G.isHyperGraph) then
    if (G.hasRelation) then
      begin
        r := graph.getAnyRelation;
        eh.processError (GXLHypergraphError.create(r));
        result := false;
      end;

  foreach Relation r of G do
    foreach RelationEnd relEnd of r do
      if (relEnd.isDirected and isUndirected) or
         ((not relEnd.isDirected) and isDirected)) then
        begin
          eh.processError (
            GXLRelEndDirectionError.create(rel, relEnd));
          result := false;
        end;
end;
```

3.4.2 Korrektheit der Attribute

Beschreibung

Elemente von GXL-Graphen (z.B. Knoten, Kanten, Relationen, Graphen) können attributiert sein. Jedes Attribut besitzt einen Bezeichner (Namen), einen Typ und einen Wert.
Damit ein Graph instanzkorrekt ist, müssen die Bezeichner aller Attribute eines jeden attributierten Elementes eindeutig sein. D.h. zwei Attribute eines Elementes dürfen nicht den gleichen Namen besitzen. Außerdem wird gefordert, dass die Attributbelegungen aller im Graph vorkommenden Attribute zu dem jeweiligen Wertebereich passen, soweit dies ohne Rückgriff auf das Schema überprüft werden kann. So darf ein Integer-Attribut z.B keinen String-Wert besitzen (`<int>abc<\int>`).

Beispiele

Ein Positivbeispiel für die Anforderung an korrekte Attribute ist der Graph in Abbildung 3.6. Dagegen verletzt der Graph in Abbildung 3.7 die Forderung nach eindeutigen Attributbezeichnern.

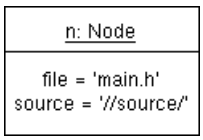

Abbildung 3.6: Graph mit eindeutigen Attributbezeichnern

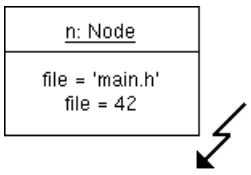

Abbildung 3.7: Graph mit nicht eindeutigen Attributbezeichnern

Pseudocode

Die nun folgenden Funktionen überprüfen, ob ein Graph G die gerade formulierten Anforderungen erfüllt. Der Funktion `testInstanceAttributes`,

die die Erfüllung der Anforderung testet, steht dazu eine weitere Funktion namens `testElement` zur Verfügung, die den Test bezogen auf ein konkretes attributiertes Element und dessen Attribute durchführt.

```
function testElement (ae: AttributedElement;
                      var attributeNames: Set of String;
                      eh: ErrorHandler): Boolean;
  foreach Attribute a of ae do
    begin
      if attributeNames.hasElement(a.getName) then
        begin
          eh.processError (
            GXLDuplicateAttributeNameError.create(ae, a));
          result := false;
        end
      else
        attributeNames.add (attrName);

      if not (a.getValue is of type a.getValueType) then
        begin
          eh.processError (
            GXLWrongAttributeValueError.create(ae, a));
          result := false;
        end;

      foreach Attribute innerAttr of a do
        result = (testElement(a, eh) and result);
    end;
end;

function testInstanceAttributes  (G: Graph;
                                  eh: ErrorHandler): Boolean;
var
  attributeNames: Set of String;
begin
  result := true;
  foreach AttributedElement ae of G do
    result := (testElement (ae, attributeNames, eh) and result);
end;
```

3.4.3 Korrekte Kanten-Endelemente

Beschreibung

Kanten verbinden Graphelemente. Dies können Knoten, Kanten oder Relationen sein[1]. In GXL werden diese Verbindungen durch Angabe der Ids der zu verbindenden Elemente realisiert. Allerdings besitzen nicht nur Graphelemente Ids, sondern auch Elemente wie z.B. Graphen.
Damit ein Graph instanzkorrekt ist, müssen die Endelemente aller Kanten Graphelemente sein. Diese Anforderung wird durch die GXLInstanceAPI (vgl. [4]) überprüft. Enthält ein Graph eine Kante, die ein Endelement besitzt, das kein Graphelement ist, so wird eine Exception ausgelöst.

3.4.4 Korrekte Relationsende-Zielelemente

Beschreibung

Analog zu den Kanten werden die Zielelemente von Relations-Enden über Ids festgehalten.
Damit ein Graph instanzkorrekt ist, müssen die Zielelemente aller Relationsenden Graphelemente sein. Ebenso wie bei den Kanten löst die verwendete GXLInstanceAPI eine Exception aus, falls das Zielelement eines Relationendes kein Graphelement ist.

3.4.5 Einhaltung der Ordnung

Beschreibung

Inzidenzen können in GXL angeordnet werden. Das bezieht sich sowohl auf Kanten wie auch Relationen.
Damit ein Graph instanzkorrekt ist, müssen bezogen auf ein Graphelement alle geordneten eingehenden (toOrder) und ausgehenden (fromOrder) Kanten, sowie Relationsenden (endOrder) die Ordnung einhalten. Das bedeutet, dass eine Ordnungszahl pro Graphelement nur einmal vergeben sein darf. Lücken in der Ordnung hingegen sind erlaubt. Handelt es sich bei dem Graphelement um eine Relation, so müssen ebenfalls die Ordnungszahlen aller geordneten Relationsenden dieser Relation (startOrder) berücksichtigt werden.

[1] GXL erlaubt also Kanten zwischen Kanten oder Relationen

Beispiele

Der Graph in Abbildung 3.8 beinhaltet ein Negativbeispiel für das Einhalten der Ordnung. Die Inzidenzen der Kanten *e5* und *e6* anm Knoten *v2* besitzen beide die gleiche Ordnungszahl. Somit ist der Graph kein korrekter GXL-Graph. Ein weiteres Negativbeispiel ist in Abbildung 3.9 zu sehen. Zwei Relationsenden der Relation *r1* weisen die gleiche Ordnungszahl (startOrder) auf.

Pseudocode

Die Funktion `testInstanceOrder` überprüft, ob ein Graph G die Anforderungen an die Einhaltung der Ordnungen erfüllt. Dazu werden die Funktionen `testEdges` und `testRelEnds` verwendet, die den Test jeweils für inzidente Kanten bzw. Relationenden durchführen.

```
function testEdges (ge: GraphElement;
                    var orders: Set of Integer;
                    G: Graph;
                    eh: ErrorHandler): Boolean;
begin
  result := true;
  foreach outgoing Edge e of ge do
    if (e.isOrdered and orders.hasElement(e.fromOrder)) then
      begin
        eh.processError (
          GXLDuplicateOrderError.create(ge, e.fromOrder));
        result := false;
      end
    else
      orders.add (fromOrder);
  foreach incoming Edge e of ge do
    if (e.isOrdered and orders.hasElement(e.toOrder)) then
      begin
        eh.processError (
          GXLDuplicateOrderError.create(ge, e.toOrder));
        result := false;
      end
    else
      orders.add (toOrder);
end;
```

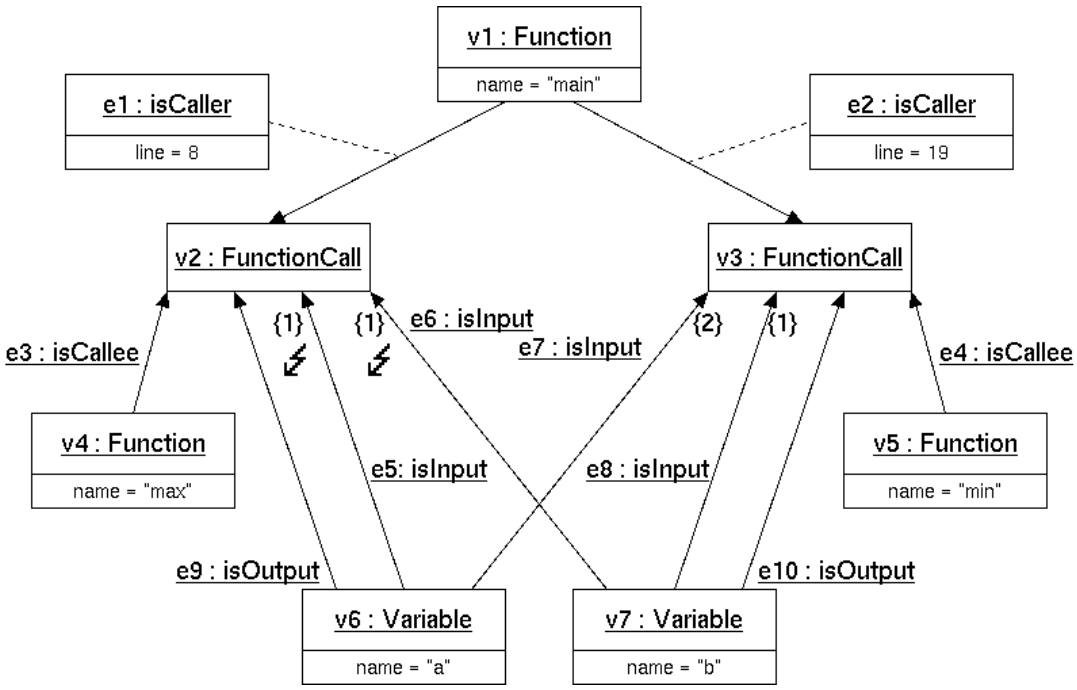

Abbildung 3.8: Graph mit fehlerhafter Kantenordnung

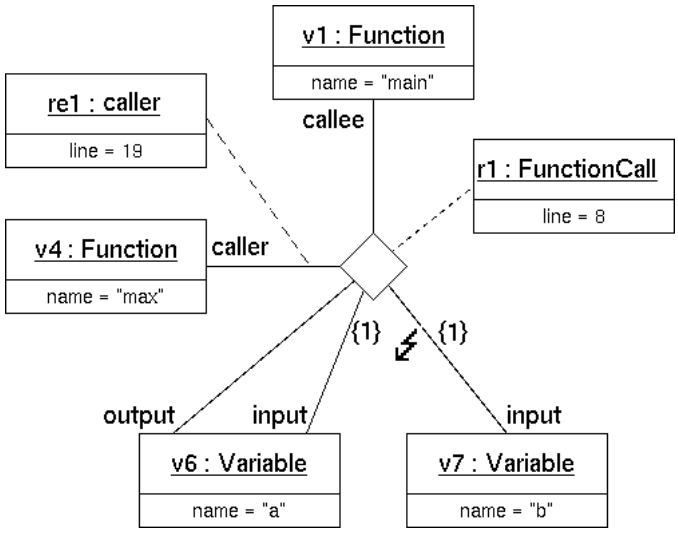

Abbildung 3.9: Graph mit fehlerhafter Relationsordnung

```
function testRelEnds (ge: GraphElement;
                      var orders: Set of Integer;
                      G: Graph;
                      eh: ErrorHandler): Boolean;
var
  relEndOrder: Integer;
begin
  result := true;
  foreach incident RelationEnd relEnd of ge do
    if (relEnd.isOrdered) then
      begin
        if (relEnd.getTarget = ge) then
          relEndOrder := relEnd.getEndOrder
        else
          relEndOrder := relEnd.getStartOrder;
        if orders.hasElement(relEndOrder) then
          begin
            eh.processError (
              GXLDuplicateOrderError.create (ge, relEndOrder));
            result := false;
          end
        else
          orders.add (relEndOrder);
      end;
end;

function testInstanceOrder (G: Graph;
                            eh: ErrorHandler): Boolean;
var
  orders: Set of Integer;
begin
  result := true;
  foreach GraphElement ge of G do
    begin
      orders.clear;
      result := (testEdges (ge, orders, G, eh) and
                 result);
      result := (testRelEnds (ge, orders, G, eh) and
                 result);
    end;
end;
```

3.5 Anforderungen an Schemakonformität

Nachdem sich der vorangegangene Abschnitt mit den Anforderungen an GXL-Graphen ohne Berücksichtigung eines Schemas beschäftigt hat, befasst sich dieser Teil der Arbeit mit den durch ein Schema aufgestellten Anforderungen.
Durch die Angabe eines Schemas werden von einem GXL-Graphen zusätzlich zur DTD-Konformität und den in Abschnitt 3.4 formulierten Bedingungen weitere Eigenschaften gefordert. Dies sind die Eigenschaften, die durch Klassendefinitionen und Beziehungen dieser Klassen untereinander aufgestellt werden. Der folgende Abschnitt beschäftigt sich mit diesen Anforderungen. Dabei wird davon ausgegangen, dass das angegebene Schema eine gültige Instanz des Metaschemas ist (vgl. Abschnitt 3.6).

3.5.1 Klassenzugehörigkeit

Beschreibung

Ein GXL-Schema definiert u.a. Graphelementklassen. Dies können im speziellen Fall Knoten-, Kanten- oder Relationsklassen sein. Jedes Graphelement eines GXL-Graphen enthält eine Angabe über den Typ, welcher mit einer Klasse im Schema übereinstimmen muss.
Damit ein Graph schemakonform ist, müssen alle im Graph vorkommenden Graphelemente Instanzen einer im Schema definierten Klasse sein, die zu der Graphklasse des Instanzgraphen gehört. Dabei darf es sich nicht um abstrakte Klassen handeln.

Beispiele

Abbildung 3.10 enthält ein Positivbeispiel für die Forderung nach Klassenzugehörigkeit. Alle Elemente des Graphen sind Instanzen aus dem Schema bekannter Klassen. Der Graph in Abbildung 3.11 hingegen ist nicht schemakonform (vgl. Abb. 3.2). Sowohl die Klasse der Kante *e1*, als auch die des Knotens *v1* existieren nicht in der Graphklasse. Analog zu diesem Beispiel gibt es Beispiele für nicht definierte Relationsklassen.
Der Graph in Abb. 3.12 enthält einen Knoten, der Instanz einer abstrakten Klasse ist. Somit ist auch dies ein Negativbeispiel für die Forderung nach Klassenzugehörigkeit. Analog gibt es Beispiele für Instanzen von abstrakten Kanten- oder Relationsklassen.

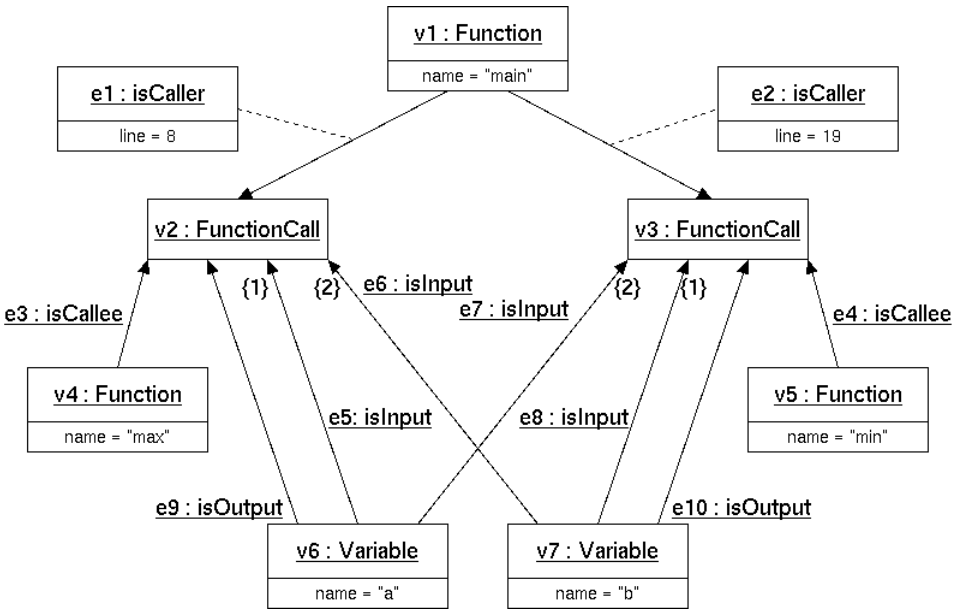

Abbildung 3.10: Schemakonformer Graph (Schema: Abb. 3.2)

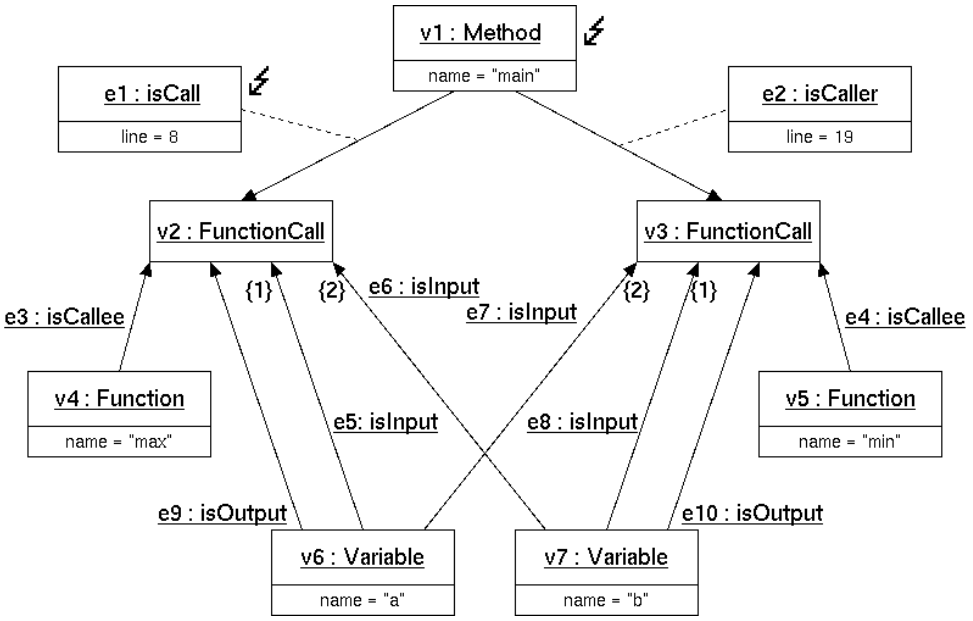

Abbildung 3.11: Graph mit undefinierten Klassen (Schema: Abb. 3.2)

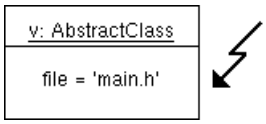

Abbildung 3.12: Instanz einer abstrakten Klasse (Schema: Abb. 3.5)

Pseudocode

Die Funktion `testInstanceVsSchemaClasses` überprüft, ob ein Graph G die Bedingung der Klassenzugehörigkeit erfüllt. Dazu wird getestet, ob der Typ eines jeden Graphelementes der Graphklasse bekannt und nicht abstrakt ist.

```
function testInstanceVsSchemaClasses (
             G: Graph;
             GC: GraphClass;
             eh: ErrorHandler): Boolean;
var
  ElementClass: Class;
begin
  result := true;
  foreach GraphElement ge of G do
    begin
      ElementClass := GC.getClassOf(ge);
      if not GC.knowsGraphElementClass (ElementClass) then
        begin
          eh.processError(
            GXLUndefinedClassError.create (ElementClass, ge));
          result := false;
        end
      else
        if ElementClass.isAbstract then
          begin
            eh.processError (
              GXLAbstractClassError.create(ge));
            result := false;
          end;
    end;
end;
```

3.5.2 Korrektheit der Kanten

Beschreibung

Die Definition einer Kantenklasse gibt an, welche Typen von Graphelementen Kanten dieser Klasse verbinden dürfen. Zusätzlich werden Multiplizitäten angegeben, die die Ober- und Untergrenze von Inzidenzen für Kanten dieser Kantenklasse bezogen auf ein Graphelement festlegen. Diese Grenzwerte werden jeweils für das From- und To-Ende einer Kantenklasse definiert. Daneben wird für eine Kantenklasse der Richtungsmodus (gerichtet/ungerichtet) für Kanten dieses Typs festgelegt.
Damit ein Graph schemakonform ist, dürfen alle im Graph vorkommenden Kanten nur Graphelemente solcher Typen miteinander verbinden, wie es durch die zugehörige Kantenklassendefinition im Schema vorgegeben ist. Dabei muss beachtet werden, dass aufgrund der Generalisierung auch Unterklassen der im Schema angegebenen Klassen zulässig sind.
Außerdem müssen alle im Graph vorkommenden Graphelemente die im Schema angegebenen Multiplizitäten einhalten. Ein Graphelement hält die in der Graphklasse definierten Kardinalitäten ein, gdw. die Summe der eingehenden und die Summe der ausgehenden Kanten eines Kantentyps die in der Graphklasse festgelegten Beschränkungen (Minimal-/Maximalwert) nicht unter- oder überschreiten. Dabei ist zu berücksichtigen, dass Kanten nicht nur Knoten miteinander verbinden können, sondern beliebige Graphelemente.
Des Weiteren wird gefordert, dass eine Kante den in der Kantenklasse angegebenen Richtungsmodus (gerichtet/ungerichtet) einhält.

Beispiele

Abbildung 3.13 zeigt ein Negativbeispiel zu den in diesem Abschnitt formulierten Anforderungen. Der gezeigte Graph ist nicht schemakonform (vgl. Abb. 3.2), da die Kante *e3* mit dem Knoten *v4* ein falsches From-Element aufweist. Analog gibt es Beispiele für inkorrekte To-Elemente einer Kante.
Der Graph in Abbildung 3.14 ist ebenfalls ein Negativbeispiel, da auch hier die durch das Schema (vgl. Abb. 3.2) aufgestellten Anforderungen verletzt werden. Der Knoten *v2* unterschreitet die untere Grenze für inzidente Kanten des Typs *isCallee*. Der Knoten *v3* hat mit den Kanten *e7* und *e10* zwei eingehende Kanten des Typs *isOutput*, womit die Obergrenze für Kanten dieses Typs überschritten wird. Zusätzlich fehlt diesem Knoten eine eingehende Kante des Typs *isInput*.

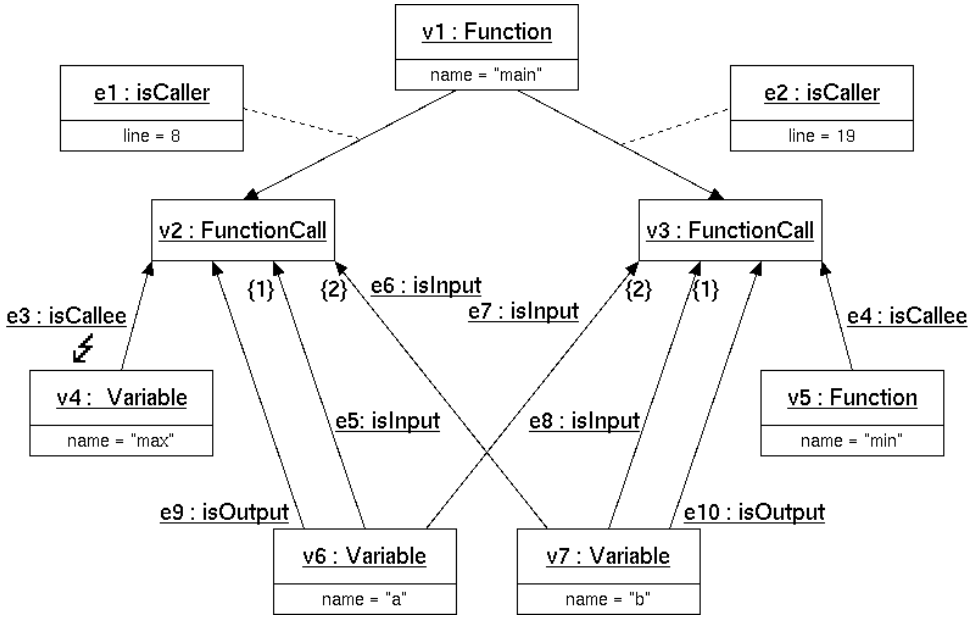

Abbildung 3.13: Graph mit fehlerhaftem Kantenende (Schema: Abb. 3.2)

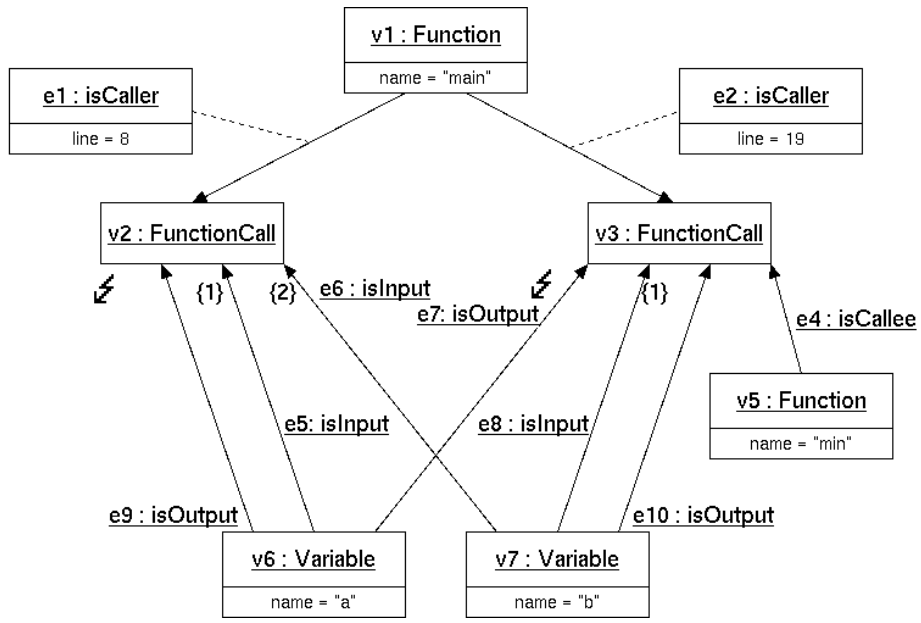

Abbildung 3.14: Graph mit falschen Kardinalitäten (Schema: Abb. 3.2)

Pseudocode

Die Funktion `testInstanceVsSchemaEdges` überprüft, ob ein Graph G die Forderung nach Korrektheit der Kantenklassen erfüllt. Dazu wird für jede Kante eines Graphen getestet, ob die Kantenendelemente Instanz der im Schema angegebenen Klasse oder einer Unterklasse dieser sind und ob die Kanten den vorgegebenen Richtungsmodus einhalten. Zusätzlich überprüft die Funktion `testCardinality`, ob ausgehend von einem Graphelement die im Schema definierten Kardinalitäten eingehalten werden.

```
function testCardinality (gec: GraphElementClass;
                          ec: EdgeClass;
                          direction: TDirection;
                          lowerLimit,
                          upperLimit: Integer;
                          G: GXLGraph;
                          eh: ErrorHandler): Boolean;
var
  edgeCount: Integer;
begin
  result := true;

  foreach GraphElement ge of G do
    begin
      if (ge is instance of gec) then
        begin
          edgeCount := 0;
          if (direction=from) then
            begin
              foreach outgoing Edge e of ge do
                if (e is instance of ec) then
                  inc(edgeCount);
            end
          else
            foreach incoming Edge e of ge do
              if (e is instance of ec ) then
                inc(edgeCount);
        end;

      if (edgeCount<lowerLimit) or
         (edgeCount>upperLimit) then
```

```
        begin
          eh.processError (
            GXLCardinalityError.create (ge,
                                        ec,
                                        lowerLimit,
                                        upperLimit,
                                        edgeCount));
          result := false;
        end;
    end;
end;

function testInstanceVsSchemaEdges (
             G: Graph;
             GC: GraphClass;
             S: Schema;
             eh: ErrorHandler): Boolean;
begin
  result := true;

  foreach EdgeClass ec of GC do
    begin
      result := (testCardinality (ec.getFromClass,
                                  ec,
                                  from,
                                  ec.getTo.getLowerLimit,
                                  ec.getTo.getUpperLimit,
                                  G,
                                  eh) and
                 result);

      result := (testCardinality (ec.getToClass,
                                  ec,
                                  to,
                                  ec.getFrom.getLowerLimit,
                                  ecFrom.getUpperLimit,
                                  G,
                                  eh) and
                 result);
    end;
```

```
  foreach Edge e of G do
    begin
      if not (e.getFrom is instance of ec.getToClass) then
        begin
          eh.processError (
            GXLWrongEdgeFromElementError.create (e, ec));
          result := false;
        end;

      if not (e.getTo is instance of ec.getToClass) then
        begin
          eh.processError (
            GXLWrongEdgeToElementError.create (e, ec));
          result := false;
        end;

      if (e.isDirected<>ec.isDirected) then
        begin
          eh.processError (
            GXLEdgeDirectionError.create (e, ec));
          result := false;
        end;
    end;
end;
```

3.5.3 Korrektheit der Relationen

Beschreibung

Genau wie Kantenklassen können in einem Schema auch Relationsklassen definiert werden. Diese bestimmen, welche Relationsenden eine Relation besitzt. Das beinhaltet die Angabe der Zielelementklassen, der Multiplizitäten und des Richtungsmodus.
Damit ein Graph schemakonform ist, dürfen alle im Graph vorkommenden Relationen nur Graphelemente solcher Typen miteinander verbinden, wie es durch die zugehörige Relationsdefinition in der Graphklasse vorgegeben ist. Dabei muss beachtet werden, dass aufgrund der Generalisierung auch Unterklassen der in der Graphklasse angegebenen Klassen zulässig sind.
Außerdem müssen die in der Graphklasse angegebenen Kardinalitäten eingehalten werden. Das bedeutet, dass die Anzahl der Relationsenden eines Typs einer Relation die in der Graphklasse festgelegten Beschränkungen (Minimal-

/Maximalwert) nicht unter- oder überschreiten darf.
Eine Relation darf auch nur solche Relationsenden besitzen, wie es durch das Schema vorgegeben ist. Außerdem müssen die Relationsenden die im Schema angegebene Richtung (*in/out/none*) einhalten.
Die Zuordnung eines Relationsendes zu seiner Klasse ist nicht so trivial, wie bei Graphelementen (Knoten, Kanten, Relationen), da Relationsenden nicht typisiert sind. Die Zuordnung geschieht vielmehr über die Rolle eines Relationsendes. Diese Rolle muss identisch sein mit der Rolle der Relationsende-Klasse im Schema. Dabei werden nur Relationende-Klassen der Relationsklasse betrachtet, die der Klasse der Relation des Relationsendes entspricht. Abbildung 3.15 veranschaulicht diesen Sachverhalt.
Ausgehend von dem Relationsende mit der Rolle *role2* und der übergeordneten Relation *r* kann zunächst im Schema die Relationsklasse *Relation* gefunden werden. Über einen Vergleich der Rollen kann dann im letzten Schritt die Relationsende-Klasse mit der Rolle *role2* als die Klasse des Relations-Endes mit der identischen Rolle bestimmt werden.
Aus dieser Art der Zuordnung ergeben sich weitere Anforderunen an die Definition von Relationsklassen, die in Abschnitt 3.6.7 thematisiert werden.

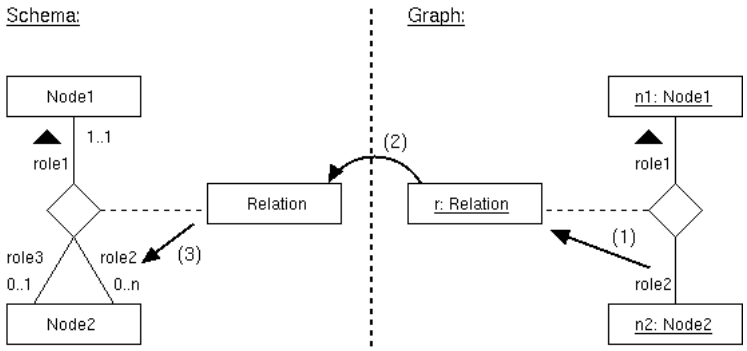

Abbildung 3.15: Zuordnung Relationsende zu entsprechender Klasse

Beispiele

In Abbildung 3.16 wird ein Graph gezeigt, der die Anforderung nach korrekten Relationen erfüllt (vgl. Schema in Abb. 3.3). Hingegen enthält Abbildung 3.17 ein Negativbeispiel, da die Relation *r1* an dem Relationsende *caller* mit dem Knoten *v4* ein falsches Ziel-Element aufweist.
Ein weiteres Negativbeispiel ist der Graph aus Abbildung 3.18. Die Relation *r1* unterschreitet die Untergrenze an Relationsenden vom Typ *caller* und überschreitet die Höchstgrenze an Relationsenden vom Typ *callee*.

Abbildung 3.20 enthält ein Positivbeispiel für die Forderung nach Einhaltung der Richtung durch ein Relationsende. Im Gegensatz dazu wird diese Anforderung durch den Graphen in 3.21 verletzt.

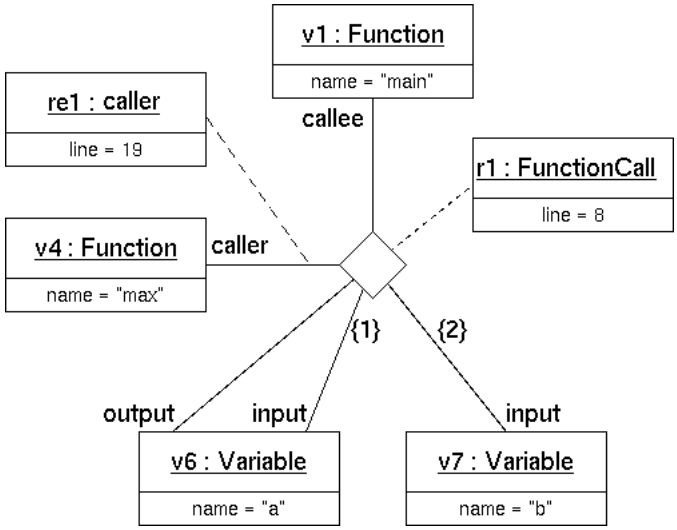

Abbildung 3.16: HyperGraph (Schema: Abb. 3.3)

Pseudocode

Die Funktion `testInstanceVsSchemaRelations` überprüft, ob ein Graph G die Forderung nach Korrektheit der Relationen erfüllt. Um dies zu erreichen, wird getestet, ob alle Relationsenden einer Relation durch eine Relationsendeklasse definiert sind. Für jede Relationsendeklasse einer Relationsklasse wird außerdem untersucht, ob im Graphen die Kardinalitäten eingehalten werden und ob die Relationsenden des jeweiligen Typs korrekt Zielelemente aufweisen. Zusätzlich wird für jedes Relationsende überprüft, ob der Richtungsmodus der zugehörigen Relationsendeklasse eingehalten wird.

```
function testInstanceVsSchemaRelations (
             G: Graph;
             S: Schema;
             GC: GraphClass;
             eh: ErrorHandler): Boolean;
var
  rc: RelationClass;
```

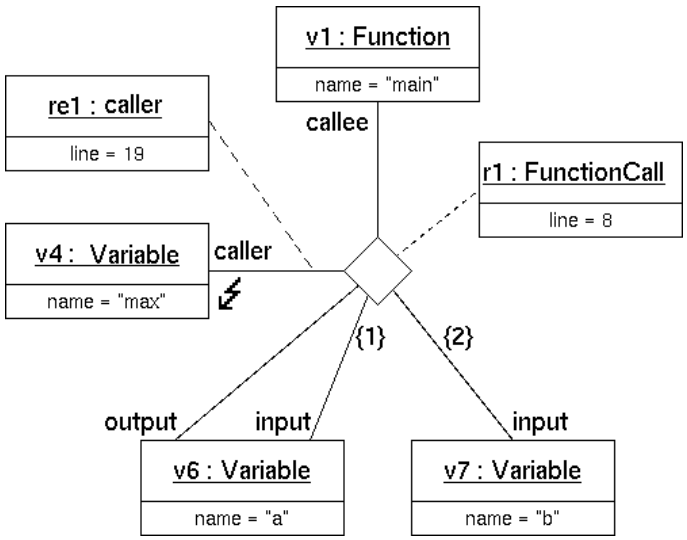

Abbildung 3.17: Graph mit falschem TargetElement (Schema: Abb. 3.3)

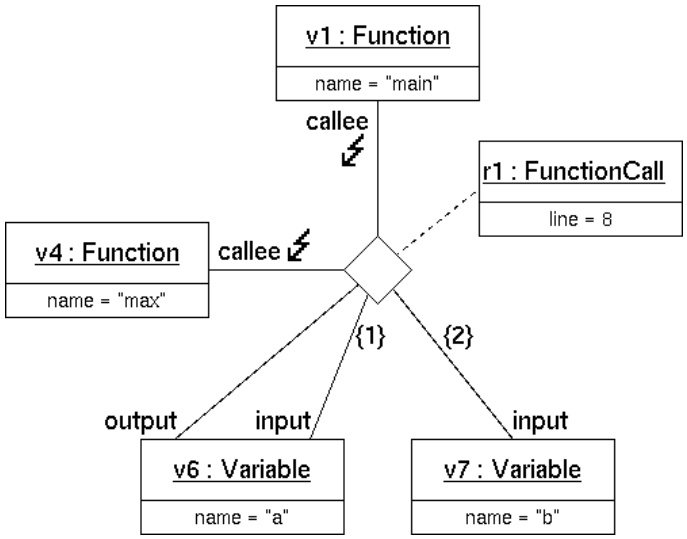

Abbildung 3.18: Graph mit falschen Relationskardinalitäten (Schema: Abb. 3.3)

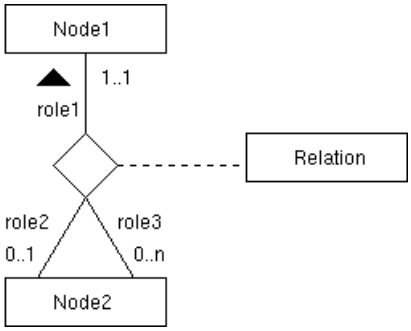

Abbildung 3.19: Schemaauszug mit Relationklasse

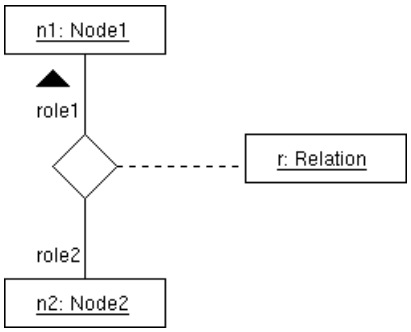

Abbildung 3.20: Graph mit korrekter Relation (Schema: Abb. 3.19)

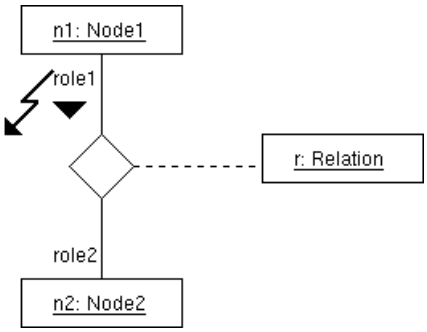

Abbildung 3.21: Graph mit inkorrekter Richtung eines Relationendes (Schema: Abb. 3.19)

```
  count: Integer;
begin
  foreach Relation r of G do
    begin
      rc := GC.getClassOf(r);
      foreach RelationEnd re of r do
        begin
          if not rc.hasRelationEndClass(re.getRole) then
            begin
              eh.processError (
                GXLUndefinedRelationEndClassError.create(re, r));
              result := false;
            end;
        end;

      foreach RelationEndClass rec of rc do
        begin
          count := 0;
          role = rec.getRole();
          foreach RelationEnd re of r with (re is instance of rec) do
            begin
              count := count+1;
              if not (re.getTarget
                        is instance of rec.getRelatesToClass) then
                begin
                  eh.processError (
                    GXLWrongRelEndTargetElementError.create  (r, re, rec));
                  result := false;
                end;

              if (re.getDirection<>rec.getDirection) then
                begin
                  eh.processError (
                    GXLRelEndDirectionError.create (r, re, rec));
                  result := false;
                end;
            end;

          if (count<rec.getRelatesTo.getLowerLimit) or
             (count>rec.getRelatesTo.getUpperLimit) then
            begin
```

```
                eh.processError (
                  GXLCardinalityError.create (
                    r,
                    rec,
                    rec.getRelatesTo.getLowerLimit,
                    rec.getRelatesTo.getUperLimit,
                    count));
                  result := false;
              end;
          end;
      end;
end;
```

3.5.4 Korrektheit der Attribute

Beschreibung

In einem GXL-Schema können Attributstrukturen für jede Klasse definiert werden. Diese Definition umfasst sowohl die Angabe des Attributnamens wie auch des Wertebereichs (Domain).
Damit ein Graph schemakonform ist, müssen alle im Graph vorkommenden attributierten Elemente die in den zugehörigen Klassen definierten Attribute besitzen. Dabei muss sowohl der Name als auch der Typ der Attribute übereinstimmen. Außer den in der Graphklasse definierten Attributen dürfen die Instanzen keine weiteren Attribute aufweisen. Ebenso dürfen keine dieser Attribute fehlen, es sei denn, es existiert für dieses Attribut ein DefaultValue. Graphelementklassen unterstützen Vererbung. Die formulierten Anforderungen sind bei Graphelementen also auf die zugehörige Klasse sowie alle Oberklassen anzuwenden. Ebenso können Attribute selbst wieder attributiert sein. Deshalb müssen die Bedingungen auch für alle Attribute eines Attributes überprüft werden.

Beispiele

Der Graph in Abbildung 3.22 beinhaltet ein Negativbeispiel für die Korrektheit der Attribute (vgl. Schema in Abb. 3.2). Der Kante *e1* fehlt ein Attribut, genauso dem Knoten *v1* und der Kante *e2*. Diese beiden Elemente haben des Weiteren jeweils ein unbekanntes Attribut. Die Attributbelegung des einzigen Attributs des Knotens *v5* passt nicht zu dem in der Graphklasse angegebenen Typ des Attributs.
Sowohl ein Positiv- wie auch ein Negativbeispiel zeigt Abbildung 3.23. Die

Belegung des Attributs *enum* im Positivbeispiel passt zur Domain, im Gegensatz zu der Attributbelegung des gleichnamigen Attributs des Knotens im Negativbeispiel.
Das Positivbeispiel in Abbildung 3.24 enthält einen Knoten, der ein Attribut besitzt, dessen Typ nicht atomar ist. Der dargestellte Knoten erfüllt die Anforderung an korrekte Attribute, der Knoten des Negativbeispiels jedoch nicht. Das zweite Tupelelement ist nicht wie gefordert ein String, sondern ein Integer-Wert.
Ein weiteres Positivbeispiel ist in Abbildung 3.25 enthalten. Das Bag-Attribut *bag* enthält wie gefordert nur Elemente des Typs Integer. Das Negativ innerhalb der gleichen Abbildung enthält einen Knoten, dessen Attribut *bag* diese Anforderung nicht erfüllt, da nicht alle Elemente den Typ Integer besitzen.

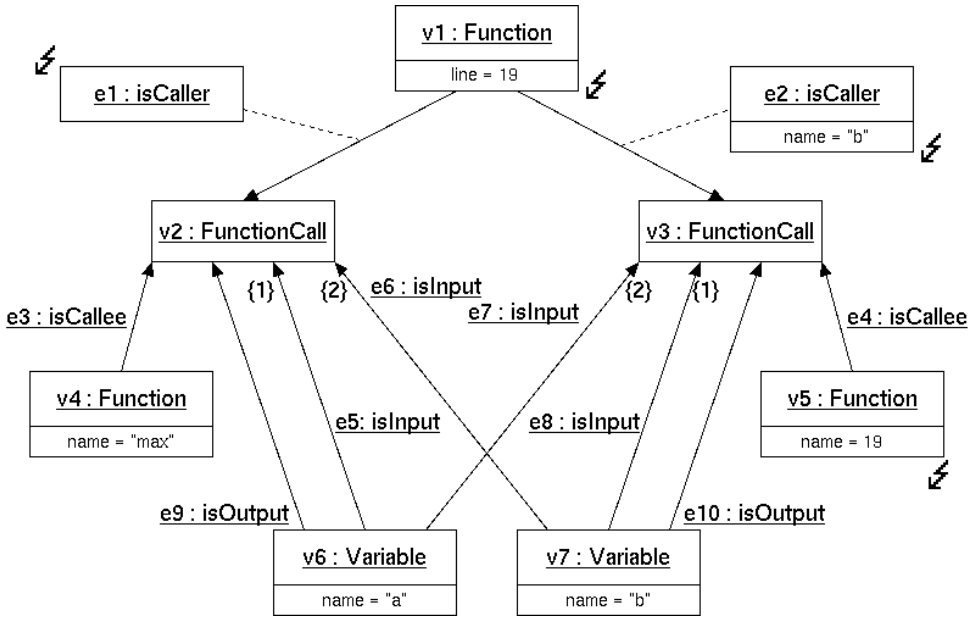

Abbildung 3.22: Graph mit falschen Attributlisten (Schema: Abb. 3.2)

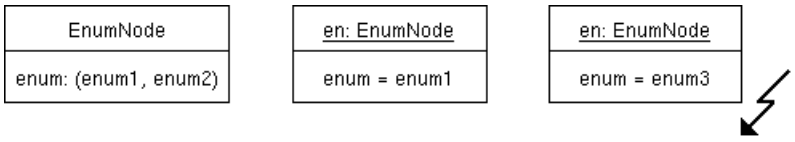

Abbildung 3.23: Knotenklasse mit Enum-Attribut, Positiv- und Negativbeispiel

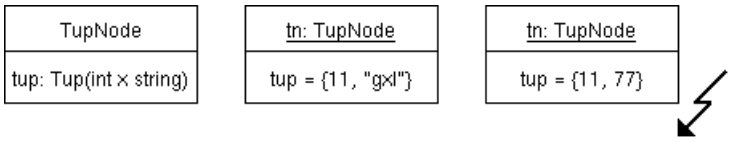

Abbildung 3.24: Knotenklasse mit Tupel-Attribut, Positiv- und Negativbeispiel

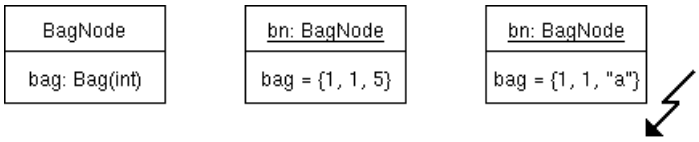

Abbildung 3.25: Knotenklasse mit Bag-Attribut, Positiv- und Negativbeispiel

Pseudocode

Die Funktion `testInstanceVsSchemaAttribute` überprüft, ob ein Graph G die Forderung nach Vollständigkeit der Attribute aller Graphelemente erfüllt. Dazu wird die Funktion `testElement` verwendet, die für ein attributiertes Element und dessen Klasse testet, ob die Attribute vollständig sind und zur jeweiligen Domain passen.

```
function testElement (S: GXLSchema;
                      eh: ErrorHandler;
                      ae: GXLAttributedElement;
                      aec: GXLAttributedElementClass): Boolean;
begin
  foreach Attribute a of ae do
    begin
      if aec.hasAttribute(a.getName) then
        begin
          if not (a.getValue matchesDomain ac.getDomain) then
            begin
              eh.processError (
                GXLValueDomainError.create (ae, a, ac.getDomain));
              result := false;
            end;
            result := (testElement (S, eh, a, ac) and result);
```

```
        end
      else
        begin
          eh.processError (
            GXLUndefinedAttributeClassError.create (ae, a));
          result := false;
        end;
    end;
  foreach AttributeClass ac of aec do
    begin
      if (not ae.hasAttribute (ac.getName)) and
         (not ac.hasDefaultValue) then
        begin
          eh.processError (
            GXLMissingAttributeError.create (ae, aec, ac));
          result := false;
        end;
     end;
end;

function testInstanceVsSchemaAttribute (
             G: GXLGraph;
             S: GXLSchema;
             GC: GXLGraphClass;
             eh: ErrorHandler): Boolean;
begin
  foreach Node n of  G do
    result := (testElement (S, eh, n, GC.getClassOf(n) and
                result);
  foreach Edge e of G do
    result := (testElement (S, eh, e, GC.getClassOf(e) and
                result);
  foreach Relation r of G do
    begin
      result := (testElement (S, eh, r, GC.getClassOf(r) and
                  result);
      foreach RelationEnd re of r do
        result := (testElement (S, eh, re, GC.getClassOf(re) and
                    result);
    end;
end;
```

3.5.5 Korrektheit der eingebetteten Graphen

Beschreibung

Graphelemente (Knoten, Kanten, Relationen) eines GXL-Graphen können eingebettete Graphen enthalten.
Damit ein Graph schemakonform ist, müssen alle in Elemente dieses Graphen eingebetteten Graphen ebenfalls schemakonform sein. Im Schema werden für eingebettete Graphen Multiplizitäten vorgeschrieben. Diese Grenzwerte müssen von eingebetteten Graphen bezogen auf ein Graphelement eines Instanzgraphen eingehalten werden, d.h. sie dürfen weder unter- noch überschritten werden.

Beispiele

In Abbildungen 3.26 ist ein Positivbeispiel für korrekte eingebette Graphen zu sehen (vgl. Schema in Abb. 3.4). Dagegen beinhalten die Abbildungen 3.27 und 3.28 Gegenbeispiele. Dem Graph aus Abb. 3.27 fehlt der von der Graphklasse geforderte eingebettete Graph. Der Graph aus Abb. 3.28 enthält zwar einen eingebetteten Graphen. Dieser passt aber nicht zu seiner Graphklasse (siehe Knoten *v4.2*).

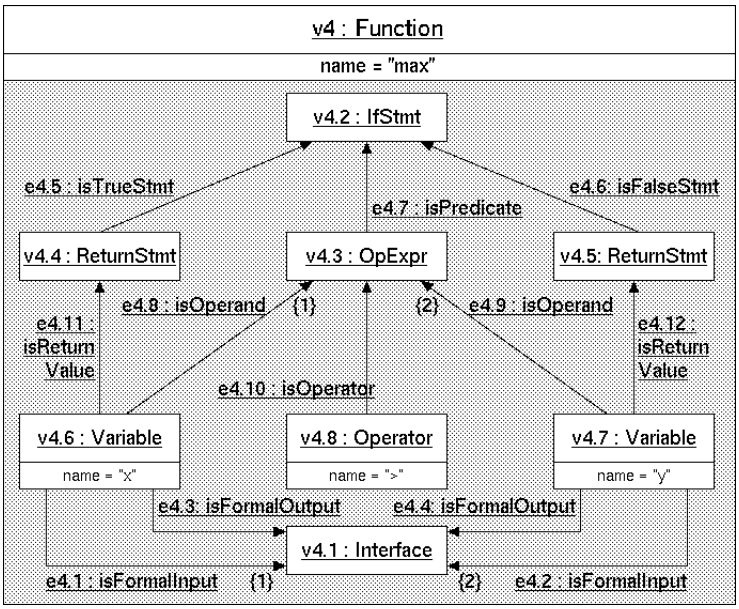

Abbildung 3.26: Hierarchischer Graph (Schema: Abb. 3.4)

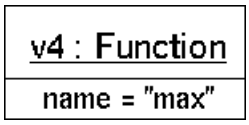

Abbildung 3.27: Graph mit fehlendem eingebetten Graph (Schema: Abb. 3.4)

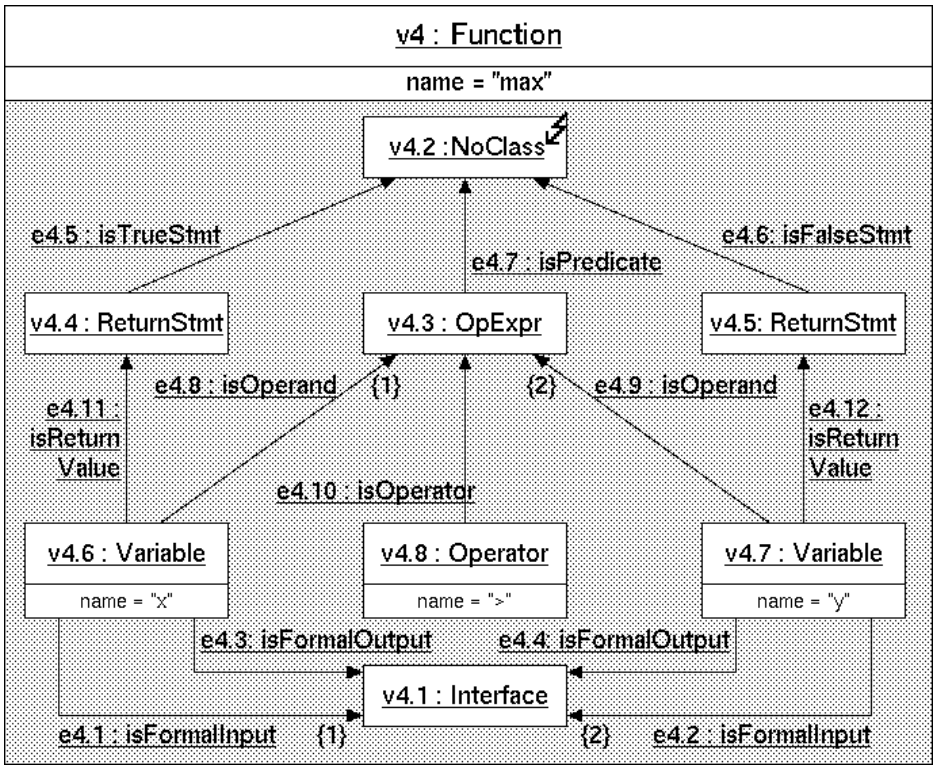

Abbildung 3.28: Graph mit fehlerhaftem eingebetteten Graph (Schema: Abb. 3.4)

Pseudocode

Die Funktion `testInstanceVsSchemaComponentGraph` überprüft, ob ein Graph G die Forderung nach Korrektheit der eingebetteten Graphen erfüllt. Für jede Art von eingebettetem Graph wird dazu getestet, ob die Graphelemente eines Graphen, die einen solchen eingebetteten Graphen enthalten können, die vorgegebenen Kardinalitäten einhalten. Außerdem wird überprüft, ob die eingebetteten Graphen selbst korrekt und schemakonform sind.

```
function testInstanceVsSchemaComponentGraph (
                G: GXLGraph;
                S: GXLSchema;
                GC: GXLGraphClass;
                eh: ErrorHandler): Boolean;
var
  componentGC: GraphClass;
begin
  foreach GraphElementClass gec of GC do
    begin
      foreach HasAsComponenGraph hasAsCG of gec do
        begin
          componentGC := hasAsCG.getComponentGraph;

          foreach GraphElement ge of G
              with (ge is instance of gec) do
            begin
              count := 0;
              foreach Graph cg of ge
                begin
                  count := count+1;
                  result := ((cg is valid GXLInstance) and
                             result);
                  if (cg.hasSchema) then
                    result := ((cg matches cg.getSchema) and
                               result);
                end;

              if (count<hasAsCG.getLowerLimit) or
                 (count>hasAsCG.getUpperLimit) then
```

```
                begin
                  eh.processError (
                    GXLCardinalityError.create (
                      ge, componentGG,
                      hasAsCG.getLowerLimit,
                      hasAsCG.getUpperLimit,
                      count));
                  result := false;
                end;
            end;
        end;
    end;
end;
```

3.6 Anforderungen an GXL-Schemata

In den beiden vorangegangenen Abschnitten wurden die Anforderungen besprochen, die erfüllt sein müssen, um die Instanzkorrektheit und Schemakonformität eines GXL-Graphen sicherzustellen.
GXL-Schemata werden ebenfalls in Form von GXL-Graphen dargestellt. Die Anforderungen aus 3.4 und 3.5 gelten demnach auch für GXL-Schemagraphen. Zusätzlich zu diesen gibt es aber weitere, die bisher nicht erfasst worden sind. Dieser Abschnitt befasst sich mit den Anforderungen, die erfüllt sein müssen, damit ein GXL-Schema korrekt ist. Dabei wird vorausgesetzt, dass das Schema selbst ein instanzkorrekter GXL-Graph ist (vgl. Abschnitt 3.4) und konform ist zum GXL-Metaschema (vgl. Abschnitt 3.5).
Die in diesem Abschnitt aufgeführten Anforderungen ergeben sich zum größten Teil aus dem GXL-Metaschema. Zusätzlich werden Anforderungen neu aufgestellt, die bisher weder im GXL-Metaschema, noch in sonstigen textuellen Beschreibungen gefordert wurden. Dazu zählt z.B. die *Korrektheit der Generalisierungen* aus 3.6.1. Diese Anforderungen wurden in Zusammenarbeit mit den Entwicklern der Sprache GXL aufgestellt.

3.6.1 Korrektheit der Generalisierungen

Beschreibung

GXL-Schemata erlauben die Generalisierung von Graphelementklassen (Knoten-, Kanten-, Relationsklassen). Dabei erben die Subklassen u.a. die Attribute ihrer Oberklassen. Dargestellt werden diese Beziehungen durch *isA*-Kanten.

Damit ein Schema schemakorrekt ist, dürfen keine zyklischen Generalisierungsabhängigkeiten existieren. Des Weitern sind *isA*-Beziehungen (Kanten) nur zwischen Instanzen erlaubt, die zur gleichen Element-Klasse gehören (NodeClass, EdgeClass, RelationClass). Das bedeutet, dass eine Knotenklasse nur Spezialisierung einer anderen Knotenklasse und nicht etwa einer Kanteklasse sein kann. Für die Spezialisierung von Kantenklassen wird gefordert, dass Kanten- zu Aggregations- oder Kompositionsklassen, Aggregations- zu Kompositionsklassen spezialisiert werden können.
Außerdem müssen alle Oberklassen einer Klasse zu der Graphklasse gehören, der diese Klasse angehört. Das bedeutet, dass alle Klassen einer Generalisierungshierarchie zur gleichen Graphklasse gehören müssen. Schemaübergreifende Generalisierungen sind somit nicht erlaubt.

Beispiel

Das Schema in Abbildung 3.30 besitzt eine zyklische Generalisierung. Daher ist das Schema nicht korrekt.
Abbildung 3.31 zeigt eine Generalisierungsbeziehung zwischen Klassen, die nicht zur gleichen Graphklasse gehören. Dies ist ebenfalls ein Widerspruch zu Korrektheit von Generalisierungen.
Dagegen enthält Abbildung 3.29 ein Positivbeispiel für diese Anforderung. Abbildung 3.32 zeigt wiederum einer Verletzung der Anforderung, da hier eine Kantenklasse Unterklasse einer Knotenklasse sein soll.

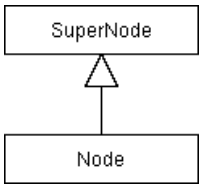

Abbildung 3.29: Spezialisierung einer Knotenklasse

Pseudocode

Die Funktion `testSchemaGeneralization` überprüft, ob ein Schemagraph die Anforderungen an korrekte Generalisierungen erfüllt. Dazu testet die Funktion `testDirectSuperClass`, ob die direkten Oberklassen einer Klasse den gleichen Typ wie ihre Unterklasse aufweisen und ob diese auch zur gleichen Graphklasse gehören.
Die Funktion `testAcyclicStructure` untersucht dagegen, ob innerhalb der

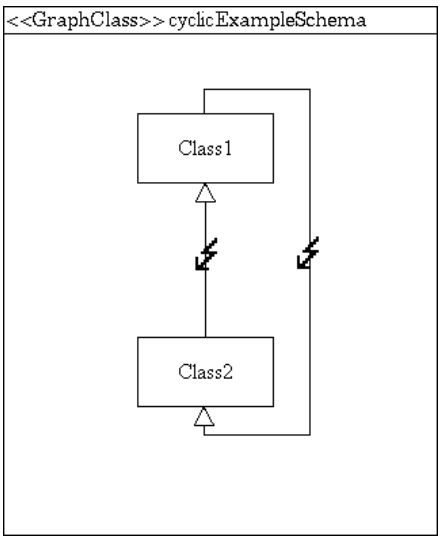

Abbildung 3.30: Schema mit zyklischer Generalisierung

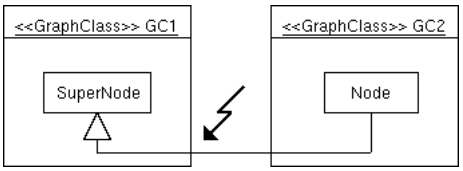

Abbildung 3.31: Schemata mit Generalisierungsabhängigkeit

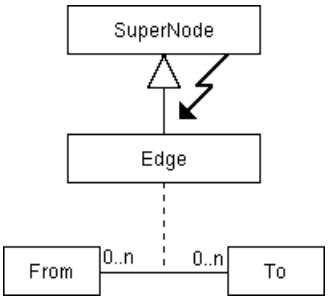

Abbildung 3.32: Nicht erlaubte Spezialisierung einer Knoten- zu einer Kantenklasse

Generalisierungshierarchien des Schemagraphen zirkuläre Strukturen existieren (vgl. [6] S. 167ff). Dabei ist darauf zu achten, dass diese Funktion eine erfolgreiche Ausführung der Funktion `testDirectSuperClass` voraussetzt.

```
function testDirectSuperClass (
             GC: GraphClass;
             eh: ErrorHandler): Boolean;
begin
  foreach GraphElementClass gec of GC do
    foreach directSuperClass GraphElementClass superClass of GEC do
      if not GC.knowsGraphElementClass(superClass) then
        begin
          eh.processError (
            GXLSuperClassOwnershipError.create (
              gec, superClass, GC));
          result := false;
        end
      else if (not (gec.getType=superClass.getType) and
               not ((gec is EdgeClass) and
            (superClass is AggregationClass)) and
       not ((gec is EgdeClass) and
            (superClass is CompositionClass)) and
       not ((gec is AggregationClass) and
            (superClass is CompositionClass))) then
        begin
          if ((gec is AggregationClass) and
              (superClass is EdgeClass)) or
             ((gec is CompositionClass) and
              (superClass is EdgeClass)) or
             ((gec is CompositionClass) and
              (superClass is AggregationClass))) then
            eh.processWarning (
              GXLWrongSuperClassError.create (gec, superClass))
          else
            begin
              eh.processErrorError (
                GXLWrongSuperClassError.create (gec, superClass));
              result := false;
            end;
        end;
```

```
end;

function testAcyclicStructure (
             G: Graph;
             GC: GraphClass;
             eh: ErrorHandler): Boolean;
var
  TNUM,
  extracted: Integer;
  inDegree: Map of [key: GraphElementClass; value: Integer];
  W: Set of GraphElementClass;
  isA: GraphElementClass;
begin
  n := GC.getNumberOfGraphElementClasses;

  foreach GraphElementClass gec of GC do
    inDegree[gec] := 0;

  isA := GC.getClassWithId('isA');
  foreach Egde e of G with (e is instance of isA) do
    inDegree[e.getTo] := inDegree[e.getTo]+1;

  W.clear;
  foreach GraphElementClass gec of GC with (inDegree[gec]=0) do
    W.add (gec);

  TNUM := 0;
  while (not W.isEmpty) do
    begin
      gec := W.extractAny;
      TNUM := TNUM+1;

      foreach directSuperClass GraphElementClass super of GEC do
        begin
          inDegree[super] = inDegree[super]-1;
          if (inDegree[super]=0) then
            W.add (super);
        end;
    end;

  if (TNUM<n) then
```

```
    begin
      eh.processError (
        GXLCyclicIsAStructureError.create);
      result := false;
    end;
end;

function testSchemaGeneralization (
             G: Graph;
             S: Schema;
             GC: GraphClass;
             eh: ErrorHandler): Boolean;
begin
  result := testDirectSuperClass (GC, eh) and
            testAcyclicStructure (G, GC, eh);
end;
```

3.6.2 Eindeutige Elementbezeichner

Beschreibung

Graphelementklassen (Knoten-, Kanten-, Relationsklassen) besitzen ein Attribut *name*. Dieses Attribut ist der Bezeichner der Klasse.
Damit ein Schema schemakorrekt ist, müssen die Bezeichner aller Graphelemente eindeutig sein. Das bedeutet, dass mehrfaches Vorkommen eines Namens innerhalb aller Graphelementklassen einer Graphklasse nicht erlaubt ist.

Beispiele

Der Graph in Abbildung 3.33 verletzt die Bedingung der eindeutigen Elementbezeichner und ist somit ein Negativbeispiel für die Forderung nach eindeutigen Elementbezeichnern, da der Bezeichner *Function* mehrfach verwendet wird.

Pseudocode

Die Funktion `testSchemaElementIdents` überprüft, ob ein Schemagraph G die Anforderungen an eindeutige Elementbezeichner erfüllt.

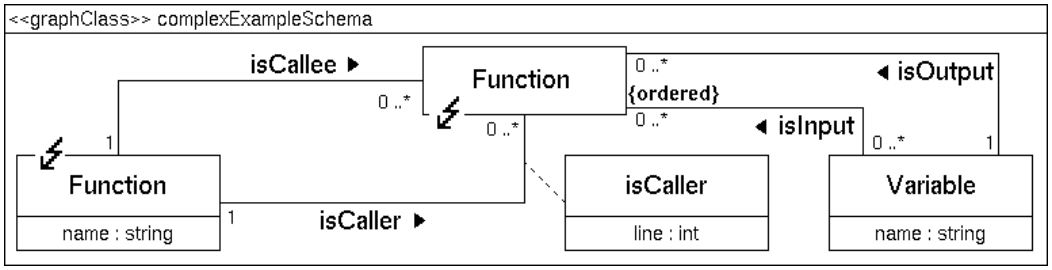

Abbildung 3.33: Schema mit nicht eindeutigen Elementbezeichnern

```
function testSchemaElementIdents (
             G: GXLGraph;
             S: Schema;
             GC: GraphClass;
             eh: ErrorHandler): Boolean;
var
  idents: Set of String;
begin
  foreach GraphElementClass gec of GC do
    begin
      if (idents.hasElement(gec.getName)) then
        begin
          eh.processError (
            GXLDuplicateElementIdentError.create(gec));
          result := false;
        end
      else
        idents.add (gec.getName);
end;
```

3.6.3 Korrektheit der Wertebereiche

Beschreibung

Innerhalb eines Schemagraphen können Wertebereiche (*Domains*) definiert werden, die Attributklassen zugeordnet werden. Diese Wertebereiche können einfache (z.B. String, Int) oder auch zusammengesetzte Bereiche (z.B. Mengen, Tupel) darstellen.
Damit ein Schema schemakorrekt ist, müssen alle Wertebereiche des Schemas korrekt sein. Das bedeutet, dass innerhalb zusammengesetzter Wertebereiche keine zyklischen Abhängigkeiten erlaubt sind. Außerdem müssen Tupel-Domains aus mindestens zwei zusammengesetzten Wertebereichen bestehen, Mengen-, Sequenz- und Multimengen-Domains aus genau einem.

Beispiele

Die Beispiele für diese Anforderung können nicht in Form von Klassendiagrammen gegeben werden, da sie durch diese Diagrammart nicht darstellbar sind. Daher werden Objektdiagramme benutzt, die Auszüge aus einem Schemagraph zeigen.
Abbildung 3.34 beinhaltet das Positivbeispiel für die Anforderung an korrekte Wertebereiche. Dagegen enthält Abbildung 3.35 ein Negativbeispiel mit Wertebereichen, die zyklisch voneinander abhängen. Auch Abbildung 3.36 enthält ein Negativbeispiel. Der dargestellte Tupel-Wertebereich weist nur ein Tupelelement auf, obwohl es mindestens zwei sein müssten.

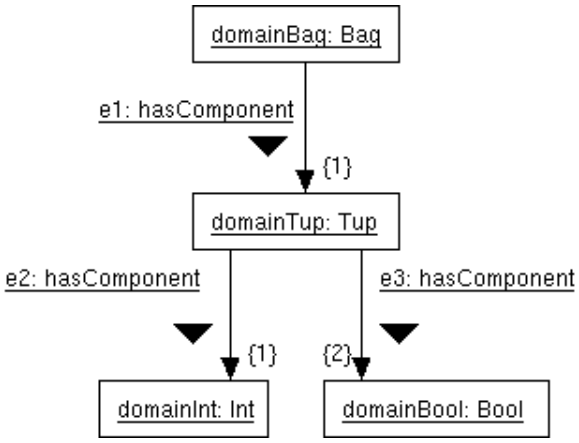

Abbildung 3.34: Zusammengesetzter Wertebereich

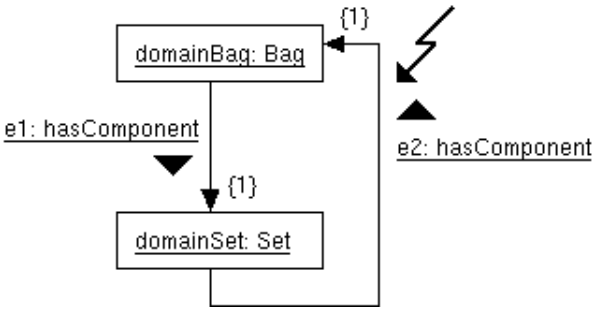

Abbildung 3.35: Zyklischer zusammengesetzter Wertebereich

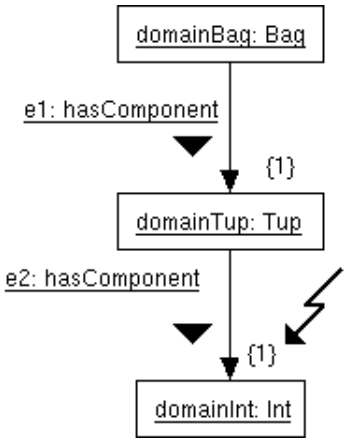

Abbildung 3.36: Tupel-Wertebereich mit nur einem Tupelelement

Pseudocode

Die Funktion `testSchemaDomain` überprüft für alle Domains eines Schemas, ob sie die Anforderung an Korrektheit erfüllen. Dazu wird jeweils die Funktion `testDomain` verwendet, die testet, ob es zirkuläre Strukturen gibt, oder ob Tupel-Domains aus mindestens zwei Komponenten und alle zusammengesetzten Domains aus genau einer Komponente bestehen.

```
funtion testDomain (d: Domain;
                    isComponentDomainOf: Set of String;
                    var visitedDomains: Set of String;
                    eh: ErrorHandler): Boolean;
var
  cd: CompositeDomain;
  domId: String;
  componentCount: Integer;
begin
  result := true;

  if visitedDomains.hasElement(d.getId) then
    exit
  else
    visitedDomains.add (d.getId);

  isComponentDomainOf.add (d.getId);
  if d.isComposite then
    begin
      cd := d as CompositeDomain;
      componentCount := 0;

      foreach Domain nextDom of cd do
        begin
          domId := nextDom.getId;
          if isComponentDomainOf.hasElement(domId) then
            begin
              eh.processError (
                GXLComponentDomainCircleError.create (cd, nextDom));
              result := false;
              break;
            end;
          else
```

```
            result := testDomain (d,
                                  isComponentDomainOf,
                                  visitedDomains,
                                  eh);
          componentCount := componentCount+1;
        end;

      if result then
        if (cd.isTup and (componentCount<2)) then
          begin
            eh.processError (
              GXLTupLowComponentDomainError.create (d));
            result = false;
          end
        else
          if (not cd.isTup) and (componentCount<>1) then
            begin
              eh.processError(
                GXLBagSetSeqComponentDomainError.create (d));
              result := false;
            end;
    end;
end;

function testSchemaDomain (G: Graph;
                           S: Schema;
                           GC: GraphClass;
                           eh: ErrorHandler): Boolean;
var
  isComponentDomainOf,
  visitedDomains: Set of String;
begin
  result := true;
  foreach Domain d of GC do
    result := (testDomain (d,
                           isComponentDomainOf,
                           visitedDomains,
                           eh) and
               result);
end;
```

3.6.4 Korrektheit der Standardwerte

Beschreibung

Ein Schema kann die Definition mehrerer Standardwerte (*DefaultValues*) umfassen. Standardwerte können Attributklassen zugeordnet werden, um so für ein Attribut einen Vorgabewert festzulegen. Ohne eine Zuordnung zu einer Attributklasse kann keine Aussage über der Wertebereich (*Domain*) eines Standardwertes getroffen werden. Diese Betrachtung beschränkt sich somit auf die Anforderungen, die ohne Kenntnis über der Wertebereich aufgestellt werden können. Die Struktur eines Standardwertes entspricht der einer Domain.
Damit ein Schema schemakorrekt ist, müssen alle Standardwerte des Schemas korrekt sein. Das bedeutet, dass einfache Werte ihrem Typ entsprechen müssen. Zusammengesetzte Werte dürfen keine zyklischen Abhängigkeiten aufweisen. Im Fall von Tupel-Werten müssen mindestens zwei Tupelelemente vorhanden sein, bei Mengen-, Sequenz- oder Multimengen müssen die einzelnen Elemente zueinander passen. So darf z.B. eine Sequenz, deren erstes Element ein *int* ist, nur Elemente dieses Typs beinhalten.

Beispiele

Ebenso wie für die Beispiele aus dem vorangegangenen Abschnitt, genügen auch für die Beispiele zu dieser Anforderung Klassendiagramme nicht. Deswegen bestehen auch diese Beispiele aus Objektdiagrammen.
Abbildung 3.38 zeigt ein Positivbeispiel für die Anforderung nach korrekten Standardwerten. Die Abbildungen 3.37, 3.39 und 3.40 hingegen stellen Negativbeispiele dar.
Der Belegung des Standardwertes in Abbildung 3.37 passt nicht zu der Art des Standardwertes. Abbildung 3.39 enthält Standardwerte, die zyklisch voneinander abhängen. Das letzte Negativbeispiel in Abbildung 3.40 zeigt einen Tupel-Standardwert, der nicht wie gefordert mindestens zwei Tupelelemente aufweist.

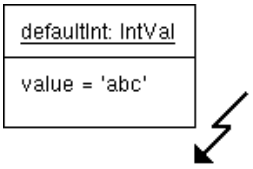

Abbildung 3.37: Standardwert mit fehlerhafter Wertbelegung

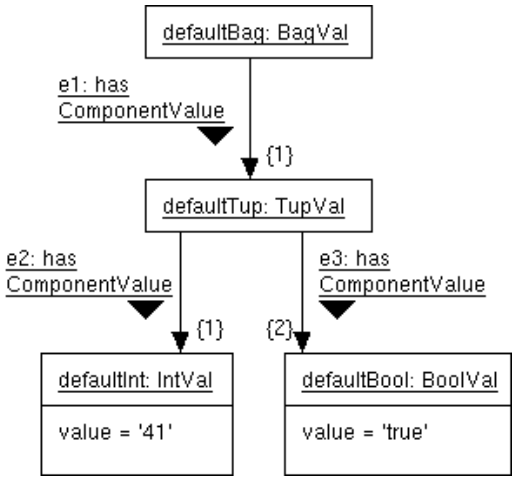

Abbildung 3.38: Zusammengesetzter Standardwert

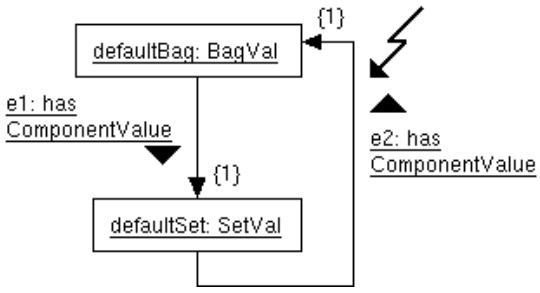

Abbildung 3.39: Zyklischer zusammengesetzter Standardwert

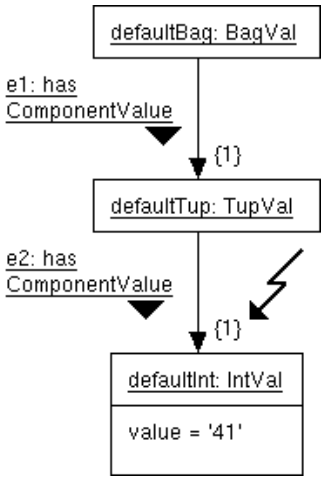

Abbildung 3.40: Tupel-Standardwert mit nur einem Tupel-Element

Pseudocode

Die Funktion `testSchemaDefaultVal` überprüft, ob alle Standardwerte eines Schemas die Anforderungen an Korrektheit erfüllen. Um zu testen, ob zirkuläre Strukturen innerhalb zusammengesetzter DefaultValues existieren, wird die Funktion `testAcyclicStructure` verwendet. Nur wenn diese Funktion für alle DefaultValues ohne Auffinden einer solchen Abhängigkeit ausgeführt worden ist, kann die Funktion `testDefaultVal` anschließend testen, ob atomare Werte zu ihrem Typ passen, Tupel-Werte aus mindestens zwei Komponenten und die übrigen zusammengesetzten Werte aus gleichartigen Elementen bestehen.

```
function testAcyclicStructure (
             dv: DefaultVal;
             isComponentDefaultValOf: Set of String;
             var visitedDefaultVals: Set of String;
             eh: ErrorHandler): Boolean;
var
  cv: DefaultCompositeVal;
begin
  result := true;
  if visitedDefaultVals.hasElement (dv.getId) then
    exit
  else
    visitedDefaultVals.add (dv.getId);
```

```
  if dv.isComposite then
    begin
      isComponentDefaultValOf.add (dv.getId);
      cv := dv as GXLDefaultCompositeVal;
      foreach DefaultVal nextVal of cv do
        begin
          if isComponentDefaultValOf.hasElement (nextVal.getId) then
            begin
              eh.processError (
                GXLComponentDefaultValCircleError.create (cv, nextVal));
              result := false;
            end
          else
            result := testAcyclicStructure (nextVal,
                                            isComponentDefaultValOf,
                                            visitedDefaultVals,
                                            eh);
        end;
    end;
end;

function testDefaultVal (dv: DefaultVal;
                         eh: ErrorHandler): Boolean;
var
  av: DefaultAtomicVal;
  cv: DefaultCompositeVal;
  tupCount: Integer;
begin
  result := true;
  if dv.isAtomic then
    begin
      av := dv as GXLDefaultAtomicVal;
      if not (av.getValue is of type av.getValueType) then
        begin
          eh.processWarning (
            GXLDefaultAtomicValError.create (av));
          result := false;
        end
    end
  else
    if dv.isComposite then
```

```
      begin
        cv := dv as DefaultCompositeVal;
        if cv.isDefaultTupVal then
          begin
            tupCount := 0;
            foreach DefaultVal nextValue of cv do
              tupCount := tupCount + 1;
            if (tupCount<2) then
              begin
                eh.processWarning (
                  GXLDefaultTupValError.create (cv));
                result := false;
              end;
          end
        else
          begin
             foreach DefaultVal nextValue of cv do
               begin
                 if not
                    (nextValue is of type cv.getComponentType) then
                   begin
                     eh.processWarning (
                       GXLDefaultBagSetSeqValError.create (cv));
                     result := false;
                   end;
               end;
          end;
      end;
end;

function testSchemaDefaultVal (G: Graph;
                               S: Schema;
                               GC: GraphClass;
                               eh: ErrorHandler): Boolean;
var
  visitedDefaultVals,
  isComponentDefaultValOf: Set of String;
begin
  result := true;
  foreach DefaultVal dv of S do
    begin
```

```
      isComponentDefaultValOf.clear;
      result := (testAcyclicStructure (dv
                                       isComponentDefaultValOf,
                                       visitedDefaultVals,
                                       eh) and result);
    end;
  if (result) then
    foreach DefaultVal dv of S do
      result := (testDefaultVal (dv, eh) and result);
end;
```

3.6.5 Korrektheit der Attributklassen

Beschreibung

GXL-Schemata erlauben die Definition von Attributklassen. Diese Klassen werden attributierbaren Klassen (*AttributedElementClass*) zugewiesen, wozu die Graphelementklasse (Knoten-, Kanten-, Relationsklassen), Relationsendeklassen und Graphklassen gehören.
Damit ein Schema schemakorrekt ist, müssen, bezogen auf eine attributierte Klasse, die Namen der Attributklassen eindeutig sein. Da Graphelementklassen Oberklassen besitzen können, ist bei diesen auf geerbte Attributklassen zu achten. Auch hier muss die Eindeutigkeit gelten. So darf z.B. eine Attributklasse einer Knotenklasse keinen Namen besitzen, den eine Attributklasse einer Oberklasse bereits aufweist.
Um den Wertebereich einer Attributklasse festzulegen, wird einer Attributklasse eine Domäne (*Domain*) zugewiesen. Einer Attributklasse kann außerdem ein Standardwert (*DefaultValue*) zugeordnet werden. Dieser Wert greift dann, wenn ein Attribut dieser Klasse in einem Instanzgraphen nicht angegeben ist. Die Belegung des Standardwertes muss zum Wertebereich der Attributklasse passen, damit das Schema korrekt ist.
Bei der Untersuchung einer Attributklasse ist darauf zu achten, dass Attributklassen ebenfalls attributiert werden können, also Attributklassen zugeordnet bekommen können. Besonders ist hier hervorzuheben, dass zirkuläre Strukturen innerhalb dieser *hasAttribute*-Beziehungen nicht erlaubt sind.

Beispiele

Der Graph in Abb. 3.43 verletzt die Forderung nach eindeutigen Attributsnamen. Der Name *name* wird in der Knotenklasse *Function* mehrfach verwendet. Analog gibt es Beispiele für nicht eindeutige Attributsnamen von

Relationsendeklassen.
Abbildung 3.41 zeigt eine Knotenklasse deren Attribut *enum* mit einem Standardwert belegt worden ist. Die Belegung ist korrekt, da der Wert in der zugehörigen Domain definiert ist. Dies ist nicht der Fall für die Knotenklasse in Abbildung 3.42, denn der Standardwert des Attributs *enum* ist nicht in der Domain definiert.

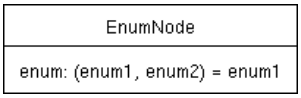

Abbildung 3.41: Knotenklasse mit korrektem Standardwert für ein Attribut

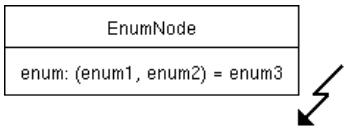

Abbildung 3.42: Knotenklasse mit undefiniertem Standardwert für ein Attribut

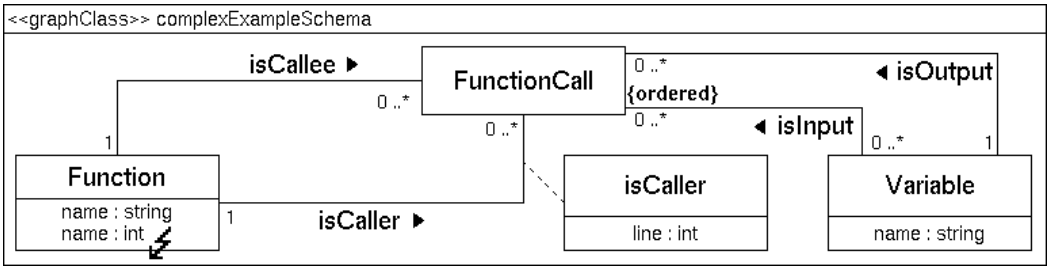

Abbildung 3.43: Schema mit nicht eindeutigen Attributnamen

Pseudocode

Die Funktion `testSchemaAttributes` überprüft, ob die Anforderung der eindeutigen Attributnamen erfüllt ist. Voraussetzung für diesen Test ist, dass es keine zirkulären Strukturen in der Hierarchie der Graphelementklassen gibt und dass sowohl die zugeordneten Wertebereiche wie auch Standardwerte auf Korrektheit überprüft worden sind.

```
function testEnumValue (aec: AttributedElementClass;
                        ac: AttributeClass;
                        v: DefaultVal;
                        d: Domain;
                        eh: ErrorHandler): Boolean;
var
  enumVal: EnumDomainVal;
  enumDom: EnumDomain;
  enumIsDefined: Boolean;
  cv: DefaultCompositeVal;
  cd: CompositeDomain

begin
  if (v.isDefaultEnumVal) then
    begin
      enumVal := v as GXLEnumDomainVal;
      enumDom := d as GXLEnumDomain;
      enumIsDefined := false;
      foreach EnumDomainVal enumDomVal of enumDom do
        begin
          enumIsDefined := (enumDomVal.getValue=enumVal.getValue);
          if enumIsDefined then break;
        end;
      if (not enumIsDefined) then
        begin
          eh.processError (
            GXLUndefinedDefaultEnumValError.create (aec, ac, d));
          result := false;
        end
    end
  else if (v.isComposite) then
    begin
      cv := v as GXLDefaultCompositeVal;
      cd := d as GXLCompositeDomain;
      nextDom := cd.getFirstComponentDomain;
      if (cv.isDefaultTupVal) then
        foreach DefaultVal nextVal of cv do
          begin
            result := (testEnumValue (
                        aec, ac, nextVal, nextDom, eh) and
                      result);
```

```
            nextDom := cd.getNextComponentDomain;
          end
      else
        foreach DefaultVal nextVal of cv do
          result := (testEnumValue (
                       aec, ac, nextVal, nextDom, eh) and
                     result);
    end;
end;

function testAttributeClass (ac: AttributeClass;
                             masterId: String;
                             var testedAC: Set of AttributeClass;
                             masterAC: Set of AttributeClass;
                             eh: ErrorHandler): Boolean;
var
  d: Domain;
  v: DefaultVal;
  innerAttrSet: Set of String;
begin
  result := true;

  if masterAC.hasElement (ac) then
    begin
      eh.processError (
        GXLAttributeClassCircleError.create (ac, masterId));
      result := false;
    end
  else
    begin
      masterAC.add (ac);
      if not testedAC.hasElement (ac) then
        begin
          testedAC.add (ac);
          d := ac.getDomain;
          if (result and ac.hasDefaultValue) then
            begin
              v := ac.getDefaultValue;
              result := testDefaultValue (ac, v, d, eh) and
                        testEnumValue (ac, v, d, eh);
            end;
```

```
          result := (testAttributedElementClass (ac,
                                                 innerAttrSet,
                                                 testedAC,
                                                 masterAC,
                                                 eh) and
                     result);
        end;
    end;
end;

function testAttributedElementClass (
             aec: AttributedElementClass;
             var attrSet,
             testedAC: Set of String;
             masterAC: Set of String;
             eh: ErrorHandler): Boolean;
begin
  result := true;
  foreach AttributeClass ac of aec do
    begin
      result := testAttributeClass (
                    ac, aec.getId, testedAC, masterAC, eh);
      if (attrSet.hasElement (ac.getName) then
        begin
          eh.processError (
            GXLDuplicateAttributeClassNameError.create (aec, ac));
          result := false;
        end
      else
        attrSet.add (ac.getName);
    end;
end;

function testGraphElementClass (
             gec: GraphElementClass;
             var attrSet,
             testedAC: Set of String;
             visited: Set of GraphElementClass;
             eh: ErrorHandler): Boolean;
var
  masterAC: Set of String;
```

```
begin
  result := true;
  if not visited.hasElement (gec) then
    begin
      visited.add (gec);
      foreach directSuperClass GraphElementClass superClass of gec do
        if result then
          result := testGraphElementClass (
                        superClass, attrSet, testedAC, eh));

      result := testAttributedElementClass (gec,
                                            attrSet,
                                            testedAC,
                                            masterAC,
                                            eh);
  end;
end;

function testSchemaAttributes (
             G: Graph;
             S: Schema;
             GC: GraphClass;
             eh: ErrorHandler): Boolean;
var
  attrSet, testedAC,
  masterAC: Set of String;
begin
  result := testAttributedElementClass (
                GC, attrSet, masterAC, eh);
  foreach RelationClass rc of GC do
    foreach RelationEndClass rec of rc do
      begin
        attrSet.clear;
        masterAC.clear;
        result := (testAttributedElementClass (rec,
                                               attrSet,
                                               testedAC,
                                               masterAC,
                                               eh) and
                   result);
      end;
```

```
  foreach GraphElementClass gec of GC do
    begin
      attrSet.clear;
      result := (testGraphElementClass (gec,
                                        attrSet,
                                        testedAC,
                                        eh) and
                 result);
    end;
end;
```

3.6.6 Korrektheit der Kantenklassen

Beschreibung

GXL-Schemata erlauben die Definition von Kantenklassen. Eine Kantenklasse bestimmt, welche Graphelemente Kanten dieses Typs miteinander verbinden können. Zusätzlich werden Multiplizitäten angegeben, die die für diesen Kantentyp gültigen Kardinalitäten bestimmen.
Damit ein Schema schemakorrekt ist, müssen die Multiplizitäten gültig sein. Das bedeutet, dass die Untergrenze die Obergrenze nicht überschreiten darf. Liegen Spezialisierungen von Kantenklassen vor, so müssen weitere Anforderungen erfüllt sein. Spezialisierte Kantenklassen dürfen nur Graphelementklassen verbinden, die mit den *From-/ToElement*-Klassen der Oberklasse identisch bzw. Unterklassen dieser sind.
Die Multiplizitäten einer spezialisierten Kantenklasse dürfen die Grenzwerte der Oberklasse nur weiter einschränken oder müssen mit denen der Oberklasse identisch sein.
Des Weiteren muss bei spezialisierten Kantenklassen auf den Richtungsmodus geachtet werden. Gerichtete Kantenklassen dürfen nur zu gerichteten, ungerichtete zu gerichteten oder ungerichteten Kantenklassen spezialisiert werden.
Neben Kantenklassen kann ein Schema Aggregations- und Kompositionsklassen enthalten. Für diese besonderen Kantenklassen gelten weitere Anforderungen an die Multiplizitäten dieser Klassen. Bei Aggregationsklassen ist darauf zu achten, dass die aggregierten Elemente von mehr als nur einem Element aggregiert werden können. Die durch eine Komposition aggregierten Elemente dürfen nur genau von einem Element aggregiert werden.

Beispiele

Abbildung 3.44 zeigt die Spezialisierung einer Kantenklasse. Die Forderungen nach korrekten Limits und korrekten Endelementklasse werden in diesem Beispiel erfüllt. Negativbeispiel für diese Anforderungen sind in den Abbildungen 3.45 und 3.46 zu sehen. Während bei der Spezialisierung in Abbildung 3.45 die Forderung an korrekte Endelementklassen verletzt wird, sind in Abbildung 3.46 ungültige Limits angegeben.

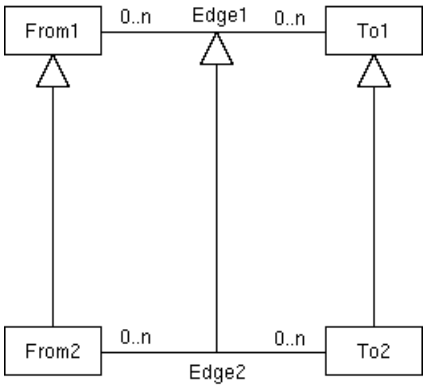

Abbildung 3.44: Spezialisierung einer Kantenklasse

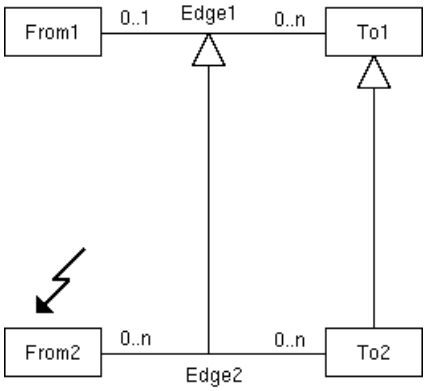

Abbildung 3.45: Spezialisierung einer Kantenklasse mit fehlerhaftem Kantenende

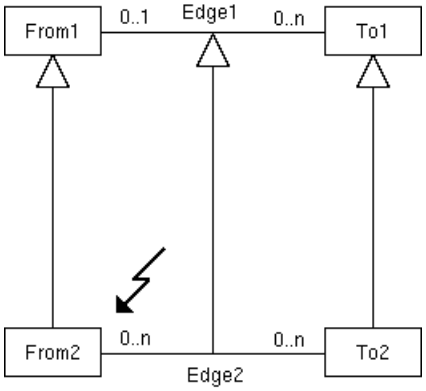

Abbildung 3.46: Spezialisierung einer Kantenklasse mit fehlerhaften Limits

Pseudocode

Die Funktion `testSchemaEdges` überprüft, ob die Anforderung an korrekte Spezialisierung erfüllt wird. Dazu testet `testEdgeClassLimits`, ob die Grenzwerte einer Kantenklasse korrekt sind, `testSuperEdgeClass` untersucht, ob die Grenzwerte in Bezug auf die Grenzwerte einer Oberklasse nur weiter eingeschränkt werden und ob die Endelementklassen mit denen der Oberklasse identisch oder Unterklassen der dort angegebenen sind. Außerdem wird überprüft, ob gerichtete Kantenklassen nur zu gerichteten spezialisiert werden.

```
const
  cFrom = 'from';
  cTo = 'to';

function testEdgeClassLimits (
             ecEnd: RelatesTo;
             ec: EdgeClass;
             ecEndType: String;
             eh: ErrorHandler): Boolean;
var
  ac: AggregationClass;
  cc: CompositionClass;
begin
  result := true;
  if (ecEnd.getLowerLimit>ecEnd.getUpperLimit) then
```

```
    begin
      eh.processError (
        GXLLimitError.create (ec,
                              ecEndType,
                              ecEnd.getLowerLimit,
                              ecEnd.getUpperLimit));
      result := false;
    end;

  if (ec is AggregationClass) then
    begin
      ac := ec as AggregationClass;
      if ((((ac.getAggregate=from) and (ecEndType=cFrom)) or
           ((ac.getAggregate=to) and (ecEndType==cTo))) and
          (lowerLimit=1) and (upperLimit=1)) then
        eh.processWarning (GXLLimitError.create (ac,
                                                 ecEndType,
                                                 lowerLimit,
                                                 upperLimit));
    end;

  if (ec is CompositionClass) then
    begin
      cc := ec as CompositionClass;
      if ((((cc.getAggregate=from) and (ecEndType=cFrom)) or
           ((cc.getAggregate=to) and (ecEndType=cTo))) and
          (lowerLimit<>1) ans (upperLimit<>1)) then
        eh.processWarning (GXLLimitError.create (cc,
                                                 ecEndType,
                                                 lowerLimit,
                                                 upperLimit));
    end;
end;

function testSuperLimits (ecEnd, superEnd: RelatesTo;
                          ecEndType: String;
                          ec, superEC: EdgeClass;
                          eh: ErrorHandler): Boolean;
begin
  result := true;
  if ((ecEnd.getLowerLimit<superEnd.getLowerLimit) or
```

```
      (ecEnd.getUpperLimit>superEnd.getUpperLimit))) then
    begin
      eh.processError (
        GXLEdgeClassSuperLimitError.create (ec, superEC,
                                            ecEnd.getLowerLimit,
                                            ecEnd.getUpperLimit,
                                            superEnd.getLowerLimit,
                                            superEnd.getUpperLimit,
                                            ecEndType));
      result := false;
    end;
end;

function testSuperEdgeClass (ec: EdgeClass;
                             superEC: EdgeClass;
                             eh: ErrorHandler): Boolean;
begin
  result := true;
  if (not ec.getFromClass=superEC.getFromClass) and
     (not ec.getFromClass isSubClassOf superEC.getFromClass)) then
    begin
      eh.processError (
        GXLEdgeInheritanceError.create (ec,
                                        ec.getFromClass,
                                        superEC.getFromClass,
                                        cFrom));
      result := false;
    end;

  if (not ec.getToClass=superEC.getToClass) and
     (not ec.getToClass isSubClassOf superEC.getToClass)) then
    begin
      eh.processError (
        GXLEdgeInheritanceError.create (ec,
                                        ec.getToClass,
                                        superEC.getToClass,
                                        cTo));
      result := false;
    end;

  result := (testSuperLimits (ec.getFrom,
```

```
                                super.getFrom,
                                cFrom,
                                ec,
                                superEC,
                                eh) and
            result);
  result := (testSuperLimits (ec.getTo,
                                super.getTo,
                                cTo,
                                ec,
                                superEC,
                                eh) and
            result);

  if (superEC.isDirected and (not ec.isDirected)) then
    begin
      eh.processWarning (
        GXLEdgeClassDirectionError.create (ec, superEC));
      result := false;
    end;
end;

function testSchemaEdges  (G: Graph;
                           S: Schema;
                           GC: GraphClass;
                           eh: ErrorHandler): Boolean;
begin
  result := true;
  foreach EdgeClass ec of GC do
    begin
      result := testEdgeClassLimits (ec.getFrom,
                                     ec,
                                     cFrom,
                                     eh);

      result := (testEdgeClassLimits (ec.getTo,
                                      ec,
                                      cTo,
                                      eh) and
                 result);
```

```
      foreach EdgeClass superEC of ec
          with (superEC is super class of ec) do
        testResult := (testSuperEdgeClass (ec,
                                           superEC,
                                           eh) and
                       result);
    end;
end;
```

3.6.7 Korrektheit der Relationsklassen

Beschreibung

Neben Kantenklassen können in einem Schema auch Relationsklassen definiert werden. Diese Definitionen geben an, welche Relationsenden für Relationen eines bestimmten Typs möglich sind. Daneben werden analog zu den Kantenklassen bei Relationsendeklassen Multiplizitäten angegeben.
Damit ein Schema schemakorrekt ist, müssen diese Grenzwertangaben gültig sein, d.h. die Untergrenze darf die Obergrenze nicht überschreiten. Außerdem müssen die Rollen der Relationsende-Klassen bezogen auf eine Relationsklasse eindeutig sein. Dabei muss berücksichtigt werden, dass Relationsklassen Vererbung unterstützen. Bei Vererbung von Relationsklassen ist ein Überschreiben von Relationsendeklassen durch Angabe einer Klasse mit identischer Rollenbezeichnung nicht möglich. Bei Spezialisierung einer Relationsklasse können die möglichen Relationsenden also nur erweitert werden. Prinzipiell wäre es möglich, ein solches Überschreiben zu erlauben. Analog zur Spezialisierung bei Kantenklassen könnte gefordert werden, dass eine spezialisierte Relationsendeklasse als Zielklasse die mit der der Oberklasse identische Klasse oder eine Unterklasse dieser besizten darf, die Limits dürften nur weiter eingeschränkt werden. Aus Gründen der Handhabbarkeit wurde dies allerdings verworfen zugunsten einer einfacheren Lösung.

Beispiele

Das Positivbeispiel in Abbildung 3.47 zeigt die Spezialisierung einer Relationsklasse. Die spezialisierte Klasse erweitert die Relationklasse um eine weitere Relationsende-Klasse.
Dagegen enthält Abbildung 3.48 eine Relationsklasse mit nicht eindeutigen Rollen, eine Relationsende-Klasse der Relationsklasse aus Abbildung 3.49 weist eine ungültige Multiplizität auf und die Spezialisierung der Relationsklasse in Abbildung 3.50 ist ebenfalls fehlerhaft. In diesem Negativbeispiel

besitzt die spezialisierte Relationsklasse eine Rolle, die von der Oberklasse bereits definiert worden ist.

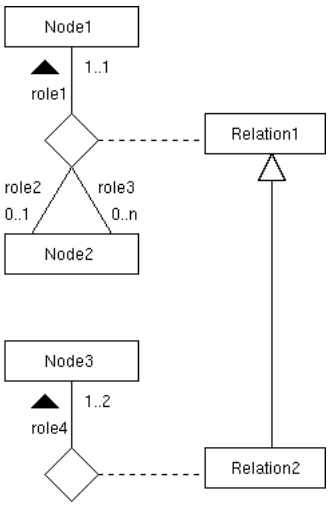

Abbildung 3.47: Definition zweier Relationsklassen

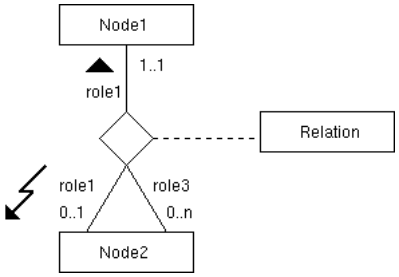

Abbildung 3.48: Relationsklasse mit nicht eindeutige Rollen

Pseudocode

Die Funktion `testSchemaRelationClass` überprüft, ob die Anforderung an korrekte Relationsklassen erfüllt wird. Dazu wird die Funktion `testRelEndClassLimits` verwendet, die testet, ob alle Relationsende-Klassen korrekte Limits besitzen. Die Funktion `testRoles` überprüft die Rolleneindeutigkeit bezogen auf eine Relationsklasse.

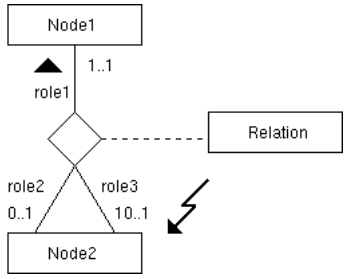

Abbildung 3.49: Relationsendeklasse mit ungültiger Multiplizität

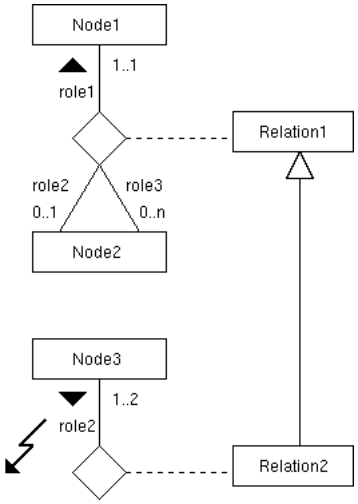

Abbildung 3.50: Spezialisierte Relationsklasse mit nicht eindeutigen Rollen

```
function testRelEndClassLimits (
             rec: RelationEndClass;
             eh: ErrorHandler): Boolean;
begin
  result := true;
  with rec.getRelatesTo do
    if (getLowerLimit>getUpperLimit) then
      begin
        eh.processError (
          GXLLimitError.create(rec, getLowerLimit, getUpperLimit));
        result := false;
      end;
end;

function testRoles (S: Schema;
                    GC: GraphClass;
                    eh: ErrorHandler;
                    rc: RelationClass;
                    var roles: Set of String): Boolean;
begin
  result := true;

  foreach directSuperClass RelationClass superRC of rc do
    result := testRoles (S, GC, eh, rc, roles) and result;

  foreach RelationEndClass rec of rc do
    begin
      if (roles.hasElement (rec.getRole) then
        begin
          eh.processError (GXLDuplicateRoleError.create (rc, rec));
          result := false;
        end
      else
        roles.add (rec.getRole);
    end;
end;

function testSchemaRelationClass (
             G: Graph;
             S: Schema;
             GC: GraphClass;
```

```
            eh: ErrorHandler): Boolean;
var
  roles: Set of String;
begin
  result := true;

  foreach RelationClass rc of GC do
    begin
      foreach RelationEndClass rec of rc do
        result := (testRelEndClassLimits (rec, eh) and
                   result);

      roles.clear;
      result := (testRoles (S, GC, eh, rc, roles) and
                 result);
    end;
end;
```

Kapitel 4

Implementation

Nachdem im letzten Kapitel die theoretischen Grundlagen für den Validierer besprochen wurden, widmet sich dieses Kapitel der praktischen Umsetzung. Das Ergebnis der Implementation soll ein Programm sein, das ein GXL-Dokument einliest und die darin enthaltenen Graphen auf Einhaltung der in Kapitel 3 aufgestellten Anforderungen hin testet und gefundene Abweichungen protokolliert.
Der folgende Abschnitt befasst sich mit der verwendeten Entwicklungsumgebung und eingesetzten Schnittstellen.

4.1 Entwicklungs- und Laufzeitumgebung

Zur Entwicklung des Programms wurde die Sprache C++ festgelegt. Maßgeblich für diese Entscheidung waren Überlegungen zur Performanz und Anbindungen an bestehende bzw. noch zu entwickelnde Schnittstellen. Bei diesen Schnittstellen handelt es sich um zwei APIs (*Application Programming Interface*), die alle Zugriffe auf GXL-Graphen und -Schemata kapseln. Durch Verwendung dieser APIs kann auf direkte XML-Zugriffe verzichtet werden.
Die erste dieser beiden ist die GXLInstanceAPI (vgl. [4]). Sie bietet eine Schnittstelle für GXL-Dokumente und alle darin enthaltenen Elemente. Für jedes Element eines GXL-Dokumentes existiert eine entsprechende Repräsentation innerhalb der API in Form von Klassen. So wird zum Beispiel ein Knoten durch die Klasse `GXLNode` repräsentiert und alle Eigenschaften eines Knoten durch sie verfügbar gemacht. Verschiedene Iteratoren bieten z.B. die Möglichkeit, alle Elemente eines Graphen oder auch zu einem Element inzidente Elemente zu durchlaufen.
Die zweite API ist die GXLSchemaAPI (vgl. [5]). Diese Schnittstelle basiert

auf der GXLInstanceAPI und kapselt GXL-Schemata für den einfachen Zugriff. Das Konzept entspricht dem der GXLInstanceAPI. So existieren auch hier für jedes Schemaelement entsprechende Klassen, die deren Eigenschaften zugreifbar machen.
Kompiliert wurde das Projekt sowohl unter Linux (gcc 3.2), als auch unter Windows (MS Visual Studio .NET). Somit stehen unter beiden Betriebssystemen ausführbare Versionen des Programmes zur Verfügung.
Um eine Idee von der Struktur des Systems zu vermitteln, beschäftigt sich der folgende Abschnitt mit der Architektur des Validierers.

4.2 Architektur

Der GXLValidator ist sehr modular aufgebaut. Für die verschiedenen Aufgabenbereiche des Systems existieren Klassen, die die jeweilige Funktionalität bereitstellen. Das Klassendiagramm in Abbildung 4.1 gibt einen Überblick über die Hauptbestandteile des GXLValidators.
Basis des Validators ist die abstrakte Klasse `GXLTest`. Alle Tests sind Spezialisierungen dieser Klasse.
Ein Test ist die Umsetzung einer Anforderung aus Kapitel 3. Durch Überschreiben der Klassenmethode `getTestName` wird einem Test ein Name zugewiesen. Die virtuelle boolsche Funktion `execute` führt die durchzuführenden Überprüfungen aus und muss demnach von jeder Testklasse überschrieben werden. Dabei benutzt sie einen `GXLErrorHandler` um gefundene Abweichungen von den überprüften Anforderungen zu protokollieren. Die Funktion `safeExecute` ruft `execute` auf, wobei mögliche Exceptions der verwendeten APIs aufgefangen und dem Errorhandler zur Protokollierung übergeben werden. Die virtuelle boolsche Klassenfunktion `canContinue` zeigt an, ob nach nicht erfolgreichem Abschluss eines Tests der gesamte Testvorgang abgebrochen werden muss oder nicht.

Die bereits erwähnte Klasse `GXLErrorHandler` orientiert sich an dem aus Abschnitt 3.1 bekannten Konzept. Fehler und Warnungen können erfasst und später ausgegeben werden. Eine Besonderheit ist die Möglichkeit, einem GXLErrorHandler erwartete Fehler mitzuteilen. Diese Eigenschaft wird von der Testumgebung des Systems genutzt (vgl. Abschnitt 4.4).
Um ganze Testgruppen bilden zu können, gibt es die Klasse `GXLTestSuite`. Diese ist ein Container für eine Sequenz von Tests. Eine `GXLTestSuite` bietet die Möglichkeit, Tests gegen einen Graphen und eine Graphklasse/Schema auszuführen. Dabei wird ein `GXLErrorHandler` eingesetzt, der an die jeweiligen Tests übergeben wird.
Wird während des Testdurchlaufs ein Test nicht erfolgreich beendet, so hängt

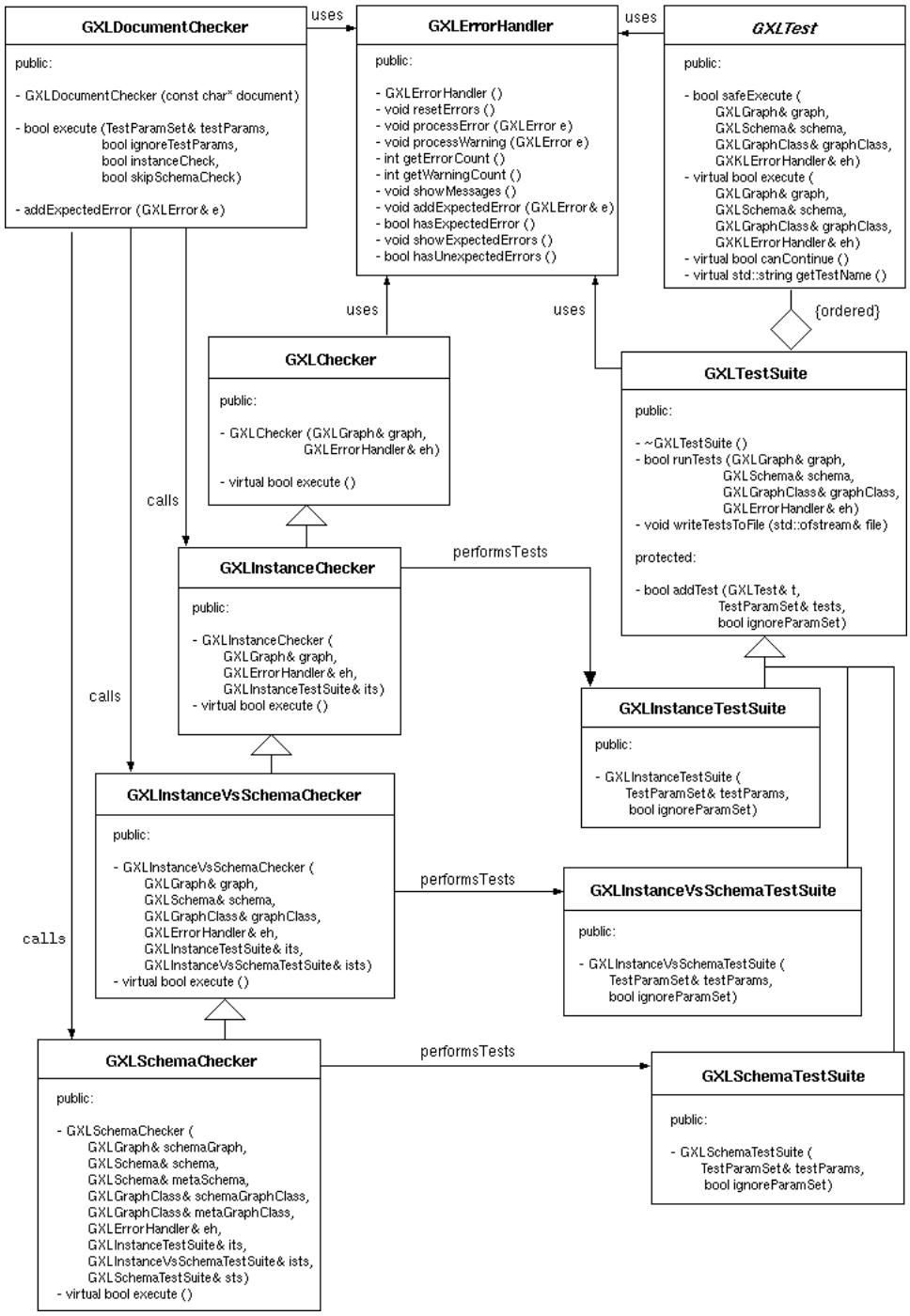

Abbildung 4.1: Klassendiagramm GXLValidator

es von dem Rückgabewert der Funktion `canContinue` des ausgeführten Tests ab, ob die Testsuite den Testdurchlauf abbricht, oder mit dem nächsten Test fortfährt. Muss der Testdurchlauf abgebrochen werden, so ist ein *kritischer* Fehler aufgetreten. Nach Durchlauf der Tests kann mithilfe des verwendeten Errorhandlers festgestellt werden, ob die Überprüfung Abweichungen gefunden hat oder nicht.
Eine weitere Eigenschaft einer `GXLTestSuite` ist, dass die Namen der bekannten Tests in eine Datei ausgegeben werden können. Diese Datei kann vom Anwender dazu verwendet werden, nur bestimmte Tests durchführen zu lassen und andere zu ignorieren. Dabei ist darauf zu achten, dass auch kritische Tests auf diese Weise umgangen werden können. Im Falle des Tests eines fehlerhaften GXL-Dokumentes besteht dann die Möglichkeit, dass ein Test nicht terminiert. Existiert z.B. eine zirkuläre Generalisierungsstruktur und der Test, der diesen Fehler erkennt, wird auf Anwenderwunsch ausgeschaltet, so terminieren die abhängigen Tests nicht (vgl. Abbildung 4.2).

Von `GXLTestSuite` gibt es verschiedene Spezialisierungen. `GXLInstanceTestSuite` beinhaltet all die Tests, die ausgeführt werden müssen, um einen Graphen auf **Instanzkorrektheit** zu überprüfen, `GXLInstanceVsSchemaTestSuite` solche, die nötig sind, um die **Schemakonformität** eines Graphen zu testen und `GXLSchemaTestSuite` analog die Tests, die zur Überprüfung der **Schemakorrektheit** herangezogen werden. Die Testsequenz einer Testsuite wird in deren Konstruktor angegeben. Dazu wird die Funktion `addTest` der Oberklasse verwendet. Diese Funktion fügt den angegebenen Test der Suite hinzu, wobei zuvor die Parameter `tests` und `ignoreParamSet` ausgewertet werden. Ist `ignoreParamSet` nicht `false`, so wird ein Test nur dann der Suite hinzugefügt, wenn sein Name Element der Menge `tests` ist. Der GXLValidator kann so die von `writeTests` erzeugte Datei nutzen, um nur solche Tests auszuführen, deren Namen in dieser Datei aufgelistet sind, sollte der Anwender dies wünschen (vgl. Anhang A.4).
Die Tests einer Testsuite werden in der Reihenfolge durchlaufen, in der sie im Konstruktor der internen Liste hinzugefügt wurden. Abbildung 4.2 gibt einen Überblick über die Testsuiten und deren Tests. Außerdem zeigt die Abbildung Abhängigkeiten unter einzelnen Tests bzw. den Testsuiten. Die Abhängigkeiten sagen aus, dass ein Test nicht ohne erfolgreiche Durchführung der Tests oder der Testsuite ausgeführt werden kann, von dem er abhängig ist.
Verwendung finden diese Suiten durch verschiedene sogenannte Checker. Oberklasse dieser Checker-Klassen ist die abstrakte Klasse `GXLChecker`.
`GXLInstanceChecker` benutzt eine Instanz der `GXLInstanceTestSuite`, um einen GXL-Graph auf Instanzkorrektheit hin zu untersuchen.
`GXLInstanceVsSchemaChecker` spezialisiert diese Klasse und erweitert sie um eine Untersuchung der Schemakonformität des Graphen. Dazu benutzt

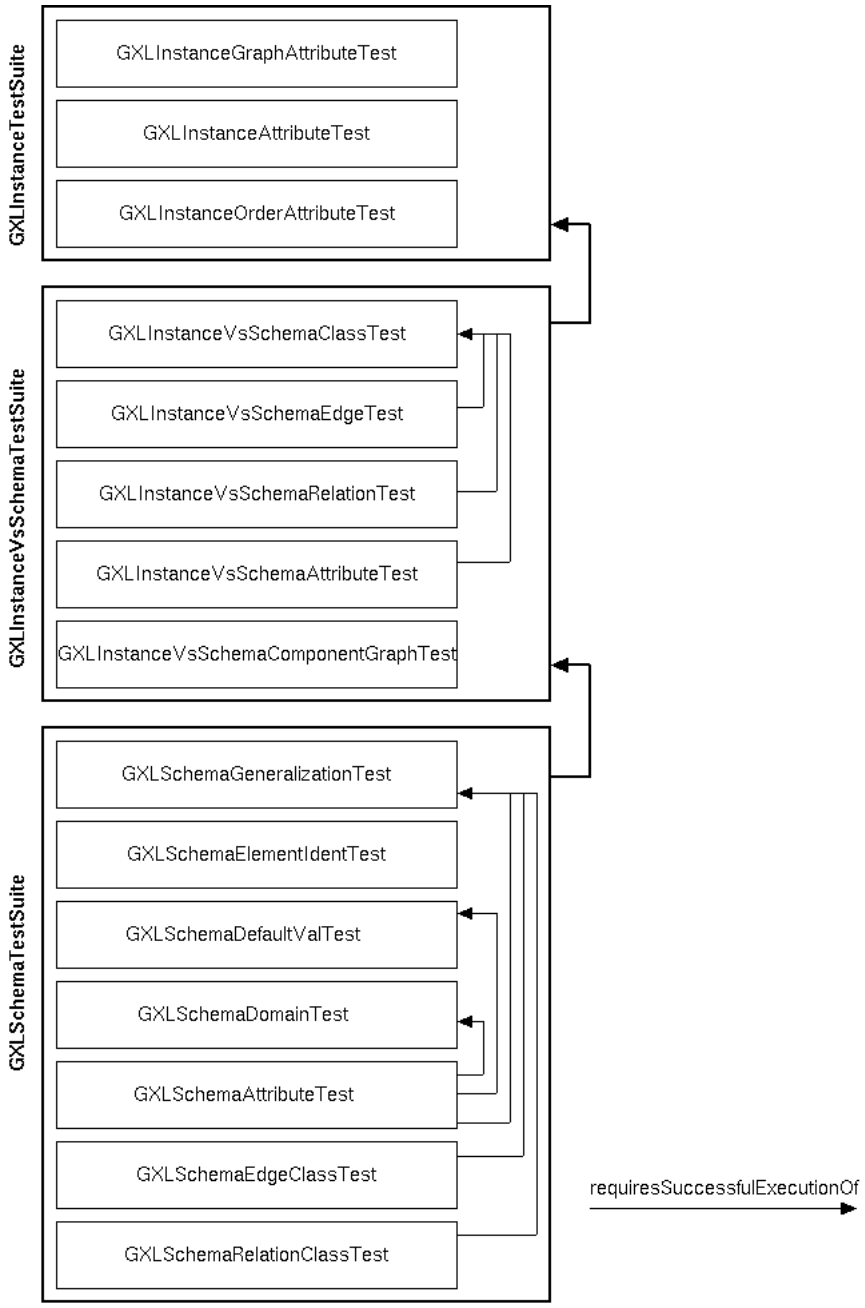

Abbildung 4.2: Abhängigkeiten zwischen Testsuiten und Tests

sie eine Instanz der `GXLInstanceVsSchemaTestSuite`. `GXLSchemaChecker` spezialisiert dieses Verhalten nochmals, um so einen Schemagraphen auf Graphkorrektheit, Metaschemakonformität und Schemakorrektheit hin zu überprüfen. Um dies zu erreichen, wird zusätzlich zur Funktionalität der Oberklasse eine `GXLSchemaTestSuite` eingesetzt.
Diese Checker werden von `GXLDocumentChecker` verwendet, um ein ganzes GXL-Dokument zu untersuchen. `GXLDocumentChecker` realisiert dabei das in 2.4 definierte Testverfahren. Je nachdem, welche Graphen in einem Dokument enthalten sind und welche Parameter (vgl. Anhang A.4) gesetzt sind, werden die nötigen oder gewünschten Überprüfungen durchgeführt. So kann über den boolschen Parameter `instanceCheck` festgelegt werden, ob die im Dokument enthaltenen Graphen nur auf Instanzkorrektheit überprüft werden sollen. Mit Hilfe des boolschen Parameters `skipSchemaCheck` hingegen kann angegeben werden, ob bei der Untersuchung der Schemakonformität eines GXL-Graphen auf die Überprüfung des angegebenen Schemas verzichtet werden soll. Standardmäßig wird die Korrektheit des Schemas getestet, bevor der Graph selbst untersucht wird.

4.3 Implementation eines Tests

Die Oberklasse aller Tests ist `GXLTest`. Diese abstrakte Klasse definiert die Schnittstelle, die allen Tests gemeinsam ist. Soll das System um einen neuen Test erweitert werden, so muss eine Unterklasse von `GXLTest` angelegt werden. Dieser Unterklasse muss ein eindeutiger Name zugewiesen werden, indem die virtuelle Klassenmethode `getTestName` überschrieben wird. Handelt es sich um einen kritischen Test, so muss ebenfalls die Klassenmethode `canContinue` überschrieben werden, die ansonsten den Rückgabewert `true` hat. Ein Test ist dann *kritisch*, wenn ein nachfolgender Test nicht ausgeführt werden kann, ohne dass die in dem Vorgängertest überprüfte Anforderung erfüllt ist.
Die eigentliche Arbeit bei der Implementation eines Tests bezieht sich auf das Überschreiben der boolschen Funktion `execute`. Liefert diese Funktion `true`, so bedeutet dies, dass die durch den Test überprüfte Anforderung erfüllt ist, `false` analog. Je nachdem ob es sich um einen Instance-, InstanceVsSchema- oder Schema-Test handelt, sind die eingehenden Paramter unterschiedlich zu interpretieren.
Die zu betrachtenden Parameter sind `GXLGraph graph`, `GXLSchema schema` und `GXLGraphClass graphClass`. Bei einem Instance-Test ist `graph` der zu überprüfende Instanzgraph, `schema` und `graphClass` sind mit nicht initia-

lisierten Werten belegt. Im Gegensatz dazu sind `schema` und `graphClass` bei InstanceVsSchema-Tests nicht undefiniert, sondern enthalten das Schema und die Graphklasse, gegen den der in `graph` übergebene Graph getestet werden soll. Anders ist dies bei Schema-Test, denn hier ist `graph` der Schemagraph, `schema` die Schemarepräsentation und `graphClass` die zu testende Graphklasse.
Ist der Test implementiert, so muss er abschließend noch einer Testsuite hinzugefügt werden, um ihn dem Validierer bekannt zu machen. Abhängig von der eben besprochenen Art des Tests muss er der GXLInstanceTestsuite, GXLInstanceVsSchemaTestSuite oder der GXLSchemaTestSuite zugeordnet werden. Der Test wird einer dieser Suiten zugewiesen, indem er im Konstruktor der Testsuite der Testliste über die Methode `addTest` angehängt wird. Dabei ist zu beachten, dass die Tests in der Reihenfolge ausgeführt werden, in der sie der Testliste hinzugefügt werden.
Nachdem dieser Abschnitt sich mit der Erweiterung eines bestehenden System befasst hat, ist das Testen des fertigen Systems Thema des folgenden Abschnitts.

4.4 Testumgebung

Um das Verhalten des GXLValidators zu testen, wurde eine Testumgebung geschaffen. Diese Testumgebung ermöglicht es, GXL-Dokumente zu testen und solche Fehler, die bekannt sind und somit erwartet werden, dem System vor Ausführung der Tests mitzuteilen.
Für jede Fehlerklasse gibt es einen Positiv- und einen Negativtest. Das bedeutet, die Testumgebung führt pro Fehlerklasse zwei Überprüfung durch. Das jeweils erste zu testende GXL-Dokument erfüllt die Anforderung, die durch die Fehlerklasse repräsentiert wird, das zweite verletzt sie.
Die erwarteten Fehler werden dem verwendeten `GXLDocumentChecker` mittels der Methode `addExpectedError (GXLError e)` vor Durchführung der Tests mitgeteilt. Dabei benutzt ein DocumentChecker die gleichnamige Methode und Funktionalität seines `GXLErrorHandler`.
Die Überprüfung eines Graphen mit erwarteten Fehlern verläuft identisch zu der eines Graphen ohne erwartete Fehler. Der Unterschied besteht in der Behandlung eines festgestellten Fehlers, da hierbei überprüft wird, ob es sich um einen erwarteten oder unerwarteten Fehler handelt. Die boolsche Funktion `execute` eines `GXLDocumentChecker` liefert genau dann ein negatives Ergebnis, wenn entweder ein unerwarteter Fehler aufgetreten ist oder ein erwarteter Fehler nicht festgestellt werden konnte.
Die Testumgebung selbst ist ein ausführbares Programm, das alle Tests

durchläuft, bis ein Fehler auftritt oder alle Test erfolgreich abgeschlossen wurden. Mit Hilfe eines Parameters kann der Benutzer auch gezielt einzelne Tests durchführen.
Der letzte nun folgende Abschnitt dieses Kapitels widmet sich der Wiederverwendung einzelner Komponenten des Validierers.

4.5 Wiederverwendung von Komponenten

Der modulare Aufbau des GXLValidators bietet die Wiederverwendung einzelner Komponenten an. Eine Möglichkeit hierfür ist die Entwicklung eines Programms, dass GXL-Graphen nicht nur auf Korrektheit, sondern zusätzlich auf weitere Eigenschaften überprüft.
Im Rahmen der gxl2g-Entwicklung kam es zu der Frage, ob die Infrastruktur und auch einzelne Tests des GXLValidators für die Überprüfung von GXL-Graphen genutzt werden könnte. So könne sichergestellt werden, dass ein Graph, der von GXL nach g konvertiert werden soll, überhaupt in dieses Format konvertiert werden kann, da GXL z.B. mit Hypergraphen Konzepte kennt, die g nicht kennt.
Im vorangegangenen Abschnitt wurde gezeigt, wie der GXLValidator um neue Tests erweitert werden kann. Um bei dem oben genannten Beispiel zu bleiben, kann der bereits vorhandene `GXLInstanceChecker` genutzt werden, um einen GXL-Graphen auf Instanzkorrektheit zu überprüfen. Für jede weitere Anforderung die sich durch die Konvertierung ergibt, wird eine neue Testklasse implementiert. Diese neuen Tests können in einer ebenfalls neuen Testsuite gesammelt werden. Um diese Testsuite in das neue System einzubinden, kann eine neue Checker-Klasse erstellt werden, die in diesem Beispiel dann eine von `GXLInstanceChecker` spezialisierte Klasse wäre. Neben diesen Erweiterungen wird ein Modul benötigt, das die Funktion des `GXLDocumentChecker` innerhalb des GXLValidators übernimmt. In Anlehnung an diesen muss eine neue Klasse erstellt werden, die dazu in der Lage ist, sowohl ein GXLDokument zu öffnen als auch den neuen Checker auszuführen.
Bei der Wiederverwendung von Komponenten des GXLValidators in diesem Sinn kann der Entwickler sich größtenteils auf die Implementierung neuer Tests beschränken ohne eine eigene Umgebung zur Ausführung dieser Tests schaffen zu müssen.
Bevor das letzte Kapitel eine Zusammenfassung der Erfahrungen während der Entwicklung des GXLValidators und einen Ausblick auf weitere Aufgaben in diesem Themengebiet gibt, folgt im nächtsten Kapitel eine Kurzanleitung für das entwickelte System.

Kapitel 5

Kurzanleitung

Dieses Kapitel enthält eine kurze Anleitung zur Verwendung des GXLValidators. Die Anleitung ist in englischer Sprache verfasst, da sie in dieser Version zusammen mit dem Tool zum Download angeboten werden kann.

5.1 How to use

GXLValidator is a command line tool. The syntax to execute it is

```
gxlvalidator (-ddocument|--document document)
             [-writeCfg|-readCfg] [--instanceCheck|-i]
             [--skipSchemaCheck|-scc] [--help|-h]
```

It uses parameters to e.g. specify the document that should be validated. This is the complete parameter list:

- document specification
  - `--document` *document*
  - `-d`*document*

  This parameter specifies the gxl-document. All graphs within this document will be validated. The GXLValidator recognizes whether each of those graphs is an instance or schema graph and performs the required tests.

- instance checking
  - `--instanceCheck`
  - `-i`

  By declaring this parameter only instance tests will be performed. Schema information will be ignored.

- assertion of schemata correctness
  - `--skipSchemaCheck`
  - `-ssc`

  By giving this parameter all schemas will be treated as if valid. The user states the assertion that all schemas of all graphs are correct and valid. Therefore schema testing is not needed and will not be performed. This parameter is useful when testing multiple graphs or documents that hold the same schema. This schema should be tested once by using the GXLValidator and succeeding tests can be called with this parameter.

- test reporting
  - `-writeCfg`

  When this parameter is declared a file named tests.cfg will be created. tests.cfg contains a list of all tests that are executed during validation.

- exclusion of tests
  - `-readCfg`

  By declaring this parameter the file tests.cfg will be read and only those tests listed in this file will be performed. This way the user can rule out those tests he does not want to be executed.

- help
  - `--help`
  - `-h`

  A complete list of all parameters and a description of their usage will be shown.

5.2 Reporting of errors and warnings

When executing the GXLValidator some reporting messages will occur. Those messages display the current status of the validator and the testing. The GXLValidator logs all located errors and warnings. After executing all tests a list of those warnings and errors is shown. These messages consist of an error text and more information about the relevant items concerning the error. For example

```
Undefined class (node: n, typeId: c)
```

is displayed when node n with typeId c is element of the graph and the schema does not contain a nodeclass named c.
This is the complete list of all possible errors sorted by test classes:

General

GXLGraphClassNotFoundError	Undefined graphClass (graphClass: "graphClassId")

GXLInstanceGraphAttributeTest

GXLEdgeIdError	Violation of edgeid attribute (edge: "edgeId")
GXLEdgeDirectionError	Violation of edgemode attribute (edge: "edgeId")
GXLEdgeDirectionError	Directionmode violated (edge: "edgeId", edgeClass: "ecId")
GXLHypergraphError	Violation of hypergraph attribute (relation: relId")
GXLRelEndDirectionError	Violation of edgemode attribute (relation: rel", relend: relend")

GXLInstanceAttributeTest

GXLDuplicateAttributeNameError	Duplicate attribute (attributedElement: "aeId", attribute: "aName")
GXLTupValError	TupValue is not a tuple (attributedElement: "aeId", attribute: "aName")
GXLValueError	Attribute value mismatches attribute type (attributedElement: "aeId", attribute: "aName")
GXLBagSetSeqValError	Bag/Set/Seq items are not uniform (attributedElement: "aeId", attribute: "aName")

GXLInstanceOrderTest

GXLDuplicateOrderError	The order of incident edges/relends is not unique (graphElement: "geId", order: 1)

GXLInstanceVsSchemaClassTest

GXLAbstractClassError	Instance of abstract class (graphElement: "geId", graphElementClass: "gecId")
GXLUndefinedClassError	Undefined class (node: teId", typeId: typeId")
	Undefined class (edge: teId", typeId: typeId")
	Undefined class (relation: teId", typeId: typeId")
	Undefined class (graph: teId", typeId: typeId")

GXLInstanceVsSchemaEdgeTest

GXLCardinalityError	Multiplicity constraint violated (graphElement: teId", edgeClass: className", expected: 0x1, found: 2)
GXLWrongEdgeFromElementError	FromElement has wrong type (edge: "eId", edgeClass: "ecName", type: "geTypeId", expected type: "gecName")
GXLWrongEdgeToElementError	ToElement has wrong type (edge: "eId", edgeClass: "ecName", type: "geTypeId", expected type: "gecName")

GXLInstanceVsSchemaRelationTest

GXLCardinalityError	Multiplicity constraint violated (graphElement: teId", relationEndClass: "className", expected: 0x1, found: 2)
GXLRelEndDirectionError	Directionmode violated (relation: relId", relend: relEnd", relationEndClass: recId")
GXLUndefinedRelationEndClassError	Undefined relation end (relation: relId", relEnd role: relEndRole", relationClass: rcId")
GXLWrongRelEndTargetElementError	TargetElement has wrong type (relationClass: rcName", role: recRole")

GXLInstanceVsSchemaAttributeTest

GXLMissingAttributeError	Missing attribute (attributedElement: “aeId”, attributeClass: “acName”)
	Missing attribute (attributedElement: “aeId”, attributedElementClass: “aecId”, attributeClass: “acName”)
GXLUndefinedAttributeClassError	Undefined attribute (attributedElement: “aeId”, attribute: “attrName”)
GXLUndefinedEnumValError	Undefined enum value (attributedElement: “aeId”, attribute: “aName”, domain: ”domainId”)
GXLValueDomainError	Value does not match attribute domain (attributedElement: “aeId”, attribute: “aName”, domain: ”dId”)

GXLInstanceVsSchemaComponentGraphTest

GXLCardinalityError	Multiplicity constraint violated (graphElement: “teId”, graphClass: className”, expected: 0x1, found: 2)

GXLSchemaGeneralizationTest

GXLCyclicIsAStructureError	Cyclic isA structure (N1–>N2–>N1)
GXLSuperClassOwnershipError	GraphClass does not contain super class (graphClass: ”gcName”, graphElementClass: ”gecId”, superClass: “superId”)
GXLWrongSuperClassError	Incompatible super class (graphElementClass: ”gecName”, superClass: “superName”)

GXLSchemaElementIdentTest

GXLDuplicateElementIdentError	Duplicate identifier (graphElementClass: ”gecName”)

GXLSchemaDefaultValTest

GXLComponentDefaultValCircleError	Cyclic default value structure (compositeDefaultVal: cvId”, defaultVal: ”dvId”)
GXLDefaultAtomicValError	Default value mismatches value type (defaultValue: “avId”)
GXLDefaultTupValError	DefaultTupVal is not a tuple (defaultTupVal: cvId”)
GXLDefaultBagSetSeqValError	DefaultBag/Set/SeqVal items are not uniform (defaultCompositeVal: cvId”)

GXLSchemaDomainTest

GXLBagSetSeqComponentDomainError	Bag/Set/Seq has not exactly one component domain (domain: "dId")
GXLComponentDomainCircleError	Cyclic domain structure (compositeDomain: cdId", domain: "dId")
GXLTupLowComponentDomainError	TupDomain needs at least two component domains (domain: "dId")

GXLSchemaAttributeTest

GXLAttributeClassCircleError	Cyclic attributeClass structure (attributeClass: "acId", master attributeClass: "masterId")
GXLDefaultValueDomainError	DefaultValue does not match attribute domain (attributeClass: "acName", domain: "dId")
GXLDuplicateAttributeClassNameError	Duplicate attributeclass name (attributedElementClass: "aecId", attributeClass: "acName")

GXLSchemaEdgeClassTest

GXLEgdeClassDirectionError	Violation of direction (edgeClass: "ecId", super edgeClass: "superId")
GXLEdgeClassSuperLimitError	Invalid limits (edgeClass: "ecId", relatesTo: "ecEndType", limits: 0x5, superEdgeClass: "superECId", limits: 1x5)
GXLEdgeInheritanceError	Inheritance violation (edgeClass: "ecId", relatesTo: "ecEndType", graphElementClass: "gecId", super GEC: "superGecId")
GXLLimitError	Invalid limits (edgeClass: "ecId", end: "ecEndType", limits: 1x0)
	Invalid limits (aggregationClass: "ecId", end: "ecEndType", limits: 1x5)
	Invalid limits (compositionClass: "ecId", end: "ecEndType", limits: 0x5)

GXLSchemaRelationClassTest

GXLDuplicateRoleError	Duplicate role (relationClass: rcName", role: recRole")
GXLLimitError	Invalid limits (relationEndClass: recId", limits: 1x0)

5.3 Critical errors

An error is critical if succeeding tests require successfully passing this test. For example the GXLGeneralizationTest must be executed without an error before GXLAttributeTest can be run. This is because GXLGeneralization-Test looks for cyclic generalization structures and GXLAttributeTest assumes that there is no cyclic structure of this kind.
There are two impacts of critical errors that a user should be aware of. First when finding a critical error testing will be stopped and only those errors located up to this point will be reported. There may be more errors within the document. Those can be found by executing GXLValidator again after eliminating the reported errors.
Secondly the user should use the exclusion of tests with caution. When excluding a test that locates critical errors, the GXLValidator may not terminate.

Kapitel 6

Ausblick

Die Entwicklung des GXLValidators ist, soweit in dieser Arbeit vorgestellt, abgeschlossen. Während der Planungs- und Entwicklungsphase ist voralllem eine Frage aufgeworfen worden, deren Beantwortung auch heute noch problematisch ist. Es ist die Frage nach der vollständigen Erfassung aller Anforderungen. Oder anders ausgedrückt: Wie kann sichergestellt werden, dass alle Anforderungen erfasst worden sind?
Für diese Arbeit wurde ein pragmatischer Ansatz gewählt. Dieser geht davon aus, dass, wenn alle Elemente sowohl des GXL-Graphmodells als auch des GXL-Metaschemas bei der Erfassung der Anforderungen berücksichtigt worden sind, die Anforderungsliste komplett sein muss. Nach diesem Verfahren wurden die aus Kapitel 3 bekannten Anforderungen aufgestellt. Bis jetzt hat sich dieser Ansatz bewährt, denn auch nach langer und intensiver Beschäftigung mit dem Thema musste die Anforderungsliste in der letzten Phase der Entwicklung nicht erweitert werden.

Obwohl die Entwicklung des GXLValidators eigentlich abgeschlossen ist, bleiben dennoch Möglichkeiten, ihn zu erweitern oder zu verbessern. Einer dieser Punkte ist das Thema Laufzeitoptimierung. Soweit möglich wurde bei der Zusammenstellung der Anforderungen und später der Implementation darauf geachtet, thematisch und strukturell zusammenpassende Überprüfungen in einem Test zusammenzufassen. Trotzdem gibt es redundant durchlaufene Graphstrukturen. Hier gibt es Optimierungspotential. Allerdings müssten hierfür die bestehenden Strukturen grundlegend geändert werden, da eine Trennung der einzelnen Tests dann nicht mehr so möglich wäre, wie es zur Zeit der Fall ist. Da bis zur Fertigstellung dieser Arbeit aber keine Performance-Probleme bekannt geworden sind, gibt es auch keinen direkten Anlass, diese großen Änderungen in Angriff zu nehmen.

Zeitgleich mit der Entwicklung des GXLValidators wurden Änderungsanforderungen zu GXL 1.0 erfasst (vgl. [2]). Diese change requests führten zu GXL 1.1, der neuesten Version des Austauschformats. Das Grundprinzip der Sprache bleibt auch in dieser Version unverändert, allerdings gibt es einige Detailänderungen. Der GXLValidator ist auf die Überprüfung von GXL 1.0-Graphen und -Schemata ausgelegt. Um Dokumente der neuen Version untersuchen zu können, wären Änderungen sowie Erweiterungen sowohl an dem Validierer selbst, wie auch an den genutzten APIs, die ebenfalls für GXL 1.0 entwickelt worden sind, nötig.
Exemplarisch für die umgesetzten change requests sind die neu eingeführten strukturierten Attribute. In der neuen Version von GXL wird es möglich sein, sogenannte *Attribute-Structs* als Attribute zu verwenden. Diese Structs sind eine besondere Art von Attributen, die eine beliebige Anzahl weiterer Attribute aufnehmen können (vgl. [2]). Abbildung 6.1 zeigt, welche Änderungen am Graphmodell zur Umsetzung dieses neuen Konzeptes nötig sind. Neben

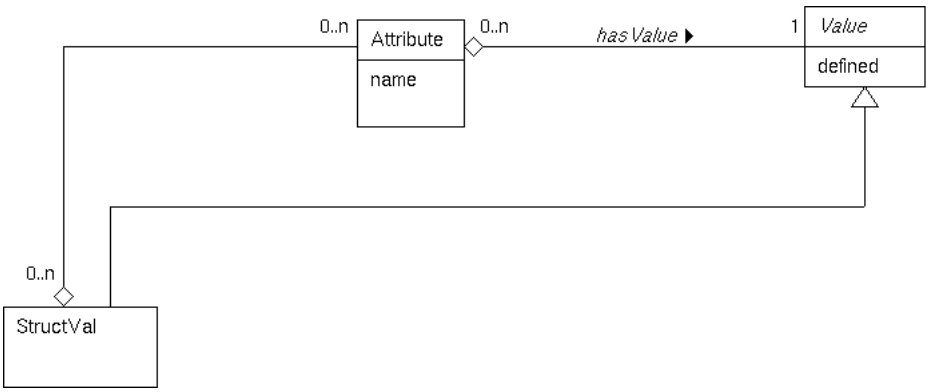

Abbildung 6.1: StructVal in GXL 1.1 (GraphModel) (Quelle: [2])

diesen Änderungen für GXL-Graphen muss auch das GXL-Metaschema erweitert werden. Diese Änderungen werden in den Abbildungen 6.2 und 6.3 dargestellt.
Soll der GXLValidator in die Lage versetzt werden neben GXL-1.0-Graphen auch GXL-1.1-Graphen untersuchen zu können, so müssen solche Änderungen berücksichtigt werden. In diesem Fall müssen alle Ebenen der Validierung (Instanzkorrektheit, Schemakonformität, Schemakorrektheit) betrachtet und überarbeitet werden. Bezogen auf dieses Beispiel müssen die Attribut-Tests die neuen Strukturen durchlaufen und wie bisher Attribute auf Korrekheit und Konformität zu den angegebenen Domänen untersuchen können. Bezogen auf ein Schema muss die neue Domainstruktur ebenso wie die Belegung

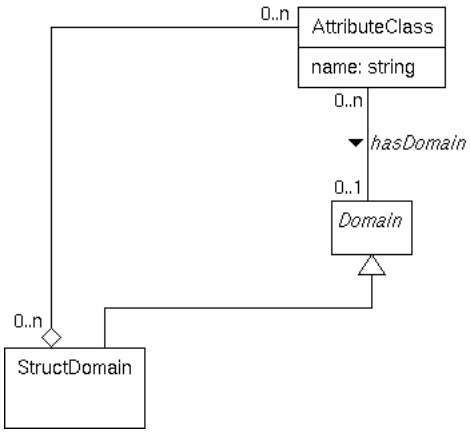

Abbildung 6.2: StructDomain in GXL 1.1 (Metaschema) (Quelle: [2])

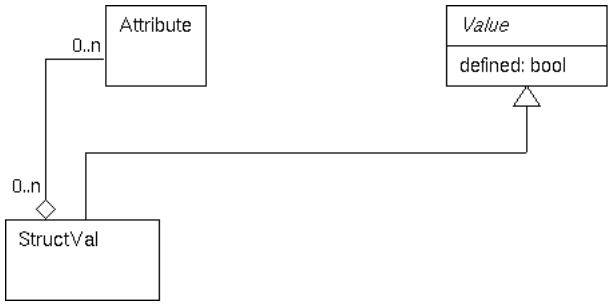

Abbildung 6.3: StructVal für Defaultwerte in GXL 1.1 (Metaschema) (Quelle: [2])

von Standardwerten (DefaultValues) überprüft werden können.
Grundlage für Erweiterungen dieser Art sind Änderungen bzw. Ergänzungen der APIs. Entweder müssten die bestehenden APIs um GXL1.1-Funktionalität erweitert werden, oder ganz neue APIs für diese Version müssten erstellt werden. Auch wenn große Teile des Validierers überarbeitet werden müssen, um GXL-1.1-Dokumente untersuchen zu können, so bleiben die grundsätzlich zu untersuchenden Strukturen doch die gleichen. Daher können das Grundgerüst des GXLValidators und auch der Großteil der bestehenden Testklassen für ein solches Projekt wiederverwendet werden.

Interessante Ergebnisse brachten die ersten Einsätze des fertigen Validierers. So wurde beispielsweise ein Fehler im GXL-Metaschema gefunden. Dieser Fehler bezieht sich auf die Vererbung von Kantenklassen. Wie in Anforderung 3.6.6 (*Korrektheit der Kantenklassen*) gefordert, darf eine spezialisierte Kantenklasse nur solche Klassen miteinander verbinden, die mit denen der Ober-Kantenklasse identisch oder Unterklassen dieser sind.
In der nicht korrigierten Version des Metaschemas gab es Vererbungsbeziehungen, die in Abbildung 6.4 dargestellt sind. Diese Beziehungen verletzen aber die gerade genannte Anforderung. Daraufhin wurde das GXL-Metaschema so geändert, dass diese Anforderungen erfüllt werden. In der überarbeiteten Version zeigt sich die Situation jetzt anders. Wie in Abbildung 6.5 dargestellt, wurden die Vererbungsbeziehungen aufgelöst, um dieses Problem zu umgehen. Ursprünglich wurden diese Generalisierungen verwendet, um die gemeinsamen Attribute nicht in jeder Klasse erneut definieren zu müssen.
Mit diesem Problem befasst sich ein change request für GXL 1.1. In der neuen GXL-Version wäre eine Oberklasse möglich, die die gemeinsamen Attribute aufnimmt, aber weder Knoten-, Kanten- noch Relationsklasse ist, sondern eine Graphelementklasse (vgl. [2]). Durch Verwendung diese Ansatzes kann in GXL 1.1 auch das Problem einer gemeinsamen Oberklasse gelöst werden.
Neben diesem Fall wurden in verschiedenen Beispielgraphen und -schemata weitere Fehler unterschiedlichster Art gefunden. Teilweise handelte es sich um einfache Tippfehler, die zu ungültigen Dokumenten führten. In anderen Fällen wurden beispielsweise Graphattribute nicht korrekt gesetzt. Diese Beispiele machen deutlich, dass Entwicklung und Einsatz eines Validierers nötig sind, um die Korrektheit von GXL-Dokumenten weitestgehend sicherzustellen. Der im Rahmen dieser Arbeit entwickelte GXLValidator hat nach heutigem Stand dieses Ziel erreicht.

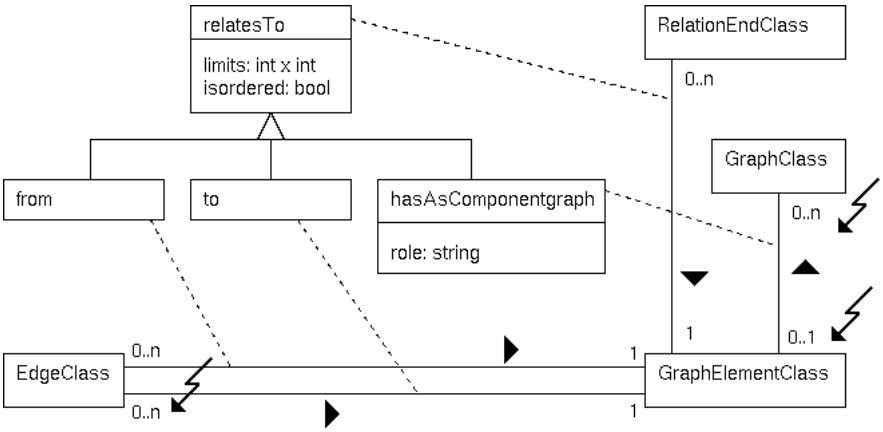

Abbildung 6.4: Fehlerhafte Generalisierung im Metaschema

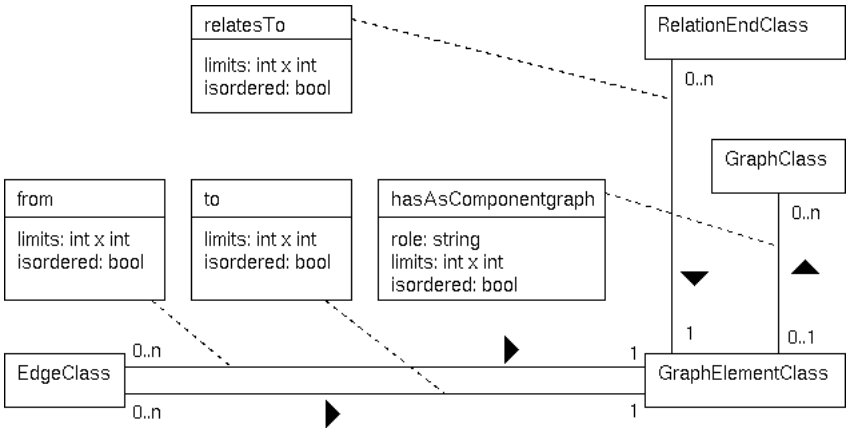

Abbildung 6.5: Aufgelöste Generalisierungen im Metaschema

Anhang A

Anhang

A.1 Definition des Pseudocodes

Der verwendete Pseudocode lehnt sich an die Sprache *Object Pascal* an. Dabei werden weitere Konstrukte benutzt, die diese Sprache nicht kennt. Diese Konstrukte werden im folgenden beschrieben.

A.1.1 Schleifen-Konstrukte

Innerhalb des Pseudocode werden für die Iterationen über verschiedenste Strukturen `foreach`-Schleifen verwendet. Dabei folgen alle Schleifen dem gleichen Prinzip:

`foreach` *ItemType ItemIdentifier* `of` *ItemizedElement* `do`

Nach dem einleitenden Schlüsselwort `foreach` folgt die Typangabe der iterierten Elemente und ein Bezeichner über den innerhalb des Schleifenblocks auf die iterierten Elemente zugegriffen werden kann. Nach dem folgenden Schlüsselwort `of` wird das Element angegeben, das die zu iterierenden Elemente besitzt.
Neben diesen einfachen Schleifen gibt es weitere, die zusätzliche Eigenschaften aufweisen. Diese Schleifen werden im Folgenden einzeln aufgeführt.

- `foreach incident Edge` *e* `of` *GraphElement ge* `do`
  Diese Schleife durchläuft alle zu dem Graphelement *ge* inzidenten Kanten
- `foreach ingoing Edge` *e* `of` *GraphElement ge* `do`
  Diese Schleife durchläuft alle in das Graphelement *ge* eingehenden Kanten

- `foreach outgoing Edge` *e* `of` *GraphElement ge* `do`
  Diese Schleife durchläuft alle von dem Graphelement *ge* ausgehenden Kanten

- `foreach directSuperClass GraphElementClass` *gec* `of` *GraphElementClass gec* `do`
  Diese Schleife durchläuft alle direkten Oberklassen der Graphelementklassen *gec*

- `foreach HasAsComponentGraphClass` *hasAsCG* `of` *GraphClass gc* `do`
  Diese Schleife durchläuft alle Klassen von eingebettete Graphen der Graphklasse *gc*

- `foreach incident RelationEnd` *re* `of` *GraphElement ge* `do`
  Diese Schleife durchläuft alle zu dem Graphelement *ge* inzidenten Relationesenden

Für alle `foreach`-Schleifen gibt es einen optionalen Zusatz. Sollen die iterierten Elemente weiter eingegrenzt werden, so ist dies möglich durch Angabe der Klausel

`with (`*condition*`)`

Ist dieser Zusatz vorhanden, so werden nur die Elemente durchlaufen, die das Prädikat *condition* erfüllen. Im Abschnitt A.1.2 werden die zur Verfügung stehenden Prädikate näher beschrieben.

A.1.2 Prädikate

In diesem Abschnitt werden die im Pseudocode verwendeten Prädikate definiert. Diese Prädikate repräsentieren teilweise umfangreichen Code, der zur Überprüfung des jeweiligen Ausdrucks nötig ist.

- *Value v* `is of type` *ValueType t*
  Dieses Prädikat ist erfüllt, gdw. die Belegung von *v* vom Typ *t* ist

- *GraphElement ge* `is instance of` *GraphElementClass GEC*
  Dieses Prädikat ist erfüllt, gdw. das Graphelement *ge* Instanz der Graphelementklasse *GEC* oder einer deren Unterklassen ist

- *DefaultValue dv* `is of type` *ValueType t*
  Dieses Prädikat ist erfüllt, gdw. die Belegung von *dv* vom Typ *t* ist

- *Value v* `matches domain` *Domain d*
  Dieses Prädikat ist erfüllt, gdw. die Belegung von *v* zum Wertebereich *d* passt

- *Graph g* `is valid GXLInstance`
  Dieses Prädikat ist erfüllt, gdw. der Graph *g* ein korrekter Instanzgraph ist (vgl. Abschnitt 3.4)

- *Graph g* `matches schema`
  Dieses Prädikat ist erfüllt, gdw. der Graph *g* schemakonform ist (vgl. Abschnitt 3.5)

A.1.3 Methoden

Neben Prädikaten verwendet der Pseudocode Methoden, die Zugriff auf Strukturen zur Verfügung stellen oder Aussagen über diese ermöglichen. Die an die GXLInstanceAPI [4] und GXLSchemaAPI [5] angelehnten Methoden werden in diesem Abschnitt einzeln beschrieben.

- `function Attribute.getName: String`

  Diese Funktion liefert den Namen eines Attributs

- `function Attribute.getValue: Value`

  Diese Funktion liefert den Wert eines Attributs

- `function Attribute.getValueType: ValueType`

  Diese Funktion liefert den Werttyp eines Attributs (z.B.: string, int, bool)

- `function AttributeClass.getDefaultValue: DefaultVal`

  Diese Funktion liefert den Defaultwert der Attributklasse, falls vorhanden

- `function AttributeClass.getDomain: Domain`

  Diese Funktion liefert die Domain der Attributklasse

- `function AttributeClass.getName: String`

  Diese Funktion liefert den Namen (Bezeichner) der Attributklasse

- `function AttributeClass.hasDefaultValue: Boolean`

  Diese Funktion sagt aus, ob die Attributklasse einen Defaultwert besitzt

- `function AttributedElement.getId: String`

  Diese Funktion liefert die Id eines Elementes

- `function AttributedElement.hasAttribute (a: String)`

  Diese Funktion sagt aus, ob das attributierte Element ein Attribut mit dem Namen `a` besitzt

- `function AttributedElementClass.hasAttribute (a: String)`

  Diese Funktion sagt aus, ob die attributierte Klasse eine Attributklasse mit dem Namen `a` besitzt

- `function CompositeDefaultValue.getComponentType: ValueType`

  Diese Funktion liefert den aussagekräftigsten Typ der Komponenten des CompositeDefaultValues. Dabei ist der aussagekräftigeste Typ der, der die längste Signatur besitzt (z.B. TUP(SET(INT)xBAG(STRING)))

- `function CompositeDomain.getFirstComponentDomain: Domain`

  Diese Funktion liefert die erste Domain der CompositeDomain

- `function CompositeDomain.getNextComponentDomain: Domain`

  Diese Funktion liefert die jeweils nächst Domain der CompositeDomain

- `function Domain.isTup: Boolean`

  Diese Funktion sagt aus, ob die CompositeDomain ein Tupel-Domain ist

- `function DefaultVal.isAtomic: Boolean`

  Diese Funktion sagt aus, ob der DefaultVal atomar ist

- `function DefaultVal.isComposite: Boolean`

  Diese Funktion sagt aus, ob der DefaultVal zusammengesetzt ist

- `function Domain.isComposite: Boolean`

  Diese Funktion sagt aus, ob die Domain zusammengesetzt ist

- `function Edge.getFrom: GraphElement`

  Diese Funktion liefert das From-Element der Kante

- `function Edge.getIsDirected: Boolean`

  Diese Funktion sagt aus, ob eine Kante gerichtet ist

- `function Edge.getTo: GraphElement`

  Diese Funktion liefert das To-Element der Kante

- `function Edge.isOrdered: Boolean`

  Diese Funktion sagt aus, ob die Kante angeordnet ist

- `function EdgeClass.getFrom: From`

  Diese Funktion liefert das From-Element der Kantenklasse

- `function EdgeClass.getFromClass: GraphElementClass`

  Diese Funktion liefert die Klasse des From-Elements der Kantenklasse

- `function EdgeClass.getTo: To`

  Diese Funktion liefert das To-Element der Kantenklasse

- `function EdgeClass.getToClass: GraphElementClass`

  Diese Funktion liefert die Klasse des To-Elements der Kantenklasse

- `function EdgeClass.isDirected: Boolean`

  Diese Funktion sagt aus, ob die Kantenklasse gerichtet ist

- `function EnumDomainVal.getValue: String`

  Diese Funktion liefert das From-Element der Kantenklasse

- `function Graph.getEdgeMode: TEdgeMode`

  Diese Funktion liefert den Richtungsmodus für alle Kanten und Relationen eines Graphen

- `function Graph.getAnyRelation: Relation`

  Diese Funktion liefert eine beliebige Relation, die zu diesem Graphen gehört

- `function Graph.hasEdgeIds: Boolean`

  Diese Funktion liefert den Wert des Graphattributs *edgeids*

- `function Graph.hasRelation: Boolean`

  Diese Funktion sagt aus, ob der Graph eine Relation besitzt oder nicht

- `function Graph.isHyperGraph: Boolean`

  Diese Funktion liefert den Wert des Graphattributs *hypergraph*

- `function GraphClass.getClassOf`
  `(ge: GraphElement): GraphElementClass`

  Diese Funktion liefert die Klasse des Graphelements `ge`

- `function GraphClass.getClassWithId`
  `(id: String): GraphElementClass`

  Diese Funktion liefert die Klasse mit der Id `id`

- `function GraphClass.getClassOf`
  `(re: RelationEnd): RelationEndClass`

  Diese Funktion liefert die Klasse des Relationsendes `re te`

- `function GraphClass.getNumberOfGraphElementClasses: Integer`

  Diese Funktion liefert die Anzahl der Graphelementklassen, die zu der Graphklasse gehören

- `function GraphClass.knowsGraphElementClass`
  `(gec: GraphElementClass): Boolean`

  Diese Funktion sagt aus, ob der Graphklasse die Graphelementklasse `gec` bekannt ist

- `function GraphElementClass.getType: ClassType`

  Diese Funktion liefert den Typ der Graphelementklasse (z.B. NodeClass, EdgeClass)

- `function GraphElementClass.isAbstract: Boolean`

  Diese Funktion sagt aus, ob die Graphelementklasse abstrakt ist

- `function GraphElementClass.isSubClassOf`
  `(gec: GraphElementClass): Boolean`

  Diese Funktion sagt aus, ob die Graphelementklasse Unterklasse von `gec` ist

- `function HasAsComponentGraph.getComponentGraph: GraphClass`

  Diese Funktion liefert die eingebette Graphklasse

- `function HasAsComponentGraph.getLowerLimit: Integer`

  Diese Funktion liefert die untere Kardinalitätsgrenze

- `function HasAsComponentGraph.getUpperLimit: Integer`

  Diese Funktion liefert die obere Kardinalitätsgrenze

- `function RelatesTo.getLowerLimit: Integer`

  Diese Funktion liefert die untere Kardinalitätsgrenze

- `function RelatesTo.getUpperLimit: Integer`

  Diese Funktion liefert die obere Kardinalitätsgrenze

- `function RelationEnd.getDirection: TDirection`

  Diese Funktion liefert die Richtung des Relationsendes *(in/out/none)*

- `function RelationEnd.getEndOrder: Integer`

  Diese Funktion liefert die Ordnungszahl des Relationsendes am Zielelement

- `function RelationEnd.getTarget: GraphElement`

  Diese Funktion liefert das Zielelement des Relationsendes

- `function RelationEnd.getStartOrder: Integer`

  Diese Funktion liefert die Ordnungszahl des Relationsendes an der Relation

- `function RelationEnd.isDirected: Boolean`

  Diese Funktion sagt aus, ob das Relationsende gerichtet ist

- `function RelationEnd.isOrdered: Boolean`

  Diese Funktion sagt aus, ob das Relationsende angeordnet ist

- `function RelationClass.hasRelationEndClass (role: String): Boolean`

  Diese Funktion sagt aus, ob die Relationsklasse eine Relationsendeklasse mit der Rolle `role` besitzt

- `function RelationEndClass.getRelatesToClass: GraphElementClass`

  Diese Funktion liefert Klasse des Zielelementes der Relationsende-Klasse

- `function RelationEndClass.getRole: String`

  Diese Funktion liefert die Rolle der Relationsende-Klasse

- `procedure Set.add (e: Element)`

  Diese Methode fügt der Menge ein weiteres Element hinzu

- `procedure Set.clear`

  Diese Methode leert die Menge

- `function Set.extractAny: Element`

  Diese Funktion liefert ein beliebiges Element der Menge

- `function Set.hasElement (e: Element): Boolean`

  Diese Funktion sagt aus, ob das Element *element* Element der Menge ist

- `function Set.isEmpty: Boolean`

  Diese Funktion sagt aus, ob die Menge leer ist

A.2 Fehlerklassen

Bei der Überprüfung eines Graphen auf Korrektheit bzw. Zugehörigkeit zu einer Graphklasse können Fehler entdeckt werden. Diese Fehler werden repräsentiert durch geeignete Fehlerklassen. Ausgehend von einer Basisklasse (GXLError) für alle möglichen Fehler, werden diese durch Spezialisierung kategorisiert. Nachfolgend werden alle Fehlerklassen zusammen mit den Konstruktorparametern und einer kurzen Beschreibung aufgelistet. Neben diesen Fehlerklassen können weitere Fehler auftreten, die hiervon nicht abgedeckt werden. Dies bezieht sich auf Exceptions der GXLInstanceAPI bzw. GXLSchemaAPI. Diese Exceptions werden in einen GXLAPIError umgewandelt und ebenfalls protokolliert.

- `GXLAbstractClassError`
  Konstruktorparameter:
  - `ge: GXLGraphElement`
  - `gec: GXLGraphElementClass`

  Dieser Fehler tritt auf, wenn das Element `ge` Instanz der abstrakten Klasse `gec` ist.

- `GXLAPIError`
  Konstruktorparameter (1):
  - `e: GXLInstanceException`

  Konstruktorparameter (2):
  - `e: GXLSchemaException`

  Dieser Fehler tritt auf, wenn die GXLAPI bzw. GXLSchemaAPI eine Exception geworfen hat.

- `GXLAttributeClassCircleError`
  Konstruktorparameter (1):
  - `ac: GXLAttributeClass`
  - `masterId: GXLString`

  Dieser Fehler tritt auf, wenn die Attributklassenstruktur der Attributklasse `ac` mit der Attributklasse `masterId` eine zyklische Abhängigkeit bildet (d.h. eine Kreis schließt).

- `GXLBagSetSeqComponentDomainError`
  Konstruktorparameter:
  - `d: GXLDomain`

  Dieser Fehler tritt auf, wenn das Compositedomain `d` (welches kein Tupeldomain ist) nicht genau ein Componentdomain besitzt.

- `GXLBagSetSeqValError`
  Konstruktorparameter:
  - `ae: GXLAttributedElement`
  - `a: GXLAttribute`

  Dieser Fehler tritt auf, wenn die einzelnen Elemente des Attributs `a` nicht den gleichen Typ aufweisen, obwohl eine Menge, Multimenge oder Sequenz dies voraussetzen.

- `GXLCardinalityError`
  Konstruktorparameter (1):
  - `ge: GXLGraphElement`
  - `ec: GXLEdgeClass`

  - `lowerLimit,`
  - `upperLimit,`
  - `count: : Integer`

  Konstruktorparameter (2):

  - `ge: GXLGraphElement`
  - `rec: GXLRelationEndClass`
  - `lowerLimit`
  - `upperLimit`
  - `count: Integer`

  Konstruktorparameter (3):

  - `ge: GXLGraphElement`
  - `gc: GXLGraphClass`
  - `lowerLimit,`
  - `upperLimit,`
  - `count: Integer`

  Dieser Fehler tritt auf, wenn die tatsächliche Anzahl von Inzidenzen eines bestimmten Typs bezogen auf ein Graphelement die im Schema definierten Grenzen unter- oder überschreiten

- `GXLComponentDefaultValCircleError`
  Konstruktorparameter:

  - `cv: GXLCompositeDefaultVal`
  - `d: GXLDefaultVal`

  Dieser Fehler tritt auf, wenn die Struktur des zusammengesetzten Default-Vals `cv` mit `d` eine zyklische Abhängigkeit bildet (d.h. einen Kreis schließt)

- `GXLComponentDomainCircleError`
  Konstruktorparameter:

  - `cd: GXLCompositeDomain`
  - `d: GXLDomain`

Dieser Fehler tritt auf, wenn die Domainstruktur der zusammengesetzten Domain `cd` mit `d` eine zyklische Abhängigkeit bildet (d.h. einen Kreis schließt)

- `GXLCyclicIsAStructureError`
  Konstruktorparameter:
  - `cycle: TStringList`

  Dieser Fehler tritt auf, wenn innerhalb der Generalisierungsstruktur ein Zirkel existiert.

- `GXLDefaultAtomicValError`
  Konstruktorparameter:
  - `av: GXLDefaultAtomicVal`

  Dieser Fehler tritt auf, wenn die Belegung des atomaren DefaultVal `av` nicht zu dessen Typ passt

- `GXLDefaultBagSetSeqValError`
  Konstruktorparameter:
  - `cv: GXLDefaultCompositeVal`

  Dieser Fehler tritt auf, wenn die Elemente des zusammengesetzten DefaultVal `cv` nicht zueinander passen. `cv` kann ein Bag-, Set- oder SeqVal sein, aber kein TupVal

- `GXLDefaulttupValError`
  Konstruktorparameter:
  - `cv: GXLDefaultCompositeVal`

  Dieser Fehler tritt auf, wenn der Tupel-DefaultVal `cv` nicht mindestens zwei Elemente besitzt

- `GXLDuplicateAttributeNameError`
  Konstruktorparameter:
  - `ae: GXLAttributedElement`
  - `a: GXLAttribute`

  Dieser Fehler tritt auf, wenn das Element `ae` bereits ein Attribut mit dem gleichen Namen wie `a` besitzt

- `GXLDuplicateAttributeClassNameError`
  Konstruktorparameter:
  - `aec: GXLAttributedElementClass`
  - `ac: GXLAttributeClass`

  Dieser Fehler tritt auf, wenn die Klasse `aec` bereits eine Attributklasse mit dem gleichen Namen wie `ac` besitzt.

- `GXLDuplicateElementIdentError`
  Konstruktorparameter:
  - `gec: GXLGraphElementClass`

  Dieser Fehler tritt auf, wenn der Bezeichner von `gec` nicht eindeutig ist.

- `GXLDuplicateOrderError`
  Konstruktorparameter:
  - `ge: GXLGraphElement`
  - `order: Integer`

  Dieser Fehler tritt auf, wenn bezogen auf ein Element die Ordnungszahl `order` für eine Inzidenz mehrfach vergeben wurde.

- `GXLDuplicateRoleError`
  Konstruktorparameter:
  - `rc: GXLRelationClass`
  - `rec: GXLRelationEndClass`

  Dieser Fehler tritt auf, wenn die Relationsklasse `rc` bereits eine Relationsende-Klasse mit der gleichen Rolle wie `rec` besitzt.

- `GXLEdgeClassSuperLimitError`
  Konstruktorparameter:
  - `ec: GXLEdgeClass`
  - `superEC: GXLEdgeClass`
  - `ecLower,`
  - `ecUpper,`
  - `superLower,`

  - `superUpper: Integer:`
  - `ecEndType: String`

  Dieser Fehler tritt auf, wenn die Limits der Kantenklasse `ec` nicht mit denen der Oberklasse `superEC` identisch sind bzw. diese weiter einschränken. Denn nur weiter einschränkende Limits sind bei Spezialisierung einer Kantenklasse erlaubt.

- `GXLEdgeDirectionError`
  Konstruktorparameter (1):

  - `e: GXLEdge`

  Dieser Fehler tritt auf, wenn die Kante `e` das Graphattribut *edgemode* verletzt.
  Konstruktorparameter (2):

  - `e: GXLEdge`
  - `ec: GXLEdgeClass`

  Dieser Fehler tritt auf, wenn die Kante `e` vom Typ `ec` die im Schema definierte Kantenrichtung nicht einhält.

- `GXLEdgeIdError`
  Konstruktorparameter:

  - `e: GXLEdge`

  Dieser Fehler tritt auf, wenn die Kante `e` das Graphattribut *edgeid* verletzt.

- `GXLGraphClassNotFoundError`
  Konstruktorparameter:

  - `gcId: GXLString`

  Dieser Fehler tritt auf, wenn eine angegebene Graphklasse mit Id `gcId` nicht gefunden werden konnte

- `GXLHypergraphError`
  Konstruktorparameter:

  - `r: GXLRelation`

Dieser Fehler tritt auf, wenn die Relation `r` das Graphattribut *hypergraph* verletzt. Dies geschieht dann, wenn das Graphattribut *hypergraph* besagt, dass der Graph keine Relationen enthält, aber `r` existiert.

- `GXLEdgeInheritanceError`
  Konstruktorparameter:
  - `ec: GXLEdgeClass`
  - `gec: GXLGraphElementClass`
  - `superGec: GXLGraphElementClass`
  - `ecEndType: String`

  Dieser Fehler tritt auf, wenn die Kantenklasse `ec` am Kantenende `ecEndType` eine aufgrund von Vererbung ungültige Klasse aufweist. Wird eine Klasse spezialisiert, so kann sie nur Klassen verbinden, die mit denen in der Ober-Kantenklasse definierten identisch oder Unterklassen dieser sind.

- `GXLLimitError`
  Konstruktorparameter (1):
  - `ec: GXLEdgeClass`
  - `ecEndType: String`
  - `lowerLimit,`
  - `upperLimit: Integer`

  Dieser Fehler tritt auf, wenn die Kantenklasse `ec` an dem Kantenende `ecEndType` ungültige Limit-Angaben besitzt. Dies ist dann der Fall, wenn die Untergrenze die Obergrenze überschreitet.
  Konstruktorparameter (2):
  - `ac: GXLAggregationClass`
  - `ecEndType: String`
  - `lowerLimit,`
  - `upperLimit: Integer`

  Dieser Fehler tritt auf, wenn die Aggregationsklasse `ac` an dem Kantenende `ecEndType` ungültige Limit-Angaben besitzt. Dies ist dann der Fall, wenn das die Limits für das Aggregat genau 1 entsprechen.
  Konstruktorparameter (3):

  - `cc: GXLCompositionClass`
  - `ecEndType: String`
  - `lowerLimit,`
  - `upperLimit: Integer`

  Dieser Fehler tritt auf, wenn die Kompositionsklasse `cc` an dem Kantenende `ecEndType` ungültige Limit-Angaben besitzt. Dies ist dann der Fall, wenn das die Limits für das Kompositum nicht genau 1 entsprechen.
  Konstruktorparameter (4):

  - `rec: GXLRelationEndClass`
  - `lowerLimit,`
  - `upperLimit: Integer`

  Dieser Fehler tritt auf, wenn die Relationsende-Klasse `rec` aungültige Limit-Angaben besitzt. Dies ist dann der Fall, wenn die Untergrenze die Obergrenze überschreitet.

- `GXLMissingAttributeError`
  Konstruktorparameter:

  - `ae: GXLAttributedElement`
  - `aec: GXLAttributedElementClass`
  - `ac: GXLAttributeClass`

  Dieser Fehler tritt auf, wenn das Element `aec` kein zu der Attributklasse `ac` passendes Attribut besitzt, obwohl dies im Schema definiert worden ist. Dabei ist `ae` Instanz der Klasse `aec`.

- `GXLRelEndDirectionError`
  Konstruktorparameter (1):

  - `r: GXLRelation`
  - `re: GXLRelEnd`

  Dieser Fehler tritt auf, wenn das Relationsende `re` der Relation `r` das Graphattribut *edgemode* verletzt.
  Konstruktorparameter (2):

  - `r: GXLRelation`

  - `re: GXLRelEnd`
  - `rec: GXLRelationEndClass`

  Dieser Fehler tritt auf, wenn das Relationsende `re` der Relation `r` nicht die in der Relationsklasse `rec` definierte Richtung einhält.

- `GXLSuperClassOwnershipError`
  Konstruktorparameter:

  - `gec: GXLGraphElementClass`
  - `super: GXLGraphElementClass`
  - `gc: GXLGraphClass`

  Dieser Fehler tritt auf, wenn die Oberklasse `super` der Klasse `gec` nicht zu der Graphklasse gehört, in der `gec` definiert ist.

- `GXLTupLowComponentDomainError`
  Konstruktorparameter:

  - `d: GXLDomain`

  Dieser Fehler tritt auf, wenn das Tupeldomain `d` nicht über mindestens zwei Elemente verfügt.

- `GXLUndefinedAttributeClassError`
  Konstruktorparameter:

  - `ae: GXLAttributedElement`
  - `a: GXLAttribute`

  Dieser Fehler tritt auf, wenn die angegebene Attributklasse des Attributs `a` im Schema nicht definiert ist.

- `GXLUndefinedClassError`
  Konstruktorparameter (1):

  - `n: GXLNode`

  Konstruktorparameter (2):

  - `e: GXLEdge`

  Konstruktorparameter (3):

  - `r: GXLRelation`

Konstruktorparameter (4):

  - `g: GXLGraph`

Dieser Fehler tritt auf, wenn der Typ des übergebenen Elementes im Schema nicht bekannt ist.

- `GXLUndefinedDefaultEnumValError`
  Konstruktorparameter:
  - `aec: GXLAttributedElementClass`
  - `ac: GXLAttributeClass`
  - `d: GXLDomain`

  Dieser Fehler tritt auf, wenn der Default-Enum-Wert der Attributklasse `ac` nicht in der zugehörigen Domain `d` definiert ist.

- `GXLUndefinedEnumValError`
  Konstruktorparameter:
  - `ae: GXLAttributedElement`
  - `a: GXLAttribute`
  - `d: GXLDomain`

  Dieser Fehler tritt auf, wenn der Enum-Wert des Attributs `a` nicht in der zugehörigen Domain `d` definiert ist.

- `GXLUndefinedRelationEndClassError`
  Konstruktorparameter:
  - `re: GXLRelEnd`
  - `r: GXLRelation`

  Dieser Fehler tritt auf, wenn die Relation `r` ein Relationsende `re` besitzt, dass im Schema nicht definiert ist.

- `GXLValueError`
  Konstruktorparameter:
  - `ae: GXLAttributedElement`
  - `a: GXLAttribute`

  Dieser Fehler tritt auf, wenn der Wert des Attributs `a` nicht zu seinem Typ passt. Dabei ist `a` ein Attribut des Elementes `ae`

- `GXLValueDomainError`
  Konstruktorparameter:
  - `ae: GXLAttributedElement`
  - `a: GXLAttribute`
  - `v: GXLValue`
  - `aec: GXLAttributedElementClass`
  - `ac: GXLAttributeClass`
  - `d: GXLDomain`

  Dieser Fehler tritt auf, wenn der Wert `value` des Attributs `a` nicht zu dem im Schema definierten Domain `d` passt.

- `GXLWrongEdgeFromElementError`
  Konstruktorparameter (1):
  - `e: GXLEdge`

  Konstruktorparameter (2):
  - `e: GXLEdge`
  - `ec: GXLEdgeClass`

  Dieser Fehler tritt auf, wenn das From-Element der Kante `e` kein Graphelement oder nicht Instanz der im Schema definierten Klasse (oder einer Unterklasse dieser) ist.

- `GXLWrongEdgeToElementError`
  Konstruktorparameter (1):
  - `e: GXLEdge`

  Konstruktorparameter (2):
  - `e: GXLEdge`
  - `ec: GXLEdgeClass`

  Dieser Fehler tritt auf, wenn das To-Element der Kante `e` kein Graphelement oder nicht Instanz der im Schema definierten Klasse (oder einer Unterklasse dieser) ist.

- `GXLWrongRelEndTargetElementError`
  Konstruktorparameter (1):

  - `re: GXLRelEnd`

  Konstruktorparameter (2):

  - `r: GXLRelation`
  - `re: GXLRelEnd`
  - `rec: GXLRelationEndClass`

  Dieser Fehler tritt auf, wenn das Zielelement des Relationsendes `re` kein Graphelement oder nicht Instanz der im Schema definierten Klasse (oder einer Unterklasse dieser) ist.

- `GXLWrongSuperClassError`
  Konstruktorparameter:

  - `gec: GXLGraphElementClass`
  - `superClass: GXLGraphElementClass`

  Dieser Fehler tritt auf, wenn die Oberklasse `superClass` von `gec` nicht zur gleichen Art gehört wie `gec`. So können Knoten nur von Knoten, Kanten nur von Kanten und Relationen nur von Relationen erben.

A.3 Zuordnung von Anforderungen zu Testklassen

Die folgende Tabelle listet die Zuordnung von Anforderungen aus Kapitel 3 zu der Implementation in einer Testklasse auf.

Anforderung	Testklasse
Korrektheit der Graphattribute	GXLInstanceGraphAttributeTest
Korrektheit der Attribute	GXLInstanceAttributeTest
Einhaltung der Ordnung	GXLInstanceOrderTest
Korrektheit der Kanten-Endelemente	API-Funktionalität
Korrektheit der Relationsende-Zielelemente	API-Funktionalität
Klassenzugehörigkeit	GXLInstanceVsSchemaClassTest
Korrektheit der Kanten	GXLInstanceVsSchemaEdgeTest
Korrektheit der Relationen	GXLInstanceVsSchemaRelationTest
Korrektheit der Attribute	GXLInstanceVsSchemaAttributeTest
Korrektheit der eingebetteten Graphen	GXLInstanceVsSchemaComponentGraphTest
Existenz der Graphklasse	vgl. GXLDocumentChecker
Azyklische Generalisierung	GXLSchemaGeneralizationTest
Typsichere Generalisierung	GXLSchemaGeneralizationTest
Besitz aller Oberklassen	GXLSchemaGeneralizationTest
Eindeutige Elementbezeichner	GXLSchemaElementIdentTest
Korrektheit der Wertebereiche	GXLSchemaDomainTest
Korrektheit der Standardwerte	GXLSchemaDefaultValTest
Korrektheit der Attributklassen	GXLSchemaAttributeTest
Korrektheit der Kantenklassen	GXLSchemaEdgeClassTest
Korrektheit der Relationenklassen	GXLSchemaRelationClassTest

A.4 GXLValidator Optionen

Die folgende Tabelle listet alle Optionen des GXLValidators auf.

Kurzform	Langform	Beschreibung
-h	–help	Anzeige aller verfügbaren Optionen
-d	–document	Angabe des GXL-Dokumentes, das untersucht werden soll
-writeCfg		Erstellt die Datei tests.cfg, die eine Auflistung aller Testklassen enthält
-readCfg		Liest die Datei tests.cfg. Nur Tests, die in dieser Datei aufgelistet sind, werden ausgeführt. Nicht bekannte Testnamen werden ignoriert
-i	–instanceCheck	Die Tests werden auf Instanztests beschränkt.
-ssc	–skipSchemaCheck	Angegebene Schemata werden nicht getestet. Schemakonformitätstests werden aber durchgeführt. Diese Option sollte nur dann verwendet werden, wenn sichergestellt ist, dass die angegebenen Schemata korrekt sind

A.5 GXL-DTD

Die jeweils aktuelle DTD von GXL-1.0 ist zu finden unter

`http://www.gupro.de/GXL/gxl-1.0.dtd`

Abbildung A.1 zeigt die zum Zeitpunkt der Veröffentlichung dieser Arbeit gültige GXL-DTD. Die Kommentare wurden leicht modifiziert, damit die Abbildung auf eine Seite passt.

```
< !– Extensions –>
< !ENTITY % gxl-extension “” >
< !ENTITY % graph-extension “” >
< !ENTITY % node-extension “” >
< !ENTITY % edge-extension “” >
< !ENTITY % rel-extension “” >
< !ENTITY % value-extension “” >
< !ENTITY % relend-extension “” >
< !ENTITY % gxl-attr-extension “” >
< !ENTITY % graph-attr-extension “” >
< !ENTITY % node-attr-extension “” >
< !ENTITY % edge-attr-extension “” >
< !ENTITY % rel-attr-extension “” >
< !ENTITY % relend-attr-extension “” >

< !– attribute values –>
< !ENTITY % val "locator | bool | int | float | string |
                 enum | seq | set | bag | tup
                 % value-extension;">

< !– gxl –>
< !ELEMENT gxl (graph* %gxl-extension;) >
< !ATTLIST gxl
    xmlns:xlink CDATA          #FIXED
                               "www.w3.org/1999/xlink"
    %gxl-attr-extension; >

< !– type –>
< !ELEMENT type EMPTY>
< !ATTLIST type
    xlink:type  (simple)       #FIXED “simple”
    xlink:href  CDATA          #REQUIRED >

< !– graph –>
< !ELEMENT graph (type? , attr* ,
                  ( node | edge | rel )*
                  %graph-extension;) >
< !ATTLIST graph
    id          ID             #REQUIRED
    role        NMTOKEN        #IMPLIED
    edgeids     ( true | false )  “false”
    hypergraph  ( true | false )  “false”
    edgemode    ( directed | undirected |
                  defaultdirected | defaultundirected)
                  "directed"
    %graph-attr-extension; >

< !– node –>
< !ELEMENT node (type? , attr*, graph*
                  %node-extension;) >
< !ATTLIST node
    id          ID             #REQUIRED
    %node-attr-extension; >

< !– edge –>
< !ELEMENT edge (type?, attr*, graph*
                  %edge-extension;) >
< !ATTLIST edge
    id          ID             #IMPLIED
    from        IDREF          #REQUIRED
    to          IDREF          #REQUIRED
    fromorder   CDATA          #IMPLIED
    toorder     CDATA          #IMPLIED
    isdirected  ( true | false )  #IMPLIED
    %edge-attr-extension; >

< !– rel –>
< !ELEMENT rel (type? , attr*, graph*, relend*
                  %rel-extension;) >
< !ATTLIST rel
    id          ID             #IMPLIED
    isdirected  ( true | false )  #IMPLIED
    %rel-attr-extension; >

< !– relend –>
< !ELEMENT relend (attr* %relend-extension;) >
< !ATTLIST relend
    target      IDREF          #REQUIRED
    role        NMTOKEN        #IMPLIED
    direction   ( in | out | none)  #IMPLIED
    startorder  CDATA          #IMPLIED
    endorder    CDATA          #IMPLIED
    %relend-attr-extension; >

< !– attr –>
< !ELEMENT attr (type?, attr*, (%val;)) >
< !ATTLIST attr
    id          IDREF          #IMPLIED
    name        NMTOKEN        #REQUIRED
    kind        NMTOKEN        #IMPLIED >

< !– locator –>
< !ELEMENT locator EMPTY >
< !ATTLIST locator
    xlink:type  (simple)       #FIXED “simple”
    xlink:href  CDATA          #IMPLIED >

< !– attribute values –>
< !ELEMENT bool (#PCDATA) >
< !ELEMENT int (#PCDATA) >
< !ELEMENT float (#PCDATA) >
< !ELEMENT string (#PCDATA) >
< !ELEMENT enum (#PCDATA) >
< !ELEMENT seq (%val;)* >
< !ELEMENT set (%val;)* >
< !ELEMENT bag (%val;)* >
< !ELEMENT tup (%val;)* >
```

Abbildung A.1: GXL Document Type Definition (DTD)

A.6 GXL-Graphmodel und -Metaschema

Die folgenden Abbildungen zeigen das GXL-Graphmodell und das GXL-Metaschema.

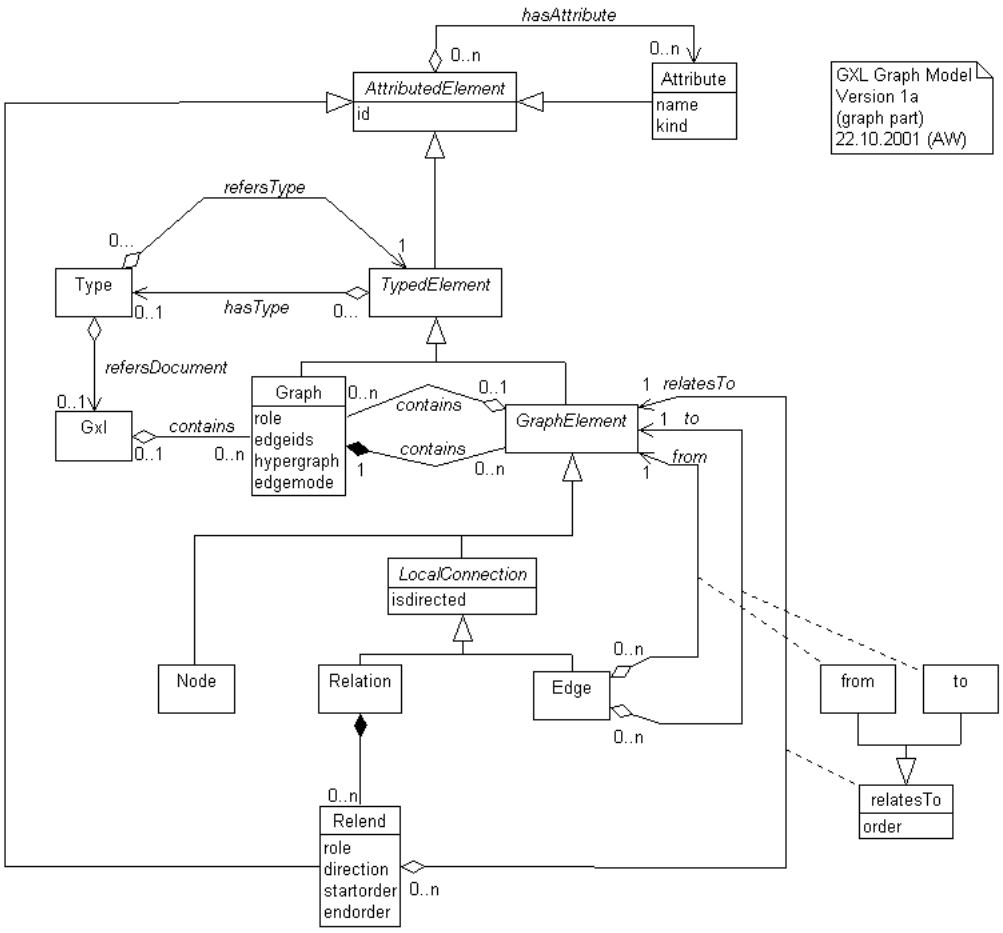

Abbildung A.2: Das GXL-Graphmodell (graphpart) (Quelle: [3])

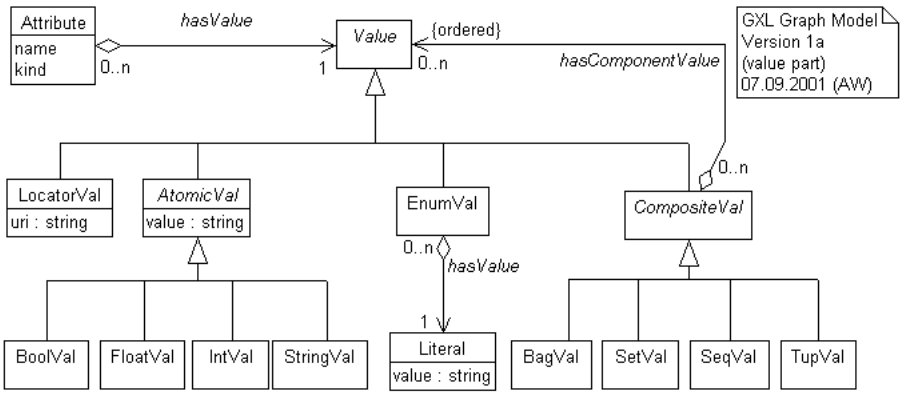

Abbildung A.3: Das GXL-Graphmodell (valuepart)(Quelle: [3])

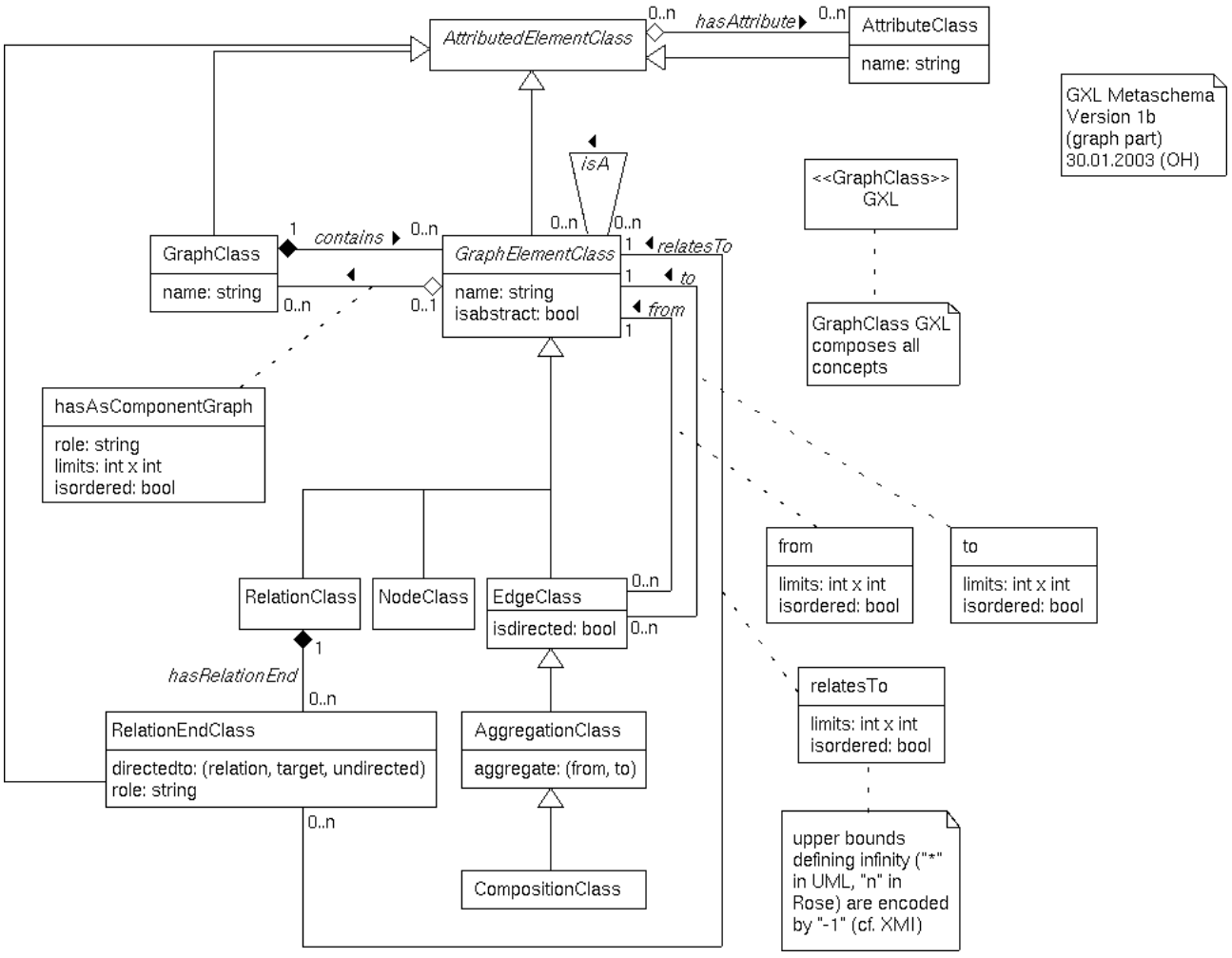

Abbildung A.4: Das GXL-Metaschema (graphpart) (Quelle: [3])

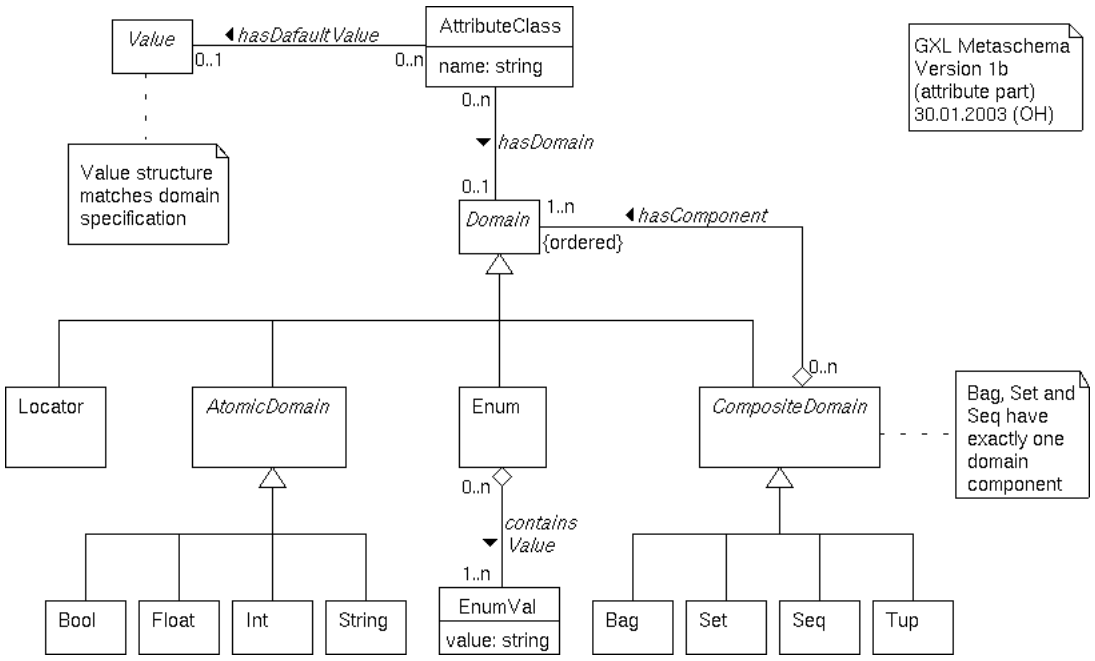

Abbildung A.5: Das GXL-Metaschema (attributepart) (Quelle: [3])

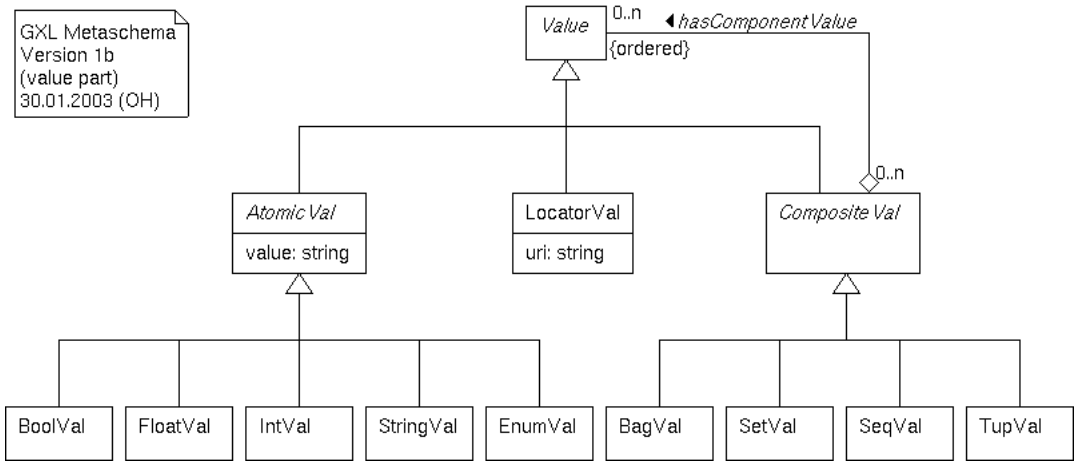

Abbildung A.6: Das GXL-Metaschema (valuepart) (Quelle: [3])

Literaturverzeichnis

[1] Winter, A., Kullbach, B., Riediger, V.
An overview of the GXL Graph Exchange Language
Springer Verlag: S. Diehl (ed.) Software Visualization, International Seminar Dagstuhl Castle, Germany, May 20-25, 2001.

[2] Heinen, O.
The XML techniques and their effects on GXL 1.1
Studienarbeit, Universität Koblenz-Landau,
Institut für Softwaretechnik, Koblenz, 2003

[3] Holt, R., Schürr, A., Elliott, S., Winter, A.
GXL-Website
http://www.gupro.de/GXL, (3. September 2003)

[4] Zavgorodnya, D., Winter, A., Riediger, V.
GXL Instance API
Projektbericht 2/03, Universität Koblenz-Landau,
Institut für Softwaretechnik, Koblenz, 2003

[5] Hirschmann, K., Winter, A., Riediger, V.
GXL Schema API
Projektbericht 2/03, Universität Koblenz-Landau,
Institut für Softwaretechnik, Koblenz, 2003

[6] Ebert, J.
Effiziente Graphenalgorithmen
Akademische Verlagsgesellschaft, Wiesbaden, 1981