Institut für Softwaretechnologie
Abteilung Programmiersprachen und Übersetzerbau
Universität Stuttgart
Universitätsstraße 38
D–70569 Stuttgart

Diplomarbeit Nr. 3357

Kombinierung von Programmiermodellen in Bauhaus IML

Sven Hendrik Glück

Studiengang:	Softwaretechnik
Prüfer:	Prof. Dr. rer. nat. /Harvard Univ. Erhard Plödereder
Betreuer:	Dipl.-Inf. Torsten Görg
begonnen am:	01. August 2012
beendet am:	30. Januar 2013
CR-Klassifikation:	D.3.4, D.3.2

Kurzfassung

In dieser Diplomarbeit wird eine Technik vorgestellt, die Spezifikation von IML dynamisch durch neue Konstrukte zu erweitern. Dabei wird eine Transformationssprache entwickelt, die es erlaubt Regeln zu formulieren anhand derer IML-Graphen in andere IML-Graphen überführt werden. Ein Beispiel-Sprachpaket für die Abbildung von Zustandsautomaten in IML wird erstellt. Diese werden mit dem UML-Werkzeug Papyrus für die Entwicklungsumgebung Eclipse entworfen und durch ein Frontend nach IML überführt. Abschließend wird ein Generator implementiert, welcher anhand der transformierten Zustandsautomaten Java-Code erzeugt. Der Ausbau zu einer "Language Workbench" erlaubt es, Bauhaus und IML mit geringem Aufwand durch neue Konstrukte und Funktionalitäten zu erweitern.

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Verzeichnis der Algorithmen

1 Grundlagen

1.1 Bauhaus

Das Projekt Bauhaus läuft seit 1996 als zentrales Forschungsvorhaben der Abteilung Programmiersprachen und Übersetzerbau des Instituts für Softwaretechnologie an der Universität Stuttgart. Ziel des Projekts ist die Entwicklung von Methoden und Werkzeugen zur Unterstützung des Programmverstehens bei der Software-Wartung. Hierzu wurde eine umfangreiche Infrastruktur geschaffen, die es erlaubt Programmanalysen auf hohen und niedrigen Abstraktionsebenen durchzuführen. Die programmiersprachenunabhängige Zwischendarstellung IML (Intermediate Language) dient als Grundlage für diverse statische Programmanalysen und enthält unter anderem den abstrakten Syntaxbaum. Zur Übersetzung von Quellprogrammen nach IML existieren bereits funktionstüchtige Frontends für die imperativen Sprachen C, C++, Ada und Java. Neben dem imperativen Paradigma gibt es zahlreiche weitere Ausführungsmodelle. Aktuelle Forschungsarbeiten untersuchen die Kombinierbarkeit der verschiedenen Ausführungsmodelle.

1.2 Aufgabenstellung

Im Rahmen dieser Diplomarbeit soll das Programmanalysen-Framework Bauhaus zu einer Language Workbench ausgebaut werden, indem es um die Möglichkeit erweitert wird, der Zwischensprache IML zusätzliche Pakete von Sprachkonstrukten hinzufügen zu können. Dieser Mechanismus soll exemplarisch angewendet werden, um ein Sprachpaket für hierarchische Zustandsautomaten in IML zu integrieren. Das Zustandsautomaten-Sprachpaket hat zum Zweck, das imperative Ausführungsmodell mit dem Ausführungsmodell von Zustandsautomaten zu koppeln. Dieses wird vom Betreuer vorbereitet und zur Verfügung gestellt. Im einzelnen sind folgende Schritte durchzuführen:

- Der bereits existierende IML-Generator ist um die Fähigkeit zu erweitern, mehrere Sprachpakete als Eingabe zu verarbeiten.
- Zur Erzeugung von IML-Repräsentationen für Zustandsautomaten ist ein Frontend zu implementieren, das UML2-Zustandsmodelle in IML übersetzt. Dabei soll davon ausgegangen werden, dass die UML2-Modelle mit dem Eclipse-UML2-Editor erstellt wurden.

- Die Semantik der Sprachkonstrukte zusätzlicher Sprachpakete soll translatorisch durch Ersetzungsmuster definierbar sein, die diese Konstrukte auf Basis-IML-Sprachkonstrukte abbilden. Für die Ersetzungsmuster ist eine Beschreibungssprache zu entwerfen.
- Es ist ein Generator zu realisieren, der auf Basis der Ersetzungsmuster-Beschreibungen Ada-Code zum automatischen Expandieren hinzugefügter Sprachkonstrukte erzeugt.
- Um die Kombination von Zustandsautomaten mit imperativem Code zu testen, soll ein Generator realisiert werden, der die für einen Test notwendigen IML-Konstrukte in Java-Quelltext transformiert.

Da die Implementierung innerhalb des Projekts Bauhaus weiter verwendet und gewartet werden soll, wird besonders auf gute Codequalität und Quelltext-Dokumentation Wert gelegt.

1.3 Intermediate Language (IML)

IML ist die abstrakte Zwischendarstellung, welche als gemeinsame Basis für die Werkzeuge des Bauhaus-Frameworks fungiert [Kna02, Pin06]. Sie erlaubt es, Bestandteile von Programmen abzubilden. Hierfür unterstützt IML verschiedene Programmiersprachen. Ursprünglich wurde IML entworfen, um Programme, welche in Ada83 geschrieben wurden, abzubilden. Dabei stützt sie sich auf die Konzeption in [Eis99]. IML befindet sich im stetigen Wachstum und wurde bereits durch mehrere (Diplom-)Arbeiten erweitert und mit neuen Konzepten versehen. Beispiele hierfür sind:

- **Erweiterung für C:** Die erste Arbeit, die eine weitere Sprache für IML entwickelte war [Roh98]. Dabei wurden Konstrukte sowie ein Frontend für die Unterstützung von C hinzugefügt.
- **Erweiterung für Java:** Diese entstand während der Diplomarbeit von Markus Knauss. Durch die reine Objektorientierung von Java wurden neue Konzepte benötigt, welche unter anderem berücksichtigen, dass auch Enumerationen und Arrays Objekte sind. Sie finden die Arbeit in [Kna02].
- **Erweiterung für C++:** Diese Erweiterung wurde von Tahir Karaca und Sebastian Setzer in ihrer Abschlussarbeit entwickelt. Nachdem vorangegangene Arbeiten es ermöglicht hatten, C Programme abzubilden, ist es durch die Arbeit von Karaca und Setzer (s. [Set03]) nun auch möglich C++-Code durch IML darzustellen.

1.3.1 Beschreibung

Mit Hilfe von IML können Programme durch in einen abstrakten Graph repräsentiert werden [Keu05]. Dabei ist es möglich Meta-Informationen, z.B. für Analysen zu speichern [Kna02]. Knoten innerhalb der IML besitzen eine Klasse, die sich durch eine Vererbungshierarchie,

ähnlich der Objektorientierung, von einer gemeinsamen Basisklasse ableitet [Pino6]. Neben einer Klasse können Knoten mehrere Interfaces definieren.

Der Einsatz von Vererbung erlaubt es, Konstrukte beliebig tief zu verschachteln und gleichzeitig durch abstrakte Basisklassen Gemeinsamkeiten zu erfassen. Als Beispiel nennt [Pino6] die Abbildung von While-Schleifen in IML. Für diese existiert ein konkreter Knotentyp `While_Loop`. Durch die Ableitung von `Loop_Statement` kann sie aber zugleich als solches behandelt werden. Dies kann für Analysen oder Code-Generatoren nötig sein. IML ist demnach abstrakt und trotzdem nah am abgebildeten Quellcode [Pino6, Keu05].

Unterteilt werden können die Knotentypen in die folgenden Kategorien [Pino6]:

- **Hierarchische Einheiten:** Hierarchische Einheiten sind eine übergeordnete Kategorie von Knoten, welche strukturellen Beziehungen von Übersetzungseinheiten beschreiben [Pino6]. Beispiele hierfür wären Routinen (bzw. Klassen-Methoden), Strukturen (z.B. `struct` in C) oder Klassen. Basisknotenklasse für alle hierarchischen Elemente ist `Hierarchical_Unit`.
- **Statements und Expressions:** Knoten, welche den Ablauf eines Programms wiedergeben, leiten sich von der abstrakten Basisklasse `Value` ab. Als Beispiele seien hier Verzweigungen, Funktionsaufrufe und Schleifen genannt.
- **Symbole:** Basis für Symbole ist die Knotenklasse `Symbol_Node`. Innerhalb der Symbole kann eine weitere Kategorisierung vorgenommen werden [Pino6]. Dabei leiten sich Typen (u.a. Klassen und Interfaces) von der Basisklasse `T_Node` ab, während Deklarationen von `O_Node` erben. Beispiele für Deklarationen sind die Aufrufparameter von Routinen oder Variablen.
- **Sonstige:** In diese Kategorie fallen alle Knotentypen, die sich in keiner der oben genannten Themengruppen wiederfinden. Dazu gehören unter anderem Knoten, welche Informationen von Analyse-Verfahren speichern. Beispiele für Elemente dieser Kategorie sind `Basic_Block` und die Abbildung von Vererbungsbeziehungen durch `Extends_Relation`.

Die Eigenschaften von Knoten werden durch Attribute definiert. Diese können unterschiedliche Typen haben und sind in allen Unterklassen sichtbar. Es ist allerdings möglich, den Typ eines Attributs in einer Unterklasse zu überschreiben. Dieser muss allerdings kompatibel, d.h. gleich oder vom ursprünglichen Typ abgeleitet sein. Neben vordefinierten Typen, sogenannten *Builtins*, können auch Knoten als Attribut spezifiziert werden.

Attribute, deren Typ eine Knotenklasse ist, spannen den Graph auf und bilden die (gerichteten) Kanten [Keu05].

Neben dem Typ können die Attribute in zwei weitere Kategorien unterteilt werden [Pino6, Knao2]. Diese ähneln der Unterscheidung zwischen Aggregationen und Komposition in der Unified Modeling Language (UML) [OMG11].

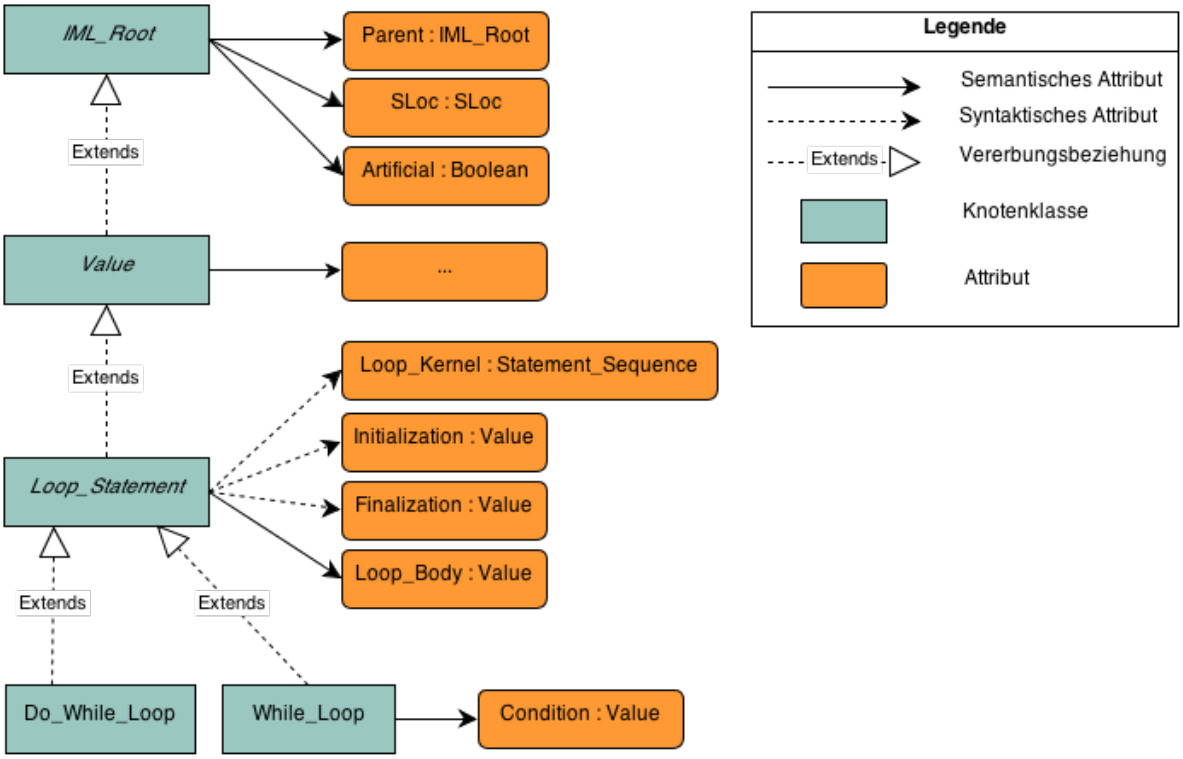

Abbildung 1.1: Ausschnitt aus der Vererbungshierarchie von IML.

- **Syntaktische Eigenschaften:** Syntaktische Attribute verhalten sich wie Aggregations-Beziehungen in UML. Die Elemente auf die sie verweisen, existieren außerhalb des attributierten Knotens. Sie sind also nicht Bestandteil des Knotens und können auch beispielsweise bestehen bleiben, wenn der Knoten entfernt wird. Syntaktische Kanten eines IML-Graphen repräsentieren die ausführbaren Elemente eines Programms und bilden einen erweiterten abstrakten Syntaxbaum [Pino6, Knao2, Keuo5].
- **Semantische Eigenschaften:** Im Gegensatz zu syntaktischen Attributen sind semantische Eigenschaften fest mit dem Knoten, in dem sie definiert sind, verbunden. Dadurch können zusätzliche knotenspezifische Informationen gespeichert werden [Knao2]. Dementsprechend verhalten sie sich wie entsprechende Kompositionen in UML. Ein typisches Beispiel für ein semantisches Attribut ist der Name eines Knotens oder das `Parent`-Attribut, welches auf den syntaktisches Vorgänger zeigt [Keuo5].

Im Schaubild 1.1 wird ein Ausschnitt aus der Hierarchie von IML dargestellt. Es zeigt, welche Abstraktionsebenen bei der Darstellung von (Do-)While-Schleifen verwendet werden und wie diese auf die Basisklasse `IML_Root` zurückführt. Zusätzlich werden die Attribute der Knoten, sowie ihre Ausprägung (semantisch oder syntaktisch) dargestellt. Die Attribute der Knotenklassen `Value` und `Do_While_Loop` wurden aus Gründen der Übersichtlichkeit weggelassen.

1.3.2 Spezifikationssprache

Für die Spezifikation der IML-Konstrukte wurde eine eigene Domain-Specific Language (s. Kapitel 2.1) entwickelt. Mit dieser können Builtins (z.B. primitive Datentypen wie `Integer`, `Double` oder auch `String`), abstrakte und konkrete Knotenklassen, Interfaces, sowie deren Attribute und Vererbungsbeziehungen in einer eigenen entsprechenden Syntax definiert werden [Pino6, Keuo5, Knao2].

Der Aufbau der Spezifikationsdatei gliedert sich in mehrere Bereiche. Begonnen wird mit einer Namensdefinition eingeleitet durch das Schlüsselwort "`name is`". Der Name wird dabei durch eine Revisionsnummer, sowie das Datum der letzten Manipulation gebildet. Im Anschluss an den Namen werden die Builtins definiert. Diese beginnen mit dem Bezeichner "`builtin`" und identifizieren Typen, deren Semantik durch den Generator definiert wird. Durch das Schlüsselwort "`root`" wird die Wurzel für alle Knoten festgelegt.

Den letzten Teil der Spezifikation bilden die Pakete. Diese besitzen einen Bezeichner und werden durch das Schlüsselwort "`package`" eingeleitet. Innerhalb der Pakete werden die Knotenklassen definiert. Die Spezifikation kann dabei aus mehreren Paketen bestehen. Das Listing in 1.1 zeigt, wie die Struktur der Spezifikationsdatei konkret aussieht.

```
1   name is "$Revision$ $Date$";
2
3   builtin String;
4   builtin Natural  is reflected c++ "int";
5   builtin Integer  is reflected c++ "int";
6   builtin Long_Float;
7   builtin Boolean  is reflected c++ "bool";
8   #...
9
10  root IML_ROOT;
11
12  package NEW_IML_BASENODES is
13      #*Klassen- und Interface-Definitionen aus der Basis-Spezifikation*
14  end package NEW_IML_BASENODES;
15
16  #Weitere Packages..
```

Listing 1.1: Aufbau der IML-Spezifikationsdatei

Mit Hilfe dieser Spezifikationsdatei ist es möglich, einen Großteil der Manipulationen an IML vorzunehmen, ohne dabei Werkzeuge verändern zu müssen. Eine der Ausnahmen ist die Einführung neuer Builtins. Diese erhalten ihre Logik durch eine entsprechende Implementierung in den Generatoren und benötigen daher nachträgliche Erweiterungen dieser.

Der Code-Ausschnitt 1.2 zeigt die Definition eines Knotens in der IML-Spezifikationssprache.

```
1  abstract class Throw_Statement
2  inherits Unconditional_Branch
3  # A 'throw' or 'raise' statement for C++, Java, Ada.
4  is
5     syntactic Expression : class Value;
6     # Expression that determines the type of the exception which
7     # is thrown. Null when no object is specified (a re-throw
8     # of the current throw object).
9  end class Throw_Statement;
```

Listing 1.2: Beispiel einer abstrakten Knotenspezifikation in IML.

Er repräsentiert eine Basisklasse zur Abbildung von Anweisung zum Auslösen einer `Exception`. In Java könnte eine solche Anweisung beispielsweise so aussehen:
`throw new IOException("File read error");`

Die Knotenklasse ist durch das Schlüsselwort `abstract` gekennzeichnet. Es können daher keine Instanzen von ihr erzeugt werden. Da sich `throw`-Anweisungen von Sprache zu Sprache unterscheiden, müssen konkrete Knotentypen von `Throw_Statement` abgeleitet werden. Der Code-Ausschnitt in 1.3 zeigt, wie dies für Java und C++ aussieht.

```
1  concrete class Cpp_Throw_Statement
2  inherits Throw_Statement
3  # A 'throw' statement that throws an exception object of the static type
4  # of 'Expression.Its_Type'.
5  is
6  end class Cpp_Throw_Statement;
7
8  concrete class Java_Throw_Statement
9  inherits Throw_Statement
10 # A 'throw' statement that throws an exception object of the
11 # dynamic type determined by evaluating 'Expression'. That type is
12 # identical to or (transitively) derived from
13 # 'Expression.Its_Type'.
14 is
15 end class Java_Throw_Statement;
```

Listing 1.3: Beispiel konkretisierender Knotenspezifikationen in IML.

Die IML-Spezifikation wird als Eingabeparameter an das Werkzeug *imlgen* übergeben. Dieses erzeugt anhand der darin definierten Knoten- und Builtin-Typen entsprechenden Ada95-Quellcode.

1.3.3 Erweiterbarkeit

Vor Beginn dieser Diplomarbeit war die Erweiterbarkeit von IML auf die Manipulation der monolithischen Basisspezifikation beschränkt. Mit Zielsetzung, der Bauhaus zu einer Language Workbench auszubauen und neue Sprach- und Programmkonzepte in IML darstellen zu können, ist es Aufgabe dieser Arbeit, die Erweiterbarkeit zu verbessern.

Hierfür wird ein neues Konzept zur modularen Erweiterung der IML-Spezifikation vorgestellt. Dieses erlaubt es, neue Pakete für IML bereitzustellen, ohne die Basis-Spezifikationsdatei zu verändern. Aktuell umfasst die Spezifikationsdatei mehr als 4000 Zeilen. Eine Aufteilung in themenbezogene Module verbessert die Wart- und Lesbarkeit.

Zu diesem Zweck wird in dieser Arbeit ein Konzept für die Modularisierung der Spezifikationsdatei entwickelt. In Kombination mit diesem wird es ermöglicht abstrakte Konzepte in eigene Sprachpakete auszulagern und diese mit einer Transformation-Beschreibung auf Basiskonstrukte abzubilden.

2 Language Workbench

Obwohl die Idee der Language Workbenches noch in den Kinderschuhen steckt, ist die ihnen zugrunde liegende Idee des Einsatzes von Domain-Specific Languages (DSL) [Fow10] nicht neu. Dieses Prinzip existiert bereits seit langer Zeit in der Unix-Umgebung in Form sogenannter "Little Languages" [Fow05]. Diese werden durch die Werkzeuge `yacc` und `lex` realisiert und generieren Code. Des Weiteren sind diese domänenspezifischen Sprachen ein weitverbreiteter Bestandteil der Programmiersprache Lisp.

Die Praxis konkretisierte, abstrakte Sprachen für den Einsatz in einem beschränkten Aufgabenbereich einzusetzen, wird als "Language-Oriented Programming" (LOP) bezeichnet. Im Laufe dieses Kapitels werden die Begriffe der Domain-Specific Languages und des Language-Oriented Programmings genauer erläutert und spezifiziert. Auf Basis dieser Definitionen wird abschließend die Funktionsweise von Language Workbenches beschrieben und charakterisiert.

2.1 Domain-Specific Languages

Charakteristisch für domänenspezifische Sprachen ist deren Beschränkung auf ein spezielles Aufgabengebiet. Dies kann dabei vielfältiger Natur sein. Tatsächlich existiert bereits ein große Menge solcher Sprachen, die zum Teil sehr weitverbreitet sind. Die prominentesten Beispiele sind dabei wohl SQL (Structured Query Language) und HTML (Hypertext Markup Language). Gerade letzteres Beispiel zeigt, dass sich DSLs stark von traditionellen Hochsprachen wie C++ oder Java unterscheiden können. Konkreter müssen DSLs nicht einmal den Anspruch der Turing-Vollständigkeit erfüllen.

In [Fow10] wird das Konzept der Domain-Specific Languages wie folgt definiert:

> Eine Computer-Programmiersprache mit eingeschränkter Ausdruckskraft, welche sich auf einen bestimmten Bereich konzentriert.

Als Programmiersprache müssen DSLs in einem für menschliche Benutzer verständlichen Syntax formuliert und trotzdem für den Computer auswert- und gegebenenfalls ausführbar sein [Fow10]. Während dieser Anspruch auch von sogenannten "General-Purpose Languages" (GPLs) wie C oder Java erfüllt wird, unterscheiden sich die beiden Konzepte doch signifikant. Während GPLs dem Entwickler ein Maximum an Freiheit bieten, konzentrieren sich DSLs auf ein spezifisches Aufgabengebiet [Fow10, Blu12]. An dieser Stelle drängt sich die Frage auf, welchem Zweck DSLs dienen, wenn die gleiche Aufgabe von den mächtigeren General

Purpose Languages übernommen werden könnte. Durch ihre Einschränkung kommen Domain-Specific Languages häufig mit einer deutlich einfacheren Syntax aus. Da GPLs mehr als einer Aufgabe gerecht werden müssen, definieren sie deutlich mächtigere, doch hier unnötige, Sprachkonstrukte. Eine Erfüllung der Anforderungen kann in einer DSL dementsprechend effizienter und einfacher sein [Fow10].

Domain-Specific Languages können in zwei Bereiche kategorisiert werden: Internal und External DSLs [Sto10, Fow05, Fow10].

2.1.1 Internal DSLs

Als Internal DSLs werden Sprachen bezeichnet, welche in den Kontext einer übergeordneten Sprache eingebettet sind. Diese wird als Host-Sprache bezeichnet [Sto10]. Internal DSLs können die Fähigkeiten der Host-Sprache für sich nutzen. Dies betrifft neben der Wiederverwendung von Sprachkonstrukten, im Idealfall, auch die Nutzung existenter Werkzeuge [Fow05, Fow10]. Funktionsfähiger Code der in einer Internal DSL geschrieben wurde, ist dabei auch immer korrekt im Sinne der Sprache in der er eingebettet ist.

Die Definition von Sprachen dieser Kategorie findet dabei häufiger in Skriptsprachen wie Ruby (in Verbindung mit Ruby On Rails), als in Hochsprachen wie C# oder Java statt [Fow05]. Dies basiert zum Teil auf dem Fehlen von Sprachelementen, welche die Definition von Internal DSLs begünstigen würden. Als Beispiel können hier Closures genannt werden. Dass es trotzdem möglich ist eine DSL in einer Sprache wie Java umzusetzen zeigt der Code-Ausschnitt in 2.1.

```
1  HTMLGenerator
2    .Get(Request.Content)
3    .Append("<p>Thanks for your purchase, %%NAME%%</p>");
4    .Append("You will be redirected shortly..");
5    .Bind("NAME", Request.Params["Name"])
6    .sendResponse();
```

Listing 2.1: Beispiel einer Internal DSL in Java.

Das hierbei verwendete Entwurfsmuster wird als *Method Chaining* bezeichnet [Fow10]. Dabei werden die Methoden eines Objekts so entworfen, dass sie den internen Zustand der Instanz verändern und diese dann als Ergebnis zurückgeben. Der Code-Ausschnitt in 2.2 zeigt, wie die Klassen- und Methodendefinition für den Beispiel-Aufruf in 2.1 aussehen könnte.

```
1  class HTMLGenerator {
2    private String htmlCode;
3
4    public HTMLGenerator Get(InputStream is) { ... }
5    public HTMLGenerator Append(String txt) { ... }
6    public HTMLGenerator Bind(String var, Object val) { ... }
```

```
7     public void sendResponse() { ... }
8
9   }
```

Listing 2.2: Beispiel für Klassen und Methoden in einer Internal DSL.

Während sich *Method Chaining* nur bei objektorientierten Sprachen einsetzen lässt, kann in prozeduralen Sprachen ein ähnlicher Effekt und Syntax durch *Function Sequences* erreicht werden [Fow10]. Bei der Verwendung von Internal DSLs mit *Method Chaining* formuliert der Anwender eine Reihe von nacheinander ausgeführten Methodenaufrufen. Charakteristisch für diesen Stil ist es, die Bezeichnungen für Funktionen mit Bezug auf den Kontext, in dem sie innerhalb dieser Reihen stehen können, zu wählen [Fow10].

Neben den Vorteilen existieren allerdings auch Nachteile. Obwohl die Wiederverwendbarkeit von Sprachkonstrukten durchaus einen Vorteil darstellt, so limitiert sie gleichzeitig die Syntax in Abhängigkeit der Basissprache. [Fow05].

Ein Punkt, in dem sich External und Internal DSLs unterscheiden, führt zu einem weiteren Nachteil: Da die interne Variante nur als Bestandteil einer übergeordneten Sprache funktioniert, muss der Anwender diese ebenfalls beherrschen. Dadurch müssen Benutzer der Sprache zusätzlich zur eigentlichen DSL noch Kenntnisse einer weiteren Programmiersprache besitzen [Fow05, Fow10]. Dies kann zu einer massiven Einschränkung der Akzeptanz und damit zu einem erheblich geringeren Nutzerkreises führen.

Während die Einbettung in eine Host-Sprache für den Benutzer zu einem erhöhten Lernaufwand führen kann, bietet sie dem Entwickler der Sprache Vorteile. Anders als bei der externen Variante werden für die Umsetzung einer Internal DSL keine Kenntnisse in den Bereichen Grammatiken und Sprach-Parsing benötigt [Fow10].

2.1.2 External DSLs

External DSLs bilden die zweite Kategorie von domänenspezifischen Sprachen. Anders als Internal DSLs werden sie als eigenständige Sprache mit eigener Grammatik realisiert [Fow05].

Ohne die Einschränkung durch eine Host-Sprache sind der Form und Ausgestaltung von External DSLs kaum Grenzen gesetzt [Fow10]. Dadurch bilden sie eine mächtige Variante der DSLs, welche optimal auf ihren jeweiligen Einsatz abgestimmt werden kann.

Größte Schwierigkeit für die Umsetzung einer External DSL ist die Entwicklung eines Übersetzers. Dieser muss in der Lage sein, die Sprache zu verstehen und entsprechenden Code zu generieren. Letzteres kann in Form ausführbaren Maschinencodes als auch als Programm in einer anderen Sprache geschehen. Die Komplexität dieser Aufgabe ist abhängig von der Mächtigkeit der zu erzeugenden Sprache [Fow05].

Durch den Verzicht auf Gestaltungsfreiheiten kann ein Trade-Off in Bezug auf die Komplexität des Übersetzungsvorgangs erreicht werden. Als Beispiel seien hier XML Dialekte genannt. Diese sind auf den XML-Syntax beschränkt und können aufgrund der weitverbreiteten Techniken (DOM, SAX, etc.) sehr einfach geparst werden [Fow05].

Ein Nachteil externer DSLs ist die *Symbolic Barrier*, die zwischen ihnen und anderen Sprachen, welche die Funktionalität der DSL einsetzen, existiert. Als Beispiel hierfür sei die Skriptingfunktionalität, welche in Java 6 durch das Paket `javax.scripting` eingeführt wurde, genannt. Diese erlaubt es Schnittstellen eines Programms externen Scripts zur Verfügung zu stellen. Innerhalb dieser Scripts können bereitgestellte Schnittstellen angesprochen werden um Variablen zu manipulieren oder Funktionen aufzurufen.

In Listing 2.3 wird eine Beispiel-Klasse gezeigt und diese in Listing 2.4 angesprochen.

```
1  class ScriptInterface {
2    private File videoFile;
3
4    public void startVideo();
5    public void stopVideo();
6    public void setVideoFile(String fileName);
7    public float getVideoLength();
8  }
```

Listing 2.3: Beispiel einer Interface-Klasse für javax.scripting.

```
1  var videoPlayer = new ScriptingInterface();
2  videoPlayer.setVideoFile("introvideo.mp4");
3  videoPlayer.startVideo();
```

Listing 2.4: Beispiel eines Scriptingaufrufs für javax.scripting.

Als Skriptsprache wird hierbei JavaScript verwendet. Um das Problem der Symbolic Barriers aufzuzeigen, sei anzunehmen, dass die Funktion `startVideo` umbenannt werden soll. Durch die Refactoring-Funktionalität neuer IDEs wird diese Änderung zwar auf alle Java-Dateien des Projekts übertragen, die Skriptdateien in Javascript bleiben davon aber unbetroffen [Fow05]. Selbstverständlich betrifft dieses Problem nicht alle Arten von External DSLs, sondern lediglich solche welche mit anderen Sprachen zusammenarbeiten. Doch der eigentliche Kernpunkt des Problems ist der Mangel an ausgereiften Werkzeugen. Während das Erstellen bzw. Bearbeiten von Skripten einer External DSL in einem reinen Texteditor zwar nicht komfortabel aber akzeptabel ist, fehlen Hilfsmittel wie Debugger meist komplett. Gerade in komplexen Anwendungen stellt dies ein massives Hindernis dar.

Da DSLs oft mit dem Ziel, auch für Nicht-Programmierer einsetzbar zu sein, entworfen werden, ist der Mangel an Unterstützung durch Syntax-Highlighting, Auto-Vervollständigung oder Refactoring nicht zu unterschätzen.

Um solche Funktionalitäten zu realisieren, ist es notwendig einen Editor bereitzustellen, welcher in der Lage ist die DSL zu verstehen und gegebenenfalls ihre abstrakte Darstellung (z.B. in Form eines ASTs) zu manipulieren [Fow05]. Selbst External DSLs, aus denen Code einer anderen Sprache generiert wird, sind von dieser Problematik betroffen. Es ist zwar möglich den generierten Code mit voller Unterstützung zu manipulieren, allerdings werden diese Änderungen nicht auf den Ausgangscode der DSL projiziert. Des weiteren kann generierter Code mit einem vorhandenen Debugger durchlaufen werden, ohne spezielle Maßnahmen ist eine Rückführung auf betroffene Stellen im DSL-Code aber nicht verfügbar [Fow05].

Ein weiterer Grund der einer breiteren Akzeptanz von DSLs im Wege steht ist die Tatsache, dass viele Programmierer davor zurückschrecken, eine weitere Sprache in ein Projekt aufzunehmen. Im Falle einer weiteren General Purpose Language wäre diese Angst berechtigter als im Falle einer simplen und leichter verständlichen Domain-Specific Language [Fow05].

Verarbeitung externer DSLs

External DSLs unterscheiden sich in den grundlegende Verarbeitungsschritten nicht von anderen Programmiersprachen. Im ersten Schritt wird der Eingabetext durch eine syntaktische oder lexikalische Analyse in seine Bestandteile zerlegt. Dabei dienen vordefinierte Schlüsselwörter (*Keywords*) als Anhaltspunkte für die Abgrenzung von Anweisungen. Dies bezeichnet man als *Delimiter-Directed Translation* [Fow10]. Dabei wird eine Grammatik für die Sprache festgelegt, welche den Syntax der DSL definiert. Der Code-Ausschnitt in 2.5 zeigt einen Auszug aus einer stark vereinfachten Grammatik für die Definition von Datenbank-Tabellen.

```
1  table_def: 'TABLE' name=ID '{'
2    entries+=entry_def*
3  '}' ';'
4
5  entry_def: name=ID ':' type=('String'|'Integer'|'Float') ';'
```

Listing 2.5: Ausschnitt aus einer Grammatik.

Auf Basis dieser Grammatik kann ein semantisches Modell definiert werden, welches mit Einträgen aus einer Textsequenz im syntaktisch korrekten Format befüllt werden kann.
Aus den Einträgen des semantischen Modells kann dann abschließend der entsprechende Code generiert werden. Angewandt auf das Beispiel 2.5 wäre dies ein entsprechendes SQL `CREATE TABLE`-Statement.

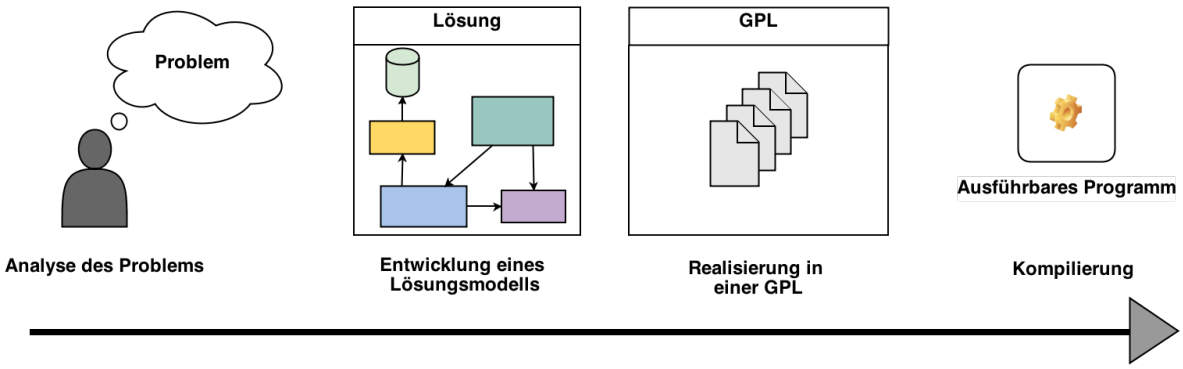

Abbildung 2.1: Traditioneller Programmierprozess ohne LOP.

2.2 Language Oriented Programming

Language Oriented Programming (LOP) bezeichnet ein Programmier-Paradigma, welches stark auf den Einsatz von Domain-Specific Languages setzt [Fow05, Blu12, Dmi05]. Das Bestreben von LOP ist es Probleme nicht durch die "Allmächtigkeit" einer GPL zu lösen, sondern stattdessen eine passendere DSL zu finden, mit der eine elegantere Lösung möglich ist. Dies zieht oft die Entwicklung einer solchen Sprache nach sich, sollte keine anwendbare existieren [Blu12, Dmi05].

Durch LOP verändert sich der traditionelle Programmierprozess. Ursprünglich konnte die Lösung einer Programmieraufgabe mit GPLs in die folgenden Phasen eingeteilt werden [Dmi05]:

1. Analyse des Problems.
2. Entwicklung einer Lösung.
3. Übertragung des Lösungsmodells auf die Konzepte der GPL.
4. Kompilierung.

Diese Phasen werden in Abbildung 2.1 dargestellt. Obwohl dieser Ablauf den meisten Programmierern intuitiv und optimal erscheint, gibt es doch Verbesserungsmöglichkeiten [Dmi05]. Dies betrifft Punkt 3 der obigen Liste, die Übertragung der Lösung auf die gewählte GPL. Dies ist oft sehr schwer und benötigt teilweise unnötig komplizierte Realisierungsmechanismen [Dmi05].

Der Einsatz des Language Oriented Programming kann dieser Problematik entgegenwirken, indem das Lösungsmodell in seine Bestandteile aufgeteilt wird. Diese Teile werden dann mit passenden DSLs realisiert. Dadurch ergibt sich eine neue Phasenaufteilung [Dmi05]:

1. Analyse des Problems.
2. Entwicklung einer Lösung.

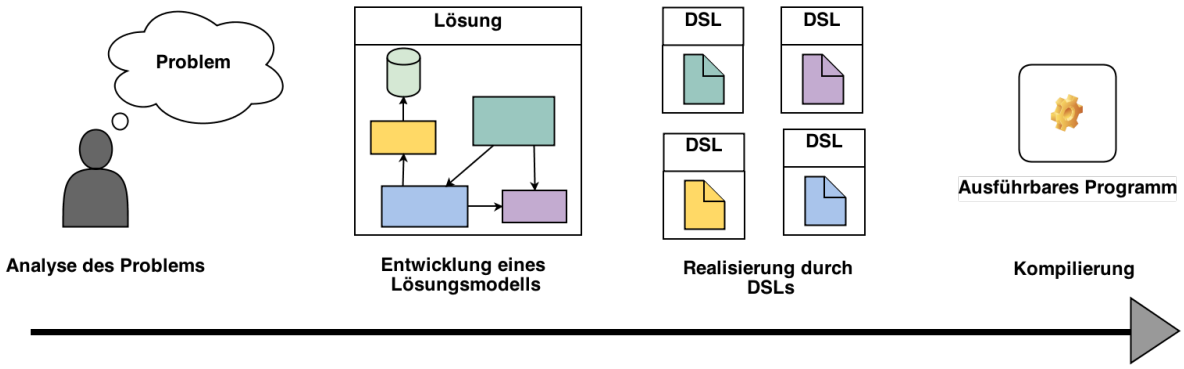

Abbildung 2.2: Programmierprozess mit LOP.

3. *Entwicklung benötiger DSLs.*
4. Übertragung des Lösungsmodells auf die entsprechenden DSLs.
5. Kompilierung/Generierung.

Durch die Einführung von LOP wird gegebenenfalls ein neuer Zwischenschritt nötig. In diesem Schritt werden eventuell benötigte DSLs konzipiert und entwickelt [Dmi05]. Können alle Bestandteile der Lösung mit bereits existenten domänenspezifischen Sprachen abgedeckt werden, so entfällt dieser Schritt. Mit Ersetzung der GPL durch DSLs ist der Übertragungsprozess nicht mehr an die Konzepte einer Sprache gebunden und wird damit einfacher und nachvollziehbarer [Dmi05]. In Abbildung 2.2 ist der Prozess mit LOP noch einmal grafisch dargestellt.

Obwohl der Begriff des Language Oriented Programming der Mehrheit der Programmierer unbekannt ist, gibt es doch einige weit verbreite Szenarien, die auf diesem Prinzip basieren.

Oberflächen-Werkzeuge:

Diese Kategorie von Werkzeugen ist mittlerweile Bestandteil vieler Programmiersprachen. Oft werden diese realisiert indem die Struktur der Oberflächen in einer separaten DSL formuliert werden [Fow05]. Dabei kommen häufig spezialisierte Editoren zum Einsatz. Ein direktes Arbeiten auf der textuellen Repräsentation ist daher nicht nötig. Da hier der Einsatz einer GPL zu Gunsten einer spezialisierteren DSL eingeschränkt wird, fallen solche Werkzeuge in den Bereich des Language Oriented Programming.

Als Beispiel sei die Entwicklungsumgebung Delphi[1] genannt. Entworfene Oberflächen werden in einer *Delphi Form Module*-Datei abgelegt. Diese wird einer Formular-Klasse durch eine entsprechende Präprozessor-Anweisung zugeordnet. Das Listing 2.6 zeigt ein Beispiel für solch eine Formular-Beschreibung.

[1] http://www.embarcadero.com/products/delphi

```
1  object FormMap : TFormMap
2      Left = 0
3      Top = 0
4      Caption = 'MapForm'
5      ClientHeight = 460
6      ClientWidth = 987
7      OnCreate = FormCreate
8      object btnListLocations : TButton
9        Left = 30
10       Top = 10
11       Width = 214
12       Height = 32
13       Caption = 'ListLocations'
14       OnClick = btnGetTownsClick
15     end
16 end
```

Listing 2.6: Beispiel einer *Delphi Form Module*-Datei.

Pro Zeile kann ein Attribut initialisiert werden oder ein Kind-Element definiert werden.

Es ist zudem möglich, Ereignissen Funktionen zuzuordnen. Diese werden über ihren Funktionsnamen identifiziert. Ein Beispiel hierfür ist die Anweisung in Abbildung 2.6, Zeile 7:

```
OnCreate = FormCreate
```

Hierbei entsteht allerdings die in 2.1.2 beschriebene Problematik der *Symbolic Barrier*. Wird eine solche Funktion im Code umbenannt, so ist die Referenzierung in der GUI-Beschreibung fehlerhaft. Um dem entgegenzuwirken übernimmt die IDE die Aufgabe solche Referenzen konsistent zu halten.

2.3 Eigenschaften von Language Workbenches

In Zeiten moderner Entwicklungsumgebungen fällt es schwer, Einsteigern eine Sprache nahe zu bringen, welche nicht über die Komfortfunktionen wie Refactoring, Fehler- und Syntax-Highlighting oder Code-Vervollständigung verfügen [Sto10]. Ohne Hilfestellung durch entsprechende Werkzeuge ist die Realisierung dieser Funktionalitäten bei der Entwicklung neuer DSLs ein sehr großer Mehraufwand.

Die Einführung von Language Workbenches soll hier Abhilfe schaffen [Sto10, Fow05]. Ihr Ziel ist es dem Entwickler von DSLs ein System für deren Erzeugung, Bearbeitung und Anwendung bereitzustellen [Sto10, Fow10]. Dabei fallen der Workbench eine Vielzahl von Aufgaben zu. Neben einem fortschrittlichen Editor für die Erzeugung bzw. Bearbeitung

von DSLs, sollte die Umgebung auch ein ebensolches Werkzeug für die erzeugte Sprache generieren. Idealerweise sind auf diese Weise realisierte Werkzeuge keine eigenständigen und isolierten Anwendungen, sondern Teil einer gemeinsamen Entwicklungsumgebung [Sto10].

Ein großes Problem weitverbreiteter (GPL-)Sprachen ist die Verpflichtung zur Rückwärtskompabilität. Soll der bestehende Sprachsyntax verändert werden, so wird dies in den meisten Fällen dazu führen, dass vormals funktionierender Code fehlerhaft wird [Sto10]. Die Folge ist, dass bestehende Sprachen lediglich wachsen, sprich neue Features bekommen, können. Sollen veraltete Elemente geändert oder gelöscht werden, so müssen die Anwender mit einem entsprechenden Zeitvorlauf darauf aufmerksam gemacht werden.

Ein Beispiel für solch einen Mechanismus ist die `@Deprecated`-Annotation [Ora04] in Java. Im Falle eigener DSLs ist diese Problematik weit schwächer ausgeprägt, da hier das Einsatzgebiet oft stark, zum Beispiel auf ein Projekt, beschränkt ist. Eigene DSLs können daher oft flexibler angepasst werden. Dies sollte eine Language Workbench ebenfalls unterstützen [Sto10].

Neben der Unterstützung des Erstellungsprozesses durch Funktionen wie Refactoring oder Code-Vervollständigung bieten IDEs heutzutage ein grafisches Interface für den Debugging-Prozess. Gerade für Einsteiger, aber auch in hochkomplexen Projekten, ist dies ein wertvolles Werkzeug für die Fehlersuche [Sto10].

2.3.1 Language Workbench Interna

Anders als einfache Texteditoren oder ältere Entwicklungsumgebungen operieren Language Workbenches nicht mehr auf der textuellen Repräsentation [Fow05] eines Programms. Statt wie ursprünglich den Programmtext durch einen Übersetzer oder Interpreter in eine ausführbare Form überführen zu lassen, setzen Language Workbenches eine abstrakte Zwischendarstellung ein [Fow05, Fow10, Sto10]. Diese stellt die in der textuellen Form vorgegebene Struktur des Codes dar. In Abbildung 2.3 wird der Ablauf einer Programmübersetzung vereinfacht dargestellt.

Hierbei wird das Programm primär durch seine textuelle Form repräsentiert. Alle Manipulationen seitens des Benutzers finden hier statt. Werden die Dateien an den Compiler übergeben, so erzeugt dieser den abstrakten Syntaxbaum (AST). Auf Basis dieser Zwischendarstellung kann der Übersetzer verschiedene Zwischenschritte wie das Entfernen toten Codes oder die Eliminierung von Hilfsvariablen durchführen. Abschließend wird aus dem AST der ausführbare Code generiert.

Moderne IDEs und Language Workbenches setzen auf einen veränderten Bearbeitungs- und Übersetzungsprozess [Fow05, Fow10]. Dies wird in Abbildung 2.4 dargestellt.

Statt einen AST erst im Falle einer anstehenden Übersetzung zu erzeugen, halten sie eine aktuelle Repräsentation des Programms im Speicher [Fow05, Fow10]. Alle Änderungen in einem Editor werden auf die abstrakte Darstellung projiziert. Im Gegenzug führen Manipulationen der Abstraktion zu einer entsprechenden Veränderung der Darstellung in den Editoren. Dabei sind diese Editoren nicht an eine textuelle Repräsentation gebunden [Fow10].

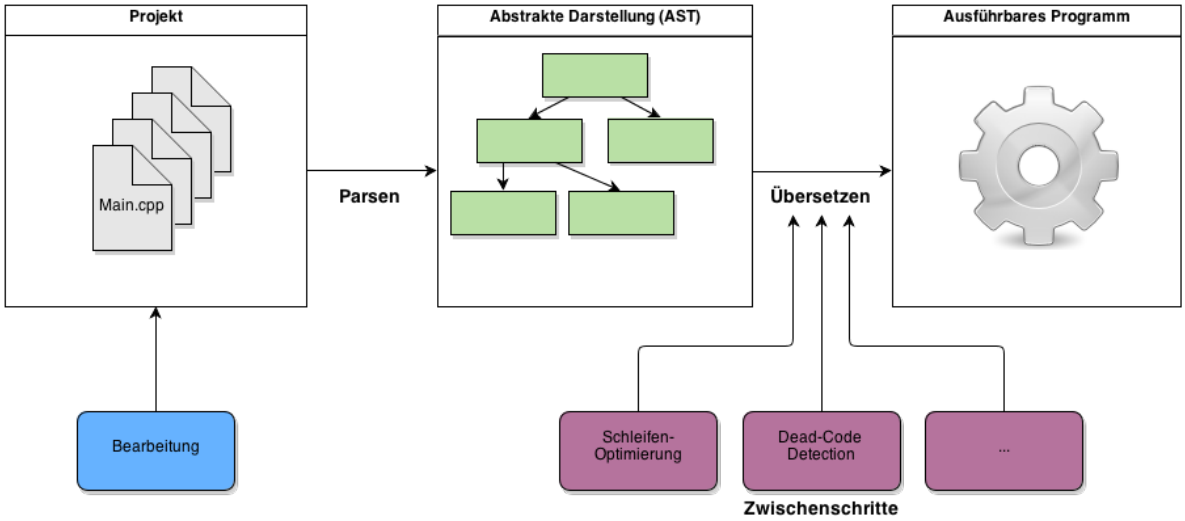

Abbildung 2.3: Traditionelle Programmübersetzung.

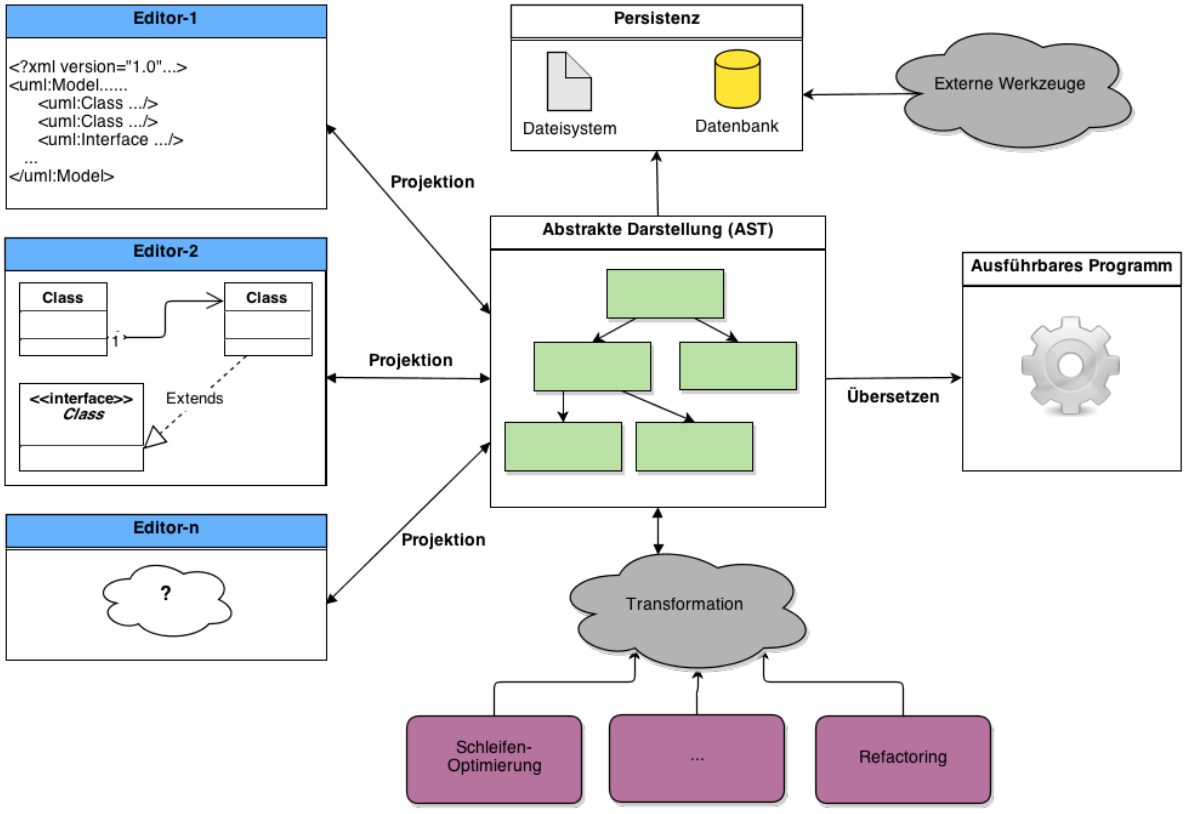

Abbildung 2.4: Arbeitsweise moderner IDEs und Language Workbenches.

Als Beispiel sei ein UML Editor genannt. Dieser könnte einen grafischen Editor besitzen, in dem die Diagramme modelliert werden können. Abgespeichert würde das Ergebnis in einer textuellen Form, z.B. als XML. Für diese Form könnte ein weiterer Editor bereitgestellt werden. Um nun aber nicht auf den XML-Knoten der textuellen Darstellung arbeiten zu müssen, wird eine abstrakte Darstellung eingesetzt. Manipulationen dieser Darstellungen werden auf die verschiedenen Darstellungsformen projiziert. Aus Sicht eines Generators könnte nun die abstrakte Zwischendarstellung in ausführbaren Code überführt werden. Neben der Verarbeitung durch IDE-interne Werkzeuge kann die Zwischendarstellung in serialisierter Form (z.B. als Datei) externen Werkzeugen zur Verfügung gestellt werden.

Obwohl die Operation auf der Zwischendarstellung einige Vorteile bietet, gibt es doch einige Schwierigkeiten zu beachten. Eine Problematik liegt in Situationen, in denen der Parser, welcher die abstrakte Zwischendarstellung erzeugt, fehlerhafte Eingaben erhält [Fow05]. Dies wäre beispielsweise ein Programmierfehler, so dass der eingegebene Code nicht mehr der Grammatik der verwendeten Programmiersprache entspricht. Zur Sicherstellung wichtiger Funktionen, wie des Refactorings muss der Parser auf eine solche Eingabe vorbereitet sein und trotzdem, soweit möglich, eine abstrakte Darstellung erzeugen [Fow05]. Auf Basis dieser Darstellung kann dann beispielsweise eine fehlerhafte Stelle im Eingabecode hervorgehoben werden.

Das Prinzip der Language Workbenches basiert auf drei grundlegenden Bestandteilen [Fow10, Fow05].

Schema des semantischen Modells:

Durch das semantische Schema wird die abstrakte Darstellung der Sprachbestandteile einer DSL beschrieben. Sie stellt den zentralen Bestandteil der Workbench dar [Fow10]. Dabei stellen Language Workbenches oft spezielle Werkzeuge bereit, um dieses Schema zu erzeugen. Im Gegensatz zum Verständnis objektorientierter Programmierung ist die abstrakte Darstellung mit keiner Verhaltenslogik verbunden [Fow10]. Diese wird extern durch das dritte Element der Language Workbenches definiert.

Editoren:

Einer der zentralen und wichtigsten Vorteile von Language Workbenches sind die mächtigen Editoren die sie bereitstellen. Diese erlauben es, das zuvor definierte semantische Modell zu befüllen. Dabei sind diese Editoren nicht auf eine textuelle Form beschränkt. Neben Texteditoren sind auch grafische oder tabellenbasierte Bearbeitungsmethoden denkbar [Fow10]. Durch diese Variation ist es möglich, auch Nicht-Programmierer mit in den produktiven Prozess einzubinden, da grafische Modellierungen oft leichter verständlich sind. Prominentes Beispiel hierfür ist Microsoft Excel [Fow10].

Verhaltenslogik des semantischen Modells:

Das letzte Element ist die Logik, mit der das semantische Modell nach seiner Instanziierung verarbeitet wird. Häufigstes Szenario ist die Generierung von Code anhand der Informationen im Modell. Es ist allerdings auch möglich, auf diese Logik zu verzichten und das semantische Modell externen Anwendungen bereitzustellen. Dadurch ist es beispielsweise möglich, einen Interpreter zu schreiben, welcher das Modell als Eingabe erhält [Fow10].

2.4 Beispiele für Language Workbenches

Obwohl das Themengebiet der Language Workbenches noch im Anfangsstadium ist, gibt es bereits einige erwähnenswerte Beispiele. Diese unterscheiden sich in Umsetzung, Arbeitsweise und den bereitgestellten Editoren. Eine große Gefahr für die Verbreitung von Language Workbenches ist der sogenannte *Vendor Lock-In* [Fow10]. Dabei kommt es zu Inkompatibilitäten der Language Workbenches zueinander. Projekte, die in einer Language Workbench erstellt sind, können dann beispielsweise in einer anderen nicht geöffnet und bearbeitet werden.

Intentional Workbench:

Intentional Workbench wird von Intentional Software unter der Leitung von Charles Simonyi entwickelt. Simonyi war als Leiter der Entwicklung der Microsoft Office Suite tätig und versucht auch mit der Intentional Workbench ebenfalls Nicht-Programmierer in den Prozess einzubinden [Fow10]. Stärken dieser Language Workbench sind ihre vielfältigen Editoren (u.a. Textuell, Tabellarisch sowie in Diagrammform) und eine Vielzahl komplexer und vordefinierter Meta-Modelle [Sto10].

Meta-Programming System (MPS):

MPS baut stark auf textuelle Editoren und fokussiert damit mehr auf Programmierer als Nutzerkreis. Dafür stellt es eine mächtige IDE mit fortschrittlichen Techniken und Werkzeugen zur Verfügung [Fow10, Sto10]. Hervorzuheben ist, dass große Teile des Meta-Programming System als Open-Source verfügbar sind. MPS wird von der Firma JetBrains entwickelt.

Meta-Edit:

Im Gegensatz zu MPS setzt Meta-Edit auf graphische Editoren und Eingaben über Tabellen [Sto10]. Dadurch ist es besonders für den Einsatz durch Nicht-Programmierer attraktiv. Verantwortlich für die Entwicklung von Meta-Edit ist die Firma MetaCase.

3 Eingesetzte Technologien

In diesem Kapitel werden einige Technologien vorgestellt, welche Einfluss auf mehrere der Projekte hatten, die im Laufe dieser Diplomarbeit realisiert wurden. Da ihr Verwendungszweck sich kaum unterscheidet, sind diese Beschreibungen in einem separaten Kapitel ausgelagert. Gibt es doch Unterschiede, so werden diese besonders hervorgehoben.

3.1 JNI

Das Java Native Interface, kurz JNI, ist eine Funktionalität innerhalb der Java Plattform mit der es möglich ist, nativen Code aus einer Java-Anwendung heraus auszuführen. Als "nativ" wird dabei Code bezeichnet, der nicht in der Java VM ausgeführt wird. Dieser kann in Sprachen wie C oder C++ geschrieben sein. Als Zwei-Wege-Interface erlaubt es JNI ebenfalls Java-Funktionen aus nativen Sprachen heraus aufzurufen [Lia99].

Dadurch ist es möglich, bereits existenten Code in neuen Projekten wiederzuverwenden oder Schnittstellen nativer Programme für andere Sprachen bereitzustellen. Letzteres könnte beispielsweise genutzt werden, um eine Java-Bridge für eine native API zu entwickeln. Ein weiteres Szenario, in dem der Einsatz von JNI möglich wäre, ist die Einbindung systeminterner Funktionalitäten (z.B. spezielle Manipulationen von Dateien) oder Gerätetreiber in Java-Programme.

Wird JNI genutzt um Java-Funktionen in nativen Programmen einzusetzen, so muss eine spezielle, native Bibliothek eingebunden werden. Diese implementiert die Java VM und kann genutzt werden, um Java-Funktionen darin aufzurufen [Lia99].

Durch den Einsatz von JNI in einer Java-Anwendung gehen allerdings auch Vorteile verloren. Während eine der großen Stärken des Java Runtime Environment seine Plattformunabhängigkeit ist, sind native Bibliotheken an eine bestimmte Plattform gebunden. Um die Unabhängigkeit beizubehalten, müssen die nativen Teile entsprechend implementiert und für alle Zielplattformen kompiliert werden [Lia99]. Des Weiteren kann es zu Problemen in Hinsicht auf die Typsicherheit kommen. Diese kann innerhalb von Java zwar sichergestellt werden, dies gilt allerdings nicht für Aufrufe nativer Funktionen [Lia99]. Um hier potentielle Fehlerquellen auszuschließen, muss der Entwickler der Java-Anwendung zusätzlichen Aufwand in Form von Typprüfungen vornehmen. Mit diesen Gefahren im Hinterkopf weist selbst [Lia99] daraufhin, zuerst eventuelle Alternativen zum Einsatz nativen Codes in Betracht zu ziehen. Mögliche Ansätze sind hierbei die Nutzung von TCP/IP- oder Interprozess-Kommunikation. Ob diese möglich sind hängt allerdings von der Natur des nativen Codes ab. Vorteile alternativer Techniken liegen besonders in der Tatsache, dass der native Code in

einem eigenen Prozess läuft. Dies sichert die aufrufende Anwendung ab, denn ein Fehler in der Ausführung der nativen Funktionen führt nicht automatisch zu einem Abbruch der Java-Anwendung [Lia99].

Der Bedarf einer Möglichkeit nativen Code aus Java heraus aufzurufen existiert bereits seit dem ersten Release des Java Development Kit (JDK). Hier wurde ein Interface bereitgestellt, mit dem Java Klassenbibliotheken (u.a. `java.lang` und `java.io`) in der Lage sind native Funktionalitäten der Host-Plattform aufzurufen [Lia99]. Aufgrund von Problemen in dieser ersten Version wurde das Java Native Interface entwickelt und als Bestandteil des *Java 2 SDK 1.2* zum ersten Mal in seiner heutigen Form ausgeliefert.

3.1.1 Einbindung des JNI

Nach der formellen Beschreibung des JNI soll an dieser Stelle gezeigt werden, wie eine Einbindung dieser Funktionalität aussieht. Dabei wird lediglich der Anwendungsfall nativen Code aus einer Java-Anwendung heraus aufzurufen betrachtet. Da alle Programme, welche während dieser Diplomarbeit entstanden sind, in Java entwickelt wurden, ist dies das relevantere Szenario.

Die Einbindung des JNI kann dabei in die folgenden Schritte unterteilt werden:

1. Klassenerzeugung mit nativer Methodendeklaration:

Im ersten Schritt müssen die nativen Methoden deklariert werden. Dies kann in einer separaten, als auch in einer bereits existenten Klasse geschehen. Zusätzlich zur Deklaration der Methode muss die Bibliothek, in der die native Funktion implementiert ist, geladen werden. Der Code-Ausschnitt 3.1 zeigt eine Beispielklasse in der eine solche native Methode deklariert und eine entsprechende Bibliothek geladen wird.

```
1  class JNIExample {
2      //Deklaration der nativen Methode:
3      private native void write();
4
5      public static void main(String[] args) {
6          //Laden der Bibliothek;
7          System.loadLibrary("JNIExample");
8
9          //Erzeugen einer Instanz und Aufruf der nativen Methode:
10         JNIExample instance = new JNIExample();
11         instance.write();
12     }
13 }
```

Listing 3.1: Beispielklasse mit nativer Methodendeklaration.

Native Methoden unterscheiden sich von der Deklaration nicht-nativer Methoden. Neben der Markierung durch das `native`-Schlüsselwort wird die Deklaration mit einem abschließenden Semikolon beendet. Da die Methode durch eine externe Bibliothek bereitgestellt wird, besitzen native Methoden in Java keine Implementierung [Lia99].

Das Laden der nativen Library wird durch den Aufruf von `System.loadLibrary(...)` ausgelöst. Hierbei wird der Name der Bibliothek übergeben. Obwohl es möglich ist eine Dateiendung anzugeben (z.B. DLL für Windows-Bibliotheken), kann dies ausgelassen werden. In diesem Fall entscheidet die Java VM in Abhängigkeit der Host-Plattform über die zu erwartende Endung [Lia99]. Wichtig ist, dass die entsprechende Bibliothek geladen wird, bevor eine native Methode aufgerufen wird.

Nach der Erstellung der Java-Klasse wird diese gespeichert und durch den *javac* Compiler übersetzt. Die dadurch entstehende Class-Datei wird im nächsten Schritt zur Generierung einer C Header-Datei benötigt. Der Aufruf des Compilers ist denkbar einfach: `javac JNIExample.java`.

2. Generierung der C Header-Datei:

Nachdem im vorangegangenen Schritt eine Klassendatei kompiliert wurde, soll nun ein entsprechender C Header generiert werden. Hierfür stellt Java ein Werkzeug bereit, welches dies automatisch erledigt [Lia99]. Dieses liegt im selben Verzeichnis wie der javac-Kompiler und wird durch den Aufruf in Listing 3.3 gestartet.

```
1    javah -jni JNIExample.class
```

Listing 3.2: Aufruf von javah für die Generierung eines C Headers.

In der dabei erzeugten Datei "JNIExample.h" befindet sich die Signatur aller als `native` deklarierten Funktionen. Wies im Bezug auf das Beispiel in Listing 3.1 aussehen würde, zeigt der Code-Ausschnitt **??**.

```
1    JNIEXPORT void JNICALL Java_JNIExample_write (JNIEnv *, jobject);
```

Listing 3.3: Aufruf von javah für die Generierung eines C Headers.

Hierbei fällt auf, dass die native Deklaration in Java keine, die C-Funktion aber zwei Parameter besitzt. Der erste Parameter bezeichnet dabei einen Pointer auf eine `JNIEnv`-Umgebung. Dass zweite Argument ist ein Zeiger auf eine Instanz des `JNIExample`. Auf die technische, interne Realisierung des JNI soll an dieser Stelle nicht weiter eingegangen werden. Dem interessierten Leser sei dafür [Lia99] nahegelegt.

3. Implementierung der nativen Methode:

Im dritten Schritt muss die native Methode nun implementiert werden. Dafür wird eine entsprechende Datei "JNIExample.c" (oder .cpp, wenn C++ verwendet wird) angelegt. Diese muss dabei als Include "jni.h", sowie die zuvor generierte "JNIExample.h" einbinden. Der Rest der Implementierung erfolgt dabei wie in sonstigen Fällen für C bzw. C++. Dies wird hier als bekannt vorausgesetzt.

4. Kompilierung als native Bibliothek:

Im letzten Schritt muss der nun erzeugte Code noch in eine Bibliothek kompiliert werden. Dieser Schritt ist demnach davon abhängig, auf welcher Plattform und mit welchem Compiler gearbeitet wird. Die entsprechende Anweisung muss demnach dem Handbuch des eingesetzten Übersetzer entnommen werden. Das Ergebnis der Ausführung ist eine sogenannte *Shared Library*. Diese Art der Bibliothek wird dann von der Java-Anwendung durch die Anweisung `System.loadLibrary(...)` geladen [Lia99].

Nach Abschluss dieses letzten Schritts kann der Java-Code nun ausgeführt werden. Liegt die Bibliothek im korrekten Verzeichnis (dem Classpath der Anwendung), dann wird der Aufruf der nativen Funktion entsprechend der Implementierung in der C bzw. C++ Bibliothek durchgeführt. Sollen weitere Funktionen ausgelagert werden, so müssen dementsprechend die Schritte wiederholt und die Bibliothek erneut übersetzt werden.

3.1.2 JNI in Bauhaus

Innerhalb des Bauhaus Frameworks wird JNI als Schnittstelle für die Nutzung von IML in Java-Programmen eingesetzt. Dafür wurden von Steffen Pingel (s. [Pino6]) Werkzeuge und Abläufe für die Generierung eines solchen Bindings implementiert. Eine ausführliche Beschreibung der technischen Realisierung, sowie der Erweiterung des Mechanismus findet sich in [Pino6].

Da in dieser Arbeit ebenfalls neue Funktionalitäten für die Java-Anbindung eingeführt wurden (s. Kapitel 7), soll an dieser Stelle lediglich auf einige ausgewählte Punkte eingegangen werden. Diese sollen helfen eventuelle Schwierigkeiten zu umgehen.

Dabei muss unterschieden welcher Art die Funktionalität ist. Ein simples Hinzufügen neuer IML-Konstrukte wird von den Werkzeugen vollständig abgedeckt und kann ohne Anpassung der Generatoren erfolgen. Komplexer gestaltet sich die Aufgabe, wenn Funktionen, die auf den IML-Graphen operieren, für Java verfügbar gemacht werden sollen. Auch hierbei muss wiederum eine Unterscheidung getroffen werden. Die folgenden Punkte behandeln zwei Kategorien, welche im Laufe dieser Diplomarbeit angetroffen wurden.

Konstante Funktionalitäten:

Diese sind fest implementiert und nicht von den Inhalten der IML-Spezifikation abhängig. Sie werden also bei einer Veränderung nicht neu generiert. Hierfür ist demnach keine Erweiterung der Werkzeuge nötig, da die Funktionssignaturen für das JNI-Binding von Hand angelegt werden können. Für die Bereitstellung einer solchen Funktionalität ist das Vorgehen identisch mit den im vorangegangenen Abschnitt erläuterten Schritten. Der Ort, an dem die einzelnen Dateien abgelegt werden, kann [Pino6] entnommen werden. Ein Beispiel für eine konstante Funktionalität ist das Paket *reflection_utils*.

Dynamisch generierte Funktionalitäten:

Im Gegensatz zu den unveränderlichen Funktionalitäten sind die dynamisch generierten erheblich komplexer in ihrer Einbindung. Da sie auf Basis veränderlicher Informationen erzeugt werden, müssen die entsprechenden Generatoren angepasst werden. Diese erzeugen in den meisten Fällen nur Ada-, und in bestimmten Situationen, C-Dateien. Existieren C-Dateien, so ist die Einbindung simpler, da das Binding auf der Basis von C-Headern erzeugt wird. Es muss also lediglich dafür gesorgt werden, dass die C-Dateien vom Generator verarbeitet werden. Existieren keine C-Dateien, so müssen die Generatoren angepasst werden. Ein Beispiel für diese Kategorie ist das Paket *iml_tables*. Dieses stellt Funktionen bereit, welche aus einem IML-Graph alle Knoten eines bestimmten Typs als `Set` zurückgeben. Das Paket wird bei jeder Generierung der IML-Klassen neu erzeugt, da es davon abhängt welche Knoten in der IML-Spezifikation enthalten sind.

3.2 Apache Commons Command Line Interface (CLI)

Das Apache Commons Command Line Interface (CLI) ist eine Bibliothek für das vereinfachte Parsen von Kommandozeilenparametern. Sie stellt Entwicklern eine Schnittstelle bereit um auf simple Weise selbst komplexe Argumente aus der Kommandozeile zu lesen. Dabei übernimmt CLI die Aufgaben des Parsen, der Validierung sowie der Anzeige von Hilfe-Texten [Ive05].

CLI ist Teil der Apache Commons Bibliothek und wird unter der Apache License 2.0 [1] bereitgestellt.

Da alle Werkzeuge, welche im Laufe dieser Diplomarbeit entstanden sind, als Konsolenanwendungen entwickelt wurden, wird CLI 1.2 in allen Situationen eingesetzt in denen Kommandozeilenparameter übergeben werden. Eingebunden wird dies, indem man die Bibliothek `commons-cli-1.2.jar` dem Buildpath einer Anwendung hinzufügt.

[1] http://www.apache.org/licenses/

Konfiguriert wird der Kommandozeilenparser, indem man Instanzen der Klasse `org.apache.commons.cli.Option` erzeugt und diese beim Parser registriert [Ive05]. Optionen können mit einer Vielzahl von Informationen ausgestattet werden:

Attribut	**Beschreibung**
ArgName	Der lesbare Name für das Argument, der in der Usage-Meldung angezeigt wird.
Opt	Das Kürzel über das der Parameter eindeutig identifiziert wird.
LongOpt	Der ausgeschriebene Name über den der Parameter ebenfalls identifiziert werden kann.
Type	Der Typ des Arguments, z.B. Integer oder Double.
Required	Flag, welches angibt ob es sich um einen Pflichtparameter handelt.
Value	Der Wert des Arguments, kann nach dem Parsen ausgelesen werden.
ValueSeparator	Das Trennzeichen welches bei der Übergabe mehrerer Werte verwendet wird.

Wird beim Parsen der Argumente ein Fehler entdeckt, beispielsweise verursacht durch ein fehlendes Pflichtargument, einen falschen Wert oder ein unbekanntes Kürzel, so wird eine `org.apache.commons.cli.ParseException` geworfen [Ive05]. Tritt dieser Fall in einem der Projekte dieser Diplomarbeit auf, so wird die Ausführung abgebrochen und eine entsprechende Fehlermeldung zusammen mit dem Hilfe-Text der Anwendung ausgegeben. Ausschnitt 3.4 zeigt die Ausgabe des Java-Code-Generators (s. 9 wenn dieser ohne Kommandozeilenparameter aufgerufen wird.

```
30.12.2012 19:33:39,297 [main] ERROR AppCore - Error parsing the commandline
    arguments!
org.apache.commons.cli.MissingOptionException: Missing required options: s, o
    at org.apache.commons.cli.Parser.checkRequiredOptions(Parser.java:299)
    at org.apache.commons.cli.Parser.parse(Parser.java:231)
    at org.apache.commons.cli.Parser.parse(Parser.java:85)
    at bauhaus.unparsej.core.AppCore.parseCmdArgs(AppCore.java:129)
    at bauhaus.unparsej.core.AppCore.main(AppCore.java:190)
usage: unparse_j.sh *args*
Processes an IML file and generates java classes from the contents found
in the file. The result is stored in the location specified in the
arguments.
 -o <arg>  Specifies the folder in which the output files will be placed.
 -s <arg>  Specifies the IML file from which the code is generated.
 -v        Toggles the verbose mode.
Missing required options: s, o
```

Listing 3.4: Ausgabe des Java-Code-Generators bei Aufruf ohne Kommandozeilenparameter.

CLI ermöglicht die Verwendung verschiedener Formatierungen für die Übergabe der Parameter [Mar08]. Möglich sind dabei unter anderem die folgenden, weitverbreiteten Formate:

Posix-Style:

Beim Posix-Stil werden die Argumente über einen einzelnen Buchstaben identifiziert. Dies erhöht die Lesbarkeit, kann aber, besonders bei komplexen Namen, zu unnötig langen Anweisungen für einen Programmaufruf führen. Argumente können dabei mit einem Wert parametriert werden. Das Beispiel in 3.5 zeigt die Konfiguration von Argumenten im Posix-Style:

```
1 public static Options configOptionsPosix()
2 {
3     final Options posixOpts = new Options();
4     posixOpts.addOption("v", false, "Logging will be verbose.");
5     posixOpts.addOption("o", true, "Specifies the ouput location.");
6     posixOpts.addOption("s", true, "Specifies the address of the server.");
7     return posixOpts;
8 }
```

Listing 3.5: Konfiguration von Kommandozeilenparametern im Posix-Style.

Hierbei werden drei Argumente registriert. Diese können mit einem vorangestellten Bindestrich, sowie ihrem Kürzel übergeben werden. Der zweite Parameter des Aufrufs von `addOption` gibt dabei an, ob das Argument mit einem Wert übergeben werden muss. Als Beispiel dient das zweite Argument im vorangegangenen Ausschnitt. Hierbei muss der Ausgabepfad für eine Datei übergeben werden. Wird kein Wert angegeben, so wird eine Exception geworfen und der Parsing-Vorgang abgebrochen.

GNU-Style:

Ein weiteres bekanntes Format für die Übergabe von Kommandozeilen-Parametern ist der sogenannte GNU-Style. Dabei werden zusätzlich ausführliche, lange Bezeichnungen für die Parameter verwendet. Diese werden mit einem doppelten Bindestrich gekennzeichnet. Im Ausschnitt 3.6 wird exemplarisch dargestellt wie CLI für die Verwendung von GNU-Style Argumenten konfiguriert wird.

```
1 public static Options configOptionsGNU()
2 {
3     final Options gnuOpts = new Options();
4     gnuOpts.addOption("v", "verbose" false, "Logging will be verbose.");
5     gnuOpts.addOption("o", "output", true, "Specifies the ouput location.");
6     gnuOpts.addOption("s", "server", true, "Specifies the address of the
          server.");
7     return gnuOpts;
8 }
```

Listing 3.6: Konfiguration von Kommandozeilenparametern im Posix-Style.

3.3 Log4J

Mit wachsender Komplexität von Projekten wächst auch der Zeitaufwand für das Auffinden von Fehlern. Um dem entgegenzuwirken ist es nötig, die Entwickler mit möglichst detaillierten Informationen zu versorgen. Um dies zu realisieren, stellen nahezu alle (größeren) Anwendungen eine Logging-Funktionalität bereit [Gue03]. Um deren Implementierung zu vereinfachen, wurden eine Vielzahl von Logging APIs entwickelt. Diese erlauben es Informationen über den Ablauf eines Programms zu erfassen, um so den Zeitpunkt und die Umstände eines aufgetretenen Fehlers zu identifizieren [Gue03]. Mit dem starken Zuwachs der Java-Community, besonders in den letzten Jahren, hat sich Log4J zu einer Art Industriestandard etabliert.

Begonnen wurde Log4j von Ceki Gülcü. Dieser war 1996 bei IBM in Zürich tätig. Dort entwickelte er die vom E.U. SEMPER Projekt [2] begonnen Arbeit an einer Logging-API zur ersten Version von Log4J [Gue03]. Log4J wurde schnell zu einer der weitverbreitetsten und erfolgreichsten Logging-APIs für Java. Bereitgestellt wird es als Open-Source-Projekt unter der Apache Software License 2.

Obwohl viele Fehler in einer Anwendung bereits in den ersten Phasen der Implementierung gefunden werden, gibt es viele Bugs, welche erst im Testcenter oder sogar nach der Produktivphase auftreten. Im Gegensatz zu ersterer Situation ist hierbei kein Debugger vorhanden mit dem konkrete Informationen über den Hergang des Fehlers gewonnen werden können. Diese Problematik betrifft ebenfalls Multithread- sowie verteile Applikationen [Gue03]. In solchen Fällen sind die Daten, welche durch das Logging bereitgestellt werden, von hohem Wert. Es ist dann möglich solche Loggings in persistenter Form, z.B. als Datei, bei der Meldung eines Fehlers den Entwicklern zur Verfügung zu stellen [Gue03].

Seit seiner Veröffentlichung wurde Log4J durch andere Autoren für eine Vielzahl anderer Sprachen portiert. Darunter befinden sich Versionen für C, C++, Python, Ruby und viele andere.

Eingebunden wird das Framework durch das Hinzufügen der Log4j-Bibliothek zum Classpath eines Projekts [Gue03]. Dadurch werden alle benötigen Funktionalitäten für den Entwickler sichtbar gemacht. Um einen Logger in einer Java-Klasse zu verwenden wird eine statische Instanz erzeugt. Dies wird im Code-Ausschnitt 3.7 beispielhaft dargestellt.

```
1 public class ExampleClass {
2     static Logger logger = Logger.getLogger(ExampleClass.class);
3     static public void main(String[] args) {
4         BasicConfigurator.configure();
5         logger.debug("Hello world.");
6     }
7 }
```

Listing 3.7: Instanziierung und Aufruf von Log4j aus Java.

[2] http://www.semper.org/

Durch die Übergabe der Zielklasse an den Aufruf von `Logger.getLogger(...)` wird die Instanz des Loggers konfiguriert. Alle Log-Einträge, die von dieser Instanz geschrieben werden, fügen den Namen der Zielklasse an der Stelle des entsprechenden Platzhalters im Log-Format ein. Standardmäßig sind keine Ausgaben in Log4j vordefiniert. Diese müssen manuell durch den Entwickler des Projekts, in dem das Framework genutzt werden soll, konfiguriert werden. Der Aufruf `BasicConfigurator.configure()` erstellt einen simplen Stdout-Logger mit dem die Ausgaben zum Beispiel in die Konsole geschrieben werden.

In den Implementierungen dieser Diplomarbeit wird Log4j standardmäßig in zwei Varianten eingesetzt. In der "normalen" Variante werden lediglich Warnungen, abgefangene Exceptions sowie kritische Fehler geloggt. Dadurch wird die Ausgabe nicht durch unnötige Informationen über Programm-Interna belastet. Dies ist beispielsweise für den Aufruf aus einem Make-Prozess sinnvoll.

In der zweiten Variante, welche durch den Kommandozeilenparameter *-v* aktiviert wird, werden zudem Informationen über Abläufe innerhalb des Programms wiedergegeben. Dies ist besonders zu Debugging-Zwecken sinnvoll, um zu erkennen an welcher Stelle ein Fehler auftritt. Dies wird in Ausschnitt 3.8, anhand eines Aufrufs des Zustandsautomaten-Frontends dargestellt.

```
1 30.12.2012 03:39:15,782 [main] INFO AppCore - Registering XMIResourceFactory.
2 30.12.2012 03:39:15,822 [main] INFO AppCore - Registering XMLResourceFactory.
3 30.12.2012 03:39:17,949 [main] INFO AppCore - Registering UML file extension
    for factory: uml
4 30.12.2012 03:39:17,955 [main] INFO AppCore - Starting AdaInterface.
```

Listing 3.8: Ausschnitt aus einer Logging-Ausgabe mit Verbose-Schalter.

Es ist bei allen Projekten, in denen das Log4j Framework zum Einsatz kommt möglich, die Eigenschaften der Logger durch eine externe Property-Datei zu konfigurieren. Dadurch ist es beispielsweise möglich, das Ausgabe-Format zu ändern oder neue Logger hinzuzufügen. Abgelegt wird diese Konfigurationsdatei dabei im Classpath der Anwendung. Standardmäßig loggen alle Projekte dieser Diplomarbeit nur auf *stdout*. Soll zudem in eine Datei geloggt werden, so kann dies über die zuvor erwähnte Property-Datei realisiert werden. Im Ausschnitt 3.9 wird gezeigt wie ein Datei-Logging mit Log4j in einer Konfigurationsdatei aktiviert wird.

```
1 # Root logger option
2 log4j.rootLogger=INFO, file
3
4 # Direct log messages to a log file
5 log4j.appender.file=org.apache.log4j.RollingFileAppender
6 log4j.appender.file.File=C:\\logging.log
7 log4j.appender.file.MaxFileSize=1MB
8 log4j.appender.file.MaxBackupIndex=1
```

```
9 log4j.appender.file.layout=org.apache.log4j.PatternLayout
10 log4j.appender.file.layout.ConversionPattern=%d{ABSOLUTE} %5p %c{1}:%L - %m%n
```

Listing 3.9: Ausschnitt aus einer Log4j-Properties-Datei.

Die Ausgabe des Loggings wird in diesem Beispiel im Pfad "C: \logging.log" erzeugt. Dabei werden alte Einträge gelöscht sobald die Datei eine Größe von 1 Megabyte erreicht hat. Durch das spezifizierte Layout wird vorgegeben wie die Einträge aussehen sollen.

4 Projekt-Übersicht

In diesem Kapitel werden die Teilaufgaben beschrieben, welche im Laufe dieser Diplomarbeit entstanden sind. Dabei wird kurz auf die einzelnen Werkzeuge, die Erweiterungen und Verwendungen externer Werkzeuge und das Musterbeispiel für den Test der Funktionalitäten eingegangen. Eine ausführliche Beschreibung der Projekte sowie des Musterbeispiels finden sich in den entsprechenden Kapiteln dieser Arbeit. Zweck dieses Abschnitts ist es, die Zusammenhänge zwischen den Werkzeugen, sowie ihre Einbettung in das Bauhaus Framework aufzuzeigen.

4.1 Aufgaben und Komponenten

Die Aufgabe dieser Diplomarbeit lässt sich in zwei Hauptkategorien unterteilen. Alle entwickelten Werkzeuge, geschriebene Skripts und sonstigen Produkte dieser Arbeit fallen in eine dieser beiden Gruppen.

Die erste Kategorie hat den Zweck der Erweiterung von IML für neue Programmiermodelle. Ziel ist es, mit der Zwischendarstellung IML abstrakte Konzepte abbilden zu können. In ihrer momentanen Form dient IML primär der Darstellung von Code verschiedener Programmiersprachen. Mit Einführung des Sprachpaket-Konzepts wird es möglich sein, abstraktere Konstrukte wie beispielsweise UML-Verhaltensdiagramme in IML zu repräsentieren. Mit diesem Schritt nähert sich Bauhaus weiter an das Prinzip einer Language Workbench an. Deren zentrales Element ist eine abstrakte Zwischendarstellung (meistens ein AST eines Programms) die mit projektionalen Editoren bearbeitet und später für die Generierung von Code genutzt wird [Fow05, Fow10]. Language Workbenches sind Werkzeuge welche das "Language-oriented Programming"-Paradigma unterstützen [Fow05, Fow10, Dmi05]. Dieses setzt auf den Einsatz (und die Eigenentwicklung) von speziellen, sogenannten "Domain-Specific Languages" (DSLs) [Fow05, Fow10, Blu12].

Mit Ausnahme der projektionalen Editoren erfüllt Bauhaus alle Kriterien einer Language Workbench. Um den Fokus und die Einsatzmöglichkeiten der IML-Zwischendarstellung zu verbessern, wird mit dieser Arbeit ein modulares Erweiterungskonzept eingeführt. Dieses ermöglicht es neue IML-Konstrukte in eigenständigen Dateien zu definieren. Durch einen Präprozessor werden diese kombiniert und dem Generator *imlgen* übergeben. Der Präprozessor und seine Verwendung sind im Kapitel 5 ausführlich beschrieben. Zusätzlich wird durch diese Arbeit eine Möglichkeit geschaffen, abstrakte IML-Konstrukte durch Transformationsanweisungen auf bestehende Basiskonstrukte abzubilden. Hierfür wird in dieser Arbeit die Sprache *IMLTransform* vorgestellt.

Der zweite Teil dieser Arbeit ist die Umsetzung eines Sprachpakets, welches als Beispiel für die Verwendung des Präprozessors und der Transformationssprache dient. Zusätzlich fungiert dies als Testfall für die korrekte Umsetzung der Werkzeuge. Thema des Sprachpakets ist die Abbildung hierarchischer Zustandsautomaten. Hierfür werden die benötigten IML-Knotenklassen in einer eigenen Spezifikationsdatei angelegt und über den Präprozessor mit den Basis-Konstrukten kombiniert. Wie auch für die Abbildung von Programmcode wird ein Frontend benötigt, welche einen Zustandsautomaten in seine IML-Darstellung überführt. Modelliert werden die Zustandsautomaten mit dem UML-Editor *Papyrus*, welcher als Plugin für die Entwicklungsumgebung Eclipse zur Verfügung steht.

Nach Generierung der Zwischendarstellung des Automaten wird dieser auf die entsprechenden Basiskonstrukte abgebildet. Um die Transformationsvorschriften anwenden zu können, wird das Skript an den Transformations-Generator übergeben. Dieser erzeugt anhand der Anweisungen der Transformation ein kompilier- und ausführbares Java-Programm. Dieses Programm führt die, im Transformations-Skript spezifizierten, Manipulationen auf einem Eingabe-Graphen durch.

In Hinsicht auf die Nutzung von Bauhaus als Language Workbench wurde für den letzten Schritt ein Generator implementiert. Dieser erzeugt aus einem transformierten Zustandsautomaten-Graphen den darin abgebildeten Java-Code.

4.2 Zusammenspiel

In Abbildung 4.1 wird das Zusammenspiel der Komponenten dieser Arbeit und externer Werkzeuge dargestellt. Der Abschnitt, in dem das Zustandsautomaten-Frontend (im Schaubild als ”Frontend” bezeichnet) aufgerufen wird, kann detaillierter im Schaubild 7.2 im Kapitel 7.4 betrachtet werden.

Begonnen wird mit der Erstellung eines Zustandsautomaten im UML-Editor *Papyrus*. Dieser wird zusammen mit einem Implementierungs-Graphen an das Frontend übergeben. Dieses erzeugt einen IML-Graphen in dem der Zustandsautomat repräsentiert wird. Hierfür werden die im Zustandsautomaten-Sprachpaket definierten Knotenklassen verwendet. Deren Beschreibung und Abhängigkeiten sind im Abschnitt 6.4 zu finden.

Für die Transformation der Konstrukte auf IML-Basisknoten muss ein entsprechendes Transformations-Skript erstellt werden. In diesem wird beschrieben wie ein Graph mit Knoten des Zustandsautomaten-Sprachpakets zu verarbeiten ist. Durch den Transformations-Generator wird das Skript verwendet und ein ausführbares Java-Programm erzeugt. Dieses ist im Schaubild als ”Transformator-Code” bezeichnet. Durch Aufruf des Java-Kompilers *javac* wird der Code übersetzt. Ergebnis ist der ”Transformator”. Dieser erwartet einen IML-Graphen als Eingabe und führt die im Ausgangs-Skript definierten Manipulationen durch. Der transformierte Graph, in der Abbildung mit ”Expandierter Zustandsautomat” betitelt, wird in einer IML-Datei gespeichert.

Im letzten Schritt ist es möglich aus der expandierten Form des Zustandsautomaten wieder Code zu generieren. Hierfür wurde in dieser Arbeit das Werkzeug *unparse_j* entwickelt.

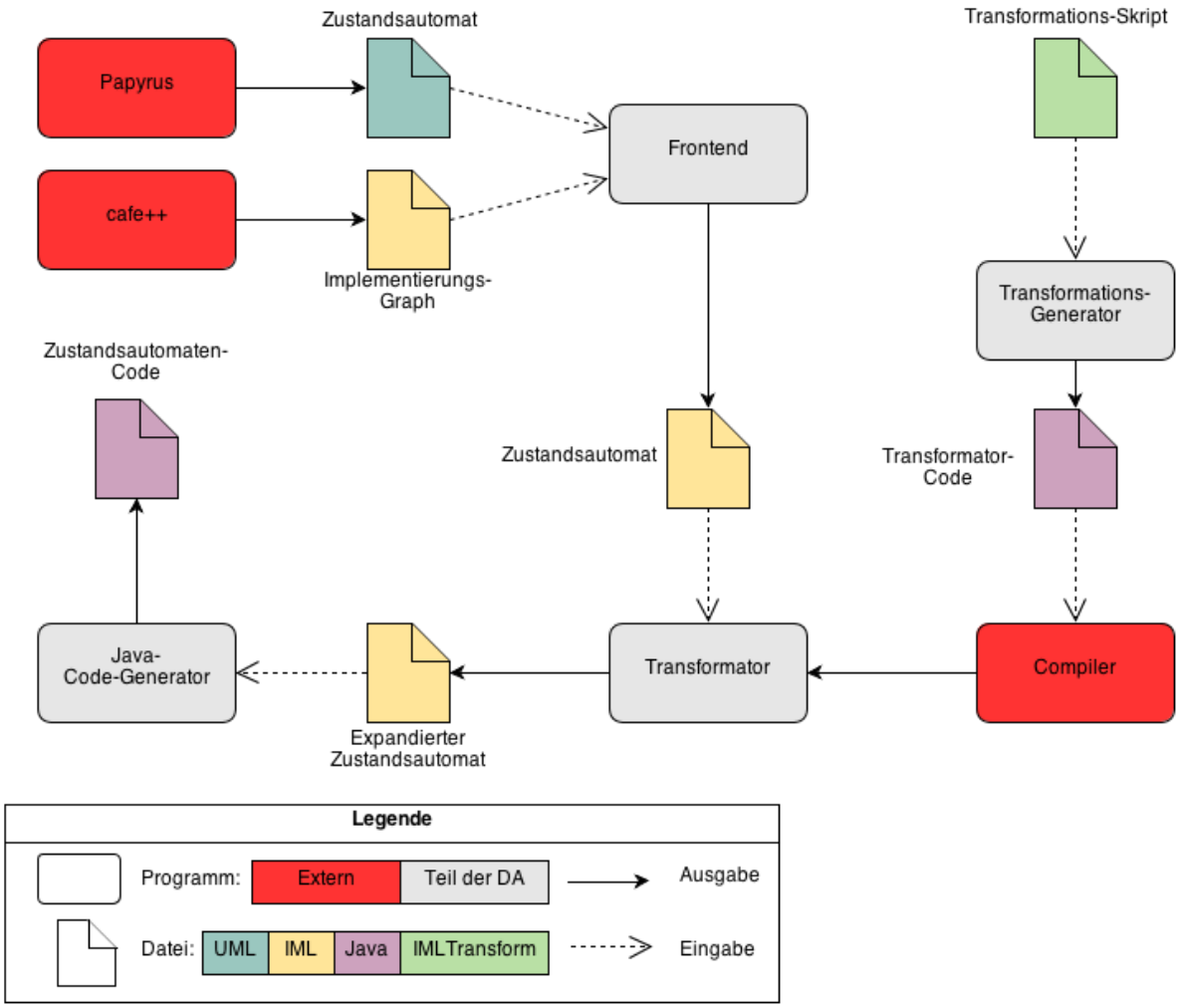

Abbildung 4.1: Zusammenspiel der Komponenten dieser Arbeit.

Dieses wird im Schaubild durch den Programm-Knoten "Java-Code-Generator" repräsentiert. Ausgabe des Generators sind Java-Klassen. Angewandt auf einen expandierten Zustandsautomaten-Graphen, werden vom Code-Generator Klassen erzeugt, deren Verhalten identisch mit der Logik des modellierten Automaten ist.

5 Sprachpaket-Präprozessor

Ziel dieser Diplomarbeit ist es, die IML Zwischensprache durch neue Konzepte zu erweitern. Dadurch soll es möglich werden auch Informationen auf höheren Abstraktionsebenen darzustellen. Um diese Erweiterungen flexibel zu gestalten, wurde ein Mechanismus entwickelt, der es erlaubt neue IML-Konstrukte modular zu definieren. Das hierfür realisierte Werkzeug, sowie seine Anwendung, werden in diesem Kapitel eingehend beschrieben.

5.1 Evaluation

Vor der Festlegung auf ein Konzept wurden die Möglichkeiten alternativer Mechanismen betrachtet. Die dabei entwickelten Lösungsansätze werden in diesem Abschnitt betrachtet.

Das Werkzeug zur Erzeugung der IML-Klassen *imlgen* ist für die Verarbeitung der Spezifikationsdatei zuständig. Alle weiteren Arbeitsschritte operieren auf den dabei entstehenden Klassen. Demnach können zwei sinnvolle und mögliche Ansatzpunkte für eine Erweiterung des IML-Erzeugungsprozesses identifiziert werden:

1. **Erweiterung der Generatoren:** Bei diesem Ansatz wird das Werkzeug *imlgen* modifiziert. Dadurch ist es möglich, komplexere Arten der Eingabe zu realisieren. Ein Beispiel hierfür wäre die Einführung eines Include-Mechanismus wie er in der Programmiersprache C eingesetzt wird. Hierfür müsste ein Präprozessor entwickelt werden, welcher den Inhalt der zu inkludierenden Zieldatei an der gewünschten Stelle einfügt. Um mehrfache Definitionen zu verhindern, würde dies weitere Makros wie `#ifndef` -Abfragen und `#define`-Anweisungen nach sich ziehen. Ein solcher Präprozessor ist dementsprechend keine triviale Aufgabe, könnte aber zusätzliche Vorteile wie bedingte Kompilierungen (z.B. für Debug-Situationen) ermöglichen. Der Einsatz eines solchen Werkzeugs ist allerdings nicht zwangsweise an die Umsetzung als Teil von *imlgen* gebunden. Er könnte ebenfalls als eigenständiges Werkzeug als Zwischenschritt vor *imlgen* geschaltet werden. Ausgabe des Präprozessors müsste dann eine einzige IML-Spezifikationsdatei sein, welche dann als Eingabe an den Generator übergeben wird. Ein Ansatz der nur auf Basis einer *imlgen*-Erweiterung realisiert werden kann, ist die Verbesserung der Eingabemöglichkeiten. Hierbei könnte *imlgen* so modifiziert werden, dass es möglich ist mehr als nur eine Spezifikationsdatei als Parameter für den Aufruf anzugeben.
2. **Hinzufügen neuer Zwischenschritte:** Grundidee dieser Lösungsstrategie ist es, weitere Werkzeuge vor *imlgen* zu schalten. Dies wurde bereits im vorangegangenen Abschnitt

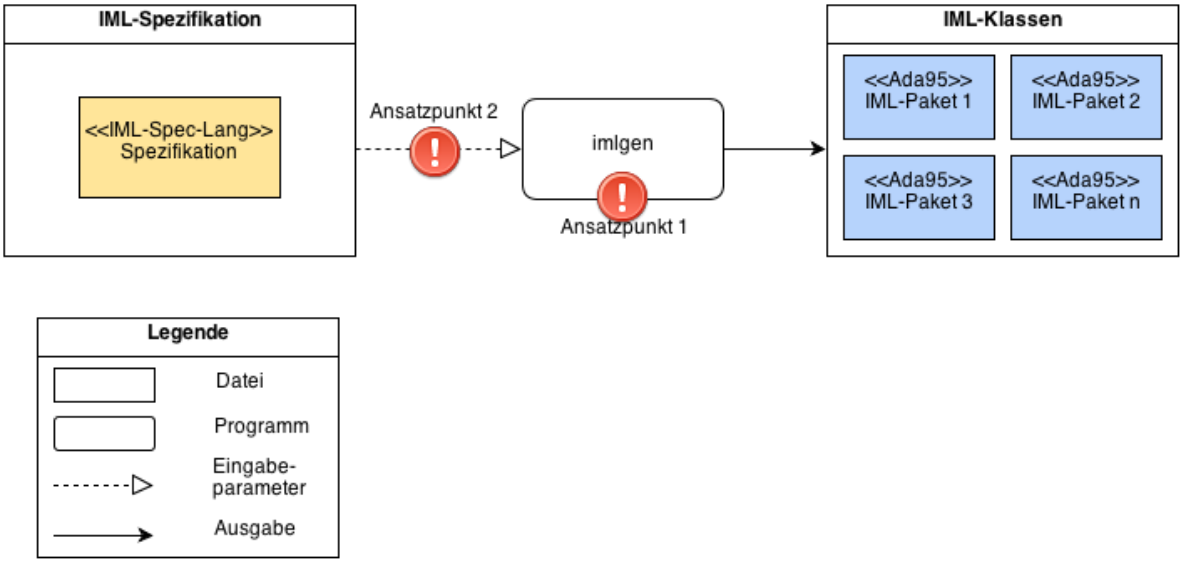

Abbildung 5.1: Arbeitsablauf der IML-Generierung vor dieser Arbeit.

für den Fall eines C-artigen Präprozessors beschrieben. Eine weitere Möglichkeit ist ein Werkzeug welches das Package-Konzept der IML-Spezifikationssprache nutzt. Dabei werden alle Spezifikationsdateien innerhalb eines bestimmten Verzeichnisses gesammelt und als separate Packages mit der Basis-Spezifikation, welche z.B. die `builtin`-Deklarationen enthält, zu einer einzelnen Datei zusammengefügt. Diese kann dann an *imlgen* übergeben werden. Vorteil dieser Kategorie ist, dass keine Modifikation an bestehenden Werkzeugen nötig ist. Zudem ist es möglich, noch weitere Programme vor den Generator zu schalten. Die Verhinderung eines einzigen, monolithischen Generators, welcher zudem in einer relativ schwach-verbreiteten Sprache wie *OCAML* geschrieben wurde, führt zu einer besseren Wartbarkeit. Dieses Konzept wurde in dieser Diplomarbeit umgesetzt.

Diese Ansatzpunkte sind in Abbildung 5.1 als rote Ausrufezeichen markiert. Sie zeigt den Verarbeitungsablauf der IML-Paketerzeugung vor Fertigstellung dieser Arbeit. Die IML-Spezifikation wird an den *imlgen*-Generator übergeben. Dieser erzeugt die Ada95-Pakete mit den IML-Knotenklassen.

Nach Einführung des neuen Werkzeugs verändert sich die Prozesskette des Erzeugungsprozesses. Wie bereits erklärt, wird das neue Programm, im Folgenden als "Merge-Präprozessor" oder nur "Präprozessor" bezeichnet, vor *imlgen* geschaltet. Die Abbildung 5.2 zeigt den neuen Ablauf. Hierbei erhält der Präprozessor zwei Eingaben: Als erstes die Basisspezifikation in der Meta-Informationen, wie Name der Spezifikation, der Root-Knotenklasse und die Deklaration der `builtins` definiert sind. Zweitens der Name eines Verzeichnisses, in dem sich die Sprachpakete befinden. Diese werden durch den Präprozessor zu einer einzelnen Datei zusammengefügt. Diese wird dann an *imlgen* übergeben, welcher daraus die entsprechenden Ada95-Pakete erzeugt.

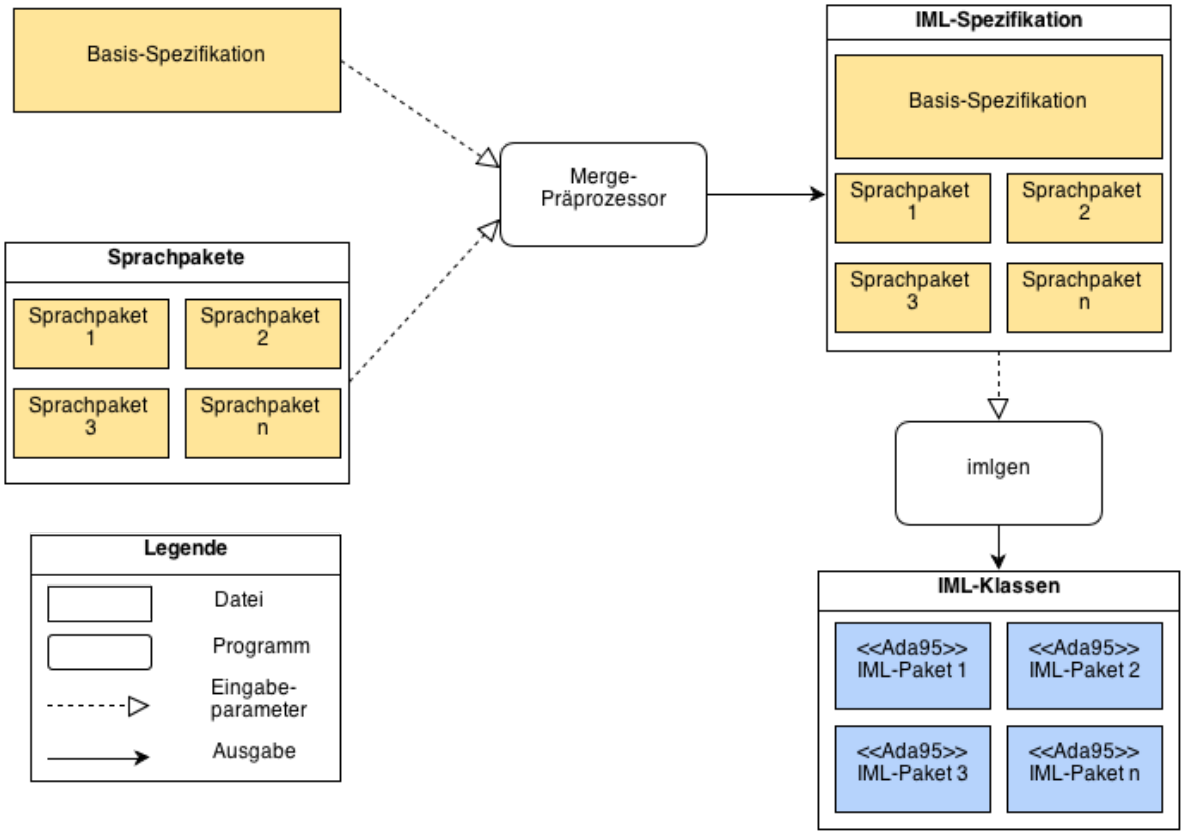

Abbildung 5.2: Arbeitsablauf der IML-Generierung nach Einführung des Merge-Präprozessors.

5.2 Entwurf und Implementierung

Aufgrund des recht simplen Aufgabenbereichs dieses Werkzeugs werden Entwurf und Implementierung in diesem Kapitel zusammengelegt. Der Merge-Präprozessor wurde in Java implementiert und nutzt das im Kapitel 3 beschriebenen Apache Commons Command Line Interface (CLI), sowie das Logging-Framework Log4j.

Die Eingabeparameter werden mit Hilfe von CLI eingelesen und in einer Instanz von `CommandLine` gespeichert. Anhand des, durch die Eingabeparameter, spezifizierten Ausgabepfads wird ein `OutputStream` erzeugt. In diesen werden alle Inhalte der Basisspezifikation und der Sprachpakete geschrieben. Da die Basisspezifikation den "Kopf" der Spezifikation enthäl,t wird diese als erstes geschrieben. Im Anschluss daran durchsucht der Präprozessor das, ebenfalls durch einen Kommandozeilenparameter definierte, Paket-Verzeichnis. Alle dabei gefundenen, passenden Dateien werden mit Hilfe eines `FileInputStreams` geöffnet. Jeglicher gelesener Inhalt der Dateien wird in den Ausgabestream geschrieben. Nachdem alle Sprachpakete verarbeitet wurden wird der Stream für die Ausgabedatei geschlossen und der Merge-Präprozessor beendet sich.

Treten Fehler bei der Ausführung des Präprozessors auf, z.B. durch Angabe falscher Eingabeparameter, so wird der Vorgang abgebrochen und das Werkzeug mit einer Fehlermeldung beendet. Werden keine Sprachpakete im angegebenen Verzeichnis gefunden, so ist die Ausgabe identisch mit der Basisspezifikation.

5.3 Anwendung

Der Merge-Präprozessor wird als eigenständige Anwendung vor der Generierung der IML-Klassen durch *imlgen* aufgerufen. Hierfür kann er in den Build-Prozess für Bauhaus integriert werden. Denkbar wäre auch ein Aufruf aus *imlgen* heraus. In letzterem Fall müsste das in Python geschrieben Startskript *imlgen.py* angepasst werden. Dadurch könnte sichergestellt werden, dass der Präprozessor in jedem Anwendungsszenario vor Ausführung von *imlgen* aufgerufen wurde.

Für den Merge-Präprozessor wird ein eigenes Startskript bereitgestellt. Dieses wird benötigt, um die Bibliotheken für die Ausführung als Classpath zu markieren. Beim Aufruf des Make-Prozesses wird dieses Startskript in das Bauhaus-Tools-Verzeichnis kopiert. Dadurch ist es in der Shell direkt aufrufbar.

Die Aufrufparameter sind im Listing 5.1 dargestellt.

```
1 usage: imlmergepreprocessor.sh *args*
2 Merges the basic iml specification with multiple extension files and
3 stores the result in a single file.
4  -d <arg>  Denotes the path in which the iml extension files reside.
5  -e <arg>  Specifies the extension of the iml extension files (default: .spec).

6  -o <arg>  Specifies where the merged ouput file should be stored.
7  -s <arg>  Specifies the file with the basic IML specifications (e.g. iml1.
       spec).
8  -v        Toggles the verbose mode.
```

Listing 5.1: Aufruf des Merge-Präprozessor-Startskripts.

Pflichtparameter sind dabei:

- **-d:** Angabe des Verzeichnisses in dem die IML Sprachpakete liegen.
- **-s:** Pfad zur Basisspezifikationsdatei.
- **-o:** Ausgabepfad für Ergebnis des Präprozessors.

6 Beispiel-Sprachpaket: Zustandsautomaten

Um die Funktionalität der einzelnen Bestandteile zu testen, wurde ein Musterbeispiel für ein Sprachpaket erstellt. In diesem Abschnitt wird dieses Beispiel sowie die verwendeten Notationen und Werkzeuge beschrieben. Zukünftige Erweiterungen können analog zu diesem Sprachpaket erzeugt werden.

Aufgabe des Sprachpakets ist die Abbildung von hierarchischen Zustandsautomaten in IML. Hierfür werden die benötigten Knotentypen in der IML-Spezifikationssprache verfasst. Nach Ausführung der Generator-Werkzeuge des Bauhaus Frameworks stehen diese als Teil der IML Zwischensprache zur Verfügung. Durch ein spezielles Frontend werden Zustandsautomaten in ihre äquivalente IML-Darstellung überführt und mit Funktionsimplementierungen versehen. Durch die Transformations-DSL werden die Konstrukte des Zustandsautomaten auf bestehende Basis-Knotenklassen der IML abgebildet. Abschließend erzeugt der Java-Code-Generator aus dem transformierten IML-Graph ausführbaren Java-Code. Das Verhalten des erzeugten Codes bei Ausführung ist funktionsidentisch mit dem modellierten Ausgangsautomaten.

6.1 Hierarchische Zustandsautomaten

Zustandsautomaten sind eine Notationsform für den Entwurf von zustandsabhängigen Systemen [Mar98]. Die Modellierung solcher Automaten ist Teil der Unified Modelling Language. Besonders häufig findet diese Form des Entwurfs Anwendung im Bereich der Hardware-Entwicklung, ist aber nicht auf diesen beschränkt.

Ein wichtiges Prinzip der Zustandsautomaten ist die sogenannte *Inversion of Control* [Sam09]. Bei diesem Prinzip wird die Steuerung "nach außen" verlagert. Das System befindet sich dabei in einem Wartezustand und reagiert erst sobald es einen Impuls erhält. Im Falle der Zustandsautomaten wird ein solcher Impuls durch *Events* symbolisiert. Man spricht daher auch von einem *Event-driven System* [Sam09]. Ausgelöst werden können Events durch verschiedene Ereignisse, wie die Betätigung einer Schaltfläche, den Empfang eines Daten-Pakets oder den Druck einer Taste. Das System reagiert auf die Events in Abhängigkeit seines aktuellen Zustands. Beispiele hierfür sind Anwendungen mit grafischer Oberfläche, bei denen Events durch die Bedienung des Benutzers ausgelöst werden.

Man unterscheidet zwei grundlegende Formen der Zustandsautomaten. Endliche Zustandsautomaten (FSM [1]) befinden sich zu jedem Zeitpunkt in genau einem Zustand. Eine Verschachtlung von Zuständen ist demnach nicht möglich. Alle Schritte des Systems werden auf einer Ebene dargestellt. Mit steigender Komplexität werden FSMs daher häufig unübersichtlich. Bei hierarchischen Zustandsautomaten (HSM [2]) ermöglichen es Zustände zu Gruppieren, indem man Unterzustände innerhalb eines Zustands definiert. Ein HSM kann sich in mehreren Zuständen gleichzeitig befinden. Ein empfangenes Event wird dabei von allen aktiven Zuständen behandelt. Dies beginnt bei der tiefsten Ebene der Verschachtelung. Wird das Event vom aktuellen Zustand nicht verarbeitet, so wird es an den nächsten übergeordneten Zustand weitergegeben.

Ein Zustandsautomat lässt sich als Programm darstellen. Hierfür existieren verschiedene Techniken zur Realisierung. Diese lassen sich in vier Kategorien unterteilen [Sam09]:

- Realisierung durch `Switch`-Konstrukte.
- Realisierung durch eine Zustands-Tabelle.
- Realisierung durch das `State`-Entwurfsmuster ([Gam95]).
- Realisierung durch Kombination der obigen Punkte.

Da eine vollständige Abdeckung der Realisierungsmöglichkeiten den Rahmen dieser Arbeit sprengen würde, werden hier nur die Umsetzung mit `Switch` und dem `State`-Pattern beschrieben. Die in dieser Arbeit umgesetzte Lösung nutzt eine Kombination dieser beiden Techniken. Bei Interesse sei dem Leser die Arbeit in [Sam09] empfohlen. Dort findet sich eine ausführliche Beschreibung aller oben genannten Realisierungen.

Realisierung mit `Switch`:

Diese Form ist die einfachste der oben genannten Möglichkeiten. Sie ist leicht nachvollziehbar, kann aber bei einer größeren Menge von Zuständen die Lesbarkeit des Codes verschlechtern. Kernstück ist ein großes `Switch`-Konstrukt, welches für jeden Zustand des Automaten eine Fallunterscheidung enthält. Bei Verarbeitung eines Events wird dieses entsprechend dem aktuellen Zustand behandelt. Dazu enthält jede Unterscheidung eine weitere `Switch`-Anweisung. Für jedes Event, welches mit einer ausgehenden Transition verknüpft ist, wird eine Unterscheidung in diesem `Switch`-Konstrukt definiert.

Als Beispiel soll ein einfacher Zustandsautomat mit zwei Zuständen dienen. Abbildung 6.1 zeigt diesen Automaten. Vom Zustand ”ON” kann durch eine Transition in den Zustand ”OFF” übergegangen werden. Dabei wird die Funktion `save` aufgerufen. Ebenso ist es möglich von ”OFF” zurück zu ”ON” zu wechseln. Dies löst die Funktion `load` aus. Jeder dieser Transitionen sei hierfür ein auslösendes Event zugeordnet. Die Zuordnung wird in Tabelle 6.1 dargestellt.

[1] Vom Englischen: Finite State Machines
[2] Vom Englischen: Hierarchical State Machines

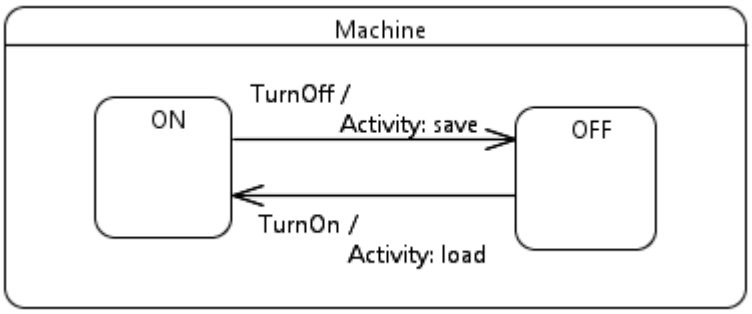

Abbildung 6.1: Einfaches Beispiel eines Zustandsautomaten mit zwei Zuständen.

Transition	Event
TurnOn	`ePowerOn`
TurnOff	`ePowerOff`

Tabelle 6.1: Zuordnung der Transitionen zu ihren Events für das `Switch`-Beispiel.

Eine Umsetzung eines solchen Automaten in Java durch eine Verschachtlung von `Switch` -Anweisungen wird im Code-Ausschnitt 6.1 gezeigt. Zu Gunsten der Übersichtlichkeit wird nur die Methode `handleEvent` gezeigt. Diese ist für die Verarbeitung von Ereignissen zuständig.

```
1  public void handleEvent(Event e) {
2
3    switch(this.currentState) {
4        case STATE_ON:
5            switch(e) {
6                case ePowerOff:
7                    this.save();
8                    this.changeState(STATE_OFF);
9                    break;
10           }
11           break;
12
13       case STATE_OFF:
14           switch(e) {
15               case ePowerOn:
16                   this.load();
17                   this.changeState(STATE_ON);
18                   break;
19           }
20           break;
```

```
21      }
22  }
```

Listing 6.1: Umsetzungsbeispiel für den Zustandsautomaten in Abbildung 6.1 mit `Switch`.

Die Effekt-Methoden `save` und `load` sind hier als Member der übergeordneten Klasse implementiert. Für den Übergang zu einem neuen Zustand wird die Methode `changeState` bereitgestellt. Diese stellt sicher, dass beim Wechsel eventuell vorhandene Exit- und Entry-Funktionen aufgerufen werden. Der aktuelle Zustand des Automaten wird in der Variable `currentState` gespeichert.

Realisierung mit dem `State`-Muster:

Eine weitere Möglichkeit zur Implementierung von Zustandsautomaten ist die Verwendung des `State`-Entwurfsmusters der *Gang of Four* (siehe [Gam95]). Für den Einsatz dieser Technik existieren mehrere Variationen. Zur Erläuterung wird hier die in [Sam09] verwendete Umsetzung benutzt.

Grundlegende Idee dieser Realisierungstechnik ist es, jeden Zustand in eine eigene Klasse auszulagern. Diese leiten sich von einer gemeinsamen Basisklasse ab, welche eine virtuelle Methode für jedes Event bereitstellt. Verarbeitet ein Zustand ein Event, so implementiert er die entsprechende Methode. Die Klasse des Zustandsautomaten speichert den aktuellen Zustand in einer Klassenvariable vom Typ der Zustandsbasisklasse. Durch *Dynamic Dispatching* wird beim Auftreten eines Events die entsprechende Implementierung der Event-Methode aufgerufen.

Als Beispiel wird der Zustandsautomat in 6.1 mit den Events aus Tabelle 6.1 verwendet.

Der Code-Ausschnitt in 6.2 zeigt wie sich der Zustandsautomat durch Einsatz des `State`-Patterns realisieren lässt.

```
1  public class Machine {
2      //Declaration of methods: save, load, changeState
3
4      private MachineState currentState;
5
6      class MachineState {
7          public void onPowerOff(Machine ctx) { };
8          public void onPowerOn(Machine ctx) { };
9      }
10
11     class MachineStateOn extends MachineState {
12         @Override
13         public void onPowerOff(Machine ctx) {
14             ctx.save();
15             ctx.changeState(MachineStateOff.class);
```

```
16          }
17      }
18
19      class MachineStateOff extends MachineState {
20          public void onPowerOn(Machine ctx) {
21              ctx.load();
22              ctx.changeState(MachineStateOn.class);
23          }
24      }
25
26      public void onPowerOn() {
27          this.currentState.onPowerOn(this);
28      }
29
30      public void onPowerOff() {
31          this.currentState.onPowerOff(this);
32      }
33  }
```

Listing 6.2: Umsetzungsbeispiel für den Zustandsautomaten in 6.1 mit `State`-Pattern.

Wie im vorherigen Umsetzungsbeispiel, wird der Zustand des Automaten in der Variable `currentState` gespeichert. Anders als bei der `Switch`-Variante, handelt es sich hier um ein Objekt, welches eine Instanz der aktuellen Zustandsklasse speichert. Der Zustandsautomat stellt als Schnittstelle eine Methode für jeden Event-Typ zur Verfügung. In diesen Methoden wird der Aufruf dann an den aktuellen Zustand weitergegeben. Der Parameter `context` übergibt dabei die aktuelle Instanz des Zustandsautomaten an die Methode. Der Wechsel der Zustände wird durch einen Aufruf von `changeState` vollzogen.

Realisierung in dieser Arbeit:

Nach Vorstellung der Umsetzung durch `Switch`-Konstrukte und `State`-Patterns soll an dieser Stelle die Technik, welche in dieser Arbeit umgesetzt wurde, beschrieben werden. Bei der Entscheidung waren dabei einige Schwierigkeiten zu beachten.

- **Lesbarkeit:** Die verwendete Umsetzung soll möglichst lesbar und verständlich sein. Dies ist wichtig, da sie das Ergebnis des Java-Code-Generators mitbestimmt. Zur Überprüfung der korrekten Funktionsweise wird der erzeugte Code betrachtet. Wäre die gewählte Technik sehr komplex, so würde dies zwangsweise zu einer komplexen Ausgabe des Java-Code-Generators führen.
- **Einfachheit:** Durch den Einsatz der Transformationssprache sollen die IML-Konstrukte des Zustandsautomaten-Sprachpakets auf Basiskonstrukte abgebildet werden. Daher führt eine komplexe Lösung für die Implementierung der Zustandsautomaten zu sehr komplexen Transformationsvorschriften. Darstellungen in IML verfügen selbst

bei kleinen Programmen über eine Vielzahl von Knoten die auf verschiedene Weisen miteinander verknüpft sind. Mit der Wahl einer möglichst simplen Lösung soll der Aufwand bei der Transformation gering gehalten werden.

- **Realisierung mit nativen Mitteln:** Teil der Aufgabenstellung dieses Projekts ist der bereits erwähnte Java-Code-Generator (s. Kapitel 9). Dieser ist so implementiert, dass er nur eine Teilmenge der IML-Knotenklassen behandelt. Dies begründet sich darin, dass eine vollständige Umsetzung in der vorgegebenen Zeit nicht machbar wäre. Aus dieser Beschränkung folgt allerdings, dass keine Standard-Libraries eingesetzt werden können. Deren Inhalt müsste bei Verwendung mit in den IML-Graphen integriert und folglich vom Java-Code-Generator behandelt werden. Um deren Funktionalität vollständig abzudecken, müssten nahezu alle IML-Knotenklassen im Code-Generator abgedeckt werden. Um dies zu vermeiden, wurde eine Lösung entwickelt, die auf zusätzliche Bibliotheken verzichtet.

Als Resultat dieser Überlegungen wurde für die Umsetzung des Zustandsautomaten eine Kombination aus `Switch`-Anweisungen und `State`-Patterns gewählt. Anstatt eine mehrfache Verschachtlung von `Switch`-Anweisungen zu verwenden, wird für jeden Zustand eine `State` -Klasse angelegt. Diese leitet sich von einer gemeinsamen, abstrakten Vaterklasse ab. Durch diesen Schritt wird die äußerste `Switch`-Anweisung, welche den aktuellen Zustand behandelt, eliminiert. Die untergeordneten `Switch`-Konstrukte, welche die Events der Zustände behandeln, werden in eine spezielle `handleEvent`-Methode in die konkreten `State`-Klassen verlagert. Im Code-Ausschnitt 6.3 ist die Vaterklasse für Zustände in C++ dargestellt.

```
1    class State
2    {
3        protected:
4            Machine* ctx;
5
6        public:
7            void setContext(Machine* context);
8            virtual HandleResponse handleEvent(Event evt);
9            virtual State* getParent();
10           virtual void exitAction();
11           virtual void entryAction();
12           virtual void enterState(bool subFinal);
13   };
14 }
```

Listing 6.3: Vaterklasse für Zustände in der gewählten Zustandsautomaten-Realisierung.

Die aktuelle Instanz des Zustandsautomaten wird als `protected`-Membervariable gespeichert. Dadurch ist es nicht mehr nötig, den Kontext an jede der Funktionen als Parameter zu übergeben. Im Folgenden werden die virtuellen Funktionen beschrieben, welche in den konkreten Zustandsklassen überschrieben werden.

- `handleEvent`: Verarbeitet eingehende Events. Enthält ein `Switch`-Konstrukt, welches die behandelten Events enthält. Der Rückgabewert signalisiert ob ein Event verarbeitet wurde oder ob es an einen übergeordneten Zustand weitergereicht werden soll.
- `getParent`: Diese Funktion gibt den direkt übergeordneten Zustand zurück.
- `exitAction`: Ruft eine eventuell vorhandene Exit-Funktion auf.
- `entryAction`: Ruft eine eventuell vorhandene Entry-Funktion auf.
- `enterState`: Wird benötigt wenn der Zustand ein *Composite-State* mit Unterzuständen ist. Sie stellt sicher dass der korrekte Zustand aktiviert wird. Der übergebene Parameter zeigt an ob der Zustand als Resultat eines erreichten `Final`-Pseudozustand betreten wird. In diesem Fall darf nicht wieder direkt in den initialen Unterzustand gewechselt werden.

Die Verarbeitung von Events wird durch eine zentrale Funktion des Zustandsautomaten realisiert. Dies ähnelt der Umsetzung der `Switch`-Variante. Der Code in Ausschnitt 6.4 zeigt die Implementierung der `dispatch`-Methode, welche die ausgelösten Ereignisse verarbeitet.

```
1    void ATM::dispatch(Event e)
2    {
3        State* current = this->currentState;
4        while( current->handleEvent(e) == NOT_HANDLED )
5        {
6            current = current->getParent();
7            if( current == null ) { break; }
8        }
9    }
```

Listing 6.4: Vaterklasse für Zustände in der gewählten Zustandsautomaten-Realisierung.

Durch die `while`-Schleife wird sichergestellt, dass das Event solange an einen Vaterzustand weitergereicht wird, bis es verarbeitet wurde oder kein solcher Zustand mehr existiert. Ob ein Zustand ein Event eines bestimmten Typs verarbeitet, wird durch die `handleEvent`-Methode bestimmt. Ein Beispiel für deren Implementierung findet sich in Listing 6.5.

```
1  Machine::HandleResponse Machine::State_Menu::handleEvent(Event evt)
2  {
3      switch(evt)
4      {
5          case EVENT_BUTTONSHOWBALANCE:
6              ctx->changeState(ctx->stateShowBalance, false);
7              return HANDLED;
8
9          case EVENT_BUTTONDRAWOUT:
10             if( verifySomething() ) {
11                 //Execute transition effect function here..
```

```
12                ctx->changeState(ctx->stateInput, false);
13            }
14            return HANDLED;
15    }
16    return NOT_HANDLED;
17 }
```

Listing 6.5: Vaterklasse für Zustände in der gewählten Zustandsautomaten-Realisierung.

Der Zustand `State_Menu`, dessen `handleEvent`-Methode als Beispiel verwendet wird, behandelt zwei Events: `EVENT_BUTTONSHOWBALANCE` und `EVENT_BUTTONDRAWOUT`. Eine eventuell vorhandene Guard-Bedingung wird als `If`-Abfrage umgesetzt. Löst das Event einen Zustandswechsel aus, so wird dieser durch einen Aufruf der `changeState`-Methode erreicht.

Der Wechsel von Zuständen ist die komplizierteste Aufgabe in hierarchischen Zustandsautomaten. Es muss darauf geachtet werden, dass beim Verlassen alle Exit-Funktionen in der richtigen Reihenfolge aufgerufen werden. Selbiges gilt analog für die Entry-Methoden beim Eintritt in den neuen Zustand. Dazu muss erkannt werden, in welchem Zweig sich der aktuelle sowie der Ziel-Zustand befinden. In Code-Ausschnitt 6.6 wird die Implementierung der `changeState`-Methode dargestellt.

```
1  void ATM::changeState(State* target, bool subFinal)
2  {
3      //Arrays which store all exited and entered states:
4      State** sourcePath = new State*[256];
5      State** targetPath = new State*[256];
6
7      int sourceIdx = 0;
8      int targetIdx = 0;
9
10     //Walk from current state to root:
11     State* current = this->currentState;
12     while( current != null )
13     {
14         sourcePath[sourceIdx++] = current;
15         current = current->getParent();
16     }
17
18     //Walk from target state to root:
19     current = target;
20     while( current != null )
21     {
22         targetPath[targetIdx++] = current;
23         current = current->getParent();
24     }
25
```

```
26    //Compare the paths and remove all states which are found in both paths:
27    for( int i = sourceIdx; i >= 0; i--)
28    {
29        for(int j = 0; j <= targetIdx; j++)
30        {
31            if( sourcePath[i] == targetPath[j] )
32            {
33                sourcePath[i] = null;
34                targetPath[j] = null;
35            }
36        }
37    }
38
39    //Fire the exit actions:
40    for(int i = 0; i <= sourceIdx; i++)
41    {
42        if(sourcePath[i] != null ) {
43            sourcePath[i]->exitAction();
44        }
45    }
46
47    //Fire the entry actions:
48    for(int i = 0; i <= targetIdx; i++)
49    {
50        if(targetPath[i] != null ) {
51            targetPath[i]->entryAction();
52        }
53    }
54
55    if(target == null)
56    {   //If null target simply set it:
57        this->currentState = null;
58    }
59    else
60    {
61        this->currentState = target;
62        target->enterState(subFinal);
63    }
```

Listing 6.6: Implementierung der `changeState`-Methode für den Wechsel von Zuständen.

Die Methode läuft dabei zuerst vom aktuellen Zustand über alle übergeordneten Zustände hin zur Wurzel. Dabei werden passierte Zustände im Array `sourcePath` gespeichert. Dies wird analog für den Pfad vom Ziel-Zustand zur Wurzel durchgeführt. Diese werden im Array `targetPath` hinterlegt. Im Anschluss wird verglichen, welche Zustände in beiden Arrays

vorhanden sind. Diese werden aus beiden Arrays entfernt. Die übrig geblieben Zustände in `sourcePath` werden verlassen und ihre Exit-Action ausgeführt. Die Einträge in `targetPath` werden betreten und ihre Entry-Action aufgerufen. Abschließend wird der Zielzustand durch den Aufruf von `enterState` betreten.

6.2 UML-Notation

Die Zustandsautomaten, welche durch das Zustandsautomaten-Frontend nach IML abgebildet werden, sind als UML Diagramm modelliert. Diese entsprechen der offiziellen Notation nach der OMG Spezifikation [OMG11].

UML stellt Entwicklern von Software Werkzeuge für die Analyse, den Entwurf und die Realisierung von Systemen bereit [OMG11]. Inspiration und Einfluss während der Entstehung von UML waren verschiedene objektorientierte Entwurfs-, Architektur- und Programmier-Methoden. Darunter waren zum Beispiel Booch[3], OMT [4] und OOSE [5].

Seit seiner Einführung hat sich UML zu einem weitverbreiteten Industriestandard entwickelt, welcher die Kompatibilität der Modellierungswerkzeuge fördert. Dies setzt voraus, dass diese Werkzeuge die durch UML definierten Regeln in Bezug auf Semantik und Notation einhalten [OMG11]. UML stellt dafür ein Meta-Modell für die abstrakte Syntax, eine Beschreibung der Semantik und eine Spezifikation der Notationen für die einzelnen Modell-Konzepte [OMG11].

Die im Abschnitt 6.12 beschriebenen hierarchischen Zustandsautomaten sind nur ein kleiner Teil der Modellierungsaufgaben, welche mit UML realisiert werden können. Durch den modularen Aufbau ist es möglich, nur die Bestandteile von UML zu verwenden, die benötigt werden [OMG11]. Im Falle dieser Diplomarbeit beschränkt sich der Einsatz von UML auf das Konzept der (hierarchischen) Zustandsautomaten.

StateMachine: Zustandsautomaten werden als Rechteck mit abgerundeten Ecken dargestellt. Der Name befindet sich zentriert am oberen Rand in einer abgetrennten Titelleiste. Dargestellt ist ein Beispiel in Abbildung 6.2.

[3] G. Booch: *Object-oriented Analysis and Design with Applications, 3rd edition*, Addison-Wesley 2007, ISBN 0-2018-9551-X

[4] J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, W. Lorensen: *Object-Oriented Modeling and Design*, Prentice Hall 1990, ISBN 0-13-629841-9

[5] I. Jacobson, M. Christerson, P. Jonsson: *Object-Oriented Software Engineering - A Use Case Driven Approach, Addison-Wesley*, 1992, ISBN 0-2015-4435-0

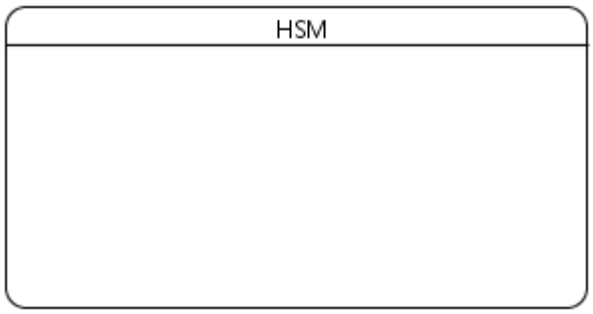

Abbildung 6.2: Leerer Zustandsautomat in Papyrus.

State: Ein Zustandsautomat kann alle darin befindlichen Zustände annehmen. Diese werden als Rechteck mit abgerundeten Ecken notiert. Der Name des Zustands steht innerhalb des Rechtecks. Das Notationsbeispiel in Abbildung 6.3 zeigt einen Zustandsautomaten mit zwei Zuständen.

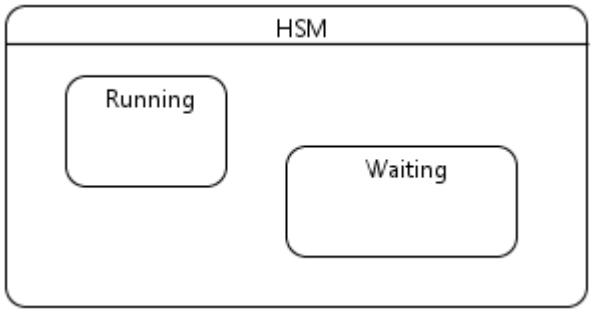

Abbildung 6.3: Zustandsautomat mit zwei Zuständen.

Verfügt ein Zustand über Entry- oder Exit-Aktionen, so werden diese ebenfalls innerhalb des Zustands gelistet. Dabei wird folgendes Format verwendet:

```
/[entry | exit] Activity *FunktionsName*
```

Zustände können als sogenannte *Composite-States* weitere (Pseudo-)Unterzustände und Transitionen enthalten. Diese werden dann, wie bei der Notation von *StateMachines*, innerhalb des Rechtecks des Zustands dargestellt. Durch *Composite-States* ist es möglich, dass ein Zustandsautomat in mehreren Zuständen gleichzeitig ist. Abbildung 6.4 zeigt einen *Composite-State* "Running" mit zwei Unterzuständen "Reading" und "Writing". Sobald ein Zustand durch Hinzufügen von Unterelementen zu einem *Composite-State* wird, verändert

sich sein Aussehen. Statt einem einfachen Rechteck wird sein Aussehen identisch mit dem des *StateMachine*-Elements.

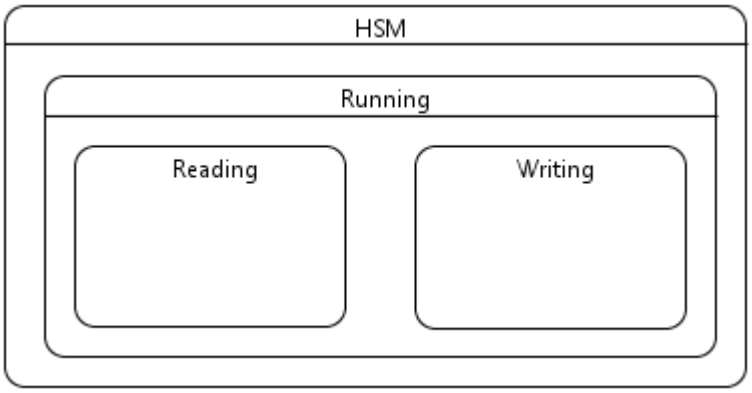

Abbildung 6.4: Zustandsautomat mit zwei Zuständen.

Transition: Transitionen bilden Übergänge zwischen Zuständen. Sie werden als gerichtete Kanten durch Pfeile repräsentiert. Transitionen können ebenfalls mit Namen versehen werden. Zudem ist es möglich ihnen eine Bedingung (engl: Guard) und einen Effekt (engl: Effect oder Activity) zuzuordnen. Durch die Guard kann sichergestellt werden, dass eine Transition nur unter bestimmten Bedingungen aktiviert werden kann. Ausgelöst werden Transitionen auf zwei Weisen. Im ersten Fall ist der Transition ein Event zugeordnet. Wird ein Event dieses Typs an den übergeordneten Zustandsautomaten gesendet, so aktiviert dieser die Transition. Dabei wird eine eventuell vorhandene Guard geprüft. Ist einer Transition kein Event zugeordnet, so bezeichnet man es als "Immediate Transitions". Diese werden aktiviert, sobald der Zustand, von dem sie ausgehen, betreten wird. Abbildung 6.5 zeigt zwei Zustände, welche durch Transitionen miteinander verbunden sind.

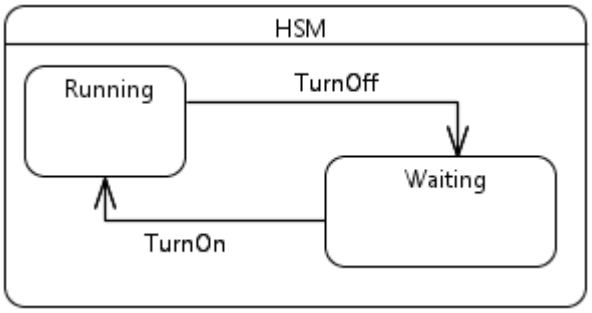

Abbildung 6.5: Zwei Zustände eines Zustandsautomats, verbunden durch Transitionen.

Pseudozustand-Choice: `Choice`-Elemente werden als Diamant dargestellt. Dieser darf genau eine eingehende und beliebig viele ausgehende Transitionen besitzen [OMG11]. Der Name wird neben dem Element vermerkt.

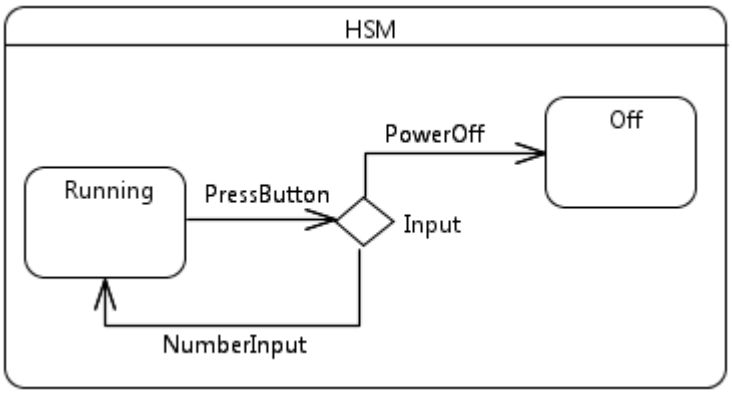

Abbildung 6.6: Zustandsautomat mit zwei Zuständen und einem *Choice*-Pseudostate.

Pseudozustand-Initial: Der Eintrittspunkt in einen Zustandsautomaten oder einen *Composite-State* wird durch einen ausgefüllten, schwarzen Punkt dargestellt. Neben einem Bezeichner muss dieser genau eine ausgehende "Immediate Transition" ohne Guard besitzen.

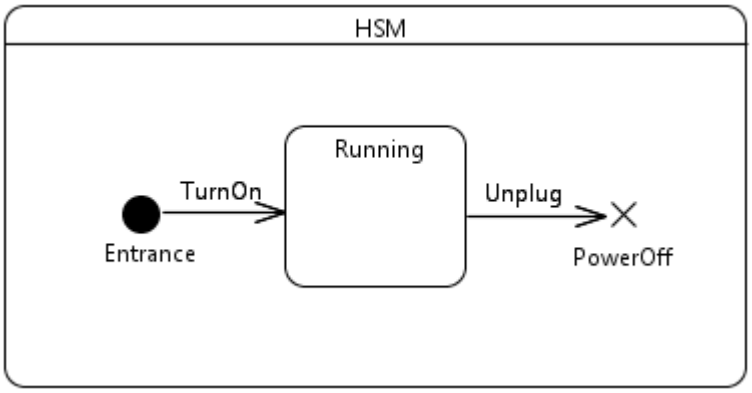

Abbildung 6.7: Zustandsautomat mit einem `Initial`-Pseudozustand.

Pseudozustand-Junction: Sollen Transitionen auf eine oder mehrere Transitionen abgebildet werden, so wird dies mit einer `Junction` realisiert. Diese wird, wie `Initials` durch einen schwarzen Punkt dargestellt.

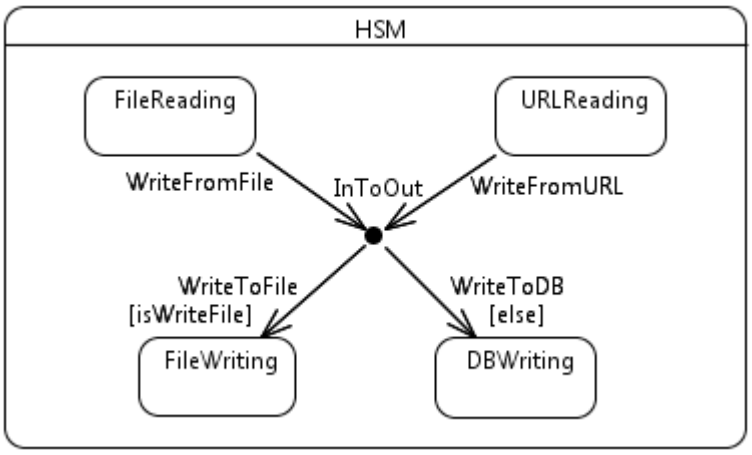

Abbildung 6.8: Verbindung mehrerer Zustände durch eine `Junction`.

Pseudozustand-Terminate: Ein `Terminate`-Pseudozustand stellt einen unerwarteten Abbruch der Ausführung dar. Die Notation für einen solchen Zustand ist ein Kreuz, welches mit dem Namen des Pseudozustands gekennzeichnet wird. Von einem `Terminate` darf es logischerweise keine ausgehenden Transitionen geben.

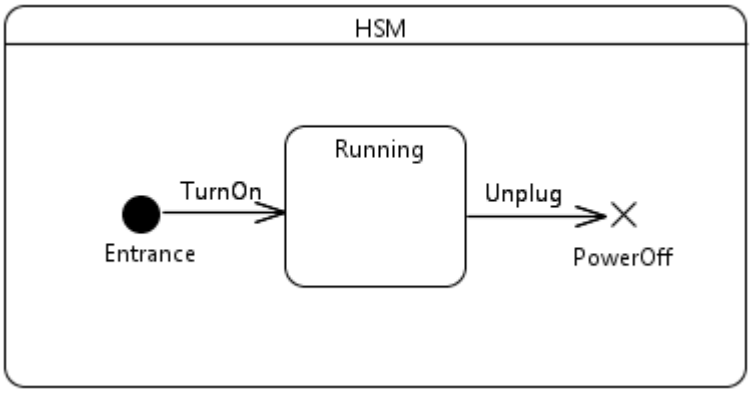

Abbildung 6.9: Zustandsautomat mit einem Zustand, sowie `Initial-` und `Terminate-`Pseudozuständen.

FinalState: Durch `FinalStates` kann die planmäßige Beendigung eines Zustandsautomatens oder eines `Composite-States` modelliert werden. Als Notation wird ein Kreis, welcher von einem Ring umgeben ist, verwendet. `FinalStates` können keine ausgehende Transitionen besitzen.

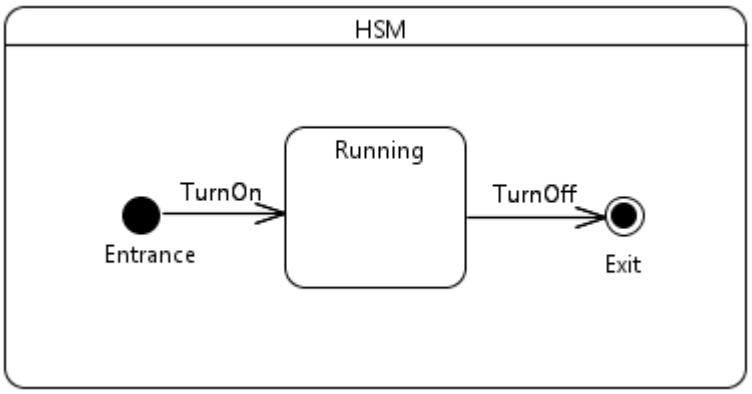

Abbildung 6.10: Zustandsautomat mit einem Zustand, sowie `Initial-` und `FinalState`-Pseudozuständen.

6.3 Papyrus UML Editor

Für die Modellierung der Zustandsautomaten wird ein UML-Werkzeug eingesetzt. Da für die Implementierung der Projekte die Entwicklungsumgebung Eclipse[6] eingesetzt wird, bietet sich der Einsatz eines Editor-Plugins für eben diese Umgebung an. Einer der aktivsten Editoren (fast 400 Commits in November und Dezember 2012) ist das *Papyrus*-Projekt. Als Teil des *Model Development Tools* (MDT) Projekts ermöglicht es die visuelle Bearbeitung verschiedener Modellierungssprachen wie UML 2, SysML und MARTE [G+]. Neben dem Entwurf von Diagrammen können mit *Papyrus* benutzer-spezifische Editoren für UML2-basierte DSLs definiert werden [G+].

Die durch *Papyrus* modellierten UML-Diagramme basieren auf der offiziellen Spezifikation der Object Management Group (OMG) [OMG11]. Ziel ist dabei eine vollständige Abdeckung derer Vorgaben [G+].

Neben der grafischen Bearbeitung ist es möglich, Modelle durch textuelle Editoren zu verändern. Diese stellen neben Syntax-Highlighting auch kontext-basierte Vervollständigungsmechanismen bereit [G+].

Zum Zeitpunkt dieser Arbeit war die Version 0.9.0 der zuletzt veröffentlichte Release. *Papyrus* ist Teil des vorkonfigurierten *Eclipse Modelling Package*.

[6] http://www.eclipse.org/

6.3.1 Installation

Wie bereits im vorangegangenen Abschnitt beschrieben, existiert eine Eclipse Version in der Papyrus vorinstalliert ist. Hier für muss diese Version lediglich heruntergeladen und in das gewünschte Ziel entpackt werden.

Soll Papyrus nachträglich für ein bestehendes Eclipse installiert werden, so muss lediglich das *Update Repository* zu den Software-Quellen hinzugefügt werden. Das Release-Repository für Eclipse Juno 4.2 findet sich unter `http://download.eclipse.org/modeling/mdt/papyrus/updates/releases/maintenance/juno`. Nach Hinzufügen der Quelle muss das Paket "Papyrus SDK Binaries" installiert werden. Nach Abschluss der Installation muss Eclipse neugestartet werden.

6.4 IML-Knoten

In diesem Kapitel werden die IML-Knoten beschrieben, welche für die Abbildung von hierarchischen Zustandsautomaten benötigt werden. Diese sind an die offizielle UML-Klassenhierarchie nach der OMG Spezifikation [OMG11] angelehnt. Sie weichen von dieser jedoch in einigen Punkten ab. Formuliert werden die Konstrukte in der IML-Spezifikationssprache mit der auch die Basis-IML-Spezifikation verfasst wurde.

Ein Grund für die Abweichungen von der offiziellen Hierarchie ist die Möglichkeit, die neuen Knoten besser in die bereits bestehenden IML-Konstrukte einzubinden. Die UML Spezifikation setzt stark auf abstrakte Basisklassen, welche in den verschiedenen Modellen wiederverwendet werden [OMG11]. Eine 1:1 Abbildung der Klassenhierarchie nach der OMG Spezifikation würde eine saubere Verankerung in den bereits vorhandenen IML-Konstrukten unnötig verkomplizieren. Zudem geht die Spezifikation der Zustandsautomaten nach OMG teilweise über die gängige Definition von StateMachines hinaus. In den folgenden Punkten weicht der Entwurf von Zustandsautomaten von der offiziellen OMG Spezifikation ab:

- Guards für `Transitions` werden als einzelner Funktionsaufruf mit boolschem Rückgabewert modelliert.
- Effekte von `Transitions` und `State`-Aktivitäten werden als Funktionsaufruf ohne Rückgabewert realisiert.
- Bei den `Pseudostates` werden `Exit`, `Entry` sowie `Fork` und `Join` nicht unterstützt. Realisiert werden `Initial`, `Terminate`, `Choice` und `Junction`.
- `FinalState`-Elemente werden von `Pseudostate` anstatt von `State` abgeleitet.
- Pro Zustandsautomat ist nur eine Region erlaubt.
- Der Name der Eventtypen die als Trigger für Transitionen fungieren werden über den Namen identifiziert.

Daraus ergeben sich die folgenden IML-Knoten für das StateMachine-Sprachpaket:

NamedElement: Diese Klasse bildet die Basisklasse für nahezu alle anderen Zustandsautomaten-Klassen. Sie definiert das jedes Element einen Namen besitzt, anhand dessen man es identifizieren kann. `NamedElement` ist eine abstrakte Klasse und es können dementsprechend keine Instanzen davon erzeugt werden.

Vertex: Als `Vertex` wird die Basisklasse für alle Zustände (`States`) und Pseudo-Zustände (`Pseudostates`) bezeichnet. Wie auch bei `NamedElement` handelt es sich um eine abstrakte Basisklasse. Allerdings werden durch `Vertex` keine neuen Attribute eingeführt sondern sie ist darauf zurückzuführen dass einige Funktionalitäten (wie beispielsweise die Deklaration von Transitionen) für Zustände als auch für Pseudo-Zustände gültig sind.

T_StateMachine: Als `T_StateMachine` wird die Definition des Zustandsautomaten beschrieben. Innerhalb einer `T_StateMachine`-Komponente können sich Subkomponenten vom Typ `Vertex` befinden. Diese repräsentieren die Zustände und Pseudo-Zustände des Zustandsautomaten. Ebenfalls Teil dieser Komponente sind die Transitionen zwischen diesen `Vertex`-Elementen sowie die Events, welche innerhalb der StateMachine auftreten können. Über das Attribut *Type_Declaration* kann die Definition mit einer Deklaration verbunden werden.

O_StateMachine: Das Gegenstück zur `T_StateMachine` ist das `O_StateMachine`-Konstrukt. Während `T_StateMachine` die Definition repräsentiert, bildet `O_StateMachine` die zugehörige Deklaration eines Zustandsautomaten.

State: Als `State` wird ein Zustand innerhalb eines Zustandsautomaten bezeichnet. Neben einem Namen besitzen Zustände drei verschiedene Aktivitäten, die zu unterschiedlichen Zeitpunkten ausgeführt werden können.

- Entry: Wird ausgeführt wenn der Zustand betreten wird.
- Do: Wird ausgeführt wenn der Zustand aktiv ist.
- Exit: Wird ausgeführt wenn der Zustand verlassen wird.

Da `State` von `Vertex` erbt können Zustände Quelle oder Ziel von Transitionen sein. Für die Abbildung von hierarchischen Zustandsautomaten kann jedes `State`-Objekt zudem noch Subkomponenten enthalten. Diese erlauben es innerhalb eines `States` weitere `States`, `Pseudostates` und `Transitions` zu definieren. Ein Zustand der Subkomponenten enthält, wird als *Composite-State* bezeichnet.

Transition: Transitionen sind Übergänge zwischen zwei (Pseudo-)Zuständen. Es handelt sich dabei um eine gerichtete Verbindung die neben einem Namen als Bezeichner mit verschiedenen Attributen ausgestattet werden können. So können Transitionen eine sogenannte "Guard", eine Bedingung die erfüllt sein muss um die `Transition` zu aktivieren, besitzen. Zusätzlich kann eine `Transition` mit einer Aktivität verknüpft werden, welche ausgeführt wird wenn die `Transition` durchgeführt wird. Der Ausführungszeitpunkt einer `Transition` ist dabei abhängig von den mit ihr verknüpften Eventtypen. Verfügt eine `Transition` über keine solche Eventtypen, so wird die `Transition` automatisch bei Betreten des Quellzustands ausgelöst.

Pseudostate: `Pseudostates` stellen eine besondere Gruppe von Zustände dar. Wie diese erben sie ebenfalls von `Vertex` und können dementsprechend mit Transitionen verknüpft werden. Anders als Zustände verfügen Pseudo-Zustände nicht über Entry-, Do- und Exit-Aktivitäten, sondern besitzen ein konstantes, vordefiniertes Verhalten. Folgende Pseudozustände werden unterstützt:

- Initial: `Initial`-Komponenten sind abgeleitet von `Pseudostate` und definieren den Einstiegspunkt eines Zustandsautomaten. Bei Start der Ausführung wird geprüft, welche der ausgehenden Transitionen des `Initial`-Zustands erfüllt ist. Diese wird aktiviert.
- Choice: Wie `Initial` sind auch `Choice`-Komponenten abgeleitete Pseudo-Zustände. Sie ermöglichen es, eine eingehende `Transition` auf mehrere ausgehende Transitionen zu verlinken. Wird die eingehende `Transition` aktiviert, so wird geprüft, welche der ausgehenden Transitionen freigeschaltet ist. Für eine Freischaltung muss die Guard-Bedingung einer Transition nach *true* auswertet werden. Die Bedingungen sollten sich dabei nicht überschneiden, da dies einen nicht-deterministischen Zustandsautomaten zur Folge hätte. Damit jede mögliche Situation abgedeckt ist, kann zusätzlich eine `Transition` mit einer vordefinierten *else*-Bedingung definiert werden.
- Terminate: `Terminate`-Zustände stellen eine Art unvorhergesehenen Abbruch dar. Geht ein Zustandsautomat in einen solchen Zustand über, so wird ihre Ausführung sofort beendet. In Zustandsautomaten nach OMG-Standard ist dieser Abbruch zudem unabhängig von der Kompositionstiefe. Dies bedeutet, dass nicht nur eine verschachtelte StateMachine beendet wird, sondern auch jede übergeordnete. `Terminate`-Zustände können keine von ihnen ausgehende Transitionen besitzen.
- Junction: Pseudozustände vom Typ `Junction` ähneln der Funktionsweise von `Choice`. In einer `Junction` können mehrere eingehende auf mehrere ausgehende Transitionen abgebildet werden. Wie der Name schon andeutet, entspricht dies dem Verhalten einer Kreuzung[7] im Straßenverkehr. Zudem werden `Junctions` genutzt um mehrere Transitionen zu einer einzigen zusammenzuführen.
- FinalState: Ähnlich wie `Terminate-Pseudostates` führen `FinalStates` bei ihrer Aktivierung zu einer Beendigung der aktuell aktiven StateMachine. Dies betrifft allerdings nur den direkt übergeordneten Zustandsautomaten. Das Betreten eines `FinalStates` entspricht einer korrekten, planmäßigen Beendigung der Ausführung eines Zustandsautomaten.

Werden die Knoten des IML-Sprachpakets durch den *imlgen*-Generator verarbeitet, so werden die entsprechenden Ada-Pakete generiert. Diese können als Klassendiagramm dargestellt werden. Dies ist in Abbildung 6.11 dargestellt.

[7] `Junction`, zu deutsch: "Kreuzung"

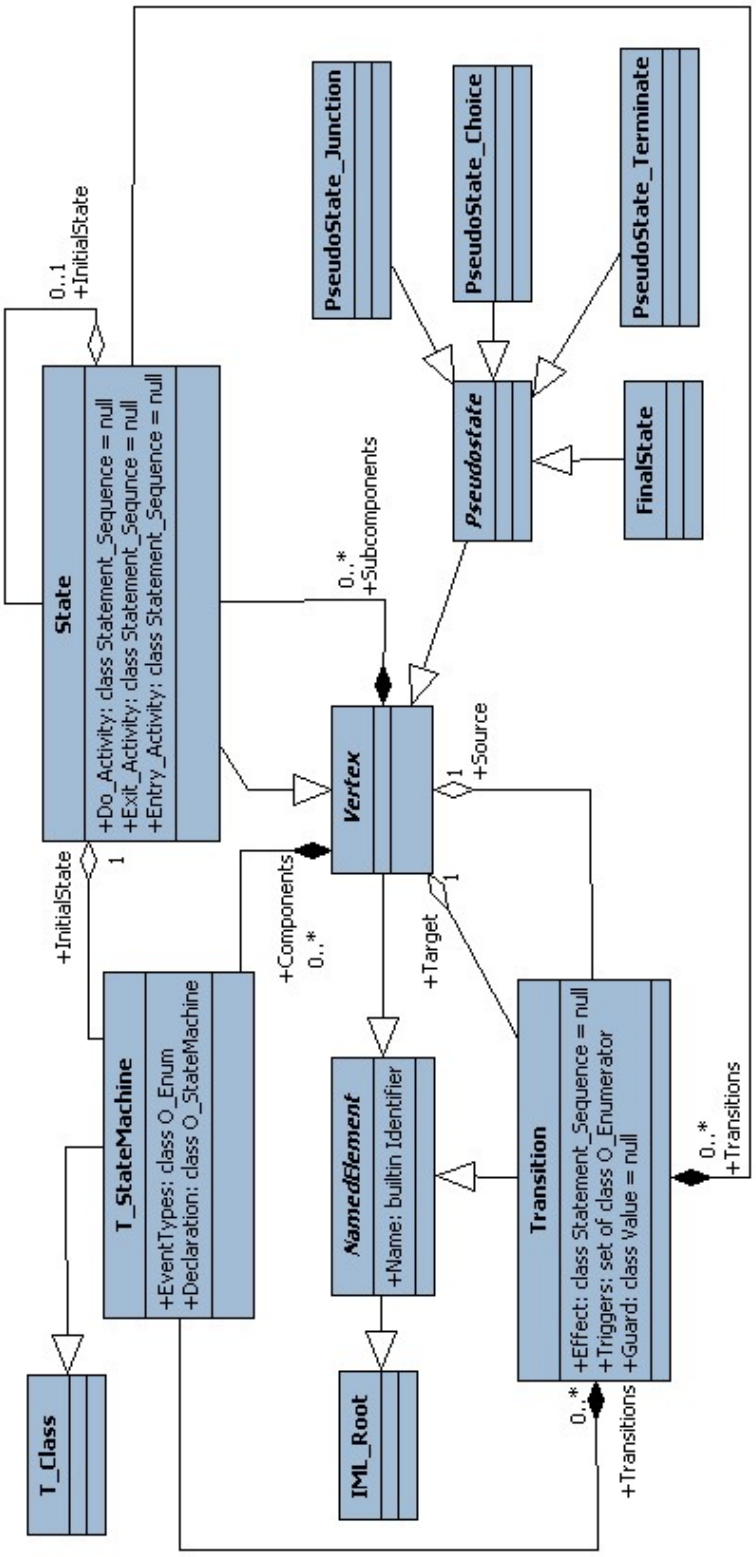

Abbildung 6.11: Klassendiagramm der IML-Knotenklassen für hierarchische Zustandsautomaten.

Alle IML-Knoten müssen auf die Basisklasse `IML_Root` zurückgeführt werden. Dies wird bei den Konstrukten für Zustandsautomaten auf verschiedene Weise. `T_StateMachine` und `O_StateMachine` sind von den bereits existenten Knotenklassen `T_Class` bzw. `O_Class` abgeleitet. Diese erben über mehrere Zwischenschritte von `IML_Root`. Alle anderen Knotenklassen für die IML-Zustandsautomaten leiten sich von der abstrakten Oberklasse `Named_Element` ab. Diese erlaubt es allen Kindklassen einen Namen zuzuweisen und ist von `IML_Root` abgeleitet.

6.5 Musterbeispiel

Um die Funktionalität und Vollständigkeit der IML-Knoten zu testen, wurde ein Musterbeispiel für einen Zustandsautomaten entworfen. Dieses enthält alle umgesetzten Bestandteile. Thema des Beispiels ist der Entwurf eines Geldautomaten (engl: "Automated Telling Machine", kurz: "ATM"). Die dabei modellierten Zustände und Verhaltensweisen bilden dessen Logik allerdings nur in vereinfachter Weise ab. Eine vollständige Umsetzung wäre unnötig komplex und macht es schwer die korrekte Abbildung zu kontrollieren. Es wird davon ausgegangen, dass von der Korrektheit des Musterbeispiels auch auf eine Korrektheit komplexerer Zustandsautomaten geschlossen werden kann.

Für das Musterbeispiel werden zwei Komponenten entwickelt. Die erste ist das UML-Diagramm für den ATM-Zustandsautomaten. Dieses wird mit dem UML-Editor *Papyrus* entworfen. *Papyrus* wurde bereits im vorangegangenen Abschnitt 6.3 beschrieben. Abbildung 6.12 zeigt die UML-Darstellung des Musterbeispiels.

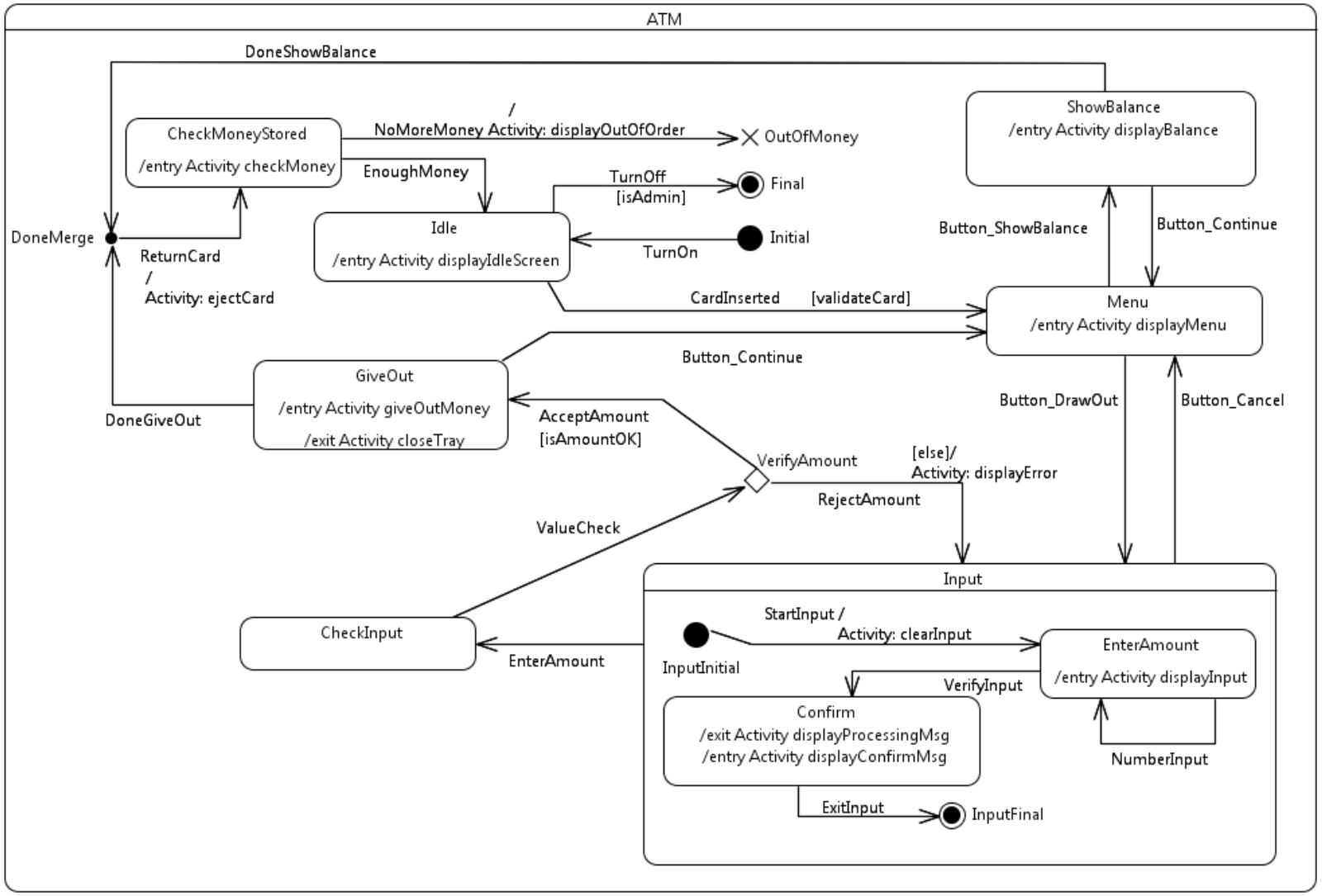

Abbildung 6.12: Musterbeispiel: UML-Darstellung für einen Geldautomaten.

Im Musterbeispiel werden die folgenden Elemente definiert:

Zustandsautomat:

Es existiert genau ein Zustandsautomat im Musterbeispiel. Dieser enthält alle Zustände des Musterbeispiels sowie die Transitionen mit denen diese verbunden werden. Der Zustandsautomat wird durch einen eindeutigen Namen identifiziert. Anhand dieses Namens bestimmt das Frontend die Klasse in der die Funktionsdefinitionen für die Guards, Effects, Entries und Exits enthalten sind. Dieser Vorgang wird im Kapitel 7 ausführlich beschrieben. Der Name des Zustandsautomaten in diesem Beispiel lautet "ATM".

Transitionen: Im Musterbeispiel des Geldautomaten ist eine Vielzahl von Transitionen enthalten. Im Gegensatz zu den Zuständen wird deren individuelle Bedeutung nicht im Einzelnen betrachtet. Die Beschreibung der Zustände erwähnt die möglichen Übergänge zwischen den Zuständen und beschreibt deren Semantik. Hier soll lediglich darauf hingewiesen werden, dass Transitionen mit einem Event verknüpft sein können. Dieses muss vom Zustandsautomaten empfangen werden, um diese Transition zu aktivieren. Im Gegensatz zum Effekt und der Guard-Bedingung einer Transition werden diese Events allerdings nicht im Diagramm sichtbar. Sie sind lediglich im zugrundeliegenden Modell definiert. Daher wird in 6.2 eine Übersicht der Transitionen und ihrer zugeordneten Events bereitgestellt.

Zustände: Innerhalb des Zustandsautomaten finden sich alle für ihn definierten Zustände. Dies umfasst neben den annehmbaren Zuständen auch alle Pseudozustände wie `Initial`, `Final`, `Terminate`, `Choice` und `Junction`.

Der Zustandsautomat des Musterbeispiels kann die folgenden Zustände annehmen:

- Idle: Der erste Zustand der zu Beginn der Ausführung angenommen wird. Bei Eintritt in den Zustand wird die Funktion `displayIdleScreen` ausgeführt. *Idle* ist der Zustand in dem der Geldautomat auf den Beginn einer Interaktion mit dem Kunden wartet. Diese kann durch das Einführen einer EC-Karte ausgeführt werden. Dies wird durch das Event `eCardInserted` signalisiert. Soll der Automat abgeschaltet werden, so kann dies von diesem Zustand aus geschehen. Dazu muss ein Event vom Typ `eTurnOff` ausgelöst werden. Die zugehörige Transition *TurnOff* wird allerdings nur aktiviert, wenn ihre Guard-Funktion `isAdmin` den Benutzer als autorisiert erachtet. Ist dies der Fall, so wird der Zustandsautomat durch den Übergang in den `FinalState`-Pseudozustand "Final" beendet.
- Menu: Dieser Zustand wird erreicht nachdem der Kunde eine EC-Karte eingeführt hat und diese durch die Guard-Funktion `validateCard` validiert wurde. Als `Entry` ist die Funktion `displayMenu` definiert. Von *Menu* aus können zwei verschiedene Zustände erreicht werden. Diese symbolisieren die gewünschte Aktion des Users. Der Zustand *ShowBalance* wird angenommen wenn das Event `eButtonShowBalance` empfangen wird. Will der Benutzer Geld abheben, so drückt er einen entsprechenden Knopf durch den ein `eButtonDrawOut`-Event an den Zustandsautomaten gesendet wird. In diesem Fall geht der Automat in den Zustand *Input* über.

Transition	Event
TurnOn	-
TurnOff	`eTurnOff`
CardInserted	`eCardInserted`
Button_ShowBalance	`eButtonShowBalance`
Button_Continue	`eButtonContinue`
Button_Cancel	`eButtonCancel`
Button_DrawOut	`eButtonDrawOut`
StartInput	-
NumberInput	`eNumberButtonPressed`
VerifyInput	`eEnterButtonPressed`
ExitInput	`eEnterButtonPressed`
EnterAmount	-
ValueCheck	-
AcceptAmount	-
RejectAmount	-
DoneGiveOut	`eGiveOutComplete`
DoneShowBalance	`eDoneShowBalance`
ReturnCard	-
EnoughMoney	`eEnoughMoney`
NoMoreMoney	`eNoMoreMoney`

Tabelle 6.2: Übersicht der Transitionen im Musterbeispiel und ihrer Events.

- ShowBalance: In diesem Zustand zeigt der Geldautomat den aktuellen Kontostand des Benutzers an. Dies wird beim Betreten des Zustands durch die Funktion `displayBalance` realisiert. Nach Anzeige des Betrags kann der Kunde entscheiden weitere Aktionen auszuführen, oder den Vorgang zu beenden. Im ersteren Fall wird ein Event vom Typ `eButtonContinue` ausgelöst und der Zustandsautomat geht wieder in den *Menu*-Zustand über. Wird dies vom Benutzer nicht signalisiert, so signalisiert ein `eDoneShowBalance`-Event den Übergang zum Pseudozustand *DoneMerge*. Dieser führt die beiden Ablaufsequenzen des Zustandsautomaten zusammen und führt in den Zustand *CheckMoneyStored*. Dabei wird die Karte des Benutzers durch die Effect-Funktion `ejectCard` ausgegeben.
- Input: *Input* ist der einzige *Composite-State* des Musterbeispiels. Er modelliert die Eingabe-Interaktion des Benutzers, wenn dieser Geld abheben möchte. Wird *Input* betreten, so beginnt dieser, wie auch bei einem Zustandsautomaten, immer am definierten `Initial`-Pseudozustand. Von dort aus wird er über die Transition *StartInput* zum Zustand *EnterAmount* weitergeleitet. Dabei wird der Effect `clearInput` der Transition ausgelöst. Durch das Auslösen des Events `eButtonCancel` kann die Eingabe abgebrochen werden. Der Zustandsautomat geht dabei aus der Verschachtelung von *Input* zurück zum Zustand *Menu*.

  - EnterAmount: In diesem Unterzustand von *Input* kann der Benutzer den gewünschten Betrag eingeben. Für jeden Tastendruck wird dabei ein Event vom Typ `eNumberInput` ausgelöst. Dies kann beliebig oft wiederholt werden, bis der Zustandsautomat das Event `eVerifyInput` empfangen wird. Dieses veranlasst einen Zustandswechsel zu *Confirm*.
  - Confirm: Dies ist der zweite Unterzustand innerhalb von *Input*. Er wird erreicht, sobald der Benutzer die Eingabe des abzuhebenden Betrags beenden möchte. Bevor der Benutzer den Betrag ausgezahlt bekommt, sollte er dies noch einmal bestätigen. *Confirm* besitzt sowohl eine Entry- als auch eine Exit-Aktion. Beim Eintritt in den Zustand, wird die Funktion `displayConfirmMsg` und beim Verlassen `displayProcessingMsg` aufgerufen.

- CheckInput: Wird die Eingabe für das Abheben eines Geldbetrags durch das Erreichen des `Final`-Pseudostates *InputFinal* erreicht, so löst dies die "Immediate Transition" *EnterAmount* aus. Dadurch geht der Zustandsautomat in den Zustand *CheckInput* über. Dieser verfügt über eine weitere "Immediate Transition" und wird folglich direkt wieder verlassen. Ziel dieser Transition ist der `Choice`-Pseudozustand *VerifyAmount*. Hier wird durch die Guard-Funktion `isAmountOK` geprüft ob der Automat über die Transition *AcceptAmount* in den Zustand *GiveOut* übergehen darf. Gibt die Funktion `false` zurück, so wird stattdessen die Transition *RejectAmount* aktiviert. Diese ist mit der speziellen `else`-Guard markiert und wird aktiviert wenn keine der anderen Transitionen der `Choice` aktiviert werden dürfen. Sie zeigt auf den Zustand *Input*.
- GiveOut: *GiveOut* wird erreicht, wenn die Guard-Funktion `isAmountOK` der Transition *AcceptAmount* `true` zurückgibt. Beim Betreten wird die Funktion `giveOutMoney` und beim Verlassen `closeTray` aufgerufen. Der Benutzer kann an dieser Stelle die Interaktion fortsetzen indem er durch einen entsprechenden Knopfdruck ein Event vom Typ `eButtonContinue` auslöst. Dieses bringt ihn zurück in das Ausgangsmenü. Bleibt der Knopfdruck aus, so leitet ein `eDoneGiveOut`-Event den Zustandsautomaten zum `Junction`-Pseudozustand *DoneMerge* weiter. Von dort aus geht der Automat in den Zustand *CheckMoneyStored* über.
- CheckMoneyStored: Dies ist der letzte Zustand der angenommen wird, bevor der Geldautomat entweder abschaltet oder in den Warte-Zustand *Idle* übergeht. Hier wird beim Eintritt die Funktion `checkMoney` aufgerufen. An dieser Stelle könnte entschieden werden, ob der Automat noch über genügend Geld für den Betrieb verfügt. Ist dies nicht der Fall, so signalisiert ein `eNoMoreMoney`-Event eine unerwartete Beendigung des Ablaufes. Hierbei wird der `Terminate`-Pseudozustand "OutOfMoney" erreicht und die Funktion `displayOutOfOrder` aufgerufen.

Nach UML Spezifikation ([OMG11]) kann ein Zustandsautomat nur genau einen `Initial`-Pseudozustand enthalten. Dieser stellt den Eintrittspunkt dar. Im Musterbeispiel ist dieser als "Initial" benannt. Von ihm aus darf es genau eine Transition geben. Diese verweist hier auf den Zustand "Idle".

Beendet werden kann die Ausführung des Zustandsautomaten auf zwei Wegen. Dabei muss entweder der `FinalState` "Final" oder der `Terminate`-Pseudozustand "OutOfMoney" erreicht werden.

Die zweite Komponenten des Musterbeispiels ist die Implementierung der Guard-, Aktivity-, Entry- und Exit-Funktionen. Diese wird dann später bei der Ausführung des Zustandsautomaten-Frontends verwendet, um die Funktionsnamen mit einer entsprechenden Funktionsdefinition zu verknüpfen. Dieser Vorgang wird detailliert im Kapitel 7 beschrieben.

Im Falle des Musterbeispiels wird die Implementierung durch eine Klasse `ATM` bereitgestellt. Diese enthält Definition und Deklaration für alle im Beispiel verwendeten Funktionen. Es ist zudem möglich Membervariablen und Konstruktoren zu definieren. Als Sprache wird C++ und für die IML-Abbildung *cafe*++ eingesetzt. Der Code-Ausschnitt in 6.7 zeigt die Definition der Klasse.

```
1  #include "StringWrapper.h"
2
3  class ATM
4  {
5      private:
6          bool adminMode;
7          int  moneyStored;
8          StringWrapper input;
9          StringWrapper output;
10
11         void appendToOutput(const StringWrapper& value);
12         void resetOutput();
13         int convertInput();
14
15     public:
16         ATM();
17         ~ATM();
18
19         //Functions used for:
20         // - Guards
21         // - Activities
22         // - Entries
23         // - Exits:
24         void displayIdleScreen();
25         void ejectCard();
26         bool isAmountOK();
27         void displayMenu();
28         void displayBalance();
```

```
29          void displayError();
30          void checkMoney();
31          void giveOutMoney();
32          void closeTray();
33          void displayOutOfOrder();
34          bool validateCard();
35          bool isAdmin();
36          void clearInput();
37          void displayInput();
38          void displayProcessingMsg();
39          void displayConfirmMsg();
40
41          //Getter & Setter for the members:
42          int getMoneyStored();
43          void setMoneyStored(int newAmount);
44
45          void setAdminMode(bool isAdmin);
46          void enterInput(char newInput);
47
48          StringWrapper getOutput();
49  };
```

Listing 6.7: Deklaration der ATM-Klasse für die Funktionen des Musterbeispiels.

Die `#include`-Anweisung für den Header "Stringwrapper.h" benötigt eine Erklärung. Teil der Aufgabenstellung dieser Diplomarbeit ist die Implementierung eines Code-Generators, welcher aus der IML-Darstellung des Zustandsautomaten funktionsfähigen Java-Code generiert. Dieser wird im Kapitel 9 beschrieben. Die Umsetzung eines vollständigen Code-Generators für alle IML-Konstrukte der Sprache C++ würde den Rahmen der Diplomarbeit sprengen. Dementsprechend wird statt des `string`-Headers der STL-Bibliothek ein Platzhalter definiert. Dieser wird dann anstelle der `string`-Klasse verwendet. Im Code-Generator ist spezifiziert, dass alle Vorkommnisse der `StringWrapper`-Klasse durch den Java-Datentyp `String` ersetzt werden. Die Definition, sowie die Implementierung von `StringWrapper` ist im Ausschnitt 6.8 dargestellt.

```
1  #ifndef __INCLUDE_STRING_WRAPPER__
2  #define __INCLUDE_STRING_WRAPPER__
3
4  class StringWrapper
5  {
6      public:
7          StringWrapper();
8          StringWrapper(char* value);
9
10         int length();
```

```
11
12          char operator[](int index);
13          char charAt(int index);
14
15          StringWrapper& operator=(char* value);
16
17          const StringWrapper& operator+(StringWrapper const& rhs);
18          const StringWrapper& operator+(char* value);
19          const StringWrapper& operator+(char value);
20
21          bool operator !=(StringWrapper const& right);
22          bool operator ==(StringWrapper const& right);
23          bool operator !=(char* value);
24          bool operator ==(char* value);
25  };
26
27  const StringWrapper& operator+(char* value, StringWrapper const& rhs);
28
29  #endif
```

Listing 6.8: Die Klasse StringWrapper. Dient als Platzhalter für String.

Die Methoden der Klasse sind dabei identisch mit denen der Java-Klasse `String`. Dadurch können diese auf die gleiche Weise aufgerufen werden und geben die gleichen Datentypen zurück. Die Operatoren sind definiert damit Instanzen von `StringWrapper` auf die gleiche Weise in Konkatenationen, Vergleichen und Zuweisungen eingesetzt werden können. Wichtig ist hier, dass die Klasse `StringWrapper` keine Implementierung besitzt. Da sie später durch einen anderen Datentyp ersetzt wird, ist sie lediglich nötig, damit die Implementierung der `ATM`-Klasse korrekt von *cafe++* verarbeitet werden kann.

7 Zustandsautomaten-Frontend

Für die Generierung einer IML-Darstellung aus einem Zustandsautomaten wird ein Frontend benötigt. Wie auch bei den Werkzeugen *cafe++* und *java2iml* verarbeitet das Zustandsautomaten-Frontend, im Folgenden auch als *Statofe*[1] bezeichnet, eine Eingabedatei und erzeugt den daraus resultierenden Graphen. Umgesetzt wurde die Anwendung in Java. Für die Modellierung der Zustandsautomaten wird das Eclipse-Plugin *Papyrus* eingesetzt. Dieses basiert auf dem Eclipse Modelling Framework (EMF). Gespeichert werden die Zustandsautomaten dabei in einem XML-Format, welches dem EMF-UML-Schema entspricht. Um eine solche Datei zu laden, müssen die EMF-Bibliotheken in die Frontend-Anwendung eingebunden werden. Die Wahl der Programmiersprache war festgelegt, da die benötigten Libraries nur für Java verfügbar sind.

7.1 Eclipse Modelling Framework (EMF)

Das Eclipse Modelling Framework (EMF) [2] stellt Modellierungs-Werkzeuge und -Mechanismen für die Eclipse Entwicklungsumgebung bereit. Auf Basis von EMF ist es möglich Modelle zu definieren, welche für verschiedene Zwecke wie Code-Generation oder Editor-Entwicklung genutzt werden können [Bud03]. Eine vollständige Beschreibung von EMF im Rahmen dieser Arbeit wäre zu aufwendig und unzweckmäßig. Dementsprechend konzentrieren sich die Ausführungen dieses Abschnitts auf die relevanten Informationen für die Nutzung von EMF im Kontext dieses Projekts.

EMF-Modelle werden im Ecore-Format definiert. Generiert werden können diese unter Anderem aus annotierten Java-Klassen oder durch Modellierungswerkzeuge (wie *Papyrus*. Durch die weite Verbreitung von EMF ist es möglich Modelle in einer Vielzahl von Werkzeugen wiederzuverwenden.

Der Einsatz von EMF für die Model-Driven Architecture (MDA) bietet dem Nutzer verschiedene Vorteile. Neben der automatisierten Serialisierung im XMI-Format können EMF-Modelle zur dynamischen Erzeugung von Java-Klassen genutzt werden [Bud03]. EMF bildet die Basis für eine Vielzahl von Modellierungs-Plugins für Eclipse. Das in dieser Diplomarbeit verwendete *Papyrus*-Plugin ist eines davon. Da Eclipse-Plugins als Jar-Dateien ausgeliefert werden, ist es möglich sie in unabhängigen Anwendungen zu verwenden.

[1] von: Statemachine Frontend
[2] http://www.eclipse.org/modeling/emf/

Für die Modellierung von UML-Diagrammen stellt EMF von Haus aus bereits fertige Ecore-Modelle bereit [Bud03]. Eine vollständige Abbildung der UML-Schaubilder nach OMG-Spezifikation [OMG11] ist eine hochkomplexe Aufgabe [Bud03]. Die Nutzung von EMF für die programmatische Generierung von UML-Diagrammen führt daher zu einer Zeitersparnis auf Seite der Entwickler. Die Entwicklungsumgebung Eclipse erfüllt alle Kriterien einer Language Workbench [Fow05]. Das Ecore-Modell dient dabei als abstrakte Zwischendarstellung auf der gearbeitet wird. Projektionale Editoren erlauben unter Anderem die textbasierte und grafische Bearbeitung. Für die Persistenz wird das XMI-Format genutzt. Durch den Einsatz von Generatoren ist es zudem möglich Code aus der Zwischendarstellung zu erzeugen.

Für die Nutzung von EMF in Standalone-Anwendungen müssen eine Vielzahl von Bibliotheken eingebunden werden. Diese stellen die Basisfunktionen von EMF sowie die UML2-Funktionalitäten bereit. Die folgende Liste enthält die benötigten Libraries. Die Versionsnummer sowie die Dateiendung ".jar" sind aus Gründen der Lesbarkeit weggelassen worden.

- org.eclipse.emf.common
- org.eclipse.emf.ecore
- org.eclipse.emf.ecore.change
- org.eclipse.emf.ecore.xmi
- org.eclipse.emf.mapping.ecore2xml
- org.eclipse.equinox.common
- org.eclipse.jdt.core
- org.eclipse.uml2.common
- org.eclipse.uml2.common.edit
- org.eclipse.uml2.uml
- org.eclipse.uml2.uml.edit
- org.eclipse.uml2.uml.resources

Ursprünglich wird EMF für die Entwicklung von Eclipse-Plugins verwendet. Beim Starten von Eclipse wird EMF dabei automatisch initialisiert. Soll EMF in einer Standalone-Anwendung eingesetzt werden, so muss die Initialisierung von Hand vorgenommen werden. Der Code-Ausschnitt in 7.1 zeigt wie dies im Zustandsautomaten-Frontend geschieht.

```
1  Resource.Factory.Registry.INSTANCE.getExtensionToFactoryMap().put(
2      "*",  new XMIResourceFactoryImpl());
3
4  Resource.Factory.Registry.INSTANCE.getExtensionToFactoryMap().put(
5      "xml", new XMLResourceFactoryImpl());
6
7  EPackage.Registry.INSTANCE.put(
8      UMLPackage.eNS_URI,
9      UMLPackage.eINSTANCE);
10
11 Resource.Factory.Registry.INSTANCE.getExtensionToFactoryMap().put(
12     UMLResource.FILE_EXTENSION,
13     UMLResource.Factory.INSTANCE);
14 }
```

Listing 7.1: Initialisierung von EMF in Standalone-Anwendungen.

Damit EMF weiß welche Factories (s. *Factory*-Pattern in [Gam95]) für die entsprechenden Dateiendungen und Formate zuständig sind, muss dies beim Datentyp `Resource` registriert werden.

7.2 Entwurf

Die Aufgabe von *Statofe* ist die Verarbeitung der Bestandteile des Zustandsautomaten. Hieraus werden die Knoten des resultierenden Graphen generiert. Übernommen wird dies von verschiedenen Parser-Klassen. Nach Einlesen des Automaten wird dieser durchlaufen und die entsprechenden Parser für die gefundenen Elemente aufgerufen. Durch die Aufteilung der Funktionalität auf verschiedene Parser wird die Wart- und Lesbarkeit verbessert. Zudem wurde eine Fassade [Gam95] für die Verarbeitung der Zustände eingesetzt. Alle Zustände leiten sich von der Basisklasse `Vertex` ab. Durch die Fassade muss nicht für jedes Element der zu verwendende Parser bestimmt und aufgerufen werden. Dies wird von der Fassade erledigt.

Der Entwurf des Frontends für Zustandsautomaten kann in die folgenden Gruppen unterteilt werden:

- **Core:** In diese Kategorie ist der Einstiegspunkt der Anwendung definiert. Hier werden die Eingabeparameter geladen und verifiziert. Auf Basis dieser Argumente werden der Implementierungs-Graph und der Zustandsautomat geladen. Weitere Aufgaben des Kerns sind die Initialisierung des JNI-Bindings für IML, sowie das Starten des Parsing- und Linking-Vorgangs.
- **Parser:** Die Verarbeitung der Bestandteile des Zustandsautomaten und die Erzeugung der entsprechenden Knoten wird von den Parsern übernommen. Diese sind auf einzelne Klassen verteilt. Die Unterteilung basiert auf dem Element-Typ für den ein Parser zuständig ist. Durch den Einsatz einer Fassade wird eine einfache Verarbeitung von `Vertices` vereinfacht und gleichzeitig eine gut erweiterbare Struktur geschaffen. Da nicht alle Pseudozustände des UML-Standards unterstützt werden, soll hier die Möglichkeit geschaffen werden in späteren Arbeiten neue Parser einzuführen.
- **Globals:** Aufgabe der `Globals`-Klasse ist die Speicherung von verarbeiteten Elementen für das spätere Linking. Das Datei-Format der Statemachines basiert auf XMI [Budo3]. Demnach kann keine bestimmte Reihenfolge für das Auftreten der Elemente angenommen werden. Dies kann zum Beispiel beim Parsen der Transitionen zu Problemen führen. Angenommen eine Transition wird verarbeitet, bevor ihre Quell- und Zielzustände verarbeitet wurden. In einem solchen Szenario können die Kanten des erzeugten IML-Knoten der Transition nicht korrekt gesetzt werden. Die Singleton-Instanz [Gam95] der `Globals`-Klasse speichert die bereits behandelten Elemente zusammen mit Informationen, welche später für das Setzen der Kanten benötigt wird.
- **Linker:** Als "Linking" werden im *Statofe*-Projekt zwei verschiedene Aufgaben bezeichnet. Der erste Teil ist die nachträgliche Verknüpfung der Elemente. Wie bereits im vorangegangenen Abschnitt beschrieben, ist dies notwendig um sicherzustellen dass

zuerst alle Bestandteile des Automaten verarbeitet wurden, bevor die Kanten des Graphen gesetzt werden.

Ebenfalls als "Linking" wird die Verknüpfung der Funktionsaufrufe in den Guards, Effekten, Exits und Entries mit ihren entsprechenden IML-Knoten. Bei der Modellierung eines Automaten in *Papyrus* ist es nicht möglich globale Variablen zu definieren. Auch die Definition von Bedingungen oder Aktivitäten ist nur schwer abzubilden. Als Lösung wurde ein spezielles Linking-Konzept entworfen.

- **Utilities:** In dieser letzten Gruppe finden sich alle Komponenten, welche nicht in den bereits genannten untergebracht werden konnten. Sie stellen Funktionen für verschiedene Zwecke bereit und werden von den Klassen der anderen Kategorien eingesetzt. Charakteristisch für *Utility*-Klassen sind ihre statischen Methoden und die Verhinderung einer Instanziierung.

Das Zusammenspiel dieser Kategorien wird im Schaubild 7.1 dargestellt. Begonnen wird hierbei am *Core*. Dieser ruft den Zustandsautomaten-Parser auf und übergibt ihm den zuvor geladenen Automaten. Während der Verarbeitung werden die verschiedenen Parser für die einzelnen Elemente verwendet. Dabei werden Elemente, welche später "gelinkt" werden sollen, in die Komponente *Globals* eingetragen. Nach Abschluss des Parsing-Vorgangs wird der *Linker* aufgerufen. Dieser verknüpft die in *Globals* registrierten Funktionsaufrufe mit den Knoten des Implementierungs-Graphen. Außerdem werden die Quell- und Ziel-Zustände der Transitionen gesetzt. Zum Schluss wird das Ergebnis als IML-Datei abgespeichert und die Anwendung beendet.

7.3 Implementierung

Bei Ausführung von *Statofe* wird die `main`-Methode der Klasse `AppCore` aufgerufen. Diese definiert den Einstiegspunkt der Anwendung und übernimmt die folgenden Aufgaben:

1. Initialisierung von EMF.
2. Laden des Zustandsautomaten.
3. Laden des Implementierungs-Graphen.
4. Aufruf des Zustandsautomaten-Parsers.
5. Aufruf des Linkers.
6. Schreiben des erzeugten Graphen in die Ausgabedatei.

Die nötigen Schritte für die Verwendung von EMF wurden bereits im Abschnitt 7.1 beschrieben. Anhand der Kommandozeilenargumente wird im zweiten Schritt der Zustandsautomat und der Implementierungsgraph geladen. EMF repräsentiert die Datei als *Resource*. Aus dieser wird später der entsprechende Automat extrahiert.

Wurden die Eingabedateien korrekt geladen, so wird eine Instanz der Klasse `UMLStateMachineParser` erzeugt. Diese ist die zentrale Eintrittsstelle für den Parsing-Prozess. Durch den Aufruf

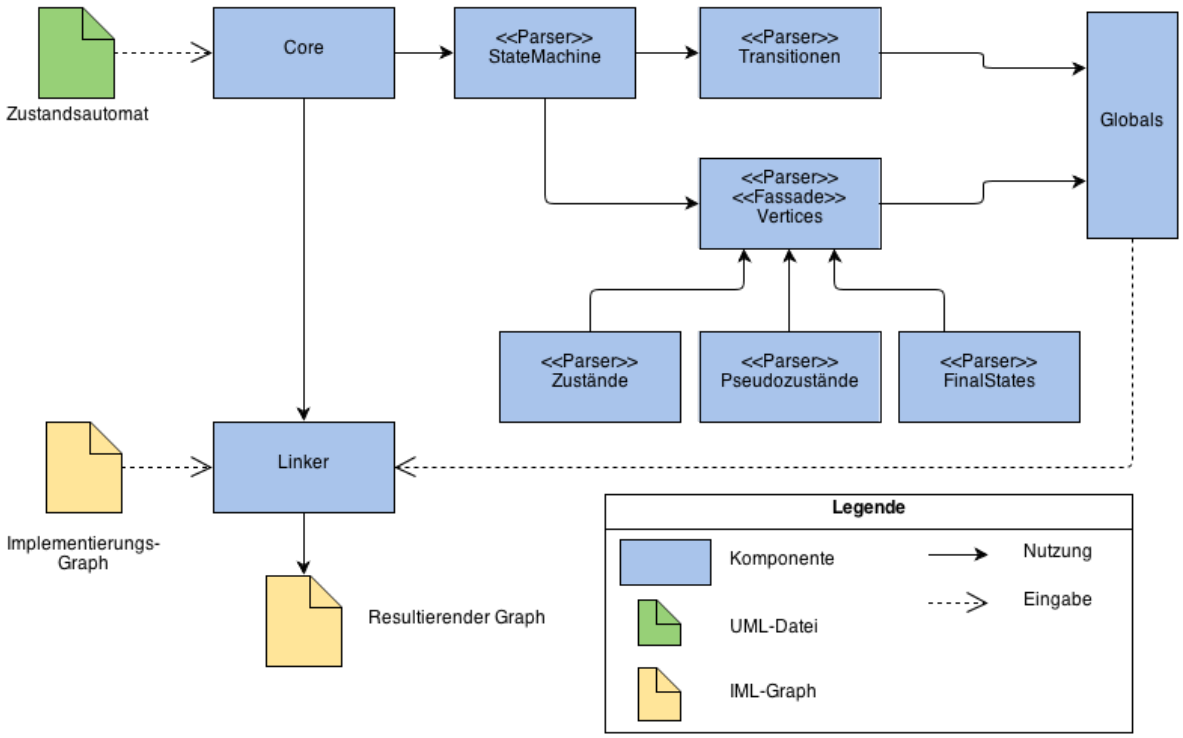

Abbildung 7.1: Zusammenspiel der Komponenten des Zustandsautomaten-Frontends.

der Methode `parseStateMachineRes` wird die Verarbeitung gestartet. Erster Schritt des Parsings ist das Extrahieren des Zustandsautomaten aus der geladenen Datei. Dieser wird als Objekt vom Typ `org.eclipse.uml2.uml.StateMachine` repräsentiert. Auf Basis dieses Objekts werden die IML-Knoten `T_StateMachine` und `O_StateMachine` erzeugt. Sie dienen als Wurzel unter die alle weiteren Elemente des generierten Zustandsautomaten gehängt werden. Im nächsten Schritt werden die Vertices des Automaten an die Klasse `UMLVertexParser` übergeben.

Der `UMLVertexParser` durchläuft die Liste der Vertices und bestimmt für jeden gefundenen Eintrag den korrekten Parser. Dies wird durch eine `Map` realisiert, in der die Vertex-Klassen auf die zugeordneten Parser abgebildet sind. Alle Parser die Vertices verarbeiten, implementieren das Interface `UMLVertexParserStrategy`. Im Code-Ausschnitt 7.2 ist die Schleife dargestellt, in der die Vertex-Elemente durchlaufen und verarbeitet werden. Der zweite Parameter des Methoden-Aufrufs `parser.parseVertex(iter, imlMachine` verweist auf den `T_StateMachine`-Wurzelknoten unter den die erzeugten Knoten gehängt werden. Zudem werden alle verarbeiteten Vertices und ihre Funktionsaufrufe (aus Entries und Exits) in der Singleton-Klasse `Globals` registriert.

Einen Sonderfall stellt das Parsen verschachtelter Zustände dar. Wird ein solcher Zustand erkannt, so werden dessen Subkomponenten und Transitionen durch entsprechende Aufrufe von `UMLVertexParser` und `UMLTransitionParser` verarbeitet. Dies passiert *depth-first*.

In hierarchischen Zustandsautomaten ist es möglich Transitionen von Subzuständen auf Zustände höherer Ebenen zu setzen. Wird eine solche Kante geparsed, so ist es möglich dass ihr Ziel noch nicht verarbeitet wurde. Diese Problematik war Grund für den Einsatz des, bereits weiter oben beschriebenen, "Linking"-Konzepts.

```
for(Vertex iter : vertices) {
    //Get the class of the vertex:
    Class<?> vertexClass = iter.getClass();

    //Find the correct parser:
    UMLVertexParserStrategy parser = vertexParsers.get(vertexClass);

    //Execute the parser:
    parser.parseVertex(iter, imlMachine);
}
```

Listing 7.2: Verarbeitung der Vertex-Knoten durch die entsprechenden Parser.

Nachdem alle Zustände des Automaten durchlaufen und die zugehörigen IML-Knoten erzeugt wurden, wird der `UMLTransitionParser` für die Kanten des Zustandsautomaten aufgerufen. Hierbei werden nur Transitionen verarbeitet, deren Quellzustand auf der obersten Ebene der Zustandshierarchie definiert sind. Alle Transitionen, welche in Subzuständen beginnen wurden bereits im vorangegangenen Schritt ausgewertet.

Mit Abschluss des Transitions-Parsings sind für alle Elemente des Zustandsautomaten die entsprechenden IML-Knoten erzeugt worden. Durch den Aufruf der Klasse `StateMachineLinker` wird nun sichergestellt dass alle Transitionen auf ihre korrekten Quell- und Zielzustände zeigen. Die zweite Aufgabe ist das Verknüpfen der Guards, Transitions-Effekte, Entries und Exits mit ihren entsprechenden Funktionsaufrufen. Während des Parsing-Vorgangs wurde für jeden solchen Aufruf ein IML-Knoten vom Typ `Direct_Call` erzeugt. Dieser wird nun vom Linker mit der Implementierung einer Funktion aus einem separaten IML-Graphen verbunden. Der Ablauf für die Bestimmung und Verlinkung eines Funktionsaufrufs wird im Algorithmus 7.1 als Pseudo-Code beschrieben.

Zur Bestimmung der Methode wird die Deklarations-Tabelle des Implementierungs-Graphen durchlaufen. Die Einträge werden dabei auf die folgenden Kriterien überprüft:

- Ist Eintrag eine Methode?
- Ist Name des Eintrags identisch mit zu verlinkender Funktion?
- Ist Klassenname der Methode identisch mit Namen des Zustandsautomaten?

Wird eine dieser Bedingungen nicht erfüllt, so setzt die Schleife die Ausführung mit dem nächsten Eintrag fort. Wurde eine passende Methode gefunden, so wird für den Aufruf ein impliziter *THIS*-Parameter erzeugt und die Verlinkung gesetzt.

Sind alle Verlinkungen gesetzt ist die Verarbeitung abgeschlossen. Um auch alle Member-Variablen auf den erzeugten Zustandsautomaten-Knoten zu übertragen werden die `O_Class-`

Algorithmus 7.1 Algorithmus zur Bestimmung und Verlinkung von Funktionsaufrufen.

```
procedure LinkDirectCall(graph, call, fctName, stateMachineName)
    Durchlaufe Deklarations-Tabelle des Graphen:
    for all Entry ∈ graph.Declaration_Table do
        if not Entry is O_Method then
            Weiter mit nächstem Eintrag.
        end if
        if ((O_Method)Entry).Name ≠ fctName then
            Weiter mit nächstem Eintrag.
        end if
        if ((O_Method))Entry.Class.Name = stateMachineName then
            CreateCopyThisIn(call, (O_Method)Entry;)
            call.RoutineDeclaration = (O_Method)Entry);
            Beende Schleife.
        end if
    end for
end procedure
```

und `T_Class`-Knoten der Klasse, welche die Implementierung der Funktionen enthält, durch die erzeugten `T_StateMachine` und `O_StateMachine` Konstrukte ersetzt. Dies wird durch einen Aufruf von `Reflection_Utils.replaceNode` erreicht. Hierbei werden alle Vorkommnisse eines Knoten durch einen anderen ersetzt. `T_StateMachine` und `O_StateMachine` leiten sich von `T_Class` bzw. `O_Class` ab. Dadurch ist eine Ersetzung möglich.

Zuletzt wird der Implementierungs-Graph, in dem die Zustandsautomaten-Knoten nun enthalten sind, in serialisierter Form als Datei abgespeichert.

7.4 Anwendung

Das Zustandsautomatenfrontend bildet eine eigenständige Anwendung. Sie wird in das Bauhaus Framework als Teil dessen Werkzeugsammlung integriert. Hierfür wurde eine Make-File geschrieben. Diese kompiliert den Quellcode der Anwendung und erzeugt die ausführbare Jar-Datei. Zudem werden die benötigten Libraries in den Bibliothekspfad von Bauhaus kopiert.

Java-Anwendungen, welche das JNI-Binding für IML nutzen, müssen die entsprechenden Bibliotheken laden. Um diese im Build-Path verfügbar zu machen wird eine spezielle Shell-File (”statofe.sh”) bereitgestellt. Diese setzt die korrekten Pfade, konfiguriert den Build-Path und führt die Jar-Datei des Frontends aus. Der folgende Ausschnitt zeigt die Ausgabe wenn die Shell-File mit fehlerhaften oder fehlenden Argumenten aufgerufen wird:

```
usage: statofe.sh *args*
Processes a statemachine uml file and generates the corresponding IML
```

```
representation. The result is written to the output file as a binary IML
file.
 -i <arg>   Specifies the implementation iml file which contains the
            functions and member variables.
 -o <arg>   Specifies the location to which the iml binary file will be
            written.
 -s <arg>   Specifies the file that contains the statemachine.
 -v         Toggles the verbose mode.
```

Pflichtparameter für den Aufruf des Zustandsautomatenfrontends sind dabei:

- **-s:** Dem Programm muss eine Zustandsautomaten-Datei im EMF-UML-Format übergeben werden.
- **-i:** Um die Funktionen der Guards, Effekte, Entries und Exits linken zu können wird eine entsprechende IML-Datei benötigt.

Das Ergebnis eines Aufrufs ist die erzeugte IML-Darstellung des Zustandsautomaten. In Abbildung 7.2 wird dargestellt wie die einzelnen Programme beim Einsatz des Frontends zusammenspielen. Die C++-Datei "ATM.cpp", implementiert eine Klasse `ATM`. Durch Aufruf von *cafe++* wird sie in einen IML-Graphen überführt. Dies ist der "Implementierungs-Graph" für den Zustandsautomaten. Dieser wird mit *Papyrus* modelliert und in der Datei "ATM.uml" gespeichert. Eingaben für das Zustandsautomaten-Frontend sind der Zustandsautomat ("ATM.uml") und der Implementierungs-Graph ("ATM.iml"). Das Ergebnis wird in der IML-Datei "ATM-StateMachine.iml" gespeichert. Die Klassennotationen zeigen beispielhaft den Inhalt der verknüpften Dateien. Nach Ausführung von *cafe++* enthält die Datei "ATM.iml" `O_Class`- und `T_Class`-Knoten für die Klasse `ATM`. Nachdem das Frontend ausgeführt wurde sind diese durch entsprechende `O_StateMachine` und `T_StateMachine`-Knoten ersetzt worden.

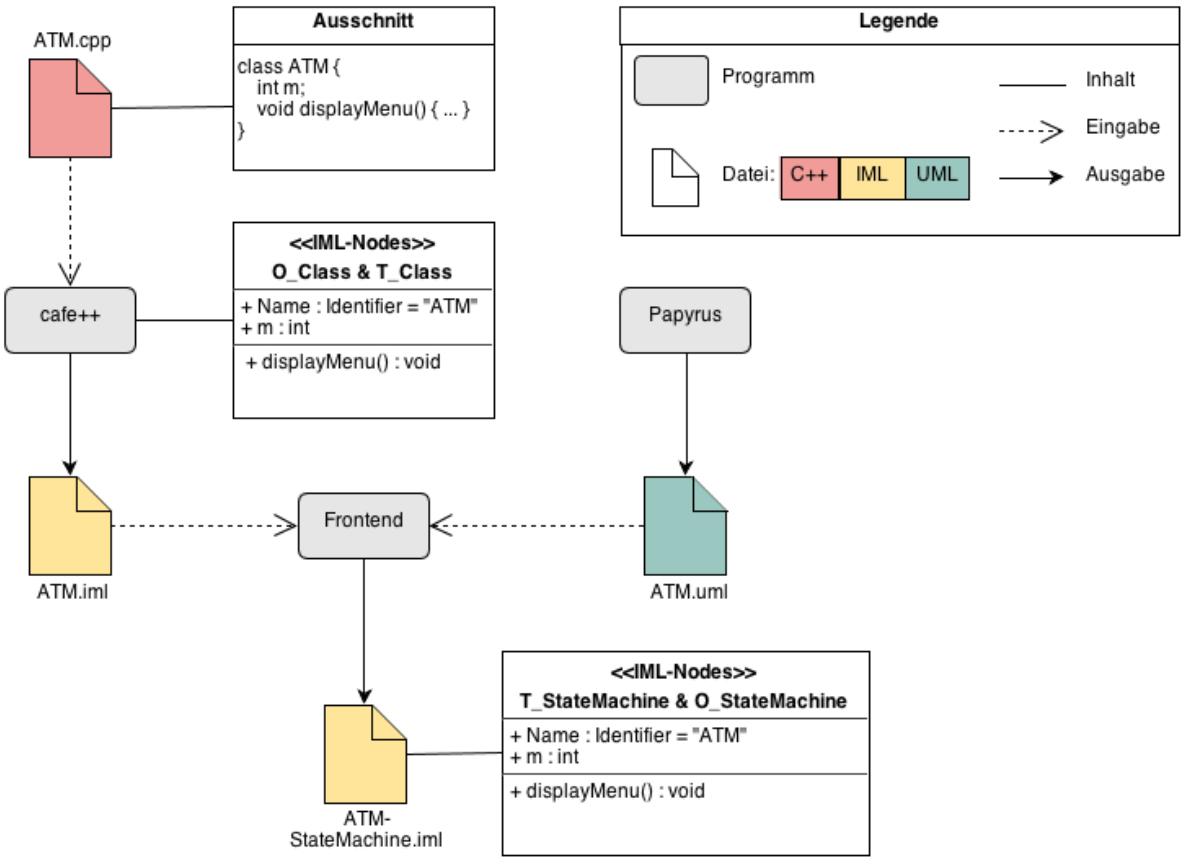

Abbildung 7.2: Genutzte Programme und entstehende Dateien für Aufruf des Frontends.

8 Transformationssprache

In diesem Kapitel wird die Transformationssprache *IMLTransform* vorgestellt. Sie wurde als Teil des Sprachpaket-Mechanismus für die Abbildung der abstrakten IML-Konstrukte auf Basis-Knotenklassen entwickelt. Ihre Anwendung ist jedoch nicht auf diesen Zweck beschränkt. Im Laufe dieses Abschnitts wird erläutert, welche Alternativen betrachtet und welche Anforderungen identifiziert wurden. Zudem werden die Technologien beschrieben, welche für die Umsetzung der Sprache verwendet wurden. Den ausführlichsten Teil dieses Kapitels nimmt die Beschreibung der Sprachkonstrukte ein.

8.1 Evaluation

Zu Beginn der Arbeit wurden mögliche Realisierungen der einzelnen Aufgaben evaluiert. Während im Kapitel 5 bereits die Überlegungen bezüglich der Sprachpaket-Verarbeitung beschrieben wurden, soll hier auf die betrachteten Alternativen und Grundlagen für die Transformationssprache eingegangen werden. Als Referenz für eine Sprache zur Transformation von Graphen wurde ATL [1] [2] betrachtet.

ATL bietet eine Syntax die auf der Definition von Ersetzungsregeln basiert. Eine Regel spezifiziert dabei Operationen die für einen bestimmten Knoten ausgeführt werden sollen. Zusätzlich können über OCL [3] [4] Filter-Bedingungen definiert werden. Die Operationen der Regel werden dann nur für Knoten ausgeführt, welche die Filterkriterien erfüllen. ATL basiert auf dem, bereits im Kapitel 7.1 beschriebenen, Eclipse Modelling Framework. Um einen Graphen transformieren zu können wird ein entsprechendes Ecore-Modell benötigt. Dieses steht in EMF für eine Vielzahl von Konzepten (u.a. UML) zur Verfügung. In der Evaluationsphase wurde geprüft ob es auch möglich wäre ATL als Transformationssprache für IML-Graphen zu verwenden. Hierfür müsste zuerst ein Werkzeug implementiert werden, welches das IML-Modell in ein äquivalentes Ecore-Modell überführt.

Ein entsprechender Versuch auf Basis der IML-Spezifikationsdatei wurde unternommen, scheiterte aber an der Definition der Builtins. Diese sind fest in den Generator-Werkzeugen implementiert. Dadurch ist es nicht möglich ein Ecore-Modell alleine auf Basis der IML-Spezifikation zu erzeugen. Ein möglicher Ansatz, der dieses Problem umgeht, wäre die

[1] von: Atlas Transformation Language
[2] http://www.eclipse.org/atl/
[3] von: Object Constraint Language
[4] http://www.omg.org/spec/OCL/2.3.1/

Verwendung der generierten Ada-Pakete als Quelle für die Modellgenerierung. Mit einer entsprechenden Ada-Grammatik könnte ein Parser generiert werden, welcher die Pakete zerlegt und aus den daraus gewonnenen Informationen das Ecore-Modell generiert. Neben der Erzeugung eines Ecore-Modells, würde für eine Verwendung von ATL ein weiteres Werkzeug zur "Befüllung" der Modelle benötigt. Ecore-Modelle werden als XMI-Datei serialisiert. Diese Serialisierung müsste auch für IML-Graphen implementiert werden und alle existierenden IML-Konstrukte verarbeiten. Ein solches Unterfangen wäre im Rahmen einer Teilaufgabe zu aufwendig.

Neben dieser Begründung gegen den Einsatz einer bestehenden Transformationssprache gibt es noch weitere Argumente welche die Implementierung einer eigenen Sprache befürworten. Bei der Umsetzung einer eigenen DSL können die konkreten Anforderungen für die Manipulation von IML berücksichtigt werden. Zudem wird kein weiterer Generator benötigt, welcher gegebenfalls für neue IML-Konstrukte nachträglich angepasst werden muss. Außerdem kann eine eigene Sprache in zukünftigen Arbeiten um weitere Konstrukte erweitert werden.

Nachdem die Evaluation zu Gunsten einer Eigenentwicklung ausgefallen war, musste im weiteren Vorgehen der Syntax der Sprache festgelegt werden. Die Überlegung einer rein regelbasierten Sprache im Stil von ATL wurde verworfen. Bei der Analyse der Transformationen für die Abbildung des Zustandsautomaten-Sprachpakets auf Basiskonstrukte wurde festgestellt, dass die diese nicht den komplexen Anforderungen der Umwandlung entsprach. ATL-Regeln definieren einen Quell- sowie einen Ziel-Knotentyp. Für jeden Quell-Knoten des Ausgangsgraphen (auf den eventuelle Filter zutreffen) wird eine Instanz des Ziel-Knotentyps erzeugt und anhand der Operationen konfiguriert. Bei der Transformation von IML gibt es viele Situationen die sich mit diesem Prinzip nur umständlich oder sogar gar nicht umsetzen lassen. Der gewählte Syntax entstand in Anlehnung an Sprachen wie Ada und Pascal. Diese setzten bei den Schlüsselworten auf lesbare, ausgeschriebene Worte. Quellcodes solcher Sprachen können beinahe Fließtext-artig gelesen werden. Für Neueinsteiger erleichtert dies den Einstieg in die Sprache.

Ebenso wie der Syntax ist auch die Funktionsweise von *IMLTransform* an die imperative Programmierung angelehnt. DSLs werden häufig mit dem Gedanken entwickelt, auch von Nutzern ohne Programmierkenntnisse einsetzbar zu sein [Fow10]. Im Falle der Transformationssprache ist ein solches Szenario eher unwahrscheinlich. Dementsprechend wurde auf eine Vereinfachung des Syntax zugunsten eines mächtigeren Funktionsumfangs verzichtet. Diese wird durch die Kombination bekannter Konstrukte (z.B. `if`-Abfragen, `For`- und `While`-Schleifen) mit speziellen, für die IML-Manipulation optimierten, Konzepten erreicht.

Um durch die Transformation der Graphen möglichst nicht destruktiv arbeiten zu müssen wurde ein Konzept zur Erhaltung der abstrakten Knoten entworfen. Dieses erlaubt es den Graphen auf Basisknoten zu transformieren ohne dabei die Informationen der abstrakteren Darstellung zu verlieren. Das Konzept stützt sich auf zwei Säulen. Zum ersten werden bestimmte Knoten entsprechend ihrer Semantik in die Vererbungshierarchie von IML eingebracht. Ein Beispiel wären die Knotenklassen `O_StateMachine` und `T_StateMachine` aus

dem Zustandsautomaten-Sprachpaket. Diese werden von `O_Class` bzw. `T_Class` abgeleitet. Dadurch können sie wie solche behandelt und verarbeitet werden. Ein Java-Code-Generator würde aus ihnen Java-Klassen generieren. Dadurch ist die Beziehung zu den Basisknoten vorhanden und es ist trotzdem möglich die abstrakte Darstellung beizubehalten.

Die zweite Säule befasst sich mit Knotenklassen die nicht von IML-Basiskonstrukten abgeleitet werden können. Ein Grund hierfür kann zum Beispiel sein dass eine solche Festlegung die Umsetzung durch den Generator zu sehr einschränken würde. Als Alternative wurde ein Interface für die Knotenklasse `IML_Root` entwickelt. Dieses erlaubt es einen Knoten über eine syntaktische Liste mit anderen zu verbinden. Dadurch kann man eine Verbindung zwischen dem abstrakten Konstrukt und den ihm zugehörigen Basisknoten schaffen. Die Liste in der diese Verbindungen gespeichert sind trägt den Bezeichner `Extension_Nodes`.

8.2 Technologien

In diesem Abschnitt werden die Technologien beschrieben mit denen die Transformations-Sprache und die Generierung des Codes umgesetzt wurde. Dabei wird kurz begründet warum diese verwendet wurden und welche Aufgaben sie erfüllen. Eine komplette Beschreibung würde den Rahmen dieses Kapitels und der gesamten Arbeit sprengen. Interessierte Leser und spätere (Weiter-)Entwickler dieses Projekts finden in den verwendeten Quellen ausführliche Informationen zu den einzelnen Themen.

8.2.1 XText

Für die Umsetzung der Transformationssprache wurde das *XText* Framework verwendet. Entwickelt wird **XText** von Mitarbeitern der itemis AG [5] als quelloffenes Eclipse Projekt [6] [AG12b].

Der Einsatzzweck von *XText* ist die Realisierung von Domain-Specific Languages (s. Kapitel 2.1). Hierfür werden eine Reihe von Werkzeugen bereitgestellt. Diese erzeugen aus einer Grammatik unter anderem einen entsprechenden Parser, das zugehörige Meta-Modell sowie einen grafischen Editor für die Eclipse Entwicklungsumgebung. Basis von *XText* ist das Eclipse Modelling Framework (EMF). Trotz der engen Verbindung zu Eclipse kann *XText* unabhängig von diesem in eigenständigen Anwendungen eingesetzt werden [AG12b].

Neben den bereitgestellten Werkzeugen zeichnet sich *XText* durch seine konfigurierbare Architektur aus. Sie erlaubt es das Verhalten der Werkzeuge durch eigene Implementierungen anzupassen [AG12b]. Hierfür wird das Dependency Injection Framework *Google Guice* [7] verwendet.

[5] http://www.itemis.com
[6] http://www.eclipse.org/Xtext/
[7] http://code.google.com/p/google-guice/

Kern eines *XText*-Projekts ist die Grammatik. Diese wird durch eine eigene Spezifikationssprache definiert, welche an EBNF angelehnt ist [AG12b]. Die Bestandteile der Grammatik werden durch Parser- und Lexer-Regeln beschrieben. Letztere bestimmen das Verhalten des Lexers bei der Zerlegung eines Eingabetextes in seine Einzelteile (*Tokens*). Standardmäßig sind Lexer-Regeln Zeichenketten. Durch die implizite Angabe eines Rückgabewerts kann eine Regel aber auch andere Typen wie Ganzzahlen repräsentieren. Eine solche Angabe ermöglicht es dem Modell-Generator den Datentyp der Attribute korrekt zu bestimmen. Beispiele für Lexer-Regeln ist die Definition des Aufbaus von Bezeichnern oder die Angabe der Whitespace-Symbole.

Mit Parser-Regeln werden die Konstrukte der Sprache beschrieben [AG12b]. Im Code-Ausschnitt 8.1 wird die Definition einer Parser-Regel für eine `While`-Schleife dargestellt.

```
1 WhileStatement :
2     'WHILE' '(' condition=Expression ')'
3     'BEGIN'
4         body+=Statement*
5     'END'
6 ;
```

Listing 8.1: Parser-Regel für `While`-Schleifen in der XText Grammatik.

Der Bezeichner vor dem initialen Doppelpunkt stellt den Namen der Regel dar. Dieser wird im generierten Modell als Name des Datentyps verwendet [AG12b]. In den Hochkommas werden die Schlüsselworte für das Konstrukt definiert. Die Bedingung der Schleife wird durch die Anweisung `condition=Expression` definiert. Im erzeugten Modell des Konstrukts wird diese in einem Attribut mit Namen "condition" abgelegt. `Expression` verweist dabei auf eine weitere Regel, welche die Formulierung von Ausdrücken festlegt. Solche Verweise können ein zwei verschiedenen Ausprägungen gesetzt werden [AG12b]. Im verwendeten Beispiel muss der Ausdruck für die Bedingung an dieser Stelle (zwischen den beiden Klammern) definiert werden. Der zweite Fall wäre der Verweis auf eine bereits existierende Instanz einer Regel. Hierfür können Regeln für eine Instanz eine eindeutige ID durch die Anweisung `name=ID` festlegen. Ein Beispiel für eine solche Regel ist die Definition von Routinen deren Bezeichner als ID verwendet wird. Für die Ausführung einer Prozedur könnte eine Regel

```
InvokeProcedure: 'CALL'method=[MethodDefinition];
```

festgelegt werden. Durch die eckigen Klammern wird nach dem "CALL"-Schlüsselwort der Name einer bereits definierten Methode anstatt einer neuen Deklaration erwartet.

Neben der Bedingung enthält eine `JavaWhile`-Schleife noch eine Folge von Anweisungen die ausgeführt werden solange die Bedingung erfüllt ist. Diese wird im beschriebenen Beispiel durch die Anweisung `body+=Statement*` beschrieben. Im generierten Modell findet sich hierfür ein Listen-Attribut "body" in dem die Anweisungen enthalten sind. Der Stern hinter dem Regel-Verweis `Statement` definiert die Kardinalität (0 oder viele).

Für die Generierung von Parser und Lexer aus der Grammatik setzt *XText* auf den Parsergenerator *ANTLR*.

8.2.2 Xtend

Für die Implementierung des Transformations-Generators wird die Sprache *Xtend* eingesetzt. Diese setzt auf Java auf und erweitert dieses durch neue Aspekte und Konstrukte [AG12a]. Wie auch *XText* wird *Xtend* von der itemis AG als Open-Source Projekt entwickelt.

Syntaktisches und semantisch ist *Xtend* an Java-Code angelehnt und erzeugt bei der Kompilierung solchen. Entstanden ist *Xtend* auf Basis von *XText* und wird als Nachfolger von *Xpand* [8] entwickelt [AG12a]. Eine vollständige Beschreibung aller Funktionalitäten und Konstrukte wäre hier bei weitem zu aufwendig. Die offizielle Referenz-Dokumentation in [AG12a] enthält diese. Interessierten Lesern sei diese daher hier nahegelegt. Sie dient auch als Quelle für die folgenden Ausführungen.

Für die Implementierung von Code-Generatoren aus *Xtext*-Modellen bietet *Xtend* eine Vielzahl von Möglichkeiten. Primär ist dies die Bereitstellung von *Template*-Ausdrücken. Diese erlauben es das Ausgabeformat für den Code genau festzulegen. Durch spezielle Platzhalter können die *Templates* dynamisch mit Inhalten befüllt werden. Dabei sind neben dem Einfügen einzelner Textelemente auch komplexere Strukturen wie `For-` und `While`-Schleifen möglich [AG12a].

Auffälligster Unterschied zu Java ist die Definition von Methoden und Variablen [AG12a]. Während in Java deren (Rückgabe-)Typ durch die Deklaration festgelegt wird, wird dieser in *Xtend* automatisch bei der Übersetzung bestimmt. Es ist möglich den Typ selbst festzulegen und die automatische Typbestimmung zu umgehen.

Xtend setzt direkt auf dem Java Development Kit (JDK) auf und arbeitet nahtlos mit diesem zusammen. So können Java-Klassen importiert und Java-Funktionen aufgerufen werden. Zudem steht das komplette Typsystem von Java auch in *Xtend* zur Verfügung.

Neben der sprachlichen Neuerungen existiert für *Xtend* ein mächtiger Editor für die Eclipse Entwicklungsumgebung. Dieser bietet die alle wichtigen Funktionalitäten wie Refactoring, Aufruf-Hierarchien und Syntax-Highlighting. Der eingesetzte Debugger ist zudem in der Lage aus dem erzeugten Java-Code auf die zugehörige Stelle in *Xtend* zu springen [AG12a].

Zusätzlich zu den bereits erwähnten *Template*-Ausdrücken bietet *Xtend* noch eine Reihe weiterer Aspekte. Für diese Arbeit waren dabei besonders die Konzepte der Erweiterungsmethoden von Bedeutung. Dieses erlaubt es Typen von außen (sprich: ohne Veränderung) mit neuen Methoden zu versehen. Eine solche Methode muss als erstes Argument eine Instanz des zu erweiternden Typs erwarten. Erweiterungsmethoden können dabei in der zu erweiternden oder als statische Methoden extern in einer anderen Klasse definiert werden. Im Ausschnitt 8.2 wird Definition und Aufruf einer solchen Methode dargestellt [AG12a].

[8] http://www.eclipse.org/modeling/m2t/?project=xpand

```
class ExtendedClass {
    def doSomething(MyClass arg, int number) {
        //...
    }

    def anotherMethod() {
        var inst = new MyClass();
        inst.doSomething(10);
    }
}
```

Listing 8.2: Definition einer Erweiterungsmethode in *Xtend*.

Die Methode `doSomething` kann hier ohne den ersten Parameter aufgerufen werden. Dieser wird implizit anhand der aufrufenden Instanz der Klasse `ExtendedClass` gesetzt.

8.2.3 ANTLR

ANTLR [9] ist ein Parsergenerator, welcher in der Lage ist aus einer Grammatik Parser und Lexer für diese in verschiedenen Sprachen zu erzeugen. Er wird seit 1989 von Terrence Parr an der Universität San Francisco entwickelt wird. Alle Informationen in diesem Abschnitt beziehen sich auf [Par07]. Ausnahmen werden explizit gekennzeichnet. Einsatzgebiete für *ANTLR* sind die Implementierungen von Interpretern, Kompilern und Übersetzern im Allgemeinen. Häufig geschieht dies im Kontext der Entwicklung von Domain-Specific Languages. Neben der Erzeugung von Parser und Lexer können mit *ANTLR* abstrakte Syntaxbäume und entsprechende *TreeParser* für diese generiert werden.

Definiert werden *ANTLR*-Grammatiken in einem regelbasierten Syntax. Ein Beispiel für eine solche Regel wird in Ausschnitt 8.3 dargestellt. Der Name der Regel wird vor dem Doppelpunkt angegeben. Die Ausdrücke "identifier" und "path" verweisen auf weitere Regeln welche an dieser Stelle ausgewertet werden. Die darauffolgenden Konstrukte bezeichnen optionale Bestandteile welche hier auftreten können. In den Anführungszeichen werden Schlüsselworte definiert.

```
builtin_name :
    identifier ('with name' path)? ('in package' path)?
;
```

Listing 8.3: Beispiel einer Grammatik-Regel in *ANTLR*.

In die Regeln können zudem Operationen eingebettet werden. Diese werden ausgeführt wenn die Regel vom Parser erkannt wurde. Wie bereits erwähnt ist *ANTLR* in der Lage Parser und Lexer für eine Vielzahl von Programmiersprachen zu erzeugen. Hierzu gehören unter anderem Java, Python, C/C++ und Ada95. Das Ausgabesprache wird dabei innerhalb

[9] von: Another Tool for Language Recognition

der Grammatik spezifiziert. Als Operationen können Anweisungen aus der Zielsprache verwendet werden.

Innerhalb der Transformationssprache wird *ANTLR* indirekt als Teil des *XText* Framework verwendet. Dieses erzeugt aus der *XText*-Grammatik der Sprache eine entsprechende Grammatik für *ANTLR*. Aus dieser werden dann Parser und Lexer erzeugt. Bei der Arbeit mit *XText* ist keine Manipulation der *ANTLR*-Grammatik nötig, da diese dynamisch erzeugt wird. Ein Verständnis von ANTLR und seiner Arbeitsweise kann aber hilfreich sein um Fehlermeldungen während der Generierung zu verstehen.

Häufigstes Problem, welches zu einem solchen Fehler führen kann, sind uneindeutige Regeln in der Grammatik. Bei diesen ist der Parser nicht in der Lage zwischen den verschieden Alternativen einer Regel zu unterscheiden. Der Ausschnitt in 8.4 zeigt eine solche fehlerhafte Regel.

```
1 method :
2       type ID '(' args ')' ';'
3     | type ID '(' args ')' '{' body '}'
4 ;
5
6 type: 'void' | 'int' ;
7 args: arg (',' arg)* ;
8 arg: 'int' ID ;
9 body: ... ;
```

Listing 8.4: Beispiel einer unentscheidbaren *ANTLR*-Grammatik.

Der verwendete Syntax ist dabei korrekt, die Erzeugung bricht allerdings mit einer Fehlermeldung ab. Grund hierfür ist die Rekursion innerhalb der Regel `args`. Standardmäßig erzeugt *ANTLR* einen $LL(k)$-Parser. Aufgrund der variablen Anzahl der Argumente ist es nicht möglich ein konstantes k für den Lookahead festzulegen. Dadurch kann der Parser nicht sicherstellen dass er in den Tokens weit genug nach vorne schaut um zu entscheiden welche der beiden Alternativen für `method` zutrifft.

Gelöst werden kann dieses Problem auf verschiedene Weisen:

- **Backtracking:** Beim *Backtracking* prüft *ANTLR* die angegeben Alternativen in der Reihenfolge ihrer Spezifikation. Wird an einer Stelle bemerkt dass sich die Alternative nicht anwenden lässt, so springt der Parser zum Ausgangspunkt zurück und versucht die nächste Option. Im Falle des Beispiels in 8.4 würde dies dazu führen, dass bei jedem Rücksprung `type ID '(' args ')'` erneut ausgewertet wird. Mit steigender Anzahl der Alternativen führt dies zu einer erheblichen Verschlechterung der Performance. Bei eingeschaltetem Backtracking liegt die Laufzeit des Parser gerade mal in $\mathcal{O}(2^n)$.
- **Left-Factoring:** Bei diesem alternativen Lösungsansatz werden die Alternativen der Regel zusammengefasst. Dadurch müssen die identischen Bestandteile nicht mehrfach ausgewertet werden. Problem dieses Konzepts ist die schlechtere Lesbarkeit der Regeln.

Zudem ist die Einbettung von Operationen in zusammengefassten Regeln schwerer. Angewandt auf das Beispiel 8.4 entsteht folgende Regel:

```
method: type ID '(' args ')' (';' '{' body '}')|
```

- **Semantische Prädikate:** Neben dem *Left-Factoring* kann die Problematik auch durch die Definition von semantischen Prädikaten umgangen werden. Hierfür muss der Entwickler Hilfsmethoden definieren, welche den Lookahead durchführen und prüfen ob eine Alternative angewandt werden kann. Dazu schauen diese im Eingabestream der Tokens ob die benötigten Elemente für die Alternative gefunden werden. Obwohl diese Methode zur Problemlösung einsetzbar ist, gibt es doch Situationen in denen sie nicht für die Entscheidung ausreicht. Ein Beispiel hierfür findet sich in [Paro7].
- **Syntaktische Prädikate:** Der Lösungsansatz, welcher von [Paro7] und [AG12b] empfohlen wird, ist der Einsatz von syntaktischen Prädikaten. Diese verhalten sich ähnlich wie ihr semantisches Pendant, und werden intern auch als ein Spezialfall dieser umgesetzt. Einfach gesagt markieren syntaktische Prädikate Situationen in denen *Backtracking* eingesetzt werden soll. Realisiert wird diese Technik durch den Einsatz von Kellerautomaten [10]. Ein klassisches Beispiel für den Einsatz von syntaktischen Prädikaten findet sich in [AG12b]. Dieses wird als *Dangling Else Problem* bezeichnet. Programmiersprachen wie C erlauben es bei `If`-Konstrukten mit einer einzelnen Anweisung die geschweiften Klammern wegzulassen. Dadurch ist der Code in 8.5 valide.

```
1 if( cond_one() )
2     if( cond_two() )
3         doSomething();
4     else
5         doSomethingElse();
```

Listing 8.5: Verschachtelte `If`-Konstrukte beim *Dangling Else Problem*.

Für den Parser ist die Entscheidung, zu welchem der `If`-Konstrukte das `Else` gehört problematisch. Gelöst werden kann dies durch den Einsatz eines syntaktischen Prädikats. Dieses wird in der *XText*-Regel in Ausschnitt 8.6 gezeigt. Dabei wird per Lookahead nach dem Schlüsselwort "else" gesucht. Wird dies gefunden werden alle sonstigen Alternativen ignoriert.

```
1 'if' '(' condition=BooleanExpression ')' then=Expression
2 (=>'else' else=Expression)?
```

Listing 8.6: Lösung des *Dangling Else Problem* mit syntaktischen Prädikaten.

[10] Im Vergleich: $LL(*)$-Parser nutzen deterministische endliche Automaten

8.3 Code-Generierung

Aufgabe von *IMLTransform* ist die Erzeugung von Dateien aus einem Eingabe-Skript. Das Format und die Aufgabe dieser Dateien wird in diesem Abschnitt beschrieben. Bei der Festlegung der Ausgabe-Sprache musste zwischen den Alternativen Java und Ada entschieden werden. Nach Abwägung verschiedener Faktoren fiel die Wahl auf Java. Gründe für diese Entscheidung waren:

- **Entwicklungsumgebung:** Für die Spezifikation der Sprache wird das Open-Source-Framework *Xtext* und für die Code-Generierung die DSL *Xtend* eingesetzt. Für beide wurden die bereitgestellten Eclipse-Plugins verwendet. Eclipse bietet eine exzellente und vielfältige Werkzeugauswahl für die Entwicklung von Java-Anwendungen. Die Möglichkeit, diese für die Fehlersuche im generierten Code einsetzen zu können, sprach für die Wahl von Java. Zudem war es so möglich die Entwicklung die in einem einzigen Werkzeug durchführen zu können.
- **Aktuelle Entwicklungen:** Mit Einführung des Bachelors und der Umstellung des Lehrplans wurde der Fokus weiter in Richtung Java gerückt. Zudem gehört Java zu den am weitesten verbreiteten Programmiersprachen der heutigen Zeit [11]. Zukünftige Mitarbeiter und Studenten, welche für die Abteilung tätig werden, sind statistisch gesehen mit hoher Wahrscheinlichkeit mit Java vertraut und können Fehler im erzeugten Code leichter entdecken und beheben. Java kann auf lange Sicht demnach zu einer besseren Wartung der Sprache führen.
- **Präferenz des Autors:** Wie bereits im vorherigen Punkt aufgeführt, nutzen eine Vielzahl von Programmierern Java und verfügen über die entsprechenden Erfahrungen und Fähigkeiten. Der Autor dieser Arbeit bildet hier keine Ausnahme. Durch eine Vielzahl von privaten, schulischen und auch beruflichen Projekten konnte ich meine Fähigkeiten mit Java verbessern. Dies erleichterte die Übertragung der Transformations-Anweisungen in ausführbaren Code und war durchaus ein Argument, welches für den Einsatz von Java sprach.
- **Mächtigkeit der Sprache:** Aufgrund ihrer Popularität befindet sich Java im stetigen Wachstum. Mit hoher Frequenz werden neue Versionen veröffentlicht, in denen die Sprache durch neue und mächtige Funktionalitäten erweitert wird. Weitere Entwicklungen der Transformations-Sprache können von diesen profitieren. Als hilfreich bei der Code-Generierung erwies sich die Möglichkeit in Java lokale Variablen nicht in einem separaten Bereich (z.B. in einem `declare`-Block oder zu Beginn eines Funktion bzw. Prozedur in Ada) definieren zu müssen. Bei der Generierung des Transformations-Codes werden häufig temporäre Variablen eingesetzt. Diese können bei Java direkt zusammen mit den entsprechenden Anweisungen an der gleichen Stelle erzeugt werden.

Trotz dieser Pro-Argumente darf nicht übersehen werden dass es genauso Punkte gibt die für Ada sprechen. Primär ist hier zu nennen dass Ada einen mächtigeren Funktionssatz

[11] Aussage basierend auf: http://www.tiobe.com/index.php/content/paperinfo/tpci/index.html

für die Arbeit mit IML besitzt. Java benötigt das JNI-Binding für die Manipulation von IML-Graphen. Ada hingegen kommt ohne den Einsatz solcher Adapter aus.

Neben der Generierung des Programm-Codes erzeugt der *IMLTransform*-Generator noch weitere Dateien. Diese werden benötigt um die generierten Transformatoren ohne manuellen Extraaufwand kompilieren und ausführen zu können.

8.3.1 Java-Code

Wie bereits erwähnt umfasst die aktuelle Implementierung des Transformations-Generators die Erzeugung von Java-Code aus den Transformationsbeschreibungen. Im Abschnitt 8.4 wurden dabei die einzelnen Sprachkonstrukte und ihre Umsetzung in Java im Detail beschrieben.

Transformations-Skripts werden immer als einzelne Java-Datei mit einer Klasse umgesetzt. Damit diese ausgeführt werden kann definiert sie eine `main`-Methode als Einstiegspunkt. Die Datei bindet dabei alle benötigten IML-Packages, sowie die zusätzlich im `@IMPORT`-Block der Transformation spezifizierten Java-Pakete ein.

Erster Schritt der Ausführung ist das Starten des JNI-Bindings. Dieses wird für alle IML-Operationen benötigt. Danach wird eine Instanz der Transformations-Klasse erzeugt. Wie weiter oben beschrieben werden alle `ACTION`-Anweisungen der Transformation als private Methoden umgesetzt. Diese werden in Reihenfolge ihrer Definition aufgerufen. Sind diese abgeschlossen so wird der resultierende Graph im, über einen Kommandozeilen-Parameter angegebenen, Zielpfad gespeichert. Die Ausführung der `ACTION`-Anweisungen ist von einem `try-catch`-Block umschlossen. Tritt ein Fehler auf bricht das Programm ab, schließt das JNI-Interface und beendet mit einer Fehlermeldung. In diesem Fall wird der Zwischenstand des transformierten Graphen nicht gespeichert.

8.3.2 Makefile

imltransform ist kein Kompiler, welcher ausführbaren Maschinen erzeugt, sondern ein Generator der anhand der Transformations-Beschreibungen Java-Code erzeugt. Um diesen Code in eine ausführbare Form zu überführen, muss der Java-Kompiler *javac* ausgerufen werden. Dieser benötigt einige spezielle Parameter um die verwendeten Bibliotheken des JNI-Bindings für IML korrekt einzubinden. Hierfür wird eine angepasste *Makefile* automatisch vom Transformations-Generator angelegt. Dies erspart dem Nutzer der Sprache unnötigen Extraaufwand.

8.3.3 Shell-File

Um das Transformator-Programm direkt nach seiner Kompilierung durch einen Aufruf des *make*-Werkzeugs ausführen zu können, erzeugt der Transformations-Generator eine

speziell angepasste Shell-Datei. Diese stellt sicher dass alle benötigten Libraries als Build-Path verfügbar sind. Dies ist notwendig um das JNI-Binding für IML im Transformator verwenden zu können. Im Abschnitt 8.7.2 wird erläutert wie die Shell-Datei aufgerufen wird und welche Parameter dieser übergeben werden müssen.

8.3.4 Ada

Wie bereits erwähnt fiel die Wahl der Ausgabesprache auf Java. Um die nachträgliche Generierung von Ada zu ermöglichen, wurden bei der Spezifikation der Sprache und der Implementierung des Generators Schnittstellen für Ada angelegt. Spätere Arbeiten können hier ansetzen und mit geringerem Aufwand die Erzeugung von Ada ermöglichen. Innerhalb einer Transformationsbeschreibung kann festgelegt werden in welche Sprache diese übersetzt werden soll. Dies wird durch das Attribut `LANGUAGE` im `@Config`-Block definiert. Bei Verarbeitung eines Skripts wird diese Einstellung ausgewertet und der entsprechende Generator geladen. Die Architektur für eine Generierung von Ada-Code ist dementsprechend bereits vorhanden und könnte, zum Beispiel als Teil einer studentischen Arbeit, nachträglich umgesetzt werden.

8.4 Syntax der Sprache

In diesem Abschnitt wird der Syntax der umgesetzten Transformationssprache beschrieben. Dabei wird auf detailliert auf die einzelnen Konstrukte, ihre Bestandteile und die Umsetzung im generierten Java-Code erläutert. Alle beschriebenen Konstrukte ihre Notation beziehen sich auf den Stand als dieses Dokument geschrieben wurde. Kleinere Änderungen, welche während der Tests nötig werden, können eventuell nicht mehr hier nachgepflegt werden.

Für eine bessere Übersichtlichkeit ist dieses Thema auf mehrere Unterkapitel unterteilt. Als Notation für die Beschreibung der Konstrukte wird eine abgewandelte Form der erweiterte Backus-Naur-Form verwendet. Hierbei wird auf die Kommas zur Trennung von Bestandteilen verzichtet. Durch umschließende Klammern können Elemente gruppiert werden.

Die Bezeichner die hinter dem aktuell beschriebenen Konzept in Klammern stehen sind der Symbolname unter dem die Konstrukte in der Syntax-Notation verwendet werden.

Transformationen basieren dabei auf *Aktionen*. Diese bezeichnen einen Verarbeitungsschritt der Transformation. Dadurch ist es gerade bei aufwendigen Manipulationen möglich die Aufgaben sinnvoll in einzelne Module aufzuteilen. *Aktionen* werden bei der Ausführung der Transformation sequentiell in Reihenfolge ihrer Definition ausgeführt.

Transformationsskripte werden als Dateien mit der Endung ”.itr” gespeichert.

8.4.1 Lexikalische Elemente

Transformationen werden in textueller Form gespeichert und müssen vor ihrer Verarbeitung durch den generierten Parser zuerst von einem Lexer in ihre Bestandteile (*Tokens* zerlegt werden. In diesem Abschnitt werden die lexikalischen Elemente anhand derer aus dem Eingabetext die einzelnen *Tokens* extrahiert werden.

Bezeichner (`name`): Für die Benennung von Elementen können eindeutige Bezeichner vergeben werden. Bezeichner müssen mit aus mindestens einem Buchstaben bestehen und mit einem solchen beginnen. Darauffolgend können beliebig viele Buchstaben, Zahlen und Unterstriche verwendet werden. Im erzeugten Java-Code werden die Bezeichner in der gleichen Schreibweise übernommen.

Beispiel für gültige Bezeichner sind:

```
counter     TestVar      timeStamp
idxOf10  create_Backup  find_1_Node
```

String-Literale (`string_literal`): Zeichenketten können in IMLTransform durch einfache und doppelte Anführungszeichen umschlossen werden. Innerhalb von Zeichenketten sind alle Zeichen erlaubt. Wird das einfache Anführungszeichen verwendet so können innerhalb der Zeichenkette auch doppelte Anführungszeichen verwendet werden und umgekehrt.

Numerische Literale (`numeric_literal`): Für numerische Literale sind rationale sowie ganze Zahlen zulässig. Als Trennzeichen für rationale Zahlen wird ein Punkt verwendet. Für negative Zahlen wird das Minus-Vorzeichen verwendet.

Kommentare: IMLTransform unterstützt ein- und mehrzeilige Kommentare. Einzeilige Kommentare werden durch zwei Schrägstriche eingeleitet. Alle weiteren Inhalte der Zeile werden ignoriert. Mehrzeilige Kommentare beginnen mit `/*` und enden mit `*/`. Hierbei wird der umschlossene Text bei der Verarbeitung nicht beachtet.

Schlüsselworte, Trennzeichen und Begrenzungszeichen: Als Begrenzungszeichen wird der Semikolon verwendet. Alle Anweisungen in IMLTransform müssen mit diesem abgeschlossen werden. Trennzeichen bestehen aus einem oder zwei Symbolen. Dabei sind die folgenden Trennzeichen definiert:

```
(   )   *    +    -    .    /    :    ;   ->  [   ]
<   >   ==   &&   ||   :=   >=   <=   !=   {   }   !
```

Schlüsselworte bestehen aus alphanumerischen Zeichen und Unterstrichen. Sie dürfen nicht als Bezeichner verwendet werden und sind unabhängig von Groß- und Kleinschreibung. IMLTransform verwendet die folgenden Schlüsselworte:

TRANSFORMATION	IS	BEGIN	END	TO	PACKAGES
LANGUAGE	JAVA	ADA	TARGET	SOURCE	SET
IMPORT	ELSE	IF	THEN	RETURN	NEW_GRAPH
AS	FOR	WHILE	DIE	CONTINUE	_TARGET_
BREAK	PROCEDURE	FUNCTION	CREATE	REMOVE	_SOURCE_
DO	SELECT	WRITE	WRITELN	APPEND	CASE
INSERT	INTO	FIRST	FROM	SET	WHERE
ON	FAILURE	SUCCESS	ACTION	IMPORT	DEFAULT
ACTIONS	IMPORTS	ROUTINES	CONFIG	INIT	SWITCH

8.4.2 Aufbau eines Transformationsskripts

Dieser Abschnitt beschreibt den generellen Aufbau einer Transformationsdatei. Als *Transformation* wird dabei eine Reihe von Aktionen beschrieben, welche einen Eingabegraphen in einen Zielgraphen überführen.

Transformation (`transformation`): In einem Transformations-Skripts ist genau eine `transformation` enthalten. Innerhalb dieser können Variablen, Unterprogramme und die Aktionen definiert werden. Für eine `transformation` können globale Variablen definiert werden. Diese sind in allen Unterkonstrukten sichtbar. Bei der Übersetzung nach Java wird für die `Transformation` eine Klasse mit dem selben Namen angelegt. Diese enthält eine `main`-Methode und akzeptiert zwei Kommandozeilenargumente: Den Pfad zur IML-Datei für den Eingabegraphen sowie den Pfad an dem der Zielgraph gespeichert werden soll.

```
transformation =
  "TRANSFORMATION" name "IS"
    {variable_def}
  "BEGIN"
    config_block
    [imports_block]
    [routines_block]
    [init_block]
    actions_block
  "END" ";"
```

Konfigurations-Block (`config_block`): Im `config_block` kann die Übersetzung des Skripts konfiguriert werden. Diese Einstellungen werden zu Beginn der Verarbeitung ausgewertet und beeinflussen den Generator. Das Format für die einzelnen Konfigurationsanweisungen ist `Key=Value`. Zusätzlich kann innerhalb des Konfigurations-Blocks eine Liste von Paket-Mappings angegeben werden. Diese wird benötigt um die Vollqualifizierung für Typen zu ermöglichen.

```
config_block =
  "CONFIG" "{"
    [config_list]
    [package_block]
  "}"
config_list = config_language | config_target | {config_list}
config_language = "LANGUAGE" "=" ("JAVA" | "ADA")
config_target  = "TARGET" "=" ("SOURCE" | "NEW_GRAPH")
```

In der aktuellen Version existieren dabei zwei verschiedene Konfigurationseinstellungen.

- Ausgabesprache (`config_language`): Legt die Ausgabesprache für den Transformations-Code fest. Möglich sind hier `JAVA` und `ADA`. Standardwert ist dabei `JAVA`.
- Zielgraph (`config_target`): Diese Einstellung legt fest ob als Zielgraph eine neuer Graph (`NEW_GRAPH`) verwendet wird oder die Transformationen auf dem Eingangsgraph durchgeführt werden. Im letzteren Fall sind die globalen Graphe-Variablen `_TARGET_` und `_SOURCE_` identisch.

PackageMapping-Block (`package_block`): Unterhalb des Konfigurations-Blocks kann ein Block für die Definition von Package-Mappings enthalten sein. Bei der Auflösung von Typnamen wird dabei zuerst geprüft ob der Typ auf ein spezielles Paket gemappt werden soll. Wird kein Mapping gefunden, so wird ein Standard-Paket angenommen. Im Falle von IML-Klassen ist dies "bauhaus.iml". Wenn für einen externen Datentyp kein Mapping definiert wurde, so wird kein vollqualifizierter Name verwendet. Für die Benutzung externer Typen muss ein entsprechender Import im Import-Block angeben werden.

```
package_block =
  "PACKAGES" "{"
      {package_mapping}
  "}"
package_mapping = (iml_type | external_type) "=" string_literal ";"
```

Beispiel für einen Konfigurations-Block mit Package-Mappings:

```
1 CONFIG {
2   LANGUAGE=JAVA;
3   TARGET=NEW_GRAPH;
4   PACKAGES {
5     IML::RaceInterface="bauhaus.concurrency";
6     EXT::Vector="java.util";
7   }
8 }
```

Import-Block (`imports_block`): Um Typen und Programme der Zielsprache nutzen zu können, werden in diesem Abschnitt die einzubindenden Pakete bzw. Klassen angegeben.

Die Angabe von Importen ist optional. Imports werden im generierten Code als Java-`Import`-Anweisung für das spezifizierte Paket umgesetzt.

```
imports_block =
  "IMPORTS" "{"
    [import_list]
  "}"
importList = import | {import_list}
import = "IMPORT" string_literal ";"
```

Routinen-Block (`routines_block`): Innerhalb einer Transformation können Unterprogramme definiert werden. Diese ermöglichen es wiederkehrende Aufgaben auszulagern. Dadurch wird eine Wiederverwendbarkeit von Code ermöglicht. Dies reduziert den Aufwand und erhöht die Lesbarkeit. In IMLTransform werden dabei Prozeduren und Funktionen unterschieden. Funktionen müssen dabei einen Rückgabewert angeben.

```
routines_block =
  "ROUTINES" "{"
    [routine_list]
  "}"
routine_list = procedure_def | function_def | {routine_list}

procedure_def =
  "PROCEDURE" name "(" [parameter_list] ")" ";"
    {variable_def}
  "BEGIN"
    {statement}
  "END" ";"

function_def =
  "FUNCTION" name "(" [parameter_list] ")" ":" data_type ";"
    {variable_def}
  "BEGIN"
    {statement}
  "END" ";"

parameter_list = parameter {"," parameter_list}
parameter = name ":" data_type
```

Umgesetzt werden Routinen im generierten Java-Code als private Funktionen der erzeugten Transformations-Klasse. Für Prozeduren wird dabei als Funktions-Typ `void` verwendet. Der Typ von Funktionen und Parametern wird entsprechend der Beschreibung der Datentypen im Abschnitt 8.4.3 umgesetzt. Da Java keine Festlegung des Übergabemodus für Parameter erlaubt werden diese alle automatisch als *Call by value* übergeben.

Beispiel für Funktionen und Prozeduren:

```
1  FUNCTION EmptySLoc() : IML::SLoc;
2  BEGIN
3      RETURN IML::SLoc(0,0,0,
4         IML::Identifier(""),
5         IML::Identifier(""), null);
6  END;
7
8  PROCEDURE sampleProc(param_a : int, param_b : boolean);
9  BEGIN
10   //...
11 END;
```

Initialisierungs-Block (`init_block`): Durch Angabe eines Initialisierungs-Blocks kann sichergestellt werden, dass bestimmter Code vor Durchführung der Aktionen in jedem Fall ausgeführt wurde. In der aktuellen Realisierung ließe sich dies auch innerhalb der zuerst definierten Aktion erledigen. Im Falle späterer Anpassungen bei der die Aktionen nicht mehr automatisch in der Reihenfolge ihrer Definition aufgerufen werden, kann der `INIT`-Block die initiale Ausführung sicherstellen. Ein Beispiel für die Verwendung dieses Blocks ist die Initialisierung der globalen Variablen.

```
init_block =
  "INIT" "{"
    {statement}
  "}"
```

Aktionen-Block (`actions_block`): Aktionen stellen das Kernstück der Transformation dar. Sie erlauben die Strukturierung des Codes. Wie bereits mehrfach erwähnt ergibt sich der Ausführungszeitpunkt aus der Reihenfolge ihrer Definition. Aktionen werden im `ACTIONS`-Block der Transformation definiert. Sie besitzen einen eindeutigen Bezeichner anhand dessen sie später im Java-Code als eigenständige Funktionen umgesetzt werden. Es ist in Aktionen zudem möglich lokale Variablen zu deklarieren.

```
actions_block =
  "ACTIONS" "{"
    {action_def}
  "}"
action_def =
  "ACTION" name ";"
    {variable_def}
  "BEGIN"
    {statement}
  "END" ";"
```

8.4.3 Datentypen

In IMLTransform werden zur Zeit drei Arten von Datentypen unterstützt. Diese können für die Definition von Variablen, Funktionstypen, Parametern, Typ-Prüfungen und Casts verwendet werden.

Native Typen (`native_type`): IMLTransform kennt die folgenden nativen Datentypen:

Nativer Datentyp	Java-Datentyp	Beschreibung
`String`	`String`	Repräsentation von Zeichenketten.
`int`	`int`	Ganzzahlen
`double`	`double`	*Double-Precision* Gleitkommazahlen
`float`	`float`	*Single-Precision* Gleitkommazahlen
`char`	`char`	Einzelne Zeichen
`boolean`	`boolean`	Boolscher Datentyp (`true` und `false`)

IML-Typen (`iml_type`): Für die Durchführung von Transformationen muss der generierte Code die durch das JNI-Binding zur Verfügung gestellten IML-Klassen nutzen können. Hierfür existiert in IMLTransform ein eigener Datentyp. Dieser wird durch den Präfix `IML::` markiert. IML-Klassen werden im generierten Java-Code mit ihrem voll-qualifizierten Klassennamen verwendet. Beispiele für IML-Typen sind `IML::Unit` und `IML::O_Class`.

```
iml_type = "IML::"name
```

Externe Typen (`external_type`): Um die Sprache flexibler und mächtiger zu gestalten wird die Einbindung externer Typen unterstützt. Da IMLTransform keine eigenen Array-Typen unterstützt kann dies genutzt werden um die Listen-Klassen von Java einzubinden und zu verwenden. Externe Typen werden mit dem Präfix `EXT::` markiert. Sie werden standardmäßig ohne voll-qualifizierten Namen im erzeugten Java-Code verwendet. Für die Benutzung eines externen Typs muss das Paket, in dem dieser definiert ist, im Import-Block eingebunden werden. In der aktuellen Implementierung werden von IMLTransform keine generischen Argumente unterstützt. Dementsprechend müssen die Java-Klassen, bei denen diese erwartet werden, ohne sie benutzt werden. Dies führt leider zum Verlust der Typensicherheit und erfordert mehr Kontrolle auf Seiten des Anwenders.

```
external_type = "EXT::"name{"." name}
```

8.4.4 Variablen

Variablen sind ein fester und elementarer Bestandteil vieler Programmiersprachen. Sie erlauben das Speichern und Abrufen von Daten.

Deklaration von Variablen (`variable_def`): Variablen werden im erzeugten Java-Code in Abhängigkeit von ihrer Position im Transformations-Skript erzeugt. In IMLTransform können diese global als auch lokal deklariert werden. Eine Variable besitzt einen Bezeichner und einen Datentyp. Als Trennzeichen zwischen diesen wird ein Doppelpunkt verwendet. Der Bezeichner wird im Java-Code übernommen. Der Datentyp der Variable wird entsprechend der Beschreibung im Abschnitt 8.4.3 umgesetzt.

```
variable_def = name ":" (native_type | iml_type | external_type) ";"
```

8.4.5 Zugriff auf Eingabe- und Ausgabegraph

Für den Zugriff auf den Eingangs- und Zielgraph stehen in IMLTransform spezielle Schlüsselworte zur Verfügung. Diese können überall im Skript verwendet werden.

Das Schlüsselwort `_SOURCE_` verweist immer auf den Eingabegraphen der transformiert werden soll. Umgesetzt ist der Eingabegraph als Member-Variable der Transformations-Klasse. Sie wird vor Beginn der Aktionen initialisiert, indem der Graph aus der übergebenen Eingabedatei geladen wird.

Um auf den Ausgabegraphen zugreifen zu können steht entsprechend als Schlüsselwort `_TARGET_` zur Verfügung. Hierbei muss beachtet werden dass die Konfigurations-Einstellung `LANGUAGE` angibt worauf `_TARGET_` zeigt. Standardmäßig ist dies ein neu erzeugte und leerer IML-Graph. Wird aber `LANGUAGE=SOURCE` gesetzt, so zeigen die beiden Schlüsselworte auf den Eingabegraphen. Dies ist nützlich wenn nur kleine Teile des Graphen transformiert werden sollen. Im generierten Java-Code ist für den Ziel-Graphen eine Member-Variable in der Transformations-Klasse definiert.

In den Notations-Beschreibungen sind die beiden Schlüsselwörter mit `input_graph` und `output_graph` bezeichnet.

8.4.6 Anweisungen

Nachdem in der Evaluation ein rein regelbasierter Syntax für die Transformationssprache verworfen wurde, musste ein Weg gefunden werden um die Manipulation von IML-Graphen effektiv umzusetzen. Bei der Betrachtung der Anforderungen für die Abbildung von Zustandsautomaten auf die Basiskonstrukte konnte einige Problematiken im Umgang mit IML erkannt werden. Diese beziehen sich primär auf die Nutzung von IML in Java. Zudem wurde versucht zu identifizieren welche häufig wiederkehrende Aufgabe durch intelligente Konstrukte vereinfacht werden können. Als Ergebnis wurde eine Reihe von Anweisungen entwickelt die Transformationen von IML-Graphen erleichtern.

Anweisungen (`statement`): In IMLTransform werden eine Reihe verschiedener Anweisungs-Konstrukte unterschieden. Diese sind detailliert in diesem und den folgenden Kapiteln

beschrieben. Die folgende Liste enthält alle Konstrukte die als Anweisungen verwendet werden können.

- Zuweisungen
- If-Then-Else-Abfragen
- For-Schleifen
- While-Schleifen
- Switch-Verzweigung
- Ausdrücke
- Insert-Node-Anweisung
- Append-Node-Anweisung
- For-Nodes-Schleife
- Select-Node-Anweisung
- Prozedur-Aufrufe
- Index-Of-Anweisung
- Remove-Node-Anweisung
- Write-Anweisung
- WriteLine-Anweisung
- Die-Anweisung

Daraus ergibt sich die folgende Syntax-Notation:

```
statement = (
  if_then_else | switch  | select_node   | index_of   |
  while_loop   | for_loop | for_nodes_loop | insert_into |
  remove_node | append_to | die          | expression |
  write       | writeline | procedure_call | assignment) ";"
```

Zuweisungen (`assignment`): Durch Zuweisungen können die Werte von Variablen und Klassen-Attributen geändert werden. Als Zuweisungs-Operator verwendet IMLTransform `:=`.

```
assignment = expression ":=" expression
```

Hierbei muss geprüft werden dass auf der linken Seite nur valide Ausdrücke stehen.

Beispiel für Zuweisungen:

```
1 unit.SLoc := IML::MakeSLoc(...);
2 myCounter := (1 + 5) * 10;
```

If-Then-Else-Abfragen (`if_then_else`): Wie auch in anderen Hochsprachen stellt IMLTransform ein Konstrukt für die Verzweigung innerhalb der Anweisungen zur Verfügung. Dabei wird eine Bedingung angegeben. Wertet diese zu *wahr* aus, so werden die Anweisungen im `Then`-Block ausgeführt. Ist dies nicht der Fall, so wird der optional angebbare `Else`-Block betreten. Die Umsetzung in Java erfolgt ebenfalls als äquivalente `If`-Verzweigung.

```
if_then_else =
  "IF" "(" expression ")" "THEN"
  "BEGIN"
    {statement}
  "END"
  ["ELSE" "BEGIN"
    {statement}
  "END"]
```

While-Schleife (`while_loop`): IMLTransform erlaubt die Nutzung von `While`-Schleifen. Neben dem Prädikat der Turing-Vollständigkeit erlauben diese die Definition flexibler Schleifen. Innerhalb des Schleifen-Blocks können neben den oben aufgelisteten Anweisungen auch die speziellen Schleifen-Anweisungen `BREAK` und `CONTINUE` verwendet werden. Erstere beendet die Schleife bei Ausführung. `CONTINUE` springt zum Beginn der Schleife und prüft erneut ob die Bedingung erfüllt ist. Umgesetzt wird dieses Konstrukt im generierten Java-Code erwartungsgemäß als `While`-Schleife.

```
while_loop =
  "WHILE" "(" expression ")"
  "BEGIN"
    {statement | break | continue}
  "END"
  break = "BREAK" ";"
  continue = "CONTINUE" ";"
```

For-Schleife (`for_loop`): `For`-Schleifen stellen eine Sonderform der `While`-Schleife und eine praktische Möglichkeit für die Iteration über festgelegte Bereiche dar. Der Kopf einer `For`-Schleife setzt sich aus drei Teilen zusammen. Erster ist die optionale Definition und Initialisierung einer Zählervariable. Im zweiten Part wird eine Bedingung formuliert. Diese wird vor jedem Schleifen-Durchlauf geprüft. Zum Abschluss ist es noch möglich eine Schrittweite anzugeben. Diese wird als einfach Zuweisung realisiert, welche nach jedem Schleifendurchlauf ausgeführt wird. Wie auch in der `While`-Schleife stehen neben den obigen Anweisungen auch die Kontroll-Anweisungen `BREAK` und `CONTINUE` zur Verfügung. Im erzeugten Java-Code sind Konstrukte dieses Typs durch eine äquivalente `For`-Schleife dargestellt.

```
for_loop =
  "FOR" "("
     variable ")"
  "BEGIN"
    {statement | break | continue}
  "END"
  break = "BREAK" ";"
  continue = "CONTINUE" ";"
```

Beispiel für For-Schleifen:

```
1 FOR(I : int := 0; i < 10; I := I + 1)
2 BEGIN
3   ...
4 END;
```

Switch-Verzweigung (`switch`): Die `Switch`-Verzweigung ist eine weitere Anweisung die in vielen bekannten Sprachen wie Java und C++ verwendet wird. Sie erlaubt es Abfragen mit vielen Verzweigungen besser lesbar darzustellen. Für die Verwendung dieses Konstrukts gelten die gleichen Einschränkungen wie für das `Switch`-Konstrukt in Java [12]. Allerdings ist es auf die Datentypen `char` und `int` beschränkt. Die Umsetzung erfolgt selbstredenderweise als entsprechendes `switch`-Konstrukt in Java.

```
switch =
  "SWITCH" "(" expression ")" "{"
     {switch_group}
  "}"
  switch_group =
    switch_label "BEGIN"
      {statement}
    "END" ";"
  switch_label =
    ("CASE" expression | "DEFAULT") ":"
```

[12] Nachzulesen unter http://docs.oracle.com/javase/tutorial/java/nutsandbolts/switch.html

Beispiel für Switch-Konstrukte:

```
SWITCH( node.visibility ) {
  CASE 0: BEGIN
    WRITELN "Public";
  END;
  DEFAULT: BEGIN
    WRITELN "Alles andere..";
  END;
}
```

For-Nodes-Schleife (`for_nodes_loop`): Die `For-Nodes`-Schleife ist das erste Konstrukt, welches speziell für die Verarbeitung von Graphen entworfen wurde. Sie ist angelehnt an die regelbasierte Methode anderer Transformationssprachen wie ATL. Mit dieser Schleife ist es möglich über eine Liste oder ein Set von Knoten zu iterieren. Der Typ der Iterator-Veriable gibt vor welche Knoten verarbeitet werden. Zusätzlich kann eine Filter-Bedingung angegeben werden. Diese wird für jeden Eintrag evaluiert. Wertet sie zu *wahr* aus, so wird der Code im Schleifen-Block ausgeführt. Dabei kann über einen Iterator auf den aktuellen Knoten zugegriffen werden. Eine weitere Besonderheit ist die automatische Typkonvertierung der durchlaufenen Elemente in den Typ des Iterators.

Im generierten Code wird für dieses Konstrukt eine Schleife erzeugt welche alle Elemente der Liste bzw. des Sets durchläuft. Zuerst geprüft ob der aktuelle Eintrag die gleiche Klasse wie die Iterator-Variable besitzt oder sich davon ableitet. Ist dies der Fall, so wird das aktuelle Elemente in den Typ des Iterators umgewandelt. Im Anschluss wird eine eventuell gesetzte Filter-Bedingung geprüft. Wird auch diese erfüllt, so wird der Schleifen-Code für dieses Element ausgeführt.

```
for_nodes_loop =
  "FOR" variable_def "FROM" ["SET"] expression
  ["WHERE" "(" expression ")"]
  "DO" "BEGIN"
      {statement | break | continue}
  "END"
  break = "BREAK" ";"
  continue = "CONTINUE" ";"
```

Beispiel für die `For-Nodes`-Schleife:

```
  FOR iter : IML::O_Node FROM SET unit.Declaration_Table
  WHERE( iter.Unmangled_Name.Name->equals("myMethod") )
  DO BEGIN
      WRITELN iter.Mangled_Name;
  END;
```

Remove-Node-Anweisung (`remove_node`): Mit diesem Statement ist es möglich auf einfache

Weise Knoten aus einer einer Liste oder einem Set zu entfernen. Hierfür wird ein Iterator verwendet dessen Typ die verarbeiteten Knoten aus der Liste definiert. Nur Einträge welche die selbe Klasse oder eine davon abgeleitete wie der Iterator besitzen werden beachtet. Durch die optionale Angabe eines Filters können zusätzliche Prüfungen vorgenommen werden. Der Filter wird für jedes Element erneut geprüft.

Im Java-Code wird diese Anweisung als Schleife über alle Einträge der Liste bzw. des Sets umgesetzt. Für jeden Eintrag wird der Typ und der Filter geprüft. Sind beide Kriterien erfüllt, so wird der Eintrag entfernt.

```
remove_node =
  "REMOVE" variable_def
  "FROM" ["SET"] expression
  ["WHERE" "("
      expression
  ")"]
```

Beispiel für die Remove-Node-Anweisung:

```
1   REMOVE node : IML::Program_Unit
2   FROM unit.Subunits
3   WHERE (node.Artificial);
```

Select-Node-Anweisung (`select_node`): Das `Select-Node`-Konstrukt kann verwendet werden um einen bestimmten Knoten in einer Liste oder einem Set zu finden. Dabei ist es möglich optional eine Filter-Bedingung anzugeben. Diese wird für jeden verarbeiteten Eintrag ausgewertet. Ist die Suche erfolgreich, so wird der aktuelle Eintrag in der Ziel-Variable gespeichert. Beim Suchen werden nur Knoten geprüft welche die gleiche Klasse wie die Ziel-Variable besitzen oder sich davon ableiten. Sobald der erste passende Eintrag gefunden wurde bricht der Suchvorgang ab.

Es ist zudem möglich Nachbedingungen für den Erfolgs- und Misserfolgs-Fall zu definieren. Als Erfolg ist ein Suchvorgang definiert, bei dem ein passender Eintrag in der Liste bzw. dem Set gefunden wurde. Je nach Ausgang der Suche werden die Anweisungen der entsprechenden Nachbedingungs-Blöcke, in der Reihenfolge ihrer Definition, ausgeführt.

Umgesetzt wird die `Select-Node`-Anweisung als Schleife die über die Menge der Knoten iteriert. Dabei werden der Typ und die Filter-Bedingung für jeden Eintrag geprüft. Wird ein passender Knoten in der Menge gefunden, so bricht die Schleife ab. Nach Beendigung der Schleife werden die Nachbedingungen durch `if`-Abfragen modelliert. Für Erfolg wird dabei geprüft ob die Ziel-Variable ungleich `null` ist.

```
select_node =
  "SELECT" "FIRST" variable_ref
  "FROM" ["SET"] expression
  ["WHERE" "("
      expression
  ")"]
  {handler}
handler =
  "ON" ("SUCCESS" | "FAILURE") "BEGIN"
    {statement}
  "END";
```

Beispiel für die Select-Node-Anweisung:

```
1   SELECT FIRST OStateMachine
2   FROM SET SourceUnit.Declaration_Table
3   ON FAILURE BEGIN
4       DIE "Could not find O_StateMachine in source IML graph!";
5   END;
```

Insert-Into-Anweisung (`insert_into`): Die `Insert-Node`-Anweisung ist eine gut lesbare und vereinfachte Methode für das Einfügen von Knoten in eine Liste oder ein Set. Für Listen muss der Index spezifiziert werden, an dem der Knoten eingefügt werden soll.

Im erzeugten Java-Code werden die Ausdrücke ausgewertet und dann die entsprechende korrekte Funktion für das Einfügen des Knotens in die Ziel-Liste bzw. das Ziel-Set aufgerufen.

```
insert_into =
  "INSERT" expression "INTO" expression
  ["AT"   expression]
```

Append-To-Anweisung (`append_to`): Ähnlich wie die `Insert-Into-Anweisung` dient dieses Konstrukt dem Einfügen von Knoten in Listen. Da Sets ungeordnete Mengen sind existiert für sie keine `Append`-Methode. Grund für die Umsetzung dieses Konstrukts war die Analyse der Anforderungen für die Abbildung der Zustandsautomaten auf ihre Basiskonstrukte. Dabei war der Anwendungsfall Knoten anzuhängen am häufigsten aufgetreten. Dementsprechend wurde hierfür ein passendes Konstrukt umgesetzt.

```
append_to = "APPEND" expression "TO" expression
```

Index-Of-Anweisung (`index_of`): Die `Index-Of`-Anweisung wird verwendet um den Index eines Knotens in einer Liste zu bestimmen. Dies funktioniert nicht für Sets, da diese ungeordnete Mengen darstellen. Wird der Knoten in der spezifizierten Liste gefunden, so wird

der Index in der Ausgabe-Variablen gespeichert. Die Variable muss dementsprechend eine Ganzzahl sein.

Umgesetzt ist dieses Konstrukt in Java als eine Schleife bei der jedes Element mit dem gesuchten Knoten verglichen wird. Wird dabei ein Treffer erzielt, so wird der aktuelle Index in der Ausgabe-Variablen abgelegt.

```
index_of =
  "SELECT" "INDEX" "OF" expression
  "FROM" expression
  "INTO" variable_def
```

Die-Statement (`die`): Um die Ausführung der Transformation vorzeitig abzubrechen kann die `Die`-Anweisung verwendet werden. Im umgesetzten Java-Code wird diese als `throw new Exception(..)` umgesetzt. Als Parameter für den Konstruktor der `Exception` wird der Ausdruck nach dem Schlüsselwort `DIE` übergeben. Bei Auftreten eines Fehlers wird dieser bis zur Main-Methode durchgereicht und die Ausführung der Transformation abgebrochen.

```
die = "DIE" expression
```

Write-Statement (`write`): Das `Write`-Statement ist ein kleines Hilfskonstrukt welches den angegebenen Ausdruck an die Funktion `java.lang.System.out.print` weitergibt. Damit ist es möglich fortlaufenden Text ohne Zeilenumbruch auszugeben. Primär ist dies für eigene Logging-Nachrichten gedacht.

```
write = "WRITE" expression
```

WriteLine-Statement (`writeline`): Dieses Konstrukt funktioniert exakt wie die `Write`-Anweisung. Allerdings ruft sie die Funktion `java.lang.System.out.println` auf. Diese fügt einen Zeilenumbruch vor dem Schreiben ein.

```
writeline = "WRITELN" expression
```

8.4.7 Aufrufe von Unterprogrammen

In diesem Abschnitt wird der Aufruf von Unterprogrammen beschrieben. Dabei werden verschiedene Arten unterschieden:

- Selbst-definierte Routinen
- Methoden
- Statische Methoden
- IML-Konstruktoren

- Externe Konstruktoren

Aufrufe selbst-definierter Routinen (`procedure_call & function_call`): Als selbst-definierte Routinen werden die Prozeduren und Funktionen bezeichnet, welche im Routinen-Block der Transformation beschrieben wurden. Sie sind innerhalb der Transformation überall verfügbar und können über ihren Namen aufgerufen werden. Funktions-Aufrufe können als Ausdrücke und Anweisungen, Prozeduren jedoch nur als Anweisungen verwendet werden.

```
procedure_call = procedure_ref "(" {args} ")"
function_call = function_ref "(" {args} ")"
args = expression | {"," expression}
```

Aufrufe von Methoden (`method_call`): Es ist in IMLTransform möglich Methoden von Klassen aufzurufen. Hierfür kann über einen Ausdruck ein Objekt ausgewählt werden. Als Operator wird `->` verwendet. Der Name der Methode muss in seiner Schreibweise identisch mit dem Namen der aufzurufenden Java-Methode sein.

```
method_call = expression "->" name "(" {args} ")"
args = expression | {"," expression}
```

Aufrufe statischer Methoden (`static_method_call`): Zusätzlich zu Member-Methoden, die auf den Instanzen aufgerufen werden, ist es möglich statische Klassen-Methoden aufzurufen. Hierfür muss der Name einer IML- oder einer externen-Klasse angegeben werden. Wie auch bei nicht-statischen Methoden wird der `->` als Operator verwendet.

```
static_method_call = (iml_type | external_type) "->" name "(" {args} ")"
args = expression | {"," expression}
```

Aufrufe von IML-Konstruktoren (`iml_constructor_call`): Zur Erzeugung von IML-Knoten können die Konstruktoren aufgerufen werden, welche durch das JNI-Binding bereitgestellt werden. Diese Erzeugen eine neue Instanz der Knoten-Klasse und geben diese zurück. Es wird kein Schlüsselwort für den Aufruf benötigt, lediglich der Name der IML-Knotenklasse mit dem Präfix `IML::` wird vorangestellt.

```
iml_constructor_call = iml_type "(" {args} ")"
args = expression | {"," expression}
```

Aufrufe von externen Konstruktoren (`ext_constructor_call`): Da es in IMLTransform möglich ist externe Pakete aus der Zielsprache einzubinden, wird eine Möglichkeit benötigt um Instanzen der darin befindlichen Klassen zu erzeugen. Analag zum Konstruktor-Aufruf für IML-Klassen wird dieser auch für externe Klassen aufgerufen. Als Präfix wird hier allerdings `EXT::` verwendet.

```
ext_constructor_call = external_type "(" {args} ")"
args = expression | {"," expression}
```

Beispiel für die verschiedenen Aufrufe von Unterprogrammen:

```
1  //Funktion:
2  node.SLoc := EmptySLoc();
3
4  //Prozedur:
5  CreateNodeAndInit();
6
7  //Methode:
8  unit->removeAllDocumentation();
9
10 //Statische Methode:
11 IML::IML->ioLoad("a_graph.iml");
12 EXT::Arrays->asList("a", "b", "c");
13
14 //Konstruktoren:
15 sloc := IML::SLoc(0,0,0,
16          IML::Identifier(""),
17          IML::Identifier(""), null);
18 list := EXT::Vector();
```

8.4.8 Ausdrücke

Beim Entwurf von IMLTransform wurde darauf Wert gelegt dass die Angabe von Ausdrücken so flexibel wie möglich gestaltet wird. Transformations-Sprachen wie ATL nutzen beispielsweise die Object Constraint Language (OCL) für die Selektion von bestimmten Knoten im Graph. Da diese für IML nicht zur Verfügung steht musste ein Weg gefunden werden ähnlich komplexe Abfragen zu ermöglichen. Als Lösung wurden die Ausdrücke in IMLTransform mit möglichst viel Freiheiten ausgestattet.

Ausdruck (`expression`): Im folgenden ist das Listing für die Definition von Ausdrücken in IMLTransform dargestellt. Hierbei kann durch Rekursion beliebig tief verschachtelt werden. Zudem können Ausdrücke beliebig geklammert werden. Diese werden bei Generierung des Java-Codes an der entsprechenden Stelle gesetzt. Die einzelnen Elemente aus denen Ausdrücke aufgebaut sind werden in den Paragraphen dieses Kapitels erläutert.

```
expression = boolean_ops

boolean_ops = comparison {("||"| "&&") comparison}

comparison = arithmic {("<"|">"|"<="|">="|"=="|"!=") arithmic}
arithmic  = cast {("+"|"-"|"*"|"/")) cast}

cast      = type_check {"AS" (iml_type | external_type)}
type_check = prefixed {"IS" (iml_type | external_type)}

prefixed = ({"!"}|{"-"}) primary

primary = numeric_literal | boolean_literal | char_literal | null | qualified

qualified =
    (function_call | method_call | static_method_call | ext_constructor_call |
     variable_ref  | input_graph | output_graph) | string_literal | "("
        expression ")"
     {
       {attribute_access | method_access} [array_op]
     }

attribute_access = {"." name}
method_access  = {"->" name "(" {args} ")"}
array_op = "[" expression "]"
args = expression | {"," expression}
```

Boolsche Operationen (`boolean_ops`): IMLTransform unterstützt zur Zeit die Verknüpfung für Konjunktion `&&` und Disjunktion `||`.

Vergleiche (`comparison`): Für den Vergleich zweier Ausdrücke stehen folgende Operatoren zur Verfügung:

< > <= >= == !=

Arithmetische Operationen (`arithmic`): Es werden die Operatoren für die vier Grundrechenarten unterstützt: `+ - * /`

Typkonvertierungen (`cast`): Die Knoten-Klassen von IML bauen auf einer starken Hierarchie-Beziehung auf. Dadurch sind sie in der Lage Gemeinsamkeiten durch abstrakte Oberklassen abzubilden und gleichzeitig beliebig konkret zu werden. Häufig findet man in IML Listen oder Sets mit Knoten die zwar verschiedenen Klassen, aber einer gemeinsamen Basisklasse haben. Wenn mit solchen Listen gearbeitet wird kann es nötig sein explizite Downcasts durchführen zu müssen. Hierfür gibt es in IMLTransform das Schlüsselwort `AS`.

Umgesetzt werden Typkonvertierungen in Java als vorangestellte Klammern die den Ziel-Typ umschließen. Darauf folgt der Ausdruck der umgewandelt werden soll. Dies ist die Standard-Notation für explizite Casts in Java.

Typprüfungen (`type_check`): Wie auch beim vorherigen Abschnitt kann es nötig sein den Typ eines Objekts zu kennen. Hierfür stellt IMLTransform den Operator `IS` zur Verfügung. Dieser gibt `true` zurück wenn die Klasse eines Objekts identisch mit der Zielklasse ist oder sich von dieser ableitet. In Java wird diese Typprüfung durch den `instanceof`-Operator realisiert.

Vorzeichen (`prefixed`): Für die boolsche Negierung eines Ausdrucks kann diesem der Operator `!` vorangestellt werden. Die arithmetische Negierung kann durch den Operator `-` vor dem Ausdruck erreicht werden.

Primäre Ausdrücke (`primary`): Als primäre Ausdrücke werden die verschiedenen Literale bezeichnet. Hierzu zählen die einzelne Zeichen, numerische Literale, die boolschen Werte `true` und `false` sowie der `null`-Literal.

Qualifizierte Ausdrücke (`qualified`): Die letzte Kategorie in diesem Kapitel sind die qualifizierten Ausdrücke. Hierbei handelt es sich um Ausdrücke, die in der Lage, sind Objekte zurückzugeben. Auf deren Attribute und Methoden kann dann über die den Attribut-Operator `.` und den Methoden-Operator `->` zugegriffen werden. Für den Zugriff auf einzelne Elemente einer Menge steht zudem der Zugriffsoperator `[]` zur Verfügung.

Die Umsetzung in Java ist für die einzelnen Bestandteile in ihren jeweiligen Paragraphen beschrieben. Dies gilt auch für den Aufruf von Methoden über den Methoden-Operator. Einen Sonderfall stellen der Attribut- und der Zugriffsoperator für Mengen dar. Diese ermöglichen es auf Attribute eines Objekts zuzugreifen ohne den entsprechenden Getter oder Setter verwenden zu müssen. Dies ist besonders im Vergleich zur Arbeit mit dem JNI-Binding von Vorteil, da hier die Eigenschaften nur durch Zugriffsmethoden erreichbar sind. Für die Umsetzung wird dabei vom Generator geprüft in welchem Kontext der qualifizierte Ausdruck steht. Steht er beispielsweise auf der linken Seite einer Zuweisung, so prüft er durch einen Stack ob es sich beim Attribut um einen nativen Datentyp handelt. Ist dies nicht der Fall, so erzeugt er automatisch einen Setter für die Zuweisung des Attributs. Wichtig hierfür ist die korrekte Benennung des Attributs. Hierbei muss ebenfalls die korrekte Groß- und Kleinschreibung beachtet werden.

Beispiel für Ausdrücke in IMLTransform:

```
1 (GetFirstElement() AS IML::T_Node)->length()
2
3 !(createRoot() IS IML::Root_Node)
4
5 (1==1) && ((2==2) && (false))
```

8.5 Entwurf

Der Entwurf der Transformationssprache umfasst die Kombination verschiedener Generatoren zur Erzeugung der Transformatoren. Ausgangspunkt ist dabei die *XText*-Grammatik in der die Transformationssprache beschrieben wird. Aus dieser wird bei beim Kompilieren das Ecore-Datenmodell für die Knoten des abstrakten Syntaxbaum erzeugt. Wie im vorherigen Abschnitt 8.2.1 beschrieben basiert *XText* auf dem Eclipse Modelling Framework (EMF). Dieses übernimmt die Erzeugung des Modells. Zusätzlich werden auf Basis der Transformations-Grammatik eine entsprechende Grammatik für *ANTLR* generiert. Diese dient als Zwischenprodukt für die automatische Erstellung von Parser und Lexer für die Transformationssprache.

Bei Verarbeitung eines Transformations-Skripts ruft *XText* zuerst den erzeugten Lexer für die Eingabedatei auf. Dessen Ausgabe wird dann an den Parser übergeben. Dieser wendet die in der Grammatik spezifizierten Regeln auf die Tokens an, welche vom Lexer übergeben werden. Ergebnis der Parser-Ausführung ist ein abstrakter Syntaxbaum (AST). Die Knoten des Baums sind die Klassen welche von EMF aus dem Ecore-Modell der Sprache erzeugt wurden.

Auf Basis dieser Zwischendarstellung wird der eigentliche Transformations-Generator aufgerufen. Dieser durchläuft den AST *depth-first* und erzeugt aus dessen Knoten den Code des Transformators. Die Abbildung 8.1 verdeutlicht diese Zusammenhänge noch einmal.

Wie bereits im Abschnitt 8.3 beschrieben, erzeugt der Transformations-Generator die Transformatoren als Java-Programme. Für eine spätere Erweiterung durch Ada-Generatoren wurde die Architektur modular entworfen. Hierfür kann im Transformations-Skript die Zielsprache spezifiziert werden. Anhand dieser Einstellung instantiiert der Transformations-Generator die entsprechenden Klassen für die Generierung des Codes. Für Java sind die entsprechenden Generator-Klassen vollständig implementiert. Diese können als Referenz für die Umsetzung der äquivalenten Ada-Komponenten verwendet werden.

Die Generierung des Transformator-Codes entsteht durch Traversierung des ASTs, welcher durch den *ANTLR*-Parser erzeugt wurden. Für jeden dabei besuchten Knoten wird eine entsprechende Methode aufgerufen. Diese verarbeiten das aktuelle Element und erzeugen den entsprechenden Code. Um die Wart- und Lesbarkeit der Generator-Klassen zu verbessern wurden diese auf mehrere Module verteilt.

8.6 Implementierung

Der Einstiegspunkt für den Transformations-Generator ist die Klasse `Main`. In dieser ist die `main`-Methode definiert. *XText* erzeugt eine solche Klasse automatisch wenn die Sprache einen eigenständigen Übersetzer für die Nutzung außerhalb von Eclipse. Da *XText* auf EMF basiert gelten hier die im Kapitel 7.1 beschriebenen Voraussetzungen. Anders als bei der Implementierung des Zustandsautomaten-Frontends wird dies aber automatisch bei Initialisierung des Generators von *XText* übernommen. Für das Starten der Generierung

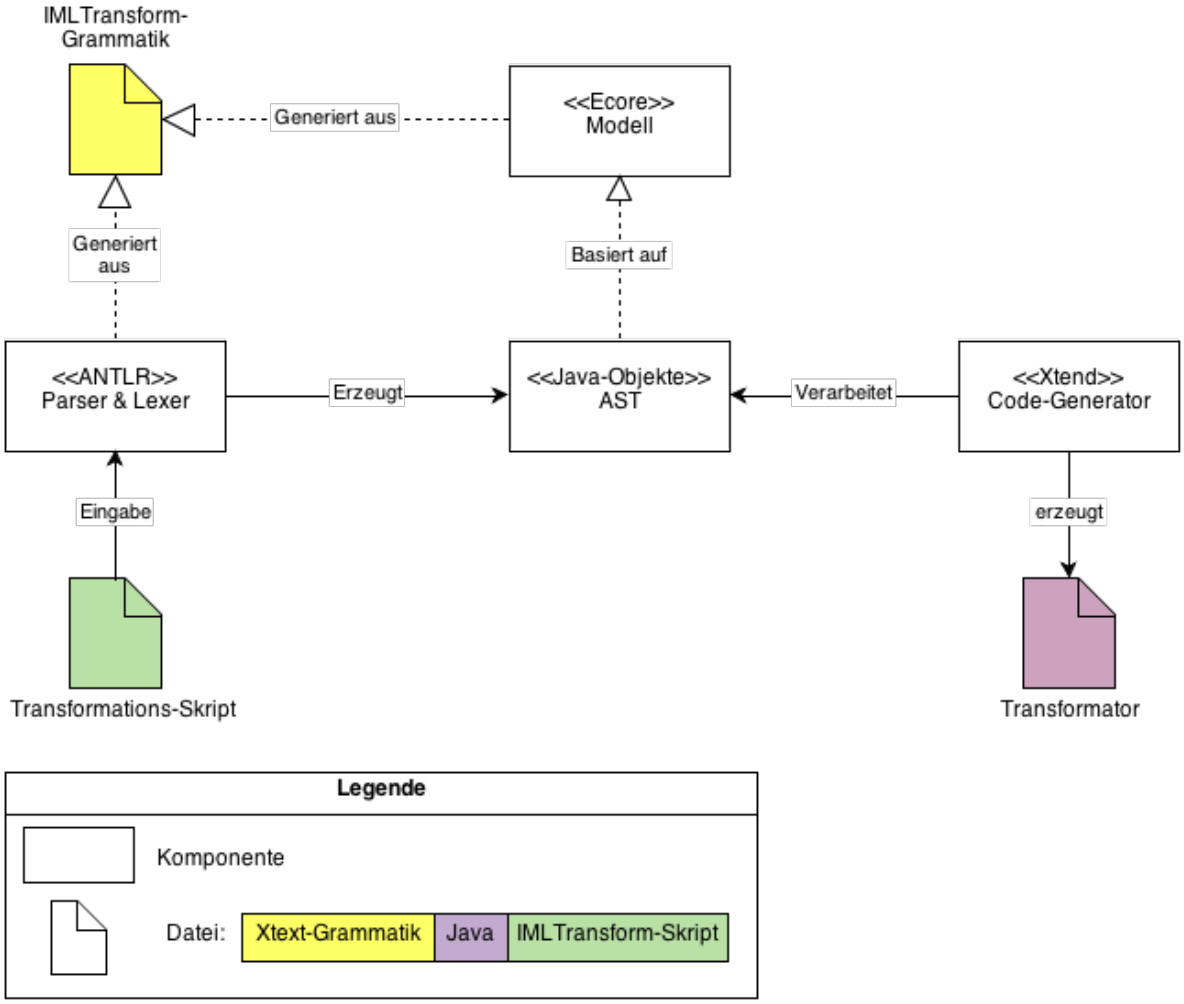

Abbildung 8.1: Zusammenhänge der Komponenten des Transformations-Generators.

werden die Kommandozeilen-Argumente an den Generator übergeben. Die `Main`-Klasse wird von *XText* nur erzeugt wenn diese noch nicht existiert. Es ist also möglich diese anzupassen ohne dass diese Änderungen bei jeder Übersetzung überschrieben werden. Im Falle dieser Arbeit wurde der Kommandozeilenparser CLI eingesetzt um die übergebenen Argumente sauber einlesen zu können. Zudem wird das Log4J-Framework für die Ausgabe von Log-Nachrichten verwendet. Eine Beschreibung der beiden Technologien finden sie im Kapitel 3.

Nach Initialisierung von EMF und Logging wird die Generierung durch Aufruf der `doGenerate` der Klasse `IMLTransformGenerator` gestartet. Im ersten Schritt initialisiert dieser die Klasse `TransformationConfig`. Diese liest die Konfigurationen aus dem Transformations-Skript und bietet Zugriffsmethoden um auf diese zuzugreifen. Realisiert ist die Klasse dabei als Singleton [Gam95] implementiert. Als nächstes wird der entsprechende Generator für die

Ausgabesprache erzeugt. `IMLTransformGenerator` agiert dabei als Fassade [Gam95] und gibt den Aufruf weiter. In Abbildung 8.2 werden die Aufrufe und Beziehungen der Generatoren beschrieben. Die Darstellung ist dabei auf Klassen beschränkt die im Kontext der Modularisierung für die Einbindung weiterer Ausgabesprachen eine Rolle spielen. Wird der Transformator als Java-Programm erzeugt, so wird die Klasse `IMLTransformGenerator` verwendet. Diese wiederum nutzt `StatementGenerationJava` für die Verarbeitung der Anweisungen. Für Ausdrücke wird `ExpressionGenerationJava` eingesetzt. Diese beiden Klassen definieren die entsprechenden Methoden für die Erzeugung von Java-Code aus Knoten des AST. Für die Generierung von Ada wird eine analoge Modularisierung verwendet.

Zusätzlich zu gibt es eine weitere Stelle die für weitere Ausgabesprachen erweitert werden muss. Die Klasse `GeneratorUtils` stellt Funktionen für verschiedene Aufgaben bereit. Diese lassen sich keiner der anderen Klassen zuordnen, werden von diesen aber verwendet. Beispiel einer solchen Funktion ist die Erzeugung voll-qualifizierter Bezeichner. Diese dient dazu die unbeabsichtigte Verwendung falscher Typen zu verhindern. Da der Syntax dieser Qualifizierung sich bei den verschiedenen Ausgabesprachen unterscheiden kann, wird auch hier eine modularisierte Architektur eingesetzt. Die Klasse `GeneratorUtils` arbeitet hierfür wieder als Fassade [Gam95], welche die Methoden-Aufrufe an eine konkrete Implementierung des `GeneratorUtils`-Interfaces weiterleitet.

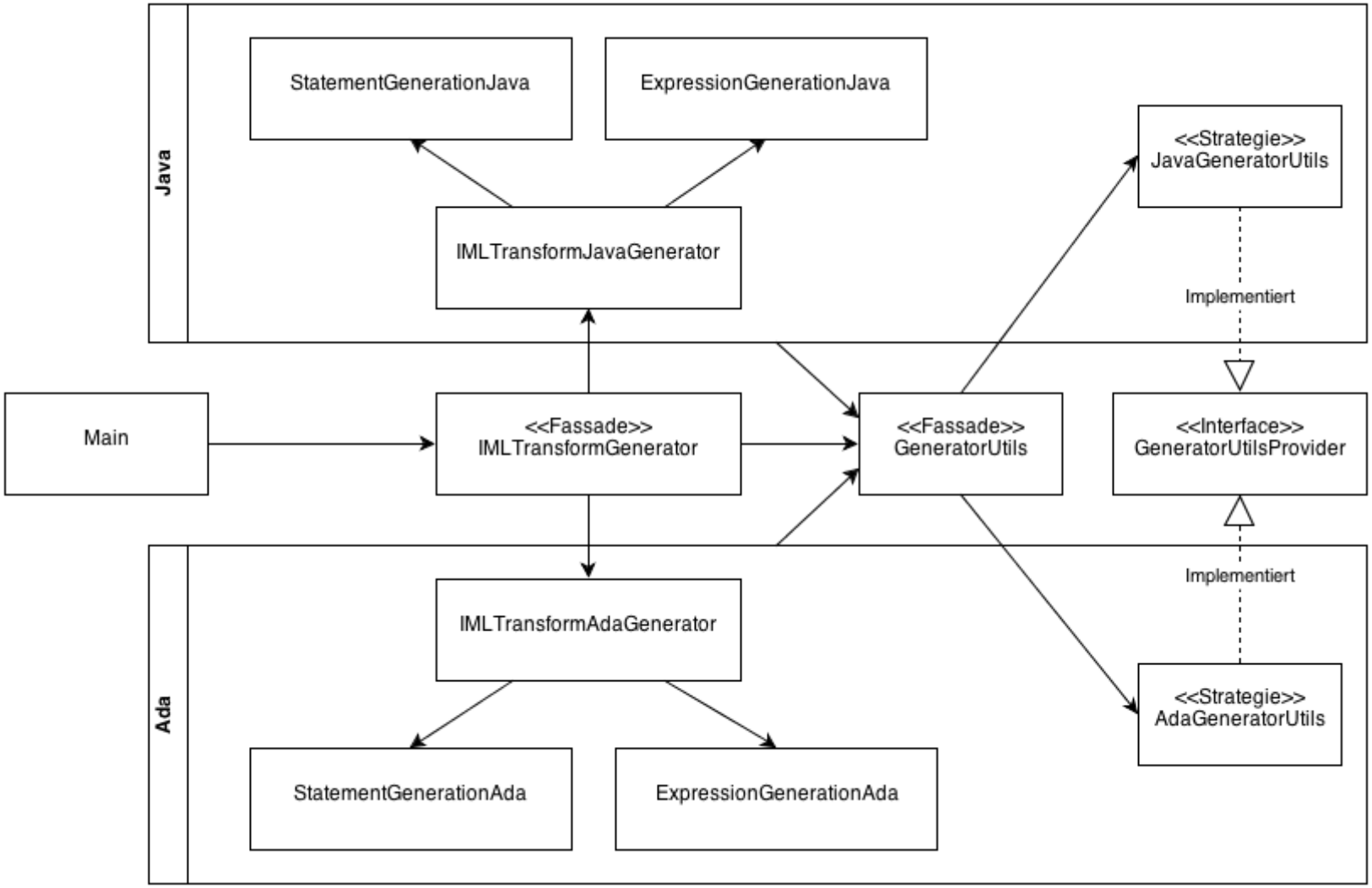

Abbildung 8.2: Aufrufbeziehungen und Abhängigkeiten zwischen den Generatoren in IMLTransform.

Nachdem der AST vollständig verarbeitet wurde werden für Java-Transformatoren eine Make- und Shellfile erzeugt. Hierfür existieren die Generatorklassen `MakeFileGenerator` und `ShellFileGenerator`. Diese werden von `IMLTransformJavaGenerator` aufgerufen. Eine Beschreibung der erzeugten Dateien findet sich im Kapitel 8.3.

Nach Abschluss des Generierungsprozesses beendet die Anwendung. Der erzeugte Code kann nun durch den Aufruf des entsprechenden Übersetzers in die ausführbare Form gebracht werden. Für Java kann hierfür die erzeugte Makefile verwendet werden.

8.7 Anwendung

In diesem Abschnitt wird beschrieben wie die Transformationssprache zu verwenden ist. Dies umfasst neben der Kompilierung und Ausführung des Transformations-Generators *imltransform* auch die Nutzung der generierten Transformator-Programme.

8.7.1 Anwendung des Transformations-Generators

Der Transformations-Generator ist für die Generierung des Programm-Codes aus den Transformationsbeschreibungen verantwortlich. Er stellt ein eigenständiges Werkzeug dar und muss dementsprechend vor Verwendung kompiliert werden. Das *XText* Framework und die *Xtend*-Sprache, mit denen der Generator umgesetzt wurde benötigen hierfür spezielle Maßnahmen.

Zuerst muss dabei die *XText*-Grammatik der Transformations-Sprache in ein EMF-Modell übersetzt werden. Hierbei werden die Java-Klassen der Regeln erzeugt, über die auf das Modell zugegriffen wird. *XText* generiert hierfür einen EMFT-Workflow [13] [14] der diese Aufgabe übernimmt. Wird der Generator aus Eclipse erzeugt, so wird dieser Workflow automatisch durch die *Run-Configuration* "Generate Language Infrastructure (bauhaus.imltransform)" ausgeführt. Da jedoch nicht davon ausgegangen werden kann dass jeder Entwickler mit dieser Entwicklungsumgebung arbeitet, wurde eine Shell-File ("mwe2runner.sh") geschrieben, die dies außerhalb von Eclipse ermöglicht. Diese nutzt die im Ordner "lib" abgelegten EMF-Bibliotheken. Die Klasse `org.eclipse.emf.mwe2.launch.runtime.Mwe2Launcher` bietet hierfür eine eigene `main`-Methode an. Diese wird aus "mwe2runner.sh" aufgerufen und startet den in der Datei "GenerateIMLTransform.mwe2" definierten Workflow zur Generierung des Sprach-Modells. Ein manueller Aufruf dieser Shell-Datei ist jedoch nicht nötig, da sie Teil des *Makefiles* für die Erzeugung des Transformations-Generators ist. Dadurch wird sichergestellt dass vor Kompilierung des Generator-Codes die Sprachkonstrukte auf dem neuesten Stand sind.

[13] Eclipse Modelling Framework Technology
[14] http://eclipse.org/modeling/emft/?project=mwe

Wie auch bei alle anderen Werkzeuge, die in dieser Arbeit entstanden sind, wird eine *Makefile* für die Einbindung in den *Make*-Prozess von Bauhaus bereitgestellt. Wie oben erwähnt führt diese zuerst den Workflow für die Generierung des Sprachmodells aus. Im Anschluss kompiliert sie den Quellcode des Transformations-Generators und erzeugt die ausführbare JAR-Datei. Die benötigten Libraries werden in das Bibliotheks-Verzeichnis von Bauhaus kopiert.

Um den Transformations-Generator aufzurufen steht eine weitere Shell-Datei mit Namen "imltransform.sh" zur Verfügung. Sie stellt vor Aufruf des Programms die korrekte Konfiguration des Build-Paths sicher. Bei Kompilierung des Generators wird sie zudem in das Tools-Verzeichnis von Bauhaus kopiert. Im folgenden Ausschnitt wird die Ausgabe der Ausführung im Falle fehlender oder falscher Kommandozeilenargumente dargestellt:

```
usage: imltransform.sh *args*
Processes a given IML transformation file and generates a JAVA or ADA
class which contains the specified transformations.
 -f <arg>   Specifies the transformation script file to be compiled.
 -v         Toggles the verbose mode.
```

Der Paramter *-f* ist dabei als Pflichtparameter anzugeben.

8.7.2 Anwendung der Transformatoren

Nach Aufruf des Transformations-Generators werden für das erzeugte Java-Programm mehrere Dateien erzeugt. Zum ersten ist dies eine Java-Datei, welche das Transformator-Programm enthält. Für die Durchführung der Transformationen muss dieses auf das JNI-Binding für IML zurückgreifen. Damit für die Kompilierung des Codes nicht manuell eine entsprechende *Makefile* angelegt werden muss, wird eine entsprechende Konfigurations-Datei für *make* automatisch generiert. Sie kompiliert das Programm und erzeugt eine ausführbare Jar-Datei. Diese kann durch die, ebenfalls automatisch angelegte Shell-Datei gestartet werden. Die Benennung der Java-Klasse, Shell-File und Jar-Datei ist abhängig vom Namen der Transformation. Diese wird innerhalb des Transformations-Skripts festgelegt.

Transformatoren benötigen beim Aufruf zwei Eingabeparameter. Als erstes wird der Dateipfad zum Quell-Graphen übergeben, auf dem die Transformationen ausgeführt werden. Zweiter Parameter ist der Ausgabepfad an dem das Resultat des Transformators abgelegt wird. Im folgenden Ausschnitt wird ein beispielhafter Aufruf einer Transformation mit Namen "ExpandIt" dargestellt:

```
ExpandIt.sh *PFAD ZUM QUELL-GRAPH* *AUSGABEPFAD*
```

9 Java-Code-Generator

Aufgabe des Java-Code-Generators *unparse_j* ist die Generierung von Java Code anhand eines gegebenen IML-Graphen. Vorab sei gesagt, dass die vollständige Umsetzung eines Code-Generators für alle IML-Klassen sehr aufwendig wäre. Dies würde den Rahmen des Anteils sprengen, den das Projekt an dieser Diplomarbeit innehat. Ziel der Implementierung ist es einen Generator zu schaffen, der in der Lage ist IML-Graphen zu verarbeiten, welche durch die Transformation von Zustandsautomaten entstanden sind. Nach Ausführung der Transformation auf einem solchen Graphen entsteht ein Graph, der den Zustandsautomaten nach der im Kapitel 6.1 vorgestellten Implementierung, wiedergibt. Dadurch ergeben sich alle umzusetzenden IML-Knotenklassen aus dieser Art der Realisierung von hierarchischen Zustandsautomaten.

9.1 Entwurf

Die Generierung von Code aus einem IML-Graphen erfordert das Durchlaufen der Knoten anhand ihrer Kanten. Dementsprechend beginnt eine Erzeugung stets bei der Wurzel und arbeitet sich anhand der von ihr ausgehenden Kanten vor. Auch wenn der Umfang der abzudeckenden Knoten in dieser Anwendung eingeschränkt wurde, so müssen doch eine Vielzahl verschiedener Konstrukte bedacht werden. Um eine monolithische Programmstruktur zu vermeiden, wurde eine Auftrennung der Verarbeitungsmethoden entworfen. Diese Methoden sind für die Generierung des Codes aus einem spezifischen Knoten zuständig.

Der Entwurf des Code-Generators lässt sich in die folgenden Kategorien unterteilen:

- **Core:** Der Kern der Anwendung enthält die Klasse welche beim Start der Anwendung aufgerufen wird. Sie enthält die statische `main`-Methode. Zusätzlich dazu ist sie für die Initialisierung des Loggings, die Auswertung der Kommandozeilen-Parameter durch CLI (s. Kapitel 3.2) und das Starten des eigentlichen Generierungsprozesses verantwortlich.
- **Unparser:** Dieser Teil stellt die Funktionalität für die Generierung des Codes bereit. Hier werden die Methoden definiert, welche die einzelnen Knoten verarbeiten. Der eingelesene Graph wird an diese Komponenten übergeben. Innerhalb dieser Kategorie sind die Methoden in einzelne Module unterteilt um eine bessere Wart- und Lesbarkeit zu gewährleisten.

- **Output:** Der generierte Code wird bei seiner Erzeugung in einen entsprechende Java-Datei geschrieben. Hierfür stellt diese Kategorie eine Klasse bereit. Von dieser wird eine neue Instanz für jede Ausgabedatei erzeugt. Beim Schreiben wird der generierte Code direkt über einen Ausgabestream an der aktuellen Position eingefügt. Um das Ergebnis lesbar zu gestalten kann die Einrückung erhöht oder verringert werden.
- **Utilities:** Unter dem Begriff *Utilities* werden verschiedene Helferklassen zusammengefasst. Diese enthalten Methoden, welche an unterschiedlichen Stellen eingesetzt werden.

Um den Java-Code-Generator später als Basis für eine vollständige Abdeckung der IML-Klassen zu verwenden, war ein Ziel eine möglichst simple Erweiterbarkeit zu gewährleisten. Hierfür wurde eine einheitliche Schnittstelle geschaffen, welche Anhand des übergebenen Knotentyps die korrekte Verarbeitungsmethode bestimmt und diese aufruft. Angelehnt wurde dieser Mechanismus an eine Kombination aus den Entwurfsmustern "Fassade" und "Strategie" [Gam95]. Um jedoch nicht für jeden Knotentyp eine konkrete Strategie-Klasse anlegen zu müssen, wurde hier eine andere Lösung gewählt. Der Einsatz von Funktionszeigern war ebenfalls nicht möglich, da dieses Konzept von Java nicht unterstützt wird. Der Mechanismus der schlussendlich umgesetzt wurde basiert auf dem seit Java 1.5 verfügbaren Konzept der `Annotations`[1]. Damit werden Methoden, welche für die Verarbeitung von Knoten zuständig sind, als solche markiert. Über einen Parameter wird mitgeteilt welche Knotenklasse von der annotierten Methode akzeptiert wird. Durch den Einsatz von *Reflections* ist es für die Fassade nun möglich die passende Verarbeitungsmethode für einen übergebenen Knoten zu bestimmen und diesen aufzurufen.

Der Code-Ausschnitt in 9.1 zeigt die Methode, welche `Return_With_Value`-Knoten verarbeitet. Die Annotation `NodeUnparsingMethod` dient dabei als Markierung und speichert zudem den Knotenklassen-Typus. Der zweite Parameter übergibt eine Instanz der `CodeOutput`-Klasse. Diese erlaubt es den erzeugten Code in die Ausgabedatei zu schreiben. Im Falle dieser Methode wird ein Knoten für eine Anweisung, welche einen Wert zurückgibt, verarbeitet. In Java wird hierfür an der aktuellen Stelle ein `return`-Statement eingefügt. Der zurückgegebene Wert ist über eine Kante *Expression* mit dem Knoten verbunden. Um diesen zu verarbeiten wird hier die Schnittstellen-Methode `unparse` der Klasse `Unparser` aufgerufen.

```
1  @NodeUnparsingMethod(target=Return_With_Value.class)
2  public void unparseReturnWithValue(
3      Return_With_Value node,
4      CodeOutput output) {
5      output.append("return ");
6      Unparser.unparse(node.getExpression(), output);
7  }
```

Listing 9.1: Beispiel einer annotierten Verarbeitungsmethode für Knotenklassen.

[1] http://docs.oracle.com/javase/tutorial/java/javaOO/annotations.html

In Java müssen Klassen, Interfaces und Enumerationen in eigenen Dateien definiert werden. Ausnahme sind verschachtelte Typdefinitionen. Diese sind innerhalb einer übergeordneten Klasse definiert. Beim Durchlaufen der Deklarationstabelle eines `Unit`-Knotens muss daher geprüft werden zu welcher Kategorie ein gefundener Typ gehört. Für alle nicht verschachtelten Typen wird eine neue Java-Datei angelegt.

9.2 Implementierung

Ausgangspunkt der Ausführung ist die Klasse `AppCore`. Bei Aufruf der `main`-Methode werden zuerst die übergebenen Kommandozeilenparameter ausgewertet. Dies geschieht durch die Methode `parseCmdArgs`. Sind diese korrekt und vollständig, so wird das JNI-Interface für IML gestartet und die Methode `run` der `AppCore`-Instanz ausgeführt.

In `run` wird zuerst der IML-Graph geladen. War dies erfolgreich, so wird ein `IMLGraphUnparser` erzeugt und die Methode `unparse` für den geladenen Graphen aufgerufen.

Der `IMLGraphUnparser` prüft zuerst ob der Graph einen `System`-Knoten besitzt. Ist dies der Fall, so werden alle gefundenen Subunits an den `IMLUnitUnparser` übergeben. Wird kein `System`-Knoten gefunden, so wird der `Raw_Graph` als einzelne `Unit` behandelt und verarbeitet.

Die Klasse `IMLUnitUnparser` durchsucht die Deklarationstabelle nach Typdefinitionen, welche nicht unterhalb eines anderen Typs definiert wurden. Für alle auf diese Weise gefundenen Klassen, Interfaces und Enumerationen wird die Methode `createOutputAndUnparseRoot` aufgerufen. Diese prüft zuerst ob es sich beim übergebenen Typ um einen *Ignored-Type* handelt. Diese sind als Platzhalter zu verstehen und werden nicht verarbeitet. Hintergrund dieser Maßnahme ist die im Abschnitt 6.5 beschriebene Problematik der Einbindung von Bibliotheken. Beispiel für einen *Ignored-Type* ist die im gleichen Kapitel beschriebene Klasse `StringWrapper`. Vorkommnisse von `StringWrapper` werden durch den Datentyp `String` ersetzt. Dementsprechend wird für `StringWrapper` keine Java-Klasse generiert. Ist die aktuelle Typdefinition nicht in *Ignored-Types* enthalten, so wird eine Instanz von `CodeOutput` erzeugt. Diese legt für den übergebenen Namen eine neue Java-Datei an und stellt die Ausgabemethoden bereit. Sie wird für alle weiteren Schritte der Verarbeitung verwendet.

Nachdem nun ein Ausgabestream besteht wird die Generierung des Codes gestartet. Hierfür wird die `Unparser`-Fassade (siehe [Gam95]) durch die Methode `unparse` angesprochen.

Aufgabe der Klasse `Unparser` ist die Identifizierung der korrekten Verarbeitungsmethode für einen bestimmten Knotentyp. Die von ihr bereitgestellte Schnittstellenmethode `unparse` ist statisch und benötigt keine Instanziierung der Klasse. Ein Aufruf dieser Methode kann in die folgenden Schritte unterteilt werden:

1. **Suche nach einem passenden** `NodeUnparsingProvider`**:** Erster Schritt ist die Bestimmung eines passenden *Providers*. Dies sind Klassen, welche das Interface `NodeUnparsingProvider` implementieren. In ihnen sind die verschiedenen Methoden

zur Verarbeitung von Knoten definiert. Ein *Provider* wird bei der `Unparser`-Fassade für eine bestimmten Knotenklasse registriert. Soll ein Knoten verarbeitet werden, so wird nach einem *Provider* gesucht der für die Klasse des Knoten oder eine seiner Oberklassen registriert wurde. Dies lässt sich am besten anhand eines Beispiels nachvollziehen:

Beispiel: Es soll ein Knoten vom Typ `T_Class_Name` verarbeitet werden. Zuerst wird geprüft ob es einen *Provider* für Knoten dieses Typs gibt. Da keiner gefunden wurde wird die Suche erneut für die Vaterklassen von `T_Class_Name` durchgeführt. Hierbei wird ein registrierter `NodeUnparsingProvider` für die Knotenklasse `T_User_Type_Name` gefunden.

2. **Suche nach einer passenden** `NodeUnparsingMethod`**:** Wurde im ersten Schritt ein *Provider* gefunden, so wird in diesem Schritt die Verarbeitungsmethode bestimmt. Hierfür werden die Methoden des *Providers*, welche mit `NodeUnparsingMethod` annotiert sind durchsucht. Wird keine passende Methode gefunden, wird wie in Schritt 1 verfahren und die Vorfahren des Knotens zur Bestimmung verwendet.
3. **Aufruf der Methode:** Konnte im vorherigen Schritt eine Verarbeitungsmethode für die Klasse des Knoten oder eine seiner Vorfahren bestimmt werde, so wird diese aufgerufen. Dabei werden der zu verarbeitende Knoten und die aktuelle Instanz der Klasse `CodeOutput` übergeben.

Die Registrierung eines *Providers* kann in `Unparser` vorgenommen werden. Hierzu wird eine neuer Eintrag in die Tabelle `PROVIDER_REGISTRY` eingefügt. Der Code in 9.2 zeigt den Ausschnitt aus `Unparser` in dem die Registrierungen vorgenommen werden.

```
1 static {
2     PROVIDER_REGISTRY.put(Value.class,        ValueUnparsingProvider.class);
3     PROVIDER_REGISTRY.put(Literal.class,      LiteralUnparsingProvider.class);
4     PROVIDER_REGISTRY.put(Operator.class,     OperatorUnparsingProvider.class);
5     PROVIDER_REGISTRY.put(Routine_Call.class, RoutineCallUnparsingProvider.
          class);
6     PROVIDER_REGISTRY.put(T_User_Type_Name.class,
          TUserTypeNameUnparsingProvider.class);
7     //...
8     //...
9 }
```

Listing 9.2: Registrierung von `NodeUnparsingProvider`-Klassen in der Fassade.

In der Registrierung werden dabei Paare aus `Storable`-Unterklassen und Klassen die das Interface `NodeUnparsingProvider` implementieren gebildet.

Mit diesem Mechanismus ist es möglich konkrete Methoden für einzelne Knotenklassen zu definieren. Gleichzeitig können Knoten, deren Verarbeitung durch eine Vaterklasse definiert wird, durch eine gemeinsame Methode behandelt werden. Durch die Verteilung auf verschiedene *Provider* kann eine monolithische Struktur vermieden werden, bei der

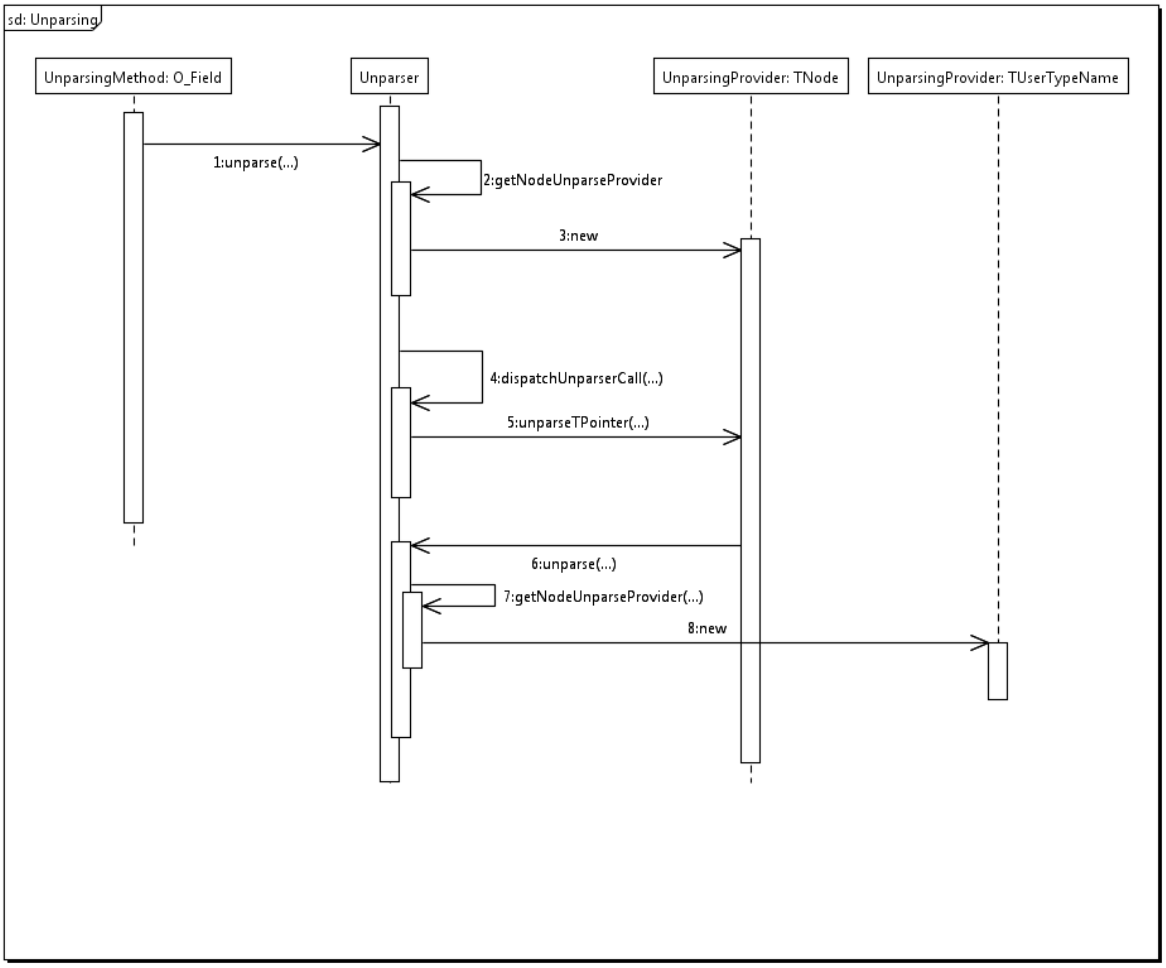

Abbildung 9.1: Sequenzdiagramm für die Verarbeitung `T_Pointer` der auf einen `T_Class_Name` zeigt.

alle Methoden in einer einzigen Klasse definiert sind. Mit Einsatz der Fassade ist die Implementierung einer Verarbeitungsmethode zudem stark vereinfacht. Statt eventuell mehrere Klassen von Hand nach der passenden Methode durchsuchen zu müssen, muss lediglich die Fassade angesprochen werden. Das Sequenzdiagramm in Abbildung 9.1 zeigt die Aufruffolge für die Verarbeitung eines `T_Pointer`-Knotens der auf einen `T_Class_Name` zeigt. Um die Übersichtlichkeit zu verbessern wurden die weiteren Schritte nach "8: new" weggelassen. Diese sind analog zu den vorangegangenen Aufrufen.

Die folgenden Tabellen listen alle aktuell implementierten `NodeUnparsingProvider` sowie ihre Verarbeitungsmethoden auf. In der ersten Zeile steht die Knotenklasse gefolgt von ihrem *Provider*. In den Spalten darunter werden die Methoden und die Knoten für die sie zuständig sind aufgeführt.

T_Node: TNodeUnparsingProvider	
Methoden	**Knotenklassen**
unparseTPointer	T_Pointer
unparseTReference	T_Reference
unparseConstQualifier	T_Const_Qualifier
unparseConstQualifier	T_Array
unparseTVoid	T_Void
unparseTBoolean	T_Boolean
unparseTCChar	TC_Char
unparseTCInteger	TC_Integer
unparseTCInt	TC_Int
unparseTFloatingPoint	T_Floating_Point
unparseTFloat	TC_Float
unparseTCDouble	TC_Double

O_User_Type_Declaration: UserTypeDeclUnparsingProvider	
Methoden	**Knotenklassen**
unparseOClass	O_Class
unparseOEnumNode	O_Enum

T_User_Type_Name: TUserTypeNameUnparsingProvider	
Methoden	**Knotenklassen**
unparseTUserTypeName	T_User_Type_Name

Routine_Call: RoutineCallUnparsingProvider	
Methoden	**Knotenklassen**
unparseDirectCall	Direct_Call
unparseVirtualCall	Virtual_Call

Operator: OperatorUnparsingProvider	
Methoden	**Knotenklassen**
unparseBinaryOperator	Binary_Operator
unparseUnaryOperator	Unary_Operator
unparseImplicitConversion	Implicit_Conversion

O_Node: ONodeUnparsingProvider	
Methoden	**Knotenklassen**
unparseOConstructor	O_Constructor
unparseOField	O_Field
unparseOMethod	O_Method
unparseOStaticMethod	O_Static_Method
unparseOStaticField	O_Static_Field
unparseOParameter	O_Parameter
unparseOThisParameter	O_This_Parameter
unparseOEnumerator	O_Enumerator

Literal: LiteralUnparsingProvider	
Methoden	**Knotenklassen**
unparseLiteral	Literal

Value: ValueUnparsingProvider	
Methoden	**Knotenklassen**
unparseStatementSequence	Statement_Sequence
unparseRead	Read
unparseReturnWithValue	Return_With_Value
unparseReturnWithOutValue	Return_Without_Value
unparseDereference	Dereference
unparseIndexedDereference	Indexed_Dereference
unparseEntityLValue	Entity_L_Value
unparseAddressOf	Address_Of
unparseAssignement	Assignment
unparseInitialize	Initialize
unparseRoutineLRConversion	Routine_LR_Conversion
unparseExitLoop	Exit_Loop
unparseContinueGoto	Continue_Goto
unparsePostfixOperator	Postfix_Operator
unparsePrefixOperator	Prefix_Operator
unparseOrElse	Or_Else
unparseAndThen	And_Then
unparseCopyIn	Copy_In
unparseArrayLRConversion	Array_LR_Conversion
unparseIfStatement	If_Statement
unparseCForLoop	C_For_Loop
unparseWhileLoop	While_Loop
unparseDoWhile	Do_While_Loop
unparseSwitch	C_Switch_Statement
unparseAnonymousLabel	Anonymous_Label
unparseValuedLabel	Valued_Label
unparseExitSwitch	Exit_Switch
unparseTryCatchFinallyStatement	Try_Catch_Finally_Statement
unparseCatchBlock	Catch_Block
unparseThrowStatement	Throw_Statement
unparseMemberSelection	Member_Selection
unparseCommonSubexpression	Common_Subexpression
unparseNullExpression	Null_Expression
unparseIntegerConstant	Integer_Constant
unparseNewOperator	New_Operator

9.3 Anwendung

Der Java-Code-Generator ist das letzte Werkzeug im Verarbeitungsprozess für Zustandsautomaten. Er wird aufgerufen nachdem der IML-Graph eines Zustandsautomaten durch die Transformationsvorschriften auf die entsprechenden Basiskonstrukte erweitert wurde. Die dabei entstehende IML-Datei wird als Eingabeparameter an den Java-Code-Generator übergeben.

Für die Integration des Generators in den Build-Prozess des Bauhaus Frameworks wird ein Konfigurationsdatei (”Makefile”) für das *Make*-Programm bereitgestellt. Dieses kompiliert den Quellcode und generiert die ausführbare JAR-Datei.

Ausgeführt wird das Programm durch Aufruf der Shelldatei ”unparse_j.sh”. Diese stellt sicher dass die benötigten Bibliotheken als Build-Path bereitgestellt werden. Im folgenden Ausschnitt wird die Ausgabe des Programms im Falle fehlender Parameter dargestellt:

```
usage: unparse_j.sh *args*
Processes an IML file and generates java classes from the contents found
in the file. The result is stored in the location specified in the
arguments.
 -o <arg>   Specifies the folder in which the output files will be placed.
 -p <arg>   Specifies the name of the package for the generated files.
 -s <arg>   Specifies the IML file from which the code is generated.
 -v         Toggles the verbose mode.
```

Außer dem Parameter *-v*, welcher die ausführliche Logging-Ausgabe aktiviert, müssen alle anderen Parameter übergeben werden.

10 Fazit und Ausblick

Bauhaus als eine vollwertige Language Workbench zu bezeichnen wäre nicht korrekt. Obwohl bereits vor dieser Arbeit einige Elemente einer Language Workbench vorhanden waren, fehlen doch andere Bestandteile. Wie auch im Falle der Language Workbenches nutzt Bauhaus eine zentrale, abstrakte Darstellung für die Abbildung der Daten. Durch die existierenden Werkzeuge zur Generierung von Code (Beispiele: *unparse_c* und der hier realisierte Java-Code-Generator) ist ebenfalls eine der Aufgaben einer Language Workbench erfüllt. Nach den Definitionen von Martin Fowler in seinen Arbeiten [Fow05] und [Fow10] fehlt in Bauhaus jedoch das Konzept der projektionalen Editoren. Die Werkzeuge des Frameworks, welche für die Erzeugung der IML-Darstellung eingesetzt werden, arbeiten nur in eine Richtung. Mit der Umsetzung dieser Diplomarbeit wurden die Mächtigkeit der IML-Zwischendarstellung stark erhöht. Durch die modulare Integration neuer Knotenklassen und die Zuordnung von Semantik durch Transformationen ist es leicht abstraktere Konzepte in IML abzubilden. Analog zu der Umsetzung des Zustandsautomaten-Sprachpakets und seiner Werkzeuge (Frontend und Transformation auf Basiskonstrukte) könnten auch andere Verhaltensdiagramme der UML in Bauhaus eingebracht werden.

Die Transformationssprache, welche im Laufe dieser Abschlussarbeit entstanden ist, kann zudem auch für andere Zwecke als lediglich zur Abbildung auf Basiskonstrukte genutzt werden. Sie ermöglicht beliebige Transformationen von IML-Graphen bei einem lesbaren und zweckmäßigen Syntax. Durch ihre Anlehnung an bekannte Sprachen wie Ada oder Pascal wird die Einstiegshürde für Nutzer gesenkt.

Aufgrund der umfangreichen Natur dieser Arbeit bieten sich eine Menge Möglichkeiten die umgesetzten Projekte zu verbessern und zu erweitern.

Eine Position ist der Merge-Präprozessor. Dieser kombiniert alle gefundenen Sprachpakete zu einer einzelnen Spezifikationsdatei. Dies könnte durch einen `Include`-Mechanismus mit dem die Spezifikationen ineinander eingebunden werden ersetzt werden. Dafür müsste allerdings der bestehende *imlgen*-Generator und (gegebenfalls) die Grammatik der Spezifikationssprache angepasst werden. Sollte der bestehende Mechanismus beibehalten werden, so könnte dieser durch verschiedene Prüfmechanismen erweitert werden. Diese könnten überprüfen, ob die in Sprachpaketen verwendeten Konstrukte überhaupt existieren. Da aktuell lediglich die Inhalte der Dateien aneinandergehängt werden, bleiben solche Prüfungen *imlgen* überlassen.

Weitere Ausbaumöglichkeiten existieren für die Transformationssprache. Diese erzeugt aktuell Java-Programme, welche die gewünschten Transformationen auf dem Eingabe-Graphen durchführen. Für eine zukünftige Erweiterung wurde eine Architektur bereitgestellt, die es erlaubt neue Ausgabesprachen leicht einzubinden. Für die Sprache Ada wurden dabei

bereits in der Sprachspezifikation und den Generator-Klassen Schnittstellen angelegt. Im Gegensatz zu anderen Transformationssprachen wie ATL [1] basiert die umgesetzte Sprache nicht auf einzelnen Regeln, welche Knoten eines bestimmten Typs durch andere Knoten ersetzt. Eine Erweiterung zur Bereitstellung solcher Regeln wurde im Laufe dieser Arbeit überlegt. Da die hierfür benötigten IML-Operationen (konkret: Paket `iml_tables`) nur durch Anpassung der existierenden Generatoren für Java bereitgestellt werden könnten, wurde dies aus Zeitgründen verworfen. Zukünftige Arbeiten könnten diesen Ansatz allerdings weiterverfolgen und in die Sprache integrieren.

Zuletzt bietet der Java-Code-Generator Ansatzpunkte für weitere Arbeiten. Die Umsetzung eines kompletten Code-Generators für alle existenten IML-Konstrukte könnte eine eigene Diplomarbeit füllen. Dementsprechend wurden nur Konstrukte beachtet, welche nötig sind, um die im Abschnitt 6.1 vorgestellte Implementierung von Zustandsautomaten zu verarbeiten. Für eine vollständige Abdeckung der IML-Knotenklassen können die restlichen Knotenklassen analog in das Projekt eingebracht werden.

[1] http://www.eclipse.org/atl/

Literaturverzeichnis

[AG12a] itemis AG. *Xtend - Documentation*. itemis AG, Am Brambusch 15-24 44536 Lünen, 2.3.0 Auflage, 2012. URL http://www.eclipse.org/xtend/documentation/2.3.0/Documentation.pdf. (Zitiert auf Seite 89)

[AG12b] itemis AG. *Xtext - Documentation*. itemis AG, Am Brambusch 15-24 44536 Lünen, 2.3.0 Auflage, 2012. URL http://www.eclipse.org/Xtext/documentation/2.3.0/Documentation.pdf. (Zitiert auf den Seiten 87, 88 und 92)

[Blu12] C. Blume. *Vergleich ausgewählter Vertreter von Language Workbenches zur Entwicklung domänenspezifischer Sprachen*. Masterarbeit, FernUniversität in Hagen. Fakultät für Mathematik und Informatik. Lehrgebiet Programmiersysteme, 2012. URL http://www.fernuni-hagen.de/imperia/md/content/ps/masterarbeit-blume.pdf. (Zitiert auf den Seiten 17, 22 und 39)

[Bud03] D. M. E. E. R. G. T. J. Budinsky, Frank; Steinberg. *Eclipse Modeling Framework: A Developer's Guide*. 0-13-142542-0. Addison-Wesley, 2nd Auflage, 2003. (Zitiert auf den Seiten 75, 76 und 77)

[Dmi05] S. Dmitriev. Language Oriented Programming: The Next Programming Paradigm. onBoard Electronic Monthly Magazine, 2005. URL http://www.onboard.jetbrains.com/is1/articles/04/10/lop/. (Zitiert auf den Seiten 22, 23 und 39)

[Eis99] E. G. J.-F. W. M. Eisenbarth, Thomas; Koschke Rainer; Plödereder. Projekt Bauhaus: Interaktive und inkrementelle Wiedergewinnung von SW-Architekturen. *Workshop Software-Reengineering. Bad Honnef, Universität Koblenz-Landau : Fachberichte Informatik*, Nr. 7-99:17–26, 1999. (Zitiert auf Seite 10)

[Fow05] M. Fowler. Language Workbenches: The Killer-App for Domain Specific Languages? Website, 2005. URL http://martinfowler.com/articles/languageWorkbench.html. (Zitiert auf den Seiten 17, 18, 19, 20, 21, 22, 23, 24, 25, 27, 39, 76 und 131)

[Fow10] M. Fowler. *Domain Specific Languages*. 0-321-71294-3. Addison-Wesley Professional, print edition Auflage, 2010. (Zitiert auf den Seiten 17, 18, 19, 21, 24, 25, 27, 28, 39, 86 und 131)

[G+] A. A. P. A. Gerard, Sebastien;Cuccuru, et al. Papyrus Eclipse Project. http://www.eclipse.org/papyrus/. URL http://www.eclipse.org/papyrus/. (Zitiert auf Seite 61)

[Gam95] R. J. R. V. J. Gamma, Erich; Helm. *Design Patterns, Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995. (Zitiert auf den Seiten 48, 50, 77, 115, 116, 122 und 123)

[Gue03] C. Guelcue. *The Complete Log4j Manual*. 9782970036906. QOS.ch, 2003. URL http://openlibrary.org/books/OL9012733M/The_Complete_Log4j_Manual. (Zitiert auf Seite 36)

[Ive05] W. Iverson. *Apache Jakarta Commons: Reusable Java(TM) Components (Bruce Perens Open Source)*. 0-13-147830-3. Prentice Hall PTR, Upper Saddle River, NJ, USA, 2005. (Zitiert auf den Seiten 33 und 34)

[Keu05] S. Keul. *Generierung der Zwischendarstellung IML für Ada95 Programme*. Diplomarbeit, Universität Stuttgart, Institut für Softwaretechnologie Abteilung Programmiersprachen und Übersetzerbau, Universitätsstraße 38 D–70569 Stuttgart, 2005. Diplomarbeit Nr. 2417. (Zitiert auf den Seiten 10, 11, 12 und 13)

[Kna02] M. Knauss. *Erweiterung und Generierung der Zwischendarstellung IML für Java-Programme.* Diplomarbeit, Universität Stuttgart, Institut für Softwaretechnologie Abteilung Programmiersprachen und Übersetzerbau, Universitätsstraße 38 D–70569 Stuttgart, 2002. Diplomarbeit Nr. 2323. (Zitiert auf den Seiten 10, 11, 12 und 13)

[Lia99] S. Liang. *The Java Native Interface: Programmer's Guide and Specification*. Addison-Wesley Java series. Prentice Hall, 1999. URL http://books.google.de/books?id=NFnBRcu8DHEC. (Zitiert auf den Seiten 29, 30, 31 und 32)

[Mar98] R. C. Martin. UML Tutorial: Finite State Machines. Engineering Notebook Column, C++ Report, 1998. (Zitiert auf Seite 47)

[Mar08] D. Marx. Command-line Parsing with Apache Commons CLI. http://marxsoftware.blogspot.de/2008/11/command-line-parsing-with-apache.html, 2008. (Zitiert auf Seite 34)

[OMG11] OMG. *OMG Unified Modeling Language (OMG UML), Infrastructure*. Object Management Group, 109 Highland Ave Needham, MA 02494 USA, version 2.4.1 Auflage, 2011. URL http://www.omg.org/spec/UML/2.4.1/Infrastructure/PDF/. (Zitiert auf den Seiten 11, 56, 59, 61, 62, 70 und 76)

[Ora04] Oracle. *How and When To Deprecate APIs*. Oracle Corporation, 500 Oracle Parkway Redwood Shores, CA 94065, 2004. URL http://docs.oracle.com/javase/1.5.0/docs/guide/javadoc/deprecation/deprecation.html. (Zitiert auf Seite 25)

[Par07] T. Parr. *The Definitive ANTLR Reference*. 0-9787392-5-6. The Pragmatic Bookshelf, Raleigh, North Carolina - Dallas, Texas, 2011-3-24 Auflage, 2007. (Zitiert auf den Seiten 90 und 92)

[Pino6] S. Pingel. *Generierung der Zwischendarstellung IML aus Java Classfiles*. Diplomarbeit, Universität Stuttgart, Institut für Softwaretechnologie Abteilung Programmiersprachen und Übersetzerbau, Universitätsstraße 38 D–70569 Stuttgart, 2006. (Zitiert auf den Seiten 10, 11, 12, 13, 32 und 33)

[Roh98] J. Rohrbach. *Erweiterung und Generierung einer Zwischendarstellung für C-Programme*. Studienarbeit nr. 1662, Universität Stuttgart, Institut für Softwaretechnologie, 1998. (Zitiert auf Seite 10)

[Samo9] M. Samek. *Practical UML Statecharts in C & C++*. Elsevier Inc., 2009. (Zitiert auf den Seiten 47, 48 und 50)

[Seto3] T. Setzer, Sebastian; Karaca. *Erweiterung und Generierung der Zwischendarstellung IML für C++ Programme.* Diplomarbeit, Universität Stuttgart, Institut für Softwaretechnologie Abteilung Programmiersprachen und Übersetzerbau, Universitätsstraße 38 D–70569 Stuttgart, 2003. Diplomarbeit Nr. 2048. (Zitiert auf Seite 10)

[Sto10] R. Stoffel. Comparing Language Workbenches. MSE-seminar: Program Analysis and Transformation, 2010. URL http://www.gamlor.info/wordpress/wp-content/uploads/2011/01/ComparingLanguageWorkbenches-Roman-Stoffel-2010-12-23.pdf. (Zitiert auf den Seiten 18, 24, 25 und 28)

Alle URLs wurden zuletzt am 25. 01. 2013 geprüft.

Erklärung

Ich versichere, diese Arbeit selbstständig verfasst zu haben. Ich habe keine anderen als die angegebenen Quellen benutzt und alle wörtlich oder sinngemäß aus anderen Werken übernommene Aussagen als solche gekennzeichnet. Weder diese Arbeit noch wesentliche Teile daraus waren bisher Gegenstand eines anderen Prüfungsverfahrens. Ich habe diese Arbeit bisher weder teilweise noch vollständig veröffentlicht. Das elektronische Exemplar stimmt mit allen eingereichten Exemplaren überein.

Ort, Datum, Unterschift