Institut für Softwaretechnologie

Universität Stuttgart
Universitätsstraße 38
D–70569 Stuttgart

Masterarbeit Nr. 108

SKilLed Bauhaus

Dennis Przytarski

Studiengang:	Softwaretechnik
Prüfer/in:	Prof. Dr. rer. nat. / Harvard Univ. Erhard Plödereder
Betreuer/in:	Dipl.-Inf. Timm Felden
Beginn am:	15. Juni 2016
Beendet am:	15. Dezember 2016
CR-Nummer:	D.2.7, D.3.4, D.4.8

Kurzfassung

Die Werkzeugkette des Projekts Bauhaus verwendet die eigenentwickelte proprietäre Bauhaus Intermediate Language (IML) als Zwischendarstellung. Die IML soll durch einen geeigneten Nachfolger ersetzt werden, der programmiersprachenunabhängiger und toleranter gegenüber einer geänderten IML-Spezifikation ist. Deshalb wurde an der Universität Stuttgart die quelloffene Sprache Serialization Killer Language (SKilL) entworfen, die die gewünschten Eigenschaften bietet.

In der vorliegenden Arbeit wird die generierte IML-Implementierung an das SKilL-Binärformat angepasst. Hierfür werden zwei Codegeneratoren entwickelt. Der erste Codegenerator generiert aus einer IML-Spezifikation die entsprechende SKilL-Spezifikation. Der zweite Codegenerator generiert aus dieser SKilL-Spezifikation die an SKilL angepasste IML-Implementierung. Anschließend wird die SKilL-basierte IML-Implementierung durch verschiedene Bauhauswerkzeuge an mehreren Programmen erfolgreich getestet.

Inhaltsverzeichnis

1 Einleitung

Das Projekt Bauhaus stellt eine größtenteils in der Programmiersprache Ada geschriebene Werkzeugkette zur Verfügung. Die Werkzeugkette erlaubt es, an in den Programmiersprachen C oder C++ entwickelten Programmen verschiedene Programmanalysen wie beispielsweise Codeklonanalysen zur Unterstützung des Reengineerings und des Programmverstehens durchzuführen. Hierfür verwendet die Werkzeugkette die eigenentwickelte proprietäre Bauhaus Intermediate Language (IML) [EKP+99, § 2] als Zwischendarstellung. Dabei werden unter anderem die semantischen und syntaktischen Programminformationen als Graph in Form einer Binärdatei abgespeichert. Die Bibliothek der IML besteht aus einer Spezifikation der zu serialisierenden Programminformationen und der dazugehörigen generierten Application Programming Interface (API), die Zugriffsmethoden auf den IML-Graphen bereitstellt. [EKP+99] [RVP06]

Laut [FW16] sind die Bauhaus-Entwickler mit IML aus folgenden zwei Gründen unzufrieden und fordern einen Nachfolger. Der erste Grund ist, dass alte Binärdateien unter Umständen nicht mehr gelesen werden können, wenn die IML-Spezifikation geändert wird. Der zweite Grund ist, dass Studenten oft universitäre Arbeiten ablehnen, die eine Verwendung von Programmiersprachen bedingen, die sie noch nie verwendet haben oder möglicherweise gar nicht kennen. Dieser Effekt wird dadurch bestärkt, dass die Universität Stuttgart die im Rahmen der ersten Vorlesung gelehrte Programmiersprache Ada durch Java ersetzt hat. Als ein weiterer Grund kann die Mächtigkeit der Beschreibungssprache der IML-Spezifikation aufgeführt werden. Da die Beschreibungssprache keine Hashmaps unterstützt, wird in der Bachelorarbeit [Gai16, § 3.1.1.4, § 3.1.2] eine ausgeflachte Darstellung der Schlüssel-Wert-Paare verwendet. Aufgrund dieser Gründe soll die IML durch SKilL ersetzt werden. Die quelloffene Sprache Serialization Killer Language (SKilL) bietet hierfür die gewünschten Kompatibilitätseigenschaften und die gewünschte Programmiersprachenunabhängigkeit [Fel13, § 1].

Da die Werkzeugkette aus über 2 Millionen Codezeilen besteht, ist es im Rahmen dieser Arbeit nicht möglich, die IML vollständig durch SKilL zu ersetzen. Daher ist das Ziel dieser Arbeit, die generierte IML-Implementierung an das SKilL-Binärformat anzupassen, sodass dieses bei der Serialisierung des IML-Graphen verwendet wird.

Gliederung

Kapitel 2 vermittelt die benötigten Grundlagen zu dem Projekt Bauhaus und zu der Serialisierungssprache SKilL.

Kapitel 3 erfasst den aktuellen Istzustand der Bibliothek `libIML` und beschreibt den Sollzustand.

Kapitel 4 beschreibt die Funktionsweise des ersten Werkzeugs namens Spezifikationsgenerator (SpecGen). Dieses Werkzeug transformiert eine gegebene IML-Spezifikation in eine SKilL-Spezifikation.

Kapitel 5 beschreibt die Funktionsweise des zweiten Werkzeugs namens Codegenerator (CodeGen). Dieses Werkzeug generiert die SKilL-basierte IML-Implementierung anhand einer gegebenen SKilL-Spezifikation.

Kapitel 6 beschreibt die signifikaten Änderungen an den handgeschriebenen Dateien in der Bibliothek `libIML`.

In Kapitel 7 wird die SKilL-basierte IML-Implementierung an verschiedenen Projekten getestet, die aus einer einzelnen Kompilierungseinheit bestehen.

Kapitel 8 bespricht die offenen Punkte, die aufgrund ihres Umfangs nicht im Rahmen dieser Arbeit gelöst wurden und gibt weitere Hinweise.

In Kapitel 9 wird eine Performance-Evaluation zwischen der SKilL-basierten IML-Implementierung und der originalen IML-Implementierung durchgeführt.

Kapitel 10 erörtert die wesentlichen Punkte bei der Übersetzung von nicht direkt in IML spezifizierten Typen nach SKilL.

Abschließend fasst Kapitel 11 die Arbeit zusammen und rundet diese mit einem Ausblick ab.

2 Grundlagen

In diesem Kapitel werden die benötigten Grundlagen für das Projekt Bauhaus und die Serialisierungssprache SKilL behandelt.

2.1 Bauhaus

Das Projekt Bauhaus[1] stellt verschiedene Werkzeuge für die Architektur- und Programmverhaltensanalyse bereit. Diese Werkzeuge können an Programmen, die in den Programmiersprachen C oder C++ entwickelt sind, angewendet werden. Bevor jedoch ein Programm durch ein Werkzeug aus dem Projekt Bauhaus analysiert werden kann, müssen verwertbare Informationen über das Programm in einer geeigneten Form vorliegen. Zu diesem Zweck wurde die proprietäre Bauhaus Intermediate Language (IML) [EKP+99, § 2] entwickelt. [EKP+99] [RVP06]

Die Abbildung 2.1 veranschaulicht die Vorgehensweise zur Generierung eines IML-Graphen aus einem in der Programmiersprache C geschriebenen Programm bestehend aus drei Quelltextdateien [EKP+99]. Beim Kompiliervorgang des Programms wird anstatt des herkömmlichen Compilers das Werkzeug `cafeCC` verwendet. Dieses ermittelt die passenden Umgebungsparameter für die Übersetzung und ruft damit das Werkzeug `cafe++` [SK03] für jede Quelltextdatei des Programms auf. Dieses übersetzt daraufhin jeweils die Quelltextdatei und speichert die semantischen und syntaktischen Informationen über die Quelltextdatei in einem IML-Graphen ab. Das Werkzeug `Linker` fügt anschließend die drei erzeugten IML-Graphen zu einem gesamten IML-Graphen zusammen [Sta09, § 11.2.1]. Das Resultat ist ein IML-Graph über das gesamte Programm. Dieser IML-Graph in Form einer Binärdatei kann nun von anderen Werkzeugen eingelesen, analysiert und weiterverarbeitet werden.

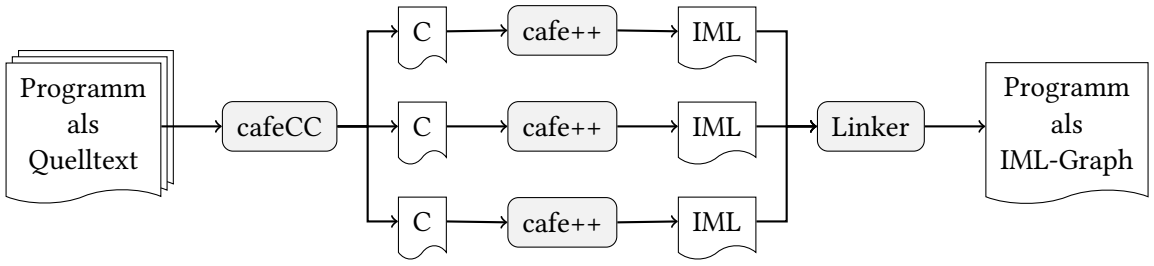


Abbildung 2.1: Der `cafeCC`-Übersetzungsprozess eines Programms vom Quelltext zum IML-Graphen [EKP+99]

[1] http://www.iste.uni-stuttgart.de/ps/projekt-bauhaus

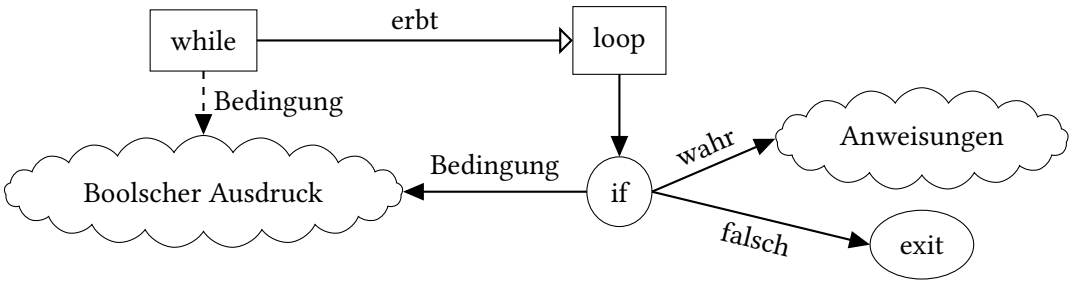


Abbildung 2.2: Die Basisklasse `loop` modelliert die unabhängige Grundstruktur einer Schleife. Die Spezialisierung der `while`-Schleife wird mithilfe einer Vererbung und einer semantischen Kante für die Bedingung modelliert. Ein durchgezogener Pfeil ist eine syntaktische Kante. Ein gestrichelter Pfeil ist eine semantische Kante [KGW98, § 4.1] [EKP+99, § 2]

2.1.1 Bauhaus Intermediate Language (IML)

Unter dem Begriff IML wird je nach Kontext die Beschreibungssprache, die Spezifikation, die daraus generierte API oder der Graph beziehungsweise die Binärdatei, die den Graphen speichert, verstanden.

Das Herzstück der IML ist die Spezifikation, die den Aufbau des Graphen mithilfe einer Klassenhierarchie [RVP06, § 3.2] für die zu speichernden Programminformationen definiert. Dabei ist die Spezifikation hinreichend mächtig, um die Programminformationen programmiersprachenübergreifend abzubilden. Gleichzeitig bleiben die Programminformationen durch Zusatzinformationen wie beispielsweise Quelltextpositionen quellennah, sodass die Analyseergebnisse der Werkzeuge noch am Programm nachvollzogen werden können. [EKP+99, § 2] [SK03, § 1.1]

In der Spezifikation repräsentiert jede Klasse ein bestimmtes Sprachkonstrukt einer Programmiersprache. Eine Instanz einer Klasse modelliert das jeweilige Vorkommen im Programm. Um in der Spezifikation die Anzahl der abbildbaren Sprachkonstrukte für verschiedene Programmiersprachen gering zu halten, werden semantisch äquivalente Sprachkonstrukte durch eine unabhängige Grundstruktur modelliert. Mithilfe der Vererbung werden dann ihre Spezialisierungen abgebildet. [KGW98, § 4.1] [EKP+99, § 2] [RVP06, § 3.2]

Dabei wird in einer Basisklasse ein Sprachkonstrukt syntaktisch vereinheitlicht abgebildet. Alle weiteren Spezialisierungen des Sprachkonstrukts werden jeweils mit den notwendigen semantischen Informationen als Unterklasse abgebildet. Die Abbildung 2.2 veranschaulicht diese Vorgehensweise an einer `while`-Schleife. [KGW98, § 4.1] [EKP+99, § 2]

Die `while`-Schleife wird durch die generelle Schleife mit einem `if-then-else`-Bedingungskonstrukt und einer `exit`-Anweisung, die die unabhängige Grundstruktur einer Schleife ist, abgebildet. Damit jedoch die Information, dass es sich um eine `while`-Schleife handelt, nicht verloren geht, wird die `while`-Schleife als Spezialisierung der generellen Schleife mit einer zusätzlichen semantischen Kante für die Bedingung modelliert. [KGW98, § 4.1] [EKP+99, § 2]

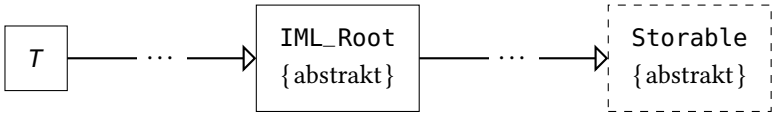


Abbildung 2.3: Die Vererbungshierarchie der Klasse *T*. Eine Klasse mit durchgezogenem Rand wird generiert. Eine Klasse mit gestricheltem Rand ist handgeschrieben

2.1.2 Bibliothek `libIML`

Damit die Spezifikation von den Werkzeugen sinnvoll genutzt werden kann, stellt die Bibliothek `libIML` das Werkzeug `imlgen` [Cze06] zur Verfügung, das eine geeignete API für die Spezifikation generiert. Die Bibliothek `libIML` besteht aus den folgenden drei Komponenten.

Werkzeug `imlgen` Das Werkzeug `imlgen` liest anhand einer der Programmiersprache Ada ähnlichen Beschreibungssprache die Spezifikation ein und generiert aus der definierten Klassenhierarchie die API.

Generierte Dateien Die Ausgabe des Werkzeugs `imlgen` sind die generierten Dateien. Für jede in der Spezifikation definierte Klasse *T* werden die Dateien *T*`_internals.adb`, *T*`_internals.ads`, *T*`s.adb` und *T*`s.ads` generiert.

Handgeschriebene Dateien Die handgeschriebenen Dateien stellen unter anderem Methoden bereit, die das Lesen eines IML-Graphen aus einer Binärdatei beziehungsweise das Schreiben eines IML-Graphen in eine Binärdatei ermöglichen. Des Weiteren werden Methoden zum Erstellen einer neuen leeren Graphinstanz und das Verwalten eines Graphzustands bereitgestellt. Da die Beschreibungssprache in ihrer jetzigen Version nicht mächtig genug ist, um alle benötigten Typen bereitzustellen, existieren für diese Fälle handgeschriebene Dateien, die dann als nicht direkt in IML spezifizierte Typen bezeichnet werden.

2.1.3 Klassenhierarchie der Spezifikation

In der Klassenhierarchie der Spezifikation können abstrakte und konkrete Klassen definiert werden, wobei von abstrakten Klassen keine Instanzen erstellt werden können. Eine generierte Klasse *T* erbt direkt oder indirekt über ihre Oberklassen von der generierten abstrakten Klasse `IML_Root`, die wiederum indirekt von der handgeschriebenen abstrakten Klasse `Storable` erbt. Diese Vererbungshierarchie wird in Abbildung 2.3 veranschaulicht.

Die handgeschriebene Klasse `Storable` definiert unter anderem Lese- und Schreibmethoden, die von der jeweiligen Klasse *T* überschrieben werden müssen. Somit wäre es theoretisch möglich, dass jede Klasse *T* über ihr eigenes Binärformat bezüglich der Speicherung ihrer Felder bestimmt. Praktisch werden die Methoden jedoch nach einem von dem Werkzeug `imlgen` festgelegten Binärformat generiert.

Die generierte Klasse `IML_Root` ist der oberste Typ der generierten Dateien und definiert unter anderem dispatchende Methoden. Eine dieser dispatchenden Methoden gibt für eine übergebene Instanz die dazugehörigen Klasseninformationen für die Reflexion zurück.

Die generierte Klasse *T* definiert ihren Typ und bietet Zugriffsmethoden auf ihre einfachen Felder. Für komplexere Felder wie Listen und Sets werden Iteratoren und weitere Methoden bereitgestellt, sodass die Datenstrukturen entsprechend ihrer Möglichkeiten verwendet und manipuliert werden können. Jede konkrete Klasse *T* stellt außerdem eine Instanziierungsmethode bereit, sodass von ihr Instanzen erstellt werden können.

2.2 Serialization Killer Language (SKilL)

Der Technische Bericht [Fel13] definiert die Beschreibungssprache [Fel13, § 2 ff] und das Serialisierungsformat [Fel13, § 6] der Serialisierungssprache SKilL[2] in englischer Sprache. In dieser Arbeit wird bereits die noch unveröffentlichte Version 1.0 [FelXX] verwendet. Aus diesem Grund werden in diesem Unterkapitel die wichtigsten Aspekte der Beschreibungssprache nochmals zusammengefasst. Das Serialisierungsformat ist für diese Arbeit uninteressant, weil sie damit nicht in Berührung kommt.

2.2.1 Beschreibungssprache

Die Beschreibungssprache ähnelt Programmiersprachen, die geschweifte Klammern für Blockgrenzen verwenden und beschreibt die zu speichernden Datenstrukturen. Der Anwender erstellt mithilfe der Beschreibungssprache eine Spezifikationsdatei. Diese wird dann einer Sprachanbindung übergeben, die unter Zuhilfenahme des dazugehörigen Codegenerators das Binding für die entsprechende Programmiersprache generiert.

Eine Spezifikationsdatei ist eine Ansammlung von Benutzertypen [Fel13, § 4.3], Schnittstellen [FelXX, § 4.4.1], Aufzählungstypen [FelXX, § 4.4.2] und Typdefinitionen [FelXX, § 4.4.3].

Benutzertyp (englisch user type) Ein Benutzertyp ist eine Ansammlung von Feldern. Ein Feld besteht aus einem Typ und einem Namen. Ein Benutzertyp kann von einem anderen Benutzertyp erben und keine oder mehrere Schnittstellen implementieren. Ein Benutzertyp kann von einem Feld als Typ verwendet werden.

Schnittstelle (englisch interface) Eine Schnittstelle ist wie auch beim Benutzertyp eine Ansammlung von Feldern. Wenn eine Schnittstelle von einem Benutzertyp implementiert wird, wird garantiert, dass die Felder aus der Schnittstelle beim Benutzertyp verfügbar sind.

Aufzählungstyp (englisch enumeration, enum) Ein Aufzählungstyp enthält eine Liste von Instanzen. Optional kann danach noch eine Ansammlung von Feldern folgen. Ein Aufzählungstyp kann von einem Feld als Typ verwendet werden.

Typdefinition (englisch type definition, typedef) Eine Typdefinition führt einen neuen Namen für einen bereits existierenden Typ ein, die gegebenenfalls mit zusätzlichen Restriktionen annotiert wird. Eine Typdefinition kann von einem Feld als Typ verwendet werden.

[2] http://www.iste.uni-stuttgart.de/ps/projekt-skill

Die weiteren möglichen Typen für ein Feld werden in die zwei Typbereiche eingebaute Typen [Fel13, § 4.1] und zusammengesetzte Typen [Fel13, § 4.2] unterteilt. Die folgenden eingebauten Typen können verwendet werden.

`annotation` Der Annotationstyp `annotation` verwaltet einen typisierten Zeiger auf einen beliebigen Benutzertyp.

`bool` Der Booleantyp `bool` verwaltet die Wahrheitswerte wahr oder falsch.

Float (`f32`, `f64`) Die Floattypen `f32` und `f64` verwalten Fließkommazahlen nach der Norm IEEE 754 [IEE08] und sind entsprechend vier oder acht Bytes groß.

Integer (`i8`, `i16`, `i32`, `i64`, `v64`) Die Integertypen verwalten Ganzzahlen und besitzen entweder eine feste oder variable Länge. Die Integertypen `i8`, `i16`, `i32` und `i64` mit fester Länge sind entsprechend ein Byte oder zwei, vier sowie acht Bytes groß. Der Integertyp `v64` mit variabler Länge wird im Hauptspeicher wie ein `i64` behandelt. Bei der Serialisierung werden je nach dem zu speichernden Wert zwischen einem Byte und neun Bytes benötigt. Dabei werden die kleinen Werte ($[0, 128) \in \mathbb{N}_0$) mit einem Byte und die großen ($\geq 2^{55}$) sowie negativen Werte mit neun Bytes serialisiert.

`string` Der Stringtyp `string` verwaltet eine Zeichenkette mit variabler Länge, die aus UTF-8 kodierten Unicode-Zeichen besteht. Dabei darf es sich um keine nullterminierte Zeichenkette handeln, weil sie bei verschiedenen Programmiersprachen zu Problemen führen kann.

Unter Verwendung eines zusammengesetzten Typs können Instanzen des gleichen Typs zusammengefasst werden. Hierfür stehen Arrays mit fixer und variabler Länge sowie Listen, Sets und Maps zur Verfügung. Arrays mit fixer Länge sind nicht größenveränderbar, Sets dürfen die selbe Instanz nur einmal beinhalten und Maps können verschachtelt werden. Ein zusammengesetzter Typ darf als Typparameter keinen anderen zusammengesetzten Typ verwenden, auch nicht über den Umweg einer Typdefinition. [Fel13, § 3.5, § 4.2]

Die Benutzertypen und ihre Felder sowie die Typdefinitionen können mit zusätzlichen Restriktionen [Fel13, § 5.1] (englisch restrictions) und Hinweisen [Fel13, § 5.2] (englisch hints) annotiert werden. Die Restriktionen werden während der Serialisierung und Deserialisierung überprüft und müssen deshalb mitserialisiert werden — auch um die gewünschten Kompatibilitätseigenschaften zu gewährleisten. Die Hinweise werden verwendet, um das generierte Binding zu optimieren und müssen deshalb nicht serialisiert werden. Die Schnittstellen oder die Aufzählungstypen dürfen nicht mit Restriktionen und Hinweisen annotiert werden.

Eine Restriktion startet mit einem `@`-Zeichen, gefolgt von einem Bezeichner und falls nötig, den Argumenten. Die folgenden Restriktionen sind für diese Arbeit interessant.

`abstract` Die Restriktion `abstract` kann auf Benutzertypen angewendet werden und gewährleistet, dass keine Instanzen von diesem Benutzertyp erstellt werden können. [FelXX, § 5.1.1.4]

`default` Die Restriktion `default` kann auf Felder angewendet werden, um Standardwerte für eingebaute Typen festzulegen. [FelXX, § 5.1.1.4]

range Die Restriktion `range` kann auf Felder angewendet werden, um die Integer- und Floattypen einzuschränken. Dabei können jeweils ein Minimum- und Maximumwert definiert sowie die Randwerte ein- oder ausgeschlossen werden. [Fel13, § 5.1]

singleton Die Restriktion `singleton` kann auf Benutzertypen angewendet werden und gewährleistet, dass maximal eine Instanz von diesem Benutzertyp existiert. [Fel13, § 5.1]

Ein Hinweis startet mit einem `!`-Zeichen, gefolgt von einem Bezeichner und falls nötig, den Argumenten. Die folgenden Hinweise sind für diese Arbeit interessant.

onDemand Der Hinweis `onDemand` kann auf Felder angewendet werden und gibt an, dass das entsprechende Feld nur deserialisiert werden soll, wenn es auch verwendet wird. [FelXX, § 5.2.2]

pragma Der Hinweis `pragma` kann auf Benutzertypen, Felder und Typdefinitionen angewendet werden, falls weitere Informationen an den Codegenerator übergeben werden sollen. [FelXX, § 5.2.14]

Ein Feld, das beim Typ das Schlüsselwort `auto` vorangestellt hat, ist flüchtig. Ein flüchtiges Feld wird nur für Berechnungen verwendet und deshalb bei der Serialisierung ignoriert. [Fel13, § 3.4]

Des Weiteren können Sichten (englisch views) auf Felder angewendet werden, um Felder umzubenennen. Dabei werden alle Zugriffe auf das Feld zum gesichteten Feld weitergeleitet. Wird ein Benutzertyp als Typ verwendet, kann dieser außerdem umtypisiert werden. [FelXX, § 4.4.4]

Das Listing 2.1 veranschaulicht eine beispielhafte Spezifikationsdatei.

Listing 2.1: Beispiel einer Spezifikationsdatei

```
/** Eine Schnittstelle namens I mit einem Feld namens a vom Typ v64 */
interface I { v64 a; }
/** Ein Benutzertyp namens A mit einem Feld namens b vom Typ string */
A { string b; }
/** Ein Benutzertyp namens B, der vom Benutzertyp A erbt und die Schnittstelle I implementiert.
    Weiterhin wird das Feld namens c definiert, das vom Typ eine Liste mit dem Typparameter f64
    ist */
B : A with I { list<f64> c; }
/** Eine Typdefinition namens Positive, die den Integertyp v64 umbenennt und mit einer
    min-Restriktion annotiert, sodass die zu speichernde Ganzzahl positiv sein muss */
typedef Positive
@min(1)
v64;
/** Ein Benutzertyp namens C, der vom Benutzertyp B erbt. Weiterhin wird eine Sicht definiert,
    die das Feld a nach d umbenennt. Anschließend wird ein flüchtiges Feld namens e vom Typ
    Positive definiert */
@singleton
C : B {
  view a as
  v64 d;

  auto Positive e;
}
```

2.2.2 Kompatibilitätseigenschaften

Bei der Entwicklung von neuen Werkzeugen oder bei der Weiterentwicklung von bisherigen Werkzeugen kann es vorkommen, dass die vorhandene Spezifikation geändert werden muss. Damit alte und neue Binärdateien unter den verschiedenen Spezifikationsversionen lesbar bleiben, bietet die Serialisierungssprache SKilL ein Maximum an Abwärts- und Aufwärtskompatibilität an [Fel13, § 1.1].

Die Abwärtskompatibilität stellt sicher, dass die Binärdateien von alten Werkzeugen auch von neuen Werkzeugen gelesen werden können. Die Aufwärtskompatibilität stellt sicher, dass die Binärdateien von neuen Werkzeugen auch von alten Werkzeugen gelesen werden können. Die Serialisierungssprache SKilL gewährleistet dies, indem unbekannte Benutzertypen und unbekannte Felder von bekannten Benutzertypen beim Lesen übersprungen werden [Fel13]. Unbekannt bedeutet hier, dass sie nicht in der verwendeten Spezifikation definiert sind.

Die Aufwärtskompatibilität soll am folgenden Beispiel veranschaulicht werden. In Listing 2.2 ist der Benutzertyp namens `Person` mit einem Feld namens `name` vom Typ `string` definiert. Ein mögliches Werkzeug könnte alle Namen der Personen in alphabetischer Reihenfolge ausgeben.

Listing 2.2: Person mit Name

```
Person { string name; }
```

Ein zweites Werkzeug, das auf der bisherigen Spezifikation aus Listing 2.2 aufbaut, reichert die Namen mit einem dazugehörigen Alter an. Dazu wird in Listing 2.3 der bisherige Benutzertyp `Person` um ein weiteres Feld namens `age` vom Typ `Natural` erweitert.

Listing 2.3: Person mit Name und Alter

```
typedef Natural
@min(0)
v64;

Person {
  string name;
  Natural age;
}
```

Alle Binärdateien, die nun mit dem zweiten Werkzeug erstellt werden, müssen und können noch weiterhin vom ersten Werkzeug gelesen werden.

Alle Binärdateien, die mit dem ersten Werkzeug erstellt wurden, können auch weiterhin vom zweiten Werkzeug eingelesen werden, weil ein bekanntes, aber nicht eingelesenes Feld mit fest vorgegebenen Standardwerten befüllt wird. In diesem Fall ist der Standardwert des Integertyps `Natural` die Ganzzahl 0.

2.2.3 Programmiersprachenunabhängigkeit

Seit der Veröffentlichung der Serialisierungssprache SKilL im Jahr 2013 wurden bereits mehrere Sprachanbindungen für verschiedene Programmiersprachen mit unterschiedlichen Programmierparadigmen entwickelt.

Die erste Sprachanbindung [ski] wurde für die Programmiersprache Scala [Ode+] entwickelt. Die Programmiersprache Scala ist der Programmiersprache Java ähnlich und vereint die funktionale und objektorientierte Programmierung in einer Programmiersprache. Die Sprachanbindung an die Programmiersprache Scala gilt zurzeit als Referenzimplementierung für andere Sprachanbindungen. Die dazugehörige Testsuite wird regelmäßig auf einem linuxbasierten Betriebssystem ausgeführt.

In der Bachelorarbeit [Prz14] wurde eine Sprachanbindung an die Programmiersprache Ada entwickelt, um die Performance mit der Sprachanbindung an die Programmiersprache Scala vergleichen zu können. Die hierfür getätigte Performance-Evaluation wurde auf dem Betriebssystem Mac OS X durchgeführt.

In der Diplomarbeit [Ung14] wurde eine Sprachanbindung an die Programmiersprache Java entwickelt, um die Nutzbarkeit der Serialisierungssprache SKilL zu evaluieren. Dabei wurde untersucht, wie die generierte API auszusehen hat, damit sie sich nahtlos in die Programmiersprache Java integriert.

In der Diplomarbeit [Har14] wurde eine Sprachanbindung an die Programmiersprache C entwickelt, um nachzuweisen, dass eine Sprachanbindung an eine nicht objektorientierte Programmiersprache möglich ist.

In einer bisher unveröffentlichten Bachelorarbeit [has] wurde eine Sprachanbindung an die Programmiersprache Haskell entwickelt, um nachzuweisen, dass eine Sprachanbindung an eine nicht objektorientierte und auch nicht statisch typisierte Programmiersprache möglich ist.

In der Masterarbeit [Rot15] wurden verschiedene Möglichkeiten zur Reduktion des Hauptspeicherverbrauchs der Referenzimplementierung untersucht. Die hierfür getätigte Evaluation wurde auf dem Betriebssystem Windows durchgeführt.

Zusammenfassend kann festgehalten werden, dass die Serialisierungssprache SKilL sprach- und plattformunabhängig ist, weil bereits verschiedene Sprachanbindungen für unterschiedliche Programmierstile existieren und diese auf unterschiedlichen Plattformen beziehungsweise Betriebssystemen ausgeführt wurden.

3 Roadmap

In diesem Kapitel werden der aktuelle Istzustand der Bibliothek `libIML` erfasst und der Sollzustand beschrieben. Das Ziel dieses Kapitels ist, eine Vorgehensweise zu ermitteln, die es ermöglicht, die generierte IML-Implementierung an das SKilL-Binärformat anzupassen.

3.1 Istzustand

Die Bibliothek `libIML` besteht im Istzustand aus der Spezifikation (*iml.spec*), den generierten Dateien (*generated*) und den handgeschriebenen Dateien (*source*). Das Werkzeug `imlgen` generiert anhand der eingelesenen Spezifikation *iml.spec* die Dateien für *generated*.

Die IML-Implementierung besteht aus den generierten und handgeschriebenen Dateien und stellt damit eine API zur Verfügung, die die gegebene IML-Spezifikation unterstützt. Dies wird in Abbildung 3.1 veranschaulicht.

3.2 Sollzustand

Bei der Analyse der Bibliothek `libIML` für den Istzustand konnte für den Sollzustand ermittelt werden, dass die handgeschriebenen Dateien teilweise angepasst werden müssen, teilweise ohne Änderungen übernommen werden können oder teilweise nicht mehr verwendbar sind beziehungsweise nicht mehr benötigt werden. Weiterhin konnte ermittelt werden, dass einige generierte Dateien ohne Änderungen übernommen werden können.

Die Bibliothek `libIML` besteht im Sollzustand aus den zwei Spezifikationen (*iml.spec*, *iml.skill*), den generierten Dateien (*generated*, *generated'*), den handgeschriebenen Dateien (*source*, *source'*) und dem Ada-Binding. Das Werkzeug `imlgen` generiert anhand der eingelesenen Spezifikation *iml.spec* die Dateien für *generated*. Das Werkzeug `SKilL` generiert anhand der eingelesenen Spezifikation *iml.skill* das Ada-Binding.

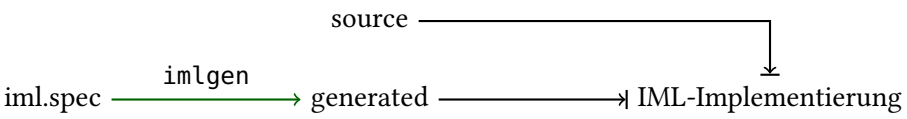


Abbildung 3.1: Der Istzustand der Bibliothek `libIML`. Ein grüner Pfeil markiert ein vorhandenes Werkzeug. Ein Pfeil mit einem Querbalken markiert eine Verwendung der Dateien

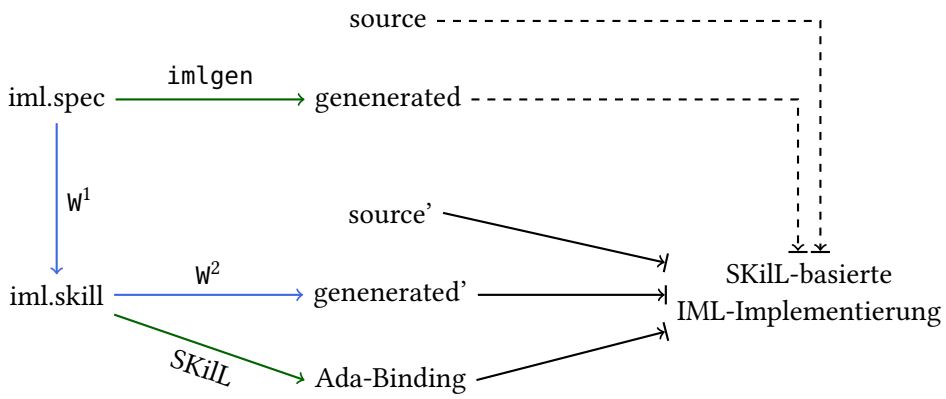


Abbildung 3.2: Der Sollzustand der Bibliothek `libIML`. Ein grüner Pfeil markiert ein vorhandenes Werkzeug. Ein blauer Pfeil markiert ein noch zu erstellendes Werkzeug. Ein durchgezogener Pfeil mit einem Querbalken markiert eine Verwendung der Dateien. Ein gestrichelter Pfeil mit einem Querbalken markiert die eingeschränkte Verwendung der Dateien

Die IML-Implementierung, die das SKilL-Binärformat unterstützt, besteht aus den neuen generierten Dateien (*generated'*), den angepassten handgeschriebenen Dateien (*source'*), dem Ada-Binding und teilweise aus den alten generierten und handgeschriebenen Dateien, die ohne Änderungen übernommen werden können. Die SKilL-basierte IML-Implementierung stellt damit eine API zur Verfügung, die das SKilL-Binärformat verwendet und die gegebene IML-Spezifikation unterstützt. Dies wird in Abbildung 3.2 veranschaulicht.

3.3 Vorgehensweise

Das Ziel ist nun eine Vorgehensweise zu ermitteln, die die Bibliothek `libIML` vom Istzustand in den Sollzustand überführt. In Abbildung 3.2 markieren die blauen Pfeile die Werkzeuge, die noch fehlen und zu erstellen sind. Dadurch ist erkennbar, dass für eine erfolgreiche Überführung vom Istzustand in den Sollzustand noch zwei Werkzeuge benötigt werden.

Das erste Werkzeug W^1 muss eine IML-Spezifikation lesen und daraus eine SKilL-Spezifikation generieren können. Das zweite Werkzeug W^2 muss aus einer SKilL-Spezifikation die Dateien generieren können, die normalerweise mithilfe des Werkzeugs `imlgen` und einer IML-Spezifikation generiert werden. Weiterhin müssen vorhandene handgeschriebene Dateien so angepasst werden, dass sie von den generierten Dateien und dem Ada-Binding verwendet werden können.

3.4 Alternative Vorgehensweise

Eine alternative Vorgehensweise ist, die IML-Typen nicht durch ihre jeweiligen SKilL-Typen zu ersetzen und somit den originalen IML-Zustand weiterhin aufzubauen. Das hat den Vorteil, dass die

vom Werkzeug `imlgen` generierten Dateien vollständig übernommen werden können. Damit jedoch der IML-Zustand aus einer Datei im SKilL-Binärformat gelesen beziehungsweise in eine Datei im SKilL-Binärformat geschrieben wird, müssen in den handgeschriebenen Dateien zwei Methoden erweitert werden.

Die erste zu erweiternde Methode ist die Lademethode eines IML-Graphen aus einer Binärdatei im Ada-Paket `IML.IO`. Dort muss der eingelesene SKilL-Zustand in einen semantisch äquivalenten IML-Zustand transformiert werden. Die zweite zu erweiternde Methode ist die Speichermethode eines IML-Graphen in eine Binärdatei im Ada-Paket `IML.IO`. Dort muss der zu speichernde IML-Zustand in einen semantisch äquivalenten SKilL-Zustand transformiert werden.

Diese alternative Vorgehensweise hat jedoch mindestens drei Nachteile. Der erste Nachteil ist, dass je nach Implementierung des Transformators im ungünstigsten Fall zwei komplette Graphinstanzen im Hauptspeicher verwaltet werden müssen. Der zweite Nachteil ist, dass die Ausführungszeit eines Werkzeugs aufgrund der Transformierung der Zustände unnötigerweise verlängert wird. Der dritte Nachteil ist, dass zwei verschiedene Binärformate gewartet werden müssen. Bereits aufgrund dieser drei Nachteile wurde diese alternative Vorgehensweise verworfen.

4 Spezifikationsgenerator (SpecGen)

In diesem Kapitel wird die Funktionsweise des ersten Werkzeugs beschrieben. Das erste Werkzeug soll eine IML-Spezifikation einlesen und daraus eine SKilL-Spezifikation generieren können.

Um dies zu erreichen, wird ein Parser benötigt, der aus einer IML-Spezifikation einen Abstract Syntax Tree (AST) erstellt. Im nächsten Schritt verwendet der Spezifikationsgenerator (SpecGen) den AST, um daraus eine SKilL-Spezifikation zu generieren. Dieses Vorgehen wird in Abbildung 4.1 veranschaulicht.

Sowohl der Parser als auch der SpecGen wurden in der Programmiersprache Scala entwickelt. Die Softwarearchitektur des SpecGens basiert auf dem SKilL-Codegenerator für das Scala-Binding.

Da im gesamten Projekt Bauhaus nur eine IML-Spezifikation existiert, wurde der entwickelte SpecGen und der dazugehörige Parser nur an dieser einen IML-Spezifikation getestet. Dabei stand bereits während der Entwicklung des SpecGens eine handübersetzte SKilL-Spezifikation von dieser einen IML-Spezifikation zur Verfügung [FW16], sodass die generierte SKilL-Spezifikation regelmäßig mit der handgeschriebenen SKilL-Spezifikation abgeglichen werden konnte.

4.1 Parser

Die Grammatik einer IML-Spezifikation stand für den zu entwickelnden Parser zu Beginn dieser Arbeit in Form einer Quelltextdatei zur Verfügung. Die Quelltextdatei gehört dem Werkzeug `imlgen` an, das in der Programmiersprache OCaml entwickelt wurde. Weiterhin wurde die Grammatik für die einzige existierende und verwendete IML-Spezifikation optimiert, sodass unter anderem ein modularer Aufbau der generierten SKilL-Spezifikation ermöglicht wurde. Vor dem Parsen wurden alle Kommentarzeilen, die mit der Zeichenkombination `##` oder `#@` eingeleitet werden, entfernt.

Eine IML-Spezifikation beginnt immer mit einem *HierarchyName*, gefolgt von mehreren *Builtin*s und der Angabe einer *Root*-Klasse sowie von mindestens einem *Package*, wie der Auszug in Listing 4.1 belegt.

Abbildung 4.1: Zuerst liest der Parser eine IML-Spezifikation ein und erstellt daraus einen AST. Danach generiert der SpecGen aus dem AST eine SKilL-Spezifikation

Listing 4.1: Auszug aus der OCaml-Quelltextdatei

```
%start description
%type <Ast.t> description
description : hierarchy_name builtins root packages EOF
{ { ast_hierarchy_name = $1; ... ast_root = $3; ... } }
```

Das Listing 4.2 stellt eine beispielhafte IML-Spezifikation für die Grammatik dar. Sie wurde anhand des Schleifen-Beispiels in Unterabschnitt 2.1.1 aufgebaut.

Listing 4.2: Beispielhafte IML-Spezifikation

```
name is "...";
builtin SLoc in package SLocs;
...
root IML_Root;

package IML is

  #CHAPTER Root
  abstract class IML_Root
  inherits ...
  is
    ...
    semantic SLoc : builtin SLoc;
    # The source location
    ...
  end class IML_Root;

  #CHAPTER Loops
  abstract class Loop
  inherits IML_Root
  is
    ...
    syntactic Initialization : class ...;
    ...
  end class Loop;

  concrete class While_Loop
  inherits Loop
  # while (...) { ... }
  is
    semantic Condition : class ...;
    # The loop condition
  end class While_Loop;

end package IML;
```

Im Paket `IML` werden drei Klassen definiert. Die erste Klasse ist eine abstrakte Klasse namens `IML_Root` mit einem semantischem Feld namens `SLoc` vom Builtin-Typ `SLoc`. Sie gibt den obersten Typ der generierten Dateien wieder. Die zweite Klasse ist eine abstrakte Klasse namens `Loop` und erbt von der abstrakten Klasse `IML_Root`. Sie besitzt ein syntaktisches Feld namens `Initialization`, dessen Typ eine in der IML-Spezifikation spezifizierte Klasse ist. Die dritte Klasse ist eine konkrete Klasse

namens `While_Loop` und erbt von der abstrakten Klasse `Loop`. Sie besitzt ein semantisches Feld namens `Condition`, dessen Typ eine in der IML-Spezifikation spezifizierte Klasse ist.

4.2 Parserkombinatoren

Der zu entwickelnde Parser wurde mithilfe von Parserkombinatoren[3] [MPO08] entwickelt. Eine Funktion höherer Ordnung ist eine Funktion, die mindestens eine Funktion als Eingabe akzeptiert oder eine Funktion zurückgibt [Ode14, § 5]. Ein Kombinator ist eine Funktion höherer Ordnung, die nur Funktionen als Eingabe akzeptiert und deren Kombinierung als neue Funktion zurückgibt [LP15, § 8]. Ein Parserkombinator ist somit ein Kombinator, der Parser und gegebenenfalls Funktionen als Eingabe akzeptiert und einen neuen Parser zurückgibt. Des Weiteren können Parsergeneratoren verwendet werden, wenn der gleiche Parser wiederholt aufgerufen werden soll.

Das Listing 4.3 zeigt den dazugehörigen Parserkombinator für das Listing 4.1. Die Variablen `BOF` und `EOF` stellen jeweils den Anfang und das Ende des eingelesenen Strings, also der IML-Spezifikation, sicher.

Listing 4.3: Dazugehöriger Parserkombinator für Listing 4.1

```
private val BOF = """\A""".r /** Stringanfang */
private val EOF = """\Z""".r /** Stringende */
private def file =
  BOF ~> hierarchyName ~! rep(builtin) ~! root ~! rep1(package) <~ EOF ^^ {
    case h ~ b ~ r ~ p => new Description(h, b, r, p)
  }
```

Das Listing 4.3 verwendet außer dem Sequenzkombinator im ersten Punkt folgende Parserkombinatoren und -generatoren [MPO08] [pcl].

~ Dieser Kombinator namens Sequenzkombinator akzeptiert zwei Parser als Eingabe und gibt einen neuen Parser zurück, der zuerst den linken und dann den rechten Parser aufruft.

~>, <~ Diese Kombinatoren akzeptieren zwei Parser als Eingabe und geben einen neuen Parser zurück, verwerfen jedoch dabei das linke beziehungsweise rechte Parserresultat.

~! Dieser Kombinator ist der Sequenzkombinator ohne Rücksetzverfahren (englisch backtracking).

rep, **rep1** Diese Generatoren akzeptieren einen Parser als Eingabe, der wiederholt bis zum ersten Fehlschlag aufgerufen wird. Dabei muss der letztere Generator mindestens einmal erfolgreich gewesen sein. Anschließend wird ein neuer Parser zurückgegeben, der in einer Liste die Parserresultate verwaltet.

^^ Dieser Kombinator akzeptiert einen Parser und eine Funktion als Eingabe und gibt einen neuen Parser zurück, wendet jedoch zuvor die Funktion auf das Parserresultat an.

[3] https://github.com/scala/scala-parser-combinators

Wenn die Spezifikation durch die Parserkombinatoren erfolgreich geparst und der AST aufgebaut werden konnte, existiert zum Schluss eine Instanz von der Klasse `Description` (siehe Listing 4.4). Diese Instanz wird vom SpecGen als Eintrittspunkt verwendet.

Listing 4.4: Klasse `Description` im AST

```
sealed abstract class Node;
final class Description(
   val hierarchyName : HierarchyName,
   val builtins : List[Builtin],
   val root : Root,
   val packages : List[Package]) extends Node
```

Die Klasse namens `Description` besteht aus dem *HierarchyName*, einer Liste von *Builtin*s, dem *Root*-Knoten und einer Liste von *Package*s.

4.3 Abstract Syntax Tree (AST)

Der interessante Teil der Klasse `Description` ist die Liste von *Package*s. Daher wird in dieser Sektion nur auf den Unterbaum eines *Package*s eingegangen.

Die Grammatik wurde so erweitert, dass ein *Chapter* jeweils seine dazugehörigen Deklarationen verwaltet, indem der Parser das Vorkommen des Kommentars `#CHAPTER` als Eröffnung eines neuen *Chapter*s behandelt und alle nachfolgenden Klassen und Schnittstellen diesem *Chapter* zuordnet. Die Abbildung 4.2 veranschaulicht den Aufbau der Klasse *Package* im AST.

Ein *Package* besteht aus mehreren *Chapter*s. Ein *Chapter* besteht aus mehreren *Declaration*s. Eine *Declaration* besteht aus mehreren *Field*s. Eine *Declaration* wird wiederum in eine *Class* und ein *Interface* unterteilt.

In der IML-Spezifikation kann eine Klasse entweder als abstrakt oder konkret definiert werden. Aus der abstrakten Klasse *Class* werden daher folgende Unterklassen abgeleitet.

AbstractClass Von einer in der IML-Spezifikation spezifizierten abstrakten Klasse können keine Instanzen erstellt werden.

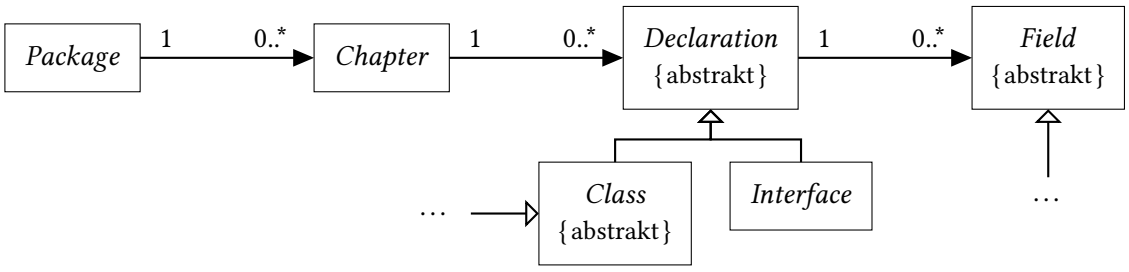


Abbildung 4.2: Der Aufbau des Unterbaums der Klasse *Package* im AST

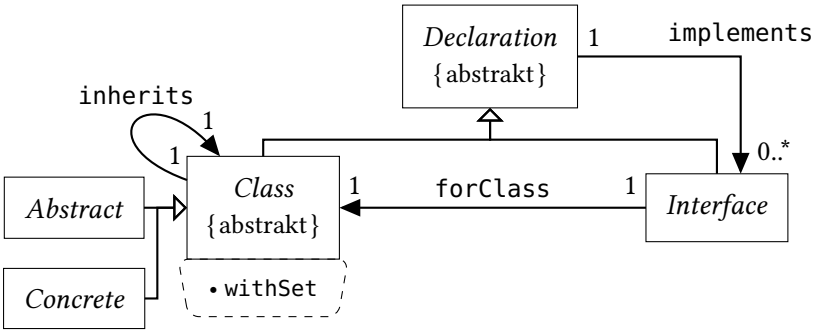


Abbildung 4.3: Der Aufbau des Unterbaums der Klasse *Declaration* im AST

ConcreteClass Von einer in der IML-Spezifikation spezifizierten konkreten Klasse können Instanzen mithilfe der Instanziierungsmethode in der jeweiligen Klasse erstellt werden.

In der Abbildung 4.3 wird der Aufbau des Unterbaums der Klasse *Declaration* im AST veranschaulicht. Eine Deklaration kann keine oder mehrere Schnittstellen implementieren. Eine Klasse muss von einer anderen Klasse erben. Eine Klasse mit der `withSet`-Anweisung in der Klassendeklaration gibt an, dass die Klasse als Typparameter eines Sets verwendet werden soll und daher die Funktionalität eines Sets bereitstellen muss. Eine Schnittstelle muss zudem die `forClass`-Anweisung in der Schnittstellendeklaration enthalten, die auf eine beliebige Klasse zeigt und angibt, dass die Schnittstelle an die angegebene Klasse gebunden ist. In anderen Worten ausgedrückt bedeutet das, dass nur die Unterklassen der angegebenen Klasse die Schnittstelle implementieren können.

In der IML-Spezifikation kann ein Feld entweder ein semantisches oder syntaktisches Feld sein sowie ein semantisches Feld überschreiben. Aus der abstrakten Klasse *Field* werden daher folgende Unterklassen abgeleitet.

SemanticField Ein semantisches Feld kann entweder einen simplen oder einen zusammengesetzten Typ verwenden. Ein simpler Typ ist entweder ein *Builtin*, eine *Class* oder ein *Interface*. Ein Feld mit einem simplen Typ kann zudem die Schlüsselwörter `default` und `omitted` deklarieren. Mit dem ersten Schlüsselwort `default` wird durch das Argument ein Standardwert für das Feld deklariert. Mit dem zweiten Schlüsselwort `omitted` wird deklariert, dass das Feld kein Parameter in der Instanziierungsmethode der jeweiligen Klasse ist. Ein zusammengesetzter Typ ist entweder eine Liste oder ein Set, die als Typparameter nur einen simplen Typ verwenden kann.

OverrideSemanticField Es kann nur ein semantisches Feld, dessen Typ eine Klasse ist, überschrieben werden.

SyntacticField Ein syntaktisches Feld kann nur eine Klasse oder eine Liste mit dem Typparameter einer Klasse als Typ verwenden. Die Besonderheit an den syntaktischen Feldern ist, dass sie einen Baum aufspannen.

In der Abbildung 4.4 wird das Typsystem für die drei verschiedenen Unterklassen der abstrakten Klasse *Field* visualisiert. Der SpecGen generiert aus dem AST die SKilL-Spezifikation, indem die

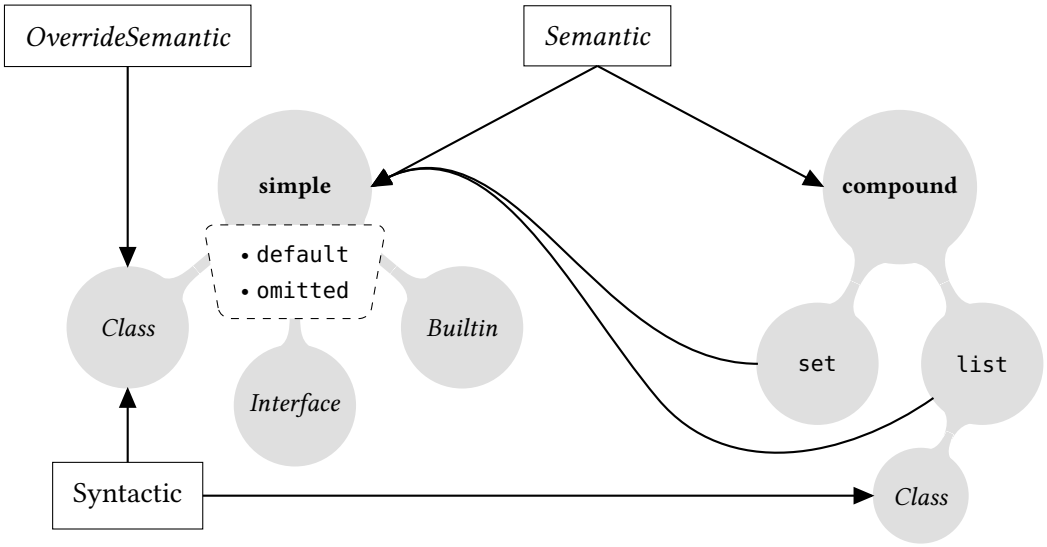


Abbildung 4.4: Das Typsystem der Beschreibungssprache einer IML-Spezifikation

Instanz der Klasse `Description` aus Listing 4.4 als Eintrittspunkt verwendet wird. Dabei werden zuerst die *Builtin*s und dann die *Package*s generiert.

4.4 Builtins

Normalerweise werden alle verwendbaren Typen in der IML-Spezifikation direkt in der Datei spezifiziert. Die Beschreibungssprache für die IML-Spezifikation stellt jedoch durch die *Builtin*s eine Möglichkeit zur Verfügung, die Typen auch außerhalb der IML-Spezifikation spezifizieren zu können. Daher werden diese Typen auch als nicht direkt in IML spezifizierte Typen bezeichnet. Damit diese Typen trotzdem ihren Weg in den SKilL-Zustand finden und serialisiert werden können, müssen diese Typen auch im generierten Ada-Binding und somit in der generierten SKilL-Spezifikation vorhanden sein. Diese Typen werden daher manuell zum SpecGen hinzugefügt.

4.4.1 `SLoc`

In der beispielhaften IML-Spezifikation in Listing 4.2 ist ein *Builtin* namens `SLoc` definiert. Dieser Typ ist Teil der handgeschriebenen Dateien und dort in dem Ada-Paket `SLocs` definiert. Das Listing 4.5 stellt die entsprechende Übersetzung nach SKilL dar.

Der Benutzertyp namens `SLoc` besteht aus mehreren Feldern. Das erste Feld heißt `Pos` und ist vom Typ `Sloc_Position`. Das zweite Feld heißt `Pathname` und ist vom Typ `Identifier`. Die Besonderheit dieses Benutzertyps ist, dass er eigenständig ist und von keinem anderen Benutzertyp abgeleitet wird. Der Benutzertyp namens `Sloc_Position` besteht aus den zwei Feldern namens `Line` und `Column`, die beide vom Typ `Natural` sind. Der Benutzertyp `SLoc` gibt die Position eines Vorkommens eines Sprachkonstrukts im Programm wieder.

Listing 4.5: `SLoc` in SKilL

```
!pragma inPackage(SLocs)
Sloc {
  /* the line/column number in the file */
  Sloc_Position Pos;
  Identifier Pathname;
  ...
}

Sloc_Position {
  Natural Line;
  Natural Column;
}
```

4.4.2 `Identifier`

Der Typ namens `Identifier` ist auch Teil der handgeschriebenen Dateien und dort in dem Ada-Paket `Identifiers` definiert. Das Listing 4.6 stellt die entsprechende Übersetzung nach SKilL dar. Der Benutzertyp namens `Identifier` besteht aus einem Feld namens `Name` vom Typ `string`. Die Besonderheit dieses Benutzertyps ist, dass er auch eine Typdefinition hätte sein können. Jedoch erwarten einige Werkzeuge, dass der Benutzertyp `Identifier` vom Benutzertyp `Storable` ableitet und das sogar überprüfen. Der Benutzertyp `Storable` ist die Wurzel, von der alle direkt spezifizierten Typen in der IML-Spezifikation indirekt über den Typ `IML_Root` ableiten.

Listing 4.6: `Identifier` in SKilL

```
!pragma inPackage(Identifiers)
Identifier : Storable {
  string Name;
}

/** Eine Typdefinition ist zurzeit nicht möglich! */
typedef Identifier
string;
```

4.4.3 Typdefinitionen

Da die IML-Spezifikation benannte Typen verwendet, die bereits mithilfe des Typsystems in SKilL abgebildet werden können, werden für diese Fälle Typdefinitionen verwendet.

Ein `Natural` spiegelt alle positiven Ganzzahlen inklusive der Null ($n \in \mathbb{N}_0$) wider. Dieser Typ wird nach SKilL somit nicht als Benutzertyp, sondern als Typdefinition übersetzt. Das Listing 4.7 stellt die entsprechende Übersetzung nach SKilL dar.

Listing 4.7: `Natural` in SKilL

```
typedef Natural
@min(0)
v64;
```

Die Typdefinition `Natural` führt einen neuen Namen für den Integertyp `v64` ein. Damit sich der Typ `Natural` in SKilL genauso wie in der Programmiersprache Ada verhält, wird dieser mit einer `min`-Restriktion annotiert. Sie gibt an, dass die Ganzzahl nicht negativ sein darf.

Die weiteren benannten Typen sind der Integertyp `Integer` und der Booleantyp `Boolean`. Die Typdefinition `Integer` führt einen neuen Namen für den Integertyp `v64` ein, jedoch ohne eine zusätzliche Restriktion. Die Typdefinition `Boolean` führt einen neuen Namen für den Booleantyp `bool` ein.

4.4.4 Aufzählungstypen

In der IML-Spezifikation kann außerdem mithilfe eines *Builtin*s ein Aufzählungstyp spezifiziert werden. Dafür wird ein Ada-Paket erstellt, das einen Aufzählungstyp definiert. Das Listing 4.8 veranschaulicht einen beispielhaften Aufzählungstyp.

Listing 4.8: Beispielhafter Aufzählungstyp in Ada

```
package Enums is
   ...
   type Enum is (A, B, C);
   ...
end Enums;
```

Der Aufzählungstyp namens `Enum` enthält die Enum-Konstanten A, B und C. Das Listing 4.9 stellt eine typische Übersetzung des Aufzählungstyps aus Listing 4.8 nach SKilL dar.

Listing 4.9: Beispielhafter Aufzählungstyp in SKilL

```
Enum { A, B, C }
```

Das Problem ist nun, dass das Ada-Binding für die SKilL-Spezifikation aus Listing 4.9 für jede Enum-Konstante einen eigenen Typ generiert (siehe Listing 4.10).

Listing 4.10: Generierte Aufzählungstypen durch das Ada-Binding

```
type Enum_a_T is new Enum_T ...
type Enum_b_T is new Enum_T ...
type Enum_c_T is new Enum_T ...
```

In der Programmiersprache Ada kann eine Enum-Konstante in einem Aufzählungstyp nicht in einen Zeiger konvertieren. Daher kann ein Aufzählungstyp in der Programmiersprache Ada nicht durch einen Aufzählungstyp in SKilL abgebildet werden. Jedoch bietet die Programmiersprache Ada das Attribut `'Pos`, um die Position einer Enum-Konstante in einem Aufzählungstyp zu ermitteln. Die

dazugehörige Inverse ist das Attribut `'Val`. Aus diesem Grund wird der beispielhafte Aufzählungstyp aus Listing 4.8 nach SKilL als Typdefinition übersetzt, die den Typ `v64` umbenennt.

Listing 4.11: Alternative Übersetzung des beispielhaften Aufzählungstyps in SKilL

```
typedef Enum
@range(0,2)
!pragma enum
!pragma inPackage(Enums)
v64
```

Zusätzlich wird sie mit einer `range`-Restriktion annotiert, die die Anzahl der Enum-Konstanten im Aufzählungstyp widerspiegelt. Des Weiteren werden zwei `pragma`-Hinweise deklariert. Der erste `pragma`-Hinweis mit dem Argument `enum` gibt an, dass es sich bei dieser Typdefinition um einen Aufzählungstyp handelt. Der zweite `pragma`-Hinweis mit dem Argument `inPackage(Enums)` gibt an, dass der Aufzählungstyp im Ada-Paket `Enums` definiert ist.

4.5 `withSet`-Anweisung

Eine Klasse, die die `withSet`-Anweisung in der Klassendeklaration verwendet, soll im Ada-Binding die Funktionalität eines Sets bereitstellen. Das Ada-Binding stellt die Funktionalität eines Sets jedoch nur für Benutzertypen zur Verfügung, die als Typparameter eines Sets verwendet werden. Daher werden für diese Klassen flüchtige Felder in dem Benutzertyp namens `With_Sets` angelegt, der mit der `abstract`-Restriktion annotiert ist. Dabei wird für jede dieser Klassen der entsprechende Benutzertyp *T* als Typparameter eines Sets verwendet. Dieses Vorgehen wird in Listing 4.12 veranschaulicht.

Listing 4.12: Bereitstellung der Set-Funktionalität mithilfe von flüchtigen Feldern

```
@abstract
With_Sets {
  auto set<T> Dummy_T;
  ...
}
```

4.6 Deklarationen

Eine Deklaration ist entweder eine Klasse oder eine Schnittstelle, wobei eine Klasse wiederum in eine abstrake und konkrete Klasse unterteilt wird. Das Listing 4.13 veranschaulicht an einem Auszug einer beispielhaften IML-Spezifikation die drei verschiedenen Arten einer Deklaration. Das Listing 4.14 stellt die entsprechende Übersetzung nach SKilL dar.

Die Schnittstelle namens `I` enthält eine `for class`-Anweisung in der Schnittstellendeklaration und wird nach SKilL als Schnittstelle namens `I` übersetzt, die den abstrakten Benutzertyp `A` als Obertyp angibt und somit an den Benutzertyp `A` gebunden ist.

Die abstrakte Klasse namens `A` erbt von der Klasse `IML_Root` und wird nach SKilL als Benutzertyp namens `A` übersetzt, der vom Benutzertyp `IML_Root` erbt und mit einer `abstract`-Restriktion annotiert ist.

Die konkrete Klasse namens `B` erbt von der Klasse `A` und implementiert die Schnittstelle `I`. Außerdem enthält sie eine `with set`-Anweisung in der Klassendeklaration. Nach SKilL wird diese Klasse als Benutzertyp namens `B` übersetzt, der vom Benutzertyp `A` erbt und die Schnittstelle `I` implementiert. Außerdem wird er mit einem `pragma`-Hinweis, der das Schlüsselwort `withSet` als Argument enthält, annotiert.

Listing 4.13: Deklarationen in IML

```
interface I
for class A
is ... end interface I;

abstract class A
inherits IML_Root
is ... end class A;

concrete class B
with set
inherits A
implements I
is ... end class B;
```

Listing 4.14: Deklarationen in SKilL

```
interface I : A {}


@abstract
A : IML_Root {}

!pragma withSet
B : A with I {}
```

4.7 Felder

Ein Feld ist entweder ein semantisches oder syntaktisches Feld, wobei ein semantisches Feld, das eine Klasse als Typ verwendet, überschrieben werden kann. Das Listing 4.15 veranschaulicht an einem Auszug einer beispielhaften IML-Spezifikation die verschiedenen Arten eines Feldes. Das Listing 4.16 stellt die entsprechende Übersetzung nach SKilL dar.

Das semantische Feld namens `SLoc` vom *Builtin*-Typ `SLoc` wird nach SKilL als Feld namens `SLoc` vom Typ `SLoc` übersetzt.

Das semantische Feld namens `N` vom *Builtin*-Typ `Natural` wird nach SKilL als Feld namens `N` vom Typ `Natural` übersetzt.

Das semantische Feld namens `P` vom *Class*-Typ `A` wird nach SKilL als Feld namens `P` vom Typ `A` übersetzt.

Das darauffolgende semantische Feld `P` typisiert den *Class*-Typ von `A` nach `B` um. Nach SKilL wird dieses Feld als Sicht übersetzt.

Das semantische Feld namens `Q` vom *Class*-Typ `A` mit dem gesetzten Standardwert `null` und dem Schlüsselwort `omitted` wird nach SKilL als Feld namens `Q` vom Typ `A` übersetzt, wobei dieser mit der entsprechenden `default`-Restriktion und dem `onDemand`-Hinweis annotiert ist.

Das semantische Feld namens `R` ist eine Liste von Instanzen vom *Builtin*-Typ `Identifier`. Nach SKilL wird dieses Feld als Liste mit dem Typparameter `Identifier` übersetzt.

Das semantische Feld namens `S` ist ein Set von Instanzen vom *Class*-Typ `A`. Nach SKilL wird dieses Feld als Set mit dem Typparameter `A` übersetzt.

Das syntaktische Feld namens `T` vom *Class*-Typ `A` wird nach SKilL als Feld namens `T` vom Typ `A` übersetzt, wobei dieses mit dem `pragma`-Hinweis `syntactic` annotiert ist.

Das syntaktische Feld namens `U` ist eine Liste von Instanzen vom *Class*-Typ `A`. Nach SKilL wird dieses Feld als Liste mit dem Typparameter `A` übersetzt, wobei dieses mit dem `pragma`-Hinweis `syntactic` annotiert ist.

Listing 4.15: Felder in IML

```
semantic SLoc : builtin SLoc;
semantic N : builtin Natural;
semantic P : class A;

override semantic P : class B;


semantic Q : class A default null omitted;



semantic R : list of builtin Identifier;
semantic S : set of class A;

syntactic T : class A;


syntactic U : list of class A;
```

Listing 4.16: Felder in SKilL

```
SLoc SLoc;
Natural N;
A P;

view P as
B P;

@default(null)
!onDemand
A Q;

list<Identifier> R;
set<A> S;

!pragma syntactic
A T;

!pragma syntactic
list<A> U;
```

4.8 Modularer Aufbau

Da ein *Package* in mehrere *Chapter*s unterteilt wird und in jedem *Chapter* die dazugehörigen Deklarationen gesammelt werden, kann ein modularer Aufbau der SKilL-Spezifikation vorgenommen werden. Dabei werden für die *Builtin*s und für jedes *Chapter* eine eigene Datei generiert. Die generierten Dateien für die beispielhafte IML-Spezifikation aus Listing 4.2 sind unter anderem *iml.skill*, *builtins.skill*, *root.skill* und *loops.skill*. Die generierte Datei *iml.skill* inkludiert alle anderen generierten Dateien der Spezifikation (siehe Listing 4.17).

Listing 4.17: iml.skill

```
include "builtins.skill"
include "root.skill"
include "loops.skill"
```

5 Codegenerator (CodeGen)

In diesem Kapitel wird die Funktionsweise des zweiten Werkzeugs beschrieben. Das zweite Werkzeug soll eine SKilL-Spezifikation einlesen und daraus die API für den IML-Graphen generieren.

Um dies zu erreichen, wird ein Parser benötigt, der eine SKilL-Spezifikation in eine Intermediate Representation (IR) überführen kann. Im nächsten Schritt verwendet der Codegenerator (CodeGen) die IR, um daraus die Dateien zu generieren, die normalerweise mithilfe des Werkzeugs `imlgen` und der IML-Spezifikation generiert werden. Dieses Vorgehen wird in Abbildung 5.1 veranschaulicht. Dabei stand der Parser zum Einlesen von SKilL-Spezifikationen und die dazugehörige IR zu Beginn dieser Arbeit zur Verfügung.

Der Parser wurde in der Programmiersprache Scala und die IR in der Programmiersprache Java entwickelt. Die IR wurde im Rahmen dieser Arbeit durch weitere Funktionalitäten erweitert. Die Softwarearchitektur des CodeGens basiert auf dem SKilL-Codegenerator für das Ada-Binding, der in der Programmiersprache Scala entwickelt wurde. Aus diesem Grund wurde auch der CodeGen in der Programmiersprache Scala entwickelt, wie bereits der SpecGen und der dazugehörige Parser zuvor. Im Vergleich zum SpecGen beanspruchte der CodeGen die vorwiegende Entwicklungszeit.

5.1 Vorgehensweise

Als Erstes wurde ein GNAT-Projekt mit dem Namen `Siml` für das generierte Ada-Binding aus der generierten SKilL-Spezifikation erstellt und damit die Bibliothek `libIML` erweitert. Dabei wurde die Erweiterung als Abhängigkeit gelöst, sodass das generierte Ada-Binding nur bei Änderungen an der IML-Spezifikation nochmals kompiliert werden muss.

Nachdem die Bibliothek `libIML` erfolgreich mit der Bibliothek `Siml` als Abhängigkeit kompiliert werden konnte, wurde im nächsten Schritt der CodeGen entwickelt, der aus der SKilL-Spezifikation eine sehr minimale Version der Dateien generieren konnte. Die minimale Version enthielt nur die Typen für die jeweiligen Benutzertypen ohne weitere Methoden, wobei die bisherigen IML-Typen durch ihre jeweiligen SKilL-Typen ersetzt wurden. Dies wurde in Form von Untertypen gelöst, sodass

Abbildung 5.1: Zuerst liest der Parser eine SKilL-Spezifikation ein und erstellt daraus eine IR. Danach generiert der CodeGen aus der IR die generierten Dateien. Ein grüner Pfeil oder Rand markiert eine vorhandene Komponente

Listing 5.1: Minimale Version eines generierten Benutzertyps *T*

```
package Ts is
   subtype T_Class is Siml.T_T; -- im Ada-Binding: type T_T is new ... with private;
   subtype T is Siml.T_Dyn; -- im Ada-Binding: type T_Dyn is access all T_T'Class;
   No_T : constant T := null;
end Ts;
```

hierfür kein neuer Typ gebildet wird. Das Listing 5.1 stellt die minimale Version einer generierten Datei für den Benutzertyp *T* dar. Der Typ *T* ist ein Untertyp von `Siml.T_Dyn`, der wiederum ein klassenweiter Zeigertyp von `Siml.T_T` ist. Der Typ `T_Class` ist ein Untertyp von `Siml.T_T`.

Von diesem Zeitpunkt an wurde versucht, einen kompilierfähigen Zustand herzustellen. Dabei wurden folgende drei Schritte gleichzeitig durchgeführt.

- Die IML-Spezifikation wurde regelmäßig gekürzt oder wieder erweitert, sodass die handgeschriebenen Dateien und das Werkzeug `cafe++` kompilierbar blieben. Durch die um teilweise 25 % gekürzte IML-Spezifikation konnte die Kompilierzeit entsprechend gekürzt werden.
- Der CodeGen wurde regelmäßig mithilfe der gegebenen Dokumentation des Werkzeugs `imlgen` [Cze06] aus dem Jahr 2006 weiterentwickelt. Da die gegebene Dokumentation nur die öffentliche API beschrieb, wurden auch die vom Werkzeug `imlgen` generierten Dateien hinzugezogen und analysiert.
- Wann immer nötig, wurden die handgeschriebenen Dateien angepasst oder sogar entfernt.

Der CodeGen wurde somit anhand der jeweils auftretenden Kompilierfehler weiterentwickelt, indem diese meist durch das Hinzufügen von Stubs in den generierten Dateien sowie durch das Anpassen oder Entfernen der handgeschriebenen Dateien behoben wurden. Nachdem die aufgetretenen Kompilierfehler eines Kompiliervorgangs beseitigt wurden, konnte ein neuer Kompiliervorgang gestartet werden. Eine auf diese Weise betriebene Entwicklung bietet keine Fortschrittsanzeige. Ein Scheitern des Gesamtprojekts aufgrund von Kompilierfehlern, die ein Hinderungsgrund für die erfolgreiche Überführung in den Sollzustand werden könnte, wurde in Kauf genommen. Nachdem ein kompilierfähiger Zustand hergestellt werden konnte, mussten die verwendeten Stubs in den generierten Dateien nur noch ausimplementiert werden.

5.2 Generierte Dateien

Für jeden Benutzertyp *T* werden durch den CodeGen folgende Dateien generiert.

***Ts*.ads/.adb** In dem Ada-Paket *Ts* werden für den Benutzertyp *T* die Untertypen wie in Listing 5.1 veranschaulicht definiert. Weiterhin werden die Zugriffsmethoden auf die Felder des Benutzertyps *T* bereitgestellt. Falls eine Schnittstelle an den Benutzertyp *T* oder an dessen Obertypen gebunden ist, so wird außerdem eine `Implements`-Funktion für diese Schnittstelle bereitgestellt. Ist der Benutzertyp *T* nicht mit einer `abstract`-Restriktion annotiert, so wird

auch eine Instanziierungsmethode bereitgestellt. Des Weiteren werden einige Methoden für die Programmiersprache C exportiert.

***T*_internals.ads/.adb** In dem Ada-Paket *T*_Internals werden für den Benutzertyp *T* die Untertypen nochmals wie in Listing 5.1 veranschaulicht definiert. Des Weiteren werden Methoden bereitgestellt, die zur Laufzeit Informationen über den Benutzertyp *T* für die Reflexion zurückgeben.

***T*_internals-lists.ads/.adb** Das Ada-Paket *T*_Internals.Lists wird nur generiert, wenn der Benutzertyp *T* in einem Feld als Typparameter einer Liste verwendet wird. Bei diesem Ada-Paket handelt es sich um einen Wrapper für die entsprechende Liste aus dem Ada-Binding. Der Wrapper stellt verschiedene Zugriffsmethoden zur Manipulation der Liste und einen Iterator zur Verfügung.

***T*_internals-sets.ads/.adb** Das Ada-Paket *T*_Internals.Sets wird nur generiert, wenn der Benutzertyp *T* in einem Feld als Typparameter eines Sets verwendet wird oder der Benutzertyp *T* mit dem pragma-Hinweis withSet annotiert ist. Bei diesem Ada-Paket handelt es sich um einen Wrapper für das entsprechende Set aus dem Ada-Binding. Der Wrapper stellt verschiedene Zugriffsmethoden zur Manipulation des Sets und einen Iterator zur Verfügung.

Außerdem werden folgende Dateien unabhängig von einem Benutzertyp generiert.

iml_classes.ads/.adb Das Ada-Paket IML_Classes hat Abhängigkeiten zu allen generierten Ada-Paketen. Außerdem wird eine Funktion bereitgestellt, die ein Array mit Informationen über alle Benutzertypen für die Reflexion zurückgibt.

iml_concrete_classes.ads Das Ada-Paket IML_Concrete_Classes stellt eine Konstante vom Typ Natural mit der Anzahl der konkreten Benutzertypen als Wert zur Verfügung. Weiterhin definiert es einen Arraytyp mit einer von dieser Konstanten vorgegebenen Länge und dem Typparameter Storables.Sets.Set für das Ada-Paket IML_Graphs.

explicit_dispatch-get_class_id.ads/.adb Dieses Ada-Paket stellt die dispatchende Funktion Get_Class_ID für das Ada-Paket IML_Roots bereit. Sie gibt für eine gegebene Instanz vom Typ IML_Root die Informationen über den entsprechenden Benutzertyp für die Reflexion zurück.

explicit_dispatch-internal_accept_visitor.ads/.adb Dieses Ada-Paket stellt die dispatchende Prozedur Internal_Accept_Visitor für das Ada-Paket IML_Roots bereit. Sie implementiert die Empfangsmethode, die aus dem Entwurfsmuster Besucher bekannt ist.

explicit_dispatch-mark_node.ads/.adb Dieses Ada-Paket stellt die dispatchende Prozedur Mark_Node für das handgeschriebene Ada-Paket Skill_Mark_Helper bereit. Sie ruft für eine gegebene Instanz vom Typ Storable die entsprechende Markierungsmethode auf.

explicit_dispatch-syntactic_children.ads/.adb Dieses Ada-Paket stellt die dispatchende Funktion Syntactic_Children für das Ada-Paket IML_Roots bereit. Sie gibt für eine gegebene Instanz vom Typ IML_Root ein Array von Instanzen zurück, die aus den entsprechenden syntaktischen Feldern eingesammelt wurden.

5.3 Erweiterung der Intermediate Representation (IR)

Die folgenden vier Erweiterungen sind während der Entwicklung des CodeGens in die IR eingeflossen.

Die erste Erweiterung ist die Implementierung der `default`-Restriktionen für die Felder. Sie wird benötigt, um bei den Parametern einer Instanziierungsmethode die mitgelieferten Standardwerte setzen zu können.

Die zweite Erweiterung ist die Abfragemöglichkeit nach einer vorhandenen `abstract`-Restriktion für einen gegebenen Benutzertyp. Sie wird benötigt, um zwischen einem abstrakten und einem konkreten Benutzertyp unterscheiden zu können.

Die dritte Erweiterung ist die Abfragemöglichkeit nach `pragma`-Hinweisen für ein gegebenes Feld. Sie wird benötigt, um unter anderem zwischen einem semantischen und einem syntaktischen Feld unterscheiden zu können.

Die IR sortiert die Benutzertypen und deren Felder nach dem Alphabet. Die vierte Erweiterung ist daher die temporäre Entfernung der Sortierung, sodass die Benutzertypen und Felder in der Reihenfolge ihrer Spezifizierung zurückgegeben werden. Diese Änderung ist insbesondere für Unterkapitel 5.4 wichtig.

5.4 Reihenfolge der Felder

Das Werkzeug `imlgen` generiert die Felder in den generierten Dateien in der Reihenfolge, wie sie in der IML-Spezifikation definiert werden. Das ist insbesondere für die Instanziierungsmethode eines konkreten Benutzertyps wichtig, weil ihre Parameter nach der Reihenfolge ihrer Spezifizierung sortiert sind. Da die Instanziierungsmethode unter anderem für die Programmiersprache `C` exportiert wird, kann dieses Problem auch nicht mithilfe von benannten Parametern umgangen werden, wie sie in der Programmiersprache Ada angeboten werden.

Des Weiteren konnte die von der IR bereitgestellte `getAllFields`-Funktion für den Benutzertyp *`T`* nicht verwendet werden, weil seine Felder in der falschen Reihenfolge zurückgegeben werden. Das Werkzeug `imlgen` sammelt die Felder mithilfe der Tiefensuche ein, wobei zuerst immer der Obertyp und dann die Schnittstellen nach der Reihenfolge ihrer Spezifizierung besucht werden. In Abbildung 5.2 wird diese Vorgehensweise an einem komplexeren Beispiel veranschaulicht.

Die alternative Implementierung der `getAllFields`-Funktion wird in Listing 5.2 veranschaulicht. Dabei wird keine Tiefensuche eingesetzt, sondern durch Rekursion eine Liste von Benutzertypen und Schnittstellen aufgebaut. Für das komplexere Beispiel in Abbildung 5.2 veranschaulicht die Abbildung 5.3 den Inhalt der Liste bei Aufruf der `getAllFields`-Funktion mit dem Benutzertyp *`T`* als Argument, kurz bevor die Benutzertypen und die Schnittstellen mithilfe der `map`-Funktion durch ihre jeweiligen Felder ersetzt werden.

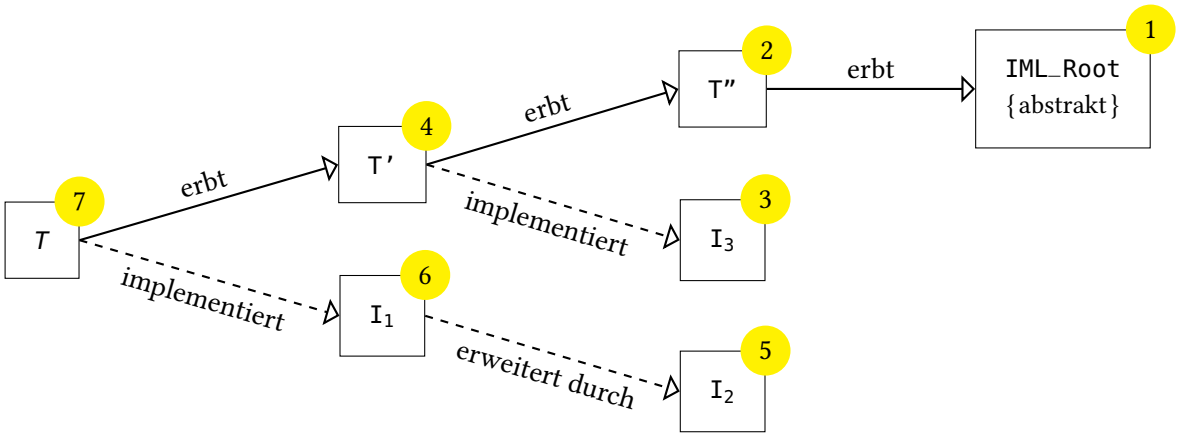


Abbildung 5.2: Die angepasste Tiefensuche an einem komplexeren Beispiel. Die Zahlen in den gelben Punkten geben die Besuchsreihenfolge der Deklarationen an. Ein durchgezogener Pfeil gibt eine Vererbung an. Ein gestrichelter Pfeil gibt die Implementierung einer Schnittstelle an

IML_Root	T''	I_3	T'	I_2	I_1	*T*

Abbildung 5.3: Der Inhalt der Liste mit den Benutzertypen und den Schnittstellen bei Aufruf der `getSuperDeclarations`-Funktion mit dem Benutzertyp *T* aus Abbildung 5.2 als Argument

Die `getSuperDeclarations`-Funktion baut für einen gegebenen Benutzertyp *T* eine Liste aus Benutzertypen und Schnittstellen auf. Anschließend werden die Benutzertypen und die Schnittstellen mithilfe der `map`-Funktion durch ihre jeweiligen Felder ersetzt.

Listing 5.2: Alternative Implementierung der `getAllFields`-Funktion

```
override protected final def getAllFields(t : UserType) : List[Field] = {
  def getSuperDeclarations(d : Declaration) : List[Declaration] = d match {
    case t : InterfaceType =>
      t.getSuperInterfaces.map(getSuperDeclarations(_)).flatten.toList ++ List(t)
    case t : UserType =>
      List(t.getSuperType).filterNot(null == _).map(getSuperDeclarations(_)).flatten.toList ++
        t.getSuperInterfaces.map(getSuperDeclarations(_)).flatten.toList ++ List(t)
  }

  (getSuperDeclarations(t)
    .map(_ match {
      case t : InterfaceType => t.getFields
      case t : UserType => t.getFields
    })
  ).flatten
}
```

Die Korrektheit der Reihenfolge der Felder wurde durch einen automatisierten Vergleich bestätigt, indem die zwei Instanziierungsmethoden der konkreten Benutzertypen in den durch die Werkzeuge `imlgen` und CodeGen generierten Dateien auf Übereinstimmung geprüft wurde.

5.5 Explizites Dispatching

Im Ada-Paket `IML_Roots` werden mehrere Methoden definiert, die einem dynamischen Dispatching unterliegen. In der Programmiersprache Ada müssen die zu dispatchenden Methoden in dem Ada-Paket definiert werden, wo auch der dazugehörige Typ definiert wird. In diesem Fall müssten die zu dispatchenden Methoden in das Ada-Binding integriert werden, das mithilfe des SKilL-Codegenerators generiert wird. Um ein Zusammenlegen der zwei Bibliotheken zu verhindern, wird im Ada-Paket `IML_Roots` das dynamische Dispatching durch das explizite Dispatching mithilfe einer Hashmap ersetzt.

Alternativ kann das Ada-Binding das Entwurfsmuster Besucher implementieren.[4] Dadurch können die zu dispatchenden Methoden in externe Besucherklassen ausgelagert werden.

In Listing 5.3 wird das explizite Dispatching anhand der dispatchenden `Do_Something`-Prozedur veranschaulicht. Für jede dynamisch dispatchende Methode wird ein eigenes Unterpaket vom Hauptpaket `Explicit_Dispatch` erstellt. Für die `Do_Something`-Prozedur wird somit das Unterpaket `Do_Something` erstellt. Für jeden Untertyp des Benutzertyps `IML_Root` wird nun in dem Ada-Paket `Do_Something` die passende Prozedur generiert. Das Ada-Binding stellt für jeden Benutzertyp die dispatchende `Skill_Name`-Funktion bereit, mit der eine Instanz zur Laufzeit den Namen ihres Benutzertyps abfragen kann. Mithilfe der `Skill_Name`-Funktion und dem Zeigertyp auf die generierten Prozeduren wird die Hashmap aufgebaut. Beim Aufruf der `Do_Something`-Prozedur wird geprüft, ob für den Benutzertyp der übergebenen Instanz ein Zeiger auf eine generierte Prozedur existiert. Ist das der Fall, wird die Prozedur mit der Instanz als Argument aufgerufen. Ansonsten wird die Ausnahme `Program_Error` ausgelöst. Anschließend muss im Ada-Paket `IML_Roots` die `Do_Something`-Prozedur nur noch umgeleitet beziehungsweise umbenannt werden. Dies wird in Listing 5.4 veranschaulicht.

Listing 5.3: Explizites Dispatching für die `Do_Something`-Prozedur

```
package body Explicit_Dispatch.Do_Something is
  procedure Do_Something_A
    (Node : access IML_Roots.IML_Root_Class'Class) is
  begin
    Node.As_A...
  end Do_Something_A;
  ...

  type Procedure_Access is access procedure
    (Node : access IML_Roots.IML_Root_Class'Class);

  package HM is new Ada.Containers.Hashed_Maps (Skill.Types.String_Access, Procedure_Access);
```

[4] Dies setzt jedoch voraus, dass die Wartung durch den *Maintainer* des SKilL-Codegenerators für das Ada-Binding übernommen wird, sodass ein Fork vermieden werden kann.

```
  HMi : HM.Map;

  function Initialize return Boolean is
  begin
    HMi.Insert (Siml.Internal_Skill_Names.A_Skill_Name, Do_Something_A'Access);
    ...
    return True;
  end Initialize;

  Unused_Initialize : Boolean := Initialize;
  pragma Unreferenced (Unused_Initialize);

  procedure Do_Something
    (Node : access IML_Roots.IML_Root_Class'Class) is
  begin
    if HMi.Contains (Node.Skill_Name) then
      return HMi.Element (Node.Skill_Name) (Node);
    else
      raise Program_Error;
    end if;
  end Do_Something;
end Explicit_Dispatch.Do_Something;
```

Listing 5.4: Umbenennung der `Do_Something`-Prozedur im Ada-Paket `IML_Roots`

```
package IML_Roots is
  ...
  procedure Do_Something
    (Node : access IML_Roots.IML_Root_Class'Class)
    renames Explicit_Dispatch.Do_Something.Do_Something;
   ...
end IML_Roots;
```

5.6 Schnittstellen

Die Schnittstellen werden in der IML auf besondere Weise gelöst. Dabei erlaubt die Beschreibungssprache der IML-Spezifikation, dass eine Schnittstelle einen abstrakten Benutzertyp als Obertyp angeben kann. Die Bedeutung davon ist, dass die Schnittstelle an den angegebenen Benutzertyp gebunden ist und dessen Untertypen die Schnittstelle implementieren können. In Listing 5.5 und in der dazugehörigen Abbildung 5.4 wird dieses Vorgehen an einem Beispiel veranschaulicht.

Listing 5.5: Schnittstellen in SKilL

```
@abstract T {}
interface I : T { Natural n; }
A : T with I {}
B : T with I {}
```

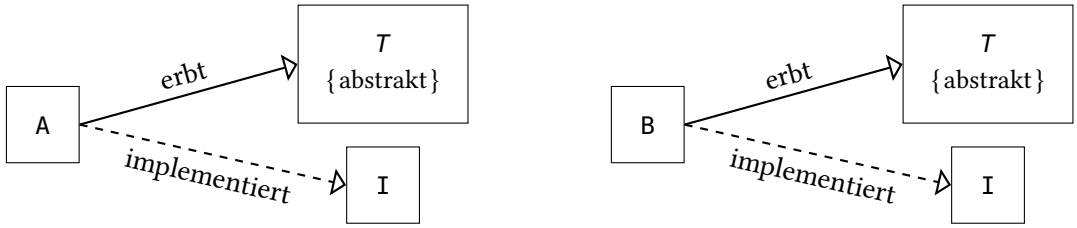


Abbildung 5.4: Die Benutzertypen `A` und `B` erben vom abstrakten Benutzertyp *T* und implementieren die Schnittstelle `I`

Es wird ein abstrakter Benutzertyp namens *T* ohne weitere Felder definiert. Die Schnittstelle namens `I` gibt den Benutzertyp *T* als Obertyp an und definiert ein Feld namens `n` vom Typ `Natural`. Die Benutzertypen `A` und `B` erben den Benutzertyp *T* und implementieren die Schnittstelle `I`.

In Listing 5.6 wird die generierte Implementierung des Ada-Pakets für den Benutzertyp *T* veranschaulicht. Dabei wird für jede Schnittstelle, die als Obertyp den Benutzertyp *T* angegeben hat, eine `Implements`-Funktion generiert. Mit dieser Funktion kann überprüft werden, ob eine Instanz eines Untertyps vom Benutzertyp *T* die entsprechende Schnittstelle implementiert. Sollte dies der Fall sein, kann die Instanz auf die implementierten Felder der überprüften Schnittstelle zugreifen, ohne der Gefahr einer Ausnahme ausgesetzt zu sein. Die Zugriffsmethoden für die Felder werden mithilfe des expliziten Dispatchings und einer Typumwandlung gelöst, weil jeder Untertyp vom Benutzertyp *T* die Schnittstelle eigenständig implementiert (siehe Abbildung 5.4 und Listing 5.6).

Listing 5.6: Implementierung des Ada-Pakets *T*

```
package body T is
  ...
  function Implements_I
   (Node : access T_Class'Class)
    return Boolean is
  begin
    if Node.all in Siml.A_T'Class | Siml.B_T'Class then
      return True;
    end if;
    return False;
  end Implements_I;

  function Get_N
   (Node : access T_Class'Class)
    return Natural is
  begin
    if Node.all in Siml.A_T'Class then
      return Natural (Node.As_A.Get_N);
    end if;
    if Node.all in Siml.B_T'Class then
      return Natural (Node.As_B.Get_N);
    end if;
    raise IML_Roots.Interface_Not_Implemented;
  end Get_N;
```

```
  procedure Set_N
    (Node : access T_Class'Class;
     New_Value : in Natural) is
  begin
    if Node.all in Siml.A_T'Class then
      Node.As_A.Set_N (Skill.Types.V64 (New_Value));
    end if;
    if Node.all in Siml.B_T'Class then
      Node.As_B.Set_N (Skill.Types.V64 (New_Value));
    end if;
    raise IML_Roots.Interface_Not_Implemented;
  end Set_N;
  ...
end package T;
```

5.7 Listen und Sets

Jeder Benutzertyp *T*, der in der SKilL-Spezifikation als Typparameter einer Liste oder eines Sets verwendet wird, muss das entsprechende Ada-Paket `T_Internals.Lists` beziehungsweise `T_Internals.Sets` bereitstellen. Im Ada-Paket *T* werden sie nochmals als Umbenennung nach `T_Internals` definiert. In der originalen IML-Implementierung wird die Liste aus dem generischen Ada-Paket `Generic_IML_List_Iterators` instanziiert. Ein Set wird aus dem generischen Ada-Paket `Storables.Sets.Wrapper` instanziiert, wobei das Ada-Paket `Storables.Sets` eine Instanz vom generischen Ada-Paket `IML_Sets.Wrapper` ist. Die Listen und Sets müssen nun durch die entsprechenden Ada-Pakete aus dem Ada-Binding ersetzt werden, sodass ihre Elemente vom SKilL-Zustand erfasst werden können. Das Ada-Binding generiert hierfür bei einer Liste das Ada-Paket `Siml.Skill_Array_T` und bei einem Set das Ada-Paket `Siml.Skill_Set_T`. Ein zwischengeschalteter Wrapper korrigiert die erwarteten Namen der Typen und implementiert die erwarteten Methoden einer Liste oder eines Sets. In Listing 5.7 wird dies an dem Wrapper für die Liste vom Benutzertyp *T* veranschaulicht. Der Wrapper für das Set vom Benutzertyp *T* ist analog implementiert.

Listing 5.7: Wrapper-Paket für die Liste vom Benutzertyp *T*

```
package T_Internals.Lists is
  package L renames Siml.Skill_Array_T;
  subtype List is L.Ref;
  subtype Item_Type is T;
  type Iterator_T is private;
  type Iterator is access all Iterator_T;
  ...
end T_Internals.Lists;
```

Sowohl der Wrapper einer Liste als auch eines Sets reimplementieren die Iteratoren. In der Dokumentation des Werkzeugs `imlgen` [Cze06] sind die Methoden angegeben, die der Iterator einer Liste bereitstellen muss. Die zu bereitstellenden Methoden für ein Set sind jedoch undokumentiert. Im Folgenden werden die Methoden nach dem Schema in [Cze06] angegeben, die ein Set für den Iterator sowie für das Set selbst bereitstellen muss.

- function `Make_Iterator` (in `Set`) return `Iterator`
- procedure `Next` (in out `Iterator`; out *item*)
- procedure `Next` (in out `Iterator`)
- function `Current` (in `Iterator`) return *item*
- function `More` (in `Iterator`) return `Boolean`
- procedure `Destroy` (in out `Iterator`)
- function `Empty_Set` return `Set`
- function `Is_Empty` (in `Set`) return `Boolean`
- procedure `Insert` (in out `Set`; in *item*)
- function `Is_Member` (in `Set`; in *item*) return `Boolean`
- procedure `Diff` (in out `Set`; in `Set`)
- procedure `Remove` (in out `Set`; in *item*)
- procedure `Remove_All` (in out `Set`)
- procedure `Destroy` (in out `Set`)

5.8 Überschriebenes semantisches Feld

Ein Feld, das vom Typ ein Benutzertyp ist, darf überschrieben werden (siehe Abbildung 4.4). Dazu werden Sichten verwendet, die es erlauben, ein Feld umzubenennen oder umzutypisieren. In Listing 5.8 wird eine typische Situation aus der SKilL-Spezifikation, bei der ein Feld umtypisiert wird, veranschaulicht.

Listing 5.8: Typische Umtypisierung in der SKilL-Spezifikation

```
A { A f; }
B : A {
  view f as
  B f;
}
```

Der Benutzertyp namens `A` enthält ein Feld namens `f` vom Typ `A`. Der Benutzertyp namens `B` erbt vom Benutzertyp `A` und deklariert eine Sicht für das Feld namens `f`, die es vom Benutzertyp `A` zum Benutzertyp `B` umtypisiert.

Der CodeGen generiert aus der Spezifikation in Listing 5.8 die Zugriffsmethoden in Listing 5.9. Die im Ada-Paket `Siml` definierte `Get_F`-Funktion gibt eine Instanz vom Benutzertyp `A` zurück. Da jedoch bekannt ist, dass die zurückgegebene Instanz nur vom Benutzertyp `B` sein kann, wird die Instanz mithilfe der `As_B`-Funktion explizit zum Benutzertyp `B` umgewandelt. Die zusätzliche Typumwandlung in der Zugriffsfunktion ist der einzige Unterschied zu einem Feld, das keine Sicht deklariert hat. Des

Weiteren stellt das Ada-Paket `Siml` die `Set_F`-Prozedur jeweils für den Benutzertyp `A` und den Benutzertyp `B` bereit. Daher reicht hierfür eine Typumwandlung vom klassenweiten Zeigertyp zum spezifischen Typ, die auch bei den Feldern ohne eine deklarierte Sicht stattfindet. Daher müssen an der `Set_F`-Prozedur keine Sicht-spezifischen Änderungen vorgenommen werden.

Listing 5.9: Generierte Zugriffsmethoden für das überschriebene Feld

```
package body B is
   ...
   function Get_F
     (Node : access B_Class'Class)
      return B_Internals.B is
   begin
      return B_Internals.B (Node.Get_F.As_B);
   end Get_F;

   procedure Set_F
     (Node : access B_Class'Class;
      New_Value : in B_Internals.B) is
   begin
      Node.Set_F (Siml.B (New_Value));
   end Set_F;
   ...
end B;
```

5.9 Syntaktisches Feld

Ein Feld, dass mit dem `pragma`-Hinweis `syntactic` annotiert ist, darf vom Typ nur ein Benutzertyp oder eine Liste mit dem Typparameter eines Benutzertyps sein (siehe Abbildung 4.4). Das Besondere an den syntaktischen Feldern ist, dass sie einen Baum aufspannen. Ein syntaktisches Feld repräsentiert hierbei einen Knoten in einem syntaktischen Baum. In Listing 5.10 wird eine typische Situation aus der SKilL-Spezifikation, bei der ein Feld ein syntaktisches Feld ist, veranschaulicht.

Listing 5.10: Typisches syntaktisches Feld in der SKilL-Spezifikation

```
IML_Root { IML_Root Parent; }
A : IML_Root {}
B : IML_Root {
  !pragma syntactic
  A f;
}
```

Der Benutzertyp namens `IML_Root` enthält ein Feld namens `Parent` vom Typ `IML_Root`. Der Benutzertyp namens `A` erbt vom Benutzertyp `IML_Root`. Der Benutzertyp namens `B` erbt vom Benutzertyp `IML_Root` und enthält ein Feld namens `f` vom Typ `A`, das mit dem `pragma`-Hinweis `syntactic` annotiert ist.

Der CodeGen generiert aus der Spezifikation in Listing 5.10 die Zugriffsprozedur `Set_F` in Listing 5.11. Die Zugriffsfunktion `Get_F` ist dabei uninteressant, weil diese den Aufbau des syntaktischen Baums

nicht beeinflusst. Bei einem syntaktischen Feld wird zusätzlich die `Safe_Set_Parent`-Prozedur im Ada-Paket `IML_Roots` aufgerufen. Die `Safe_Set_Parent`-Prozedur akzeptiert zwei Parameter als Eingabe. Beide Parameter sind jeweils eine Instanz vom Typ `IML_Root`, wobei der Erste als Kind und der Zweite als Mutter fungiert. Dort wird das Kind einer Nullzeigerprüfung unterzogen. Handelt es sich beim Kind nicht um einen Nullzeiger, so wird beim Kind die `Set_Parent`-Prozedur mit der Mutter als Argument aufgerufen. Bei einer Liste wird die `Safe_Set_Parent`-Prozedur für jede Instanz aus der Liste aufgerufen.

Listing 5.11: Generierte Zugriffsprozedur für das syntaktische Feld

```
package body B is
  ...
  procedure Set_F
    (Node : access B_Class'Class;
     New_Value : in A_Internals.A) is
  begin
    Node.Set_F (Siml.A (New_Value));
    IML_Roots.Safe_Set_Parent -- Nullzeigerprüfung + Child.Set_Parent (Parent);
      (IML_Root_Internals.IML_Root (New_Value), -- Kind
       IML_Root_Internals.IML_Root (Node)); -- Mutter
  end Set_F;
  ...
end B;
```

5.10 Reflexion

Für jeden Benutzertyp *T* werden im Ada-Paket *T*`_Internals` Methoden für die Reflexion bereitgestellt. Die Konstante namens `Class_Data` vom Typ `IML_Reflection.Class_Ptr` beinhaltet alle benötigten Informationen über den Benutzertyp *T*. Das sind unter anderem der Name, ob es sich hierbei um einen abstrakten oder konkreten Benutzertyp handelt, die Anzahl der Felder, ein Zeiger auf die Informationen des Obertyps, eine Liste von Zeigern auf die Informationen der Felder, eine Liste von Zeigern auf die Informationen der Untertypen und ein Zeiger auf die `Get_All_Nodes`-Prozedur, die alle Instanzen vom Benutzertyp *T* zurückgibt. Auf diese Konstante kann mithilfe der `Get_Class_ID`-Funktion zugegriffen werden.

Außerdem wird für jedes Feld *F* vom Benutzertyp *T* die Konstante namens *F*`_Field_Data` bereitgestellt, die alle benötigten Informationen über das Feld *F* beinhaltet. Das sind unter anderem der Name, die Zeiger auf die Zugriffsmethoden, ein Zeiger auf die Informationen des Typs sowie ein Wahrheitswert, ob es sich hierbei um ein syntaktisches Feld handelt. Auf diese Konstante kann mithilfe der *F*`_Field`-Funktion zugegriffen werden.

Weiterhin wird die `Initialize_And_Get_Class_Ptr`-Funktion bereitgestellt, die vom Ada-Paket `IML_Classes` aufgerufen wird. Sie initialisiert die Konstante `Class_Data`, indem der Zeiger auf die Informationen des entsprechenden Obertyps gesetzt und die Liste von Zeigern auf die Informationen der Felder entsprechend befüllt werden. Anschließend wird die Konstante zurückgegeben.

6 Handgeschriebene Dateien

In diesem Kapitel werden die signifikanten Änderungen an den handgeschriebenen Dateien beschrieben. Wann immer nötig wurden die handgeschriebenen Dateien während der Entwicklung des CodeGens angepasst. Dabei kam es auch vor, dass handgeschriebene Dateien entfernt wurden, weil sie entweder im ganzen Projekt Bauhaus nicht verwendet werden oder Teil eines Werkzeugs sind und daher verschoben werden sollten. Die Verschiebung kann entweder direkt zum Werkzeug oder durch eine Auslagerung in eine neue Bibliothek erfolgen, die dann vom jeweiligen Werkzeug als Abhängigkeit eingebunden wird.

6.1 SKilL-Zustand

Für alle konkreten Benutzertypen in den generierten Dateien existieren Instanziierungsmethoden, mit deren Hilfe Instanzen vom jeweiligen Benutzertyp erstellt werden können. Damit diese Instanzen auch ihren Weg in den SKilL-Zustand finden, muss dieser für die konkreten Benutzertypen zur Verfügung stehen.

Der Typ namens `IML_Graph_Class` im Ada-Paket `IML_Graphs` ist dafür zuständig, den aktuellen Zustand eines IML-Graphen zu verwalten. Das Ada-Paket `IML_Graphs` wird zudem in allen generierten Dateien als Abhängigkeit eingebunden. Indem dieser Typ nun um das Feld für den SKilL-Zustand erweitert wird, kann der SKilL-Zustand in den generierten Dateien von allen Benutzertypen aus erreicht werden. Die Erweiterung wird in Listing 6.1 veranschaulicht.

Listing 6.1: SKilL-Zustand als neues Feld in `IML_Graph_Class`

```
package IML_Graphs is
  ...
  type IML_Graph_Class is new ... with record
    ...
    Root_Node : IML_Roots.IML_Root;
    Sf : Siml.Api.File;
  end record;
  type IML_Graph is access all IML_Graph_Class'Class;
  ...
end IML_Graphs;
```

Das Feld namens `Root_Node` vom Typ `IML_Root` zeigt auf den Wurzelknoten des Graphen und war bereits vor der Erweiterung vorhanden. Dieser Wurzelknoten ist der Startknoten, um durch den Graphen zu iterieren. Das Feld namens `Sf` vom Typ `Siml.Api.File` zeigt auf den SKilL-Zustand

des entsprechenden IML-Graphen. Der Typ `IML_Graph` ist ein klassenweiter Zeigertyp für den Typ `IML_Graph_Class`.

Damit der Wurzelknoten nicht verloren geht, muss dieser auch im SKilL-Zustand hinterlegt werden. Dazu erstellt der SpecGen bei der Übersetzung einer IML-Spezifikation einen weiteren Benutzertyp namens `IML_Graph` mit einem Feld namens `Root_Node` vom Typ `IML_Root`. Dieser Benutzertyp ist mit der *singleton*-Restriktion annotiert, sodass in einem SKilL-Zustand nur eine Instanz dieses Benutzertyps existieren darf.

Listing 6.2: Benutzertyp `IML-Graph` als Singleton

```
@singleton
IML_Graph { IML_Root Root_Node; }
```

Aus diesem Grund muss beim Lesen oder Schreiben einer SKilL-Binärdatei darauf geachtet werden, dass der Wurzelknoten entsprechend aus dem SKilL-Zustand gesetzt oder zuvor in den SKilL-Zustand übernommen wird.

6.2 Erstellen einer neuen Graphinstanz

Das Ada-Paket `IML_Graphs` stellt eine `Make`-Funktion zur Verfügung, um eine neue leere Graphinstanz zu erstellen. Die Implementierung dieser Funktion wird nun so erweitert, dass zusätzlich ein neuer leerer SKilL-Zustand für die neue Graphinstanz erzeugt wird. Die Erweiterung der Implementierung wird in Listing 6.3 veranschaulicht.

Listing 6.3: Erstellen einer neuen Graphinstanz

```
package body IML_Graphs is
   ...
   function Make
    (New_Root : IML_Roots.IML_Root)
     return IML_Graph is
     Graph : IML_Graph;
   begin
     Graph := new IML_Graph_Class;
     Graph.Root_Node := New_Root;
     Graph.Sf := Siml.Api.Open ("a.out.sf", Skill.Files.Create, Skill.Files.Write);
     ...
     return Graph;
   end Make;
   ...
end IML_Graphs;
```

Die `Make`-Funktion akzeptiert einen Parameter als Eingabe. Dieser Parameter namens `New_Root` vom Typ `IML_Root` ist der Wurzelknoten des neuen Graphen. Als Nächstes wird eine neue Instanz vom Typ `IML_Graph_Class` erstellt. Das Feld `Root_Node` zeigt auf den neuen Wurzelknoten des Graphen. Das Feld `Sf` zeigt auf den neu erstellten SKilL-Zustand, bei dem der Dateiname aufgrund einer fehlenden Alternative bereits vordefiniert ist. Anschließend wird die neue Graphinstanz zurückgegeben.

6.3 Markieren einer Graphinstanz

Das Ada-Paket `Storables` stellt eine `Mark_Graph`-Prozedur zur Verfügung, um alle erreichbaren Knoten im IML-Graphen zu markieren. Dabei wird für eine Instanz vom Typ `Storable` mithilfe des dynamischen Dispatchings die entsprechende `Mark_Node`-Prozedur aufgerufen. Die `Mark_Node`-Prozedur markiert alle Felder vom Typ `Storable` als erreichbar und ruft daraufhin deren `Mark_Node`-Prozedur auf. Da ein dynamisches Dispatching nicht möglich ist, wird die Implementierung so angepasst, dass das explizite Dispatching verwendet wird. Dies erfolgt jedoch mit der Einschränkung, dass für die `Mark_Graph`-Prozedur als Eingabe nur eine Instanz vom Typ `IML_Graph` verwendet werden darf. Aufgrund der Typumwandlung und des notwendigen Aufrufs der `Mark_Node`-Prozedur für die Graphinstanz muss das Ada-Paket `Storables` eine Abhängigkeit mit dem Ada-Paket `IML_Graphs` eingehen. Dadurch entsteht in der *Elaboration Order* ein Zyklus. Um diesen Zyklus zu durchbrechen, wird die Prozedur in das Hilfspaket `Skill_Mark_Helper` ausgelagert. In Listing 6.4 wird die angepasste Implementierung der `Mark_Graph`-Prozedur im Ada-Paket `Skill_Mark_Helper` veranschaulicht.

Listing 6.4: Markieren einer Graphinstanz

```
package body Skill_Mark_Helper is
  Current_Generation : Skill.Types.V64 := 1;
  package Reference_Array is new Dyn_Array (Storable);
  IO_Reference_Table : Reference_Array.D_Array_Ptr := Reference_Array.Make_Array;
  Node_Count : Natural;
  ...
  procedure Mark_Graph
    (The_Graph : IML_Graphs.IML_Graph) is
    Mark_Node_Count : Natural;
    use type Skill.Types.V64;
  begin
    -- Current_Generation um eins erhöhen
    -- IO_Reference_Table und Node_Count zurücksetzen
    ...
    Mark_Storable_Ptr (Storable (The_Graph.Root_Node));
    IML_Graphs.Mark_Node (The_Graph);
    while Mark_Node_Count < Node_Count loop
      Mark_Node_Count := Mark_Node_Count + 1;
      Explicit_Dispatch.Mark_Node.Mark_Node (IO_Reference_Table (Mark_Node_Count));
    end loop;
  end Mark_Graph;
  ...
end Skill_Mark_Helper;
```

Die `Mark_Graph`-Prozedur akzeptiert einen Parameter als Eingabe. Dieser Parameter namens `The_Graph` vom Typ `IML_Graph` ist die Graphinstanz, deren erreichbare Knoten markiert werden sollen. Der Markierungsalgorithmus arbeitet mithilfe von Generationen. Jeder gefundene Knoten vom Typ `Storable` wird in die nächste Generation überführt. Die `Mark_Storable_Ptr`-Prozedur akzeptiert als Argument eine Instanz vom Typ `Storable`. Dort wird die Generation der Instanz um eins erhöht und anschließend zum Array `IO_Reference_Table` hinzugefügt. In der Schleife akzeptiert die `Mark_Node`-Prozedur als Argument eine Instanz vom Typ `Storable` und ruft für deren Felder vom Typ `Storable` die `Mark_Storable_Ptr`-Prozedur beziehungsweise vom Typ `SLoc` die

SLoc_Mark-Prozedur auf. Der Typ SLoc besitzt eine eigene Prozedur, weil sie nicht vom Typ Storable erbt. Als Erstes wird der Wurzelknoten der Graphinstanz und die Graphinstanz selbst markiert. Anschließend werden nacheinander die Instanzen aus dem Array IO_Reference_Table markiert. Während durch das Array iteriert wird, werden durch den Aufruf der Mark_Storable_Ptr-Prozedur und der SLoc_Mark-Prozedur in der Mark_Node-Prozedur weitere Instanzen vom Typ Storable zum Array hinzugefügt. Sobald die Schleife an das Ende des Arrays angelangt ist, weil keine weiteren Instanzen vom Typ Storable mehr gefunden werden, wird sie verlassen. Demnach wurden alle erreichbaren Knoten des Graphen in die neue Generation überführt.

6.4 Schreiben einer SKilL-Binärdatei

Das Ada-Paket IML.IO stellt eine Save-Prozedur zur Verfügung, um einen IML-Graphen in eine Binärdatei zu speichern. Die Implementierung dieser Prozedur wird nun so ausgetauscht, dass eine Datei vom SKilL-Binärformat geschrieben wird. Der Austausch der Implementierung wird in Listing 6.5 veranschaulicht.

Listing 6.5: Speichern einer SKilL-Binärdatei

```
package body IML.IO is
  ...
  procedure Save
    (Filename : String;
     Graph : IML_Graphs.IML_Graph) is
  begin
    Skill_Mark_Helper.Mark_Graph (Graph);
    Skill_Mark_Helper.Delete_Unreachable_Storables (Graph.Sf);
    Graph.Sf.Change_Path (Filename);
    if 0 = Graph.Sf.IML_Graphs.Size then
      Graph.Sf.IML_Graphs.Make (F_Root_Node => IML_Roots.Siml_IML_Root (Graph.Root_Node));
    else
      Graph.Sf.IML_Graphs.Get (1).Set_Root_Node (IML_Roots.Siml_IML_Root (Graph.Root_Node));
    end if;
    Graph.Sf.Close;
  end Save;
  ...
end IML.IO;
```

Die Save-Prozedur akzeptiert zwei Parameter als Eingabe. Der erste Parameter namens Filename vom Typ string ist der Dateiname, der für die zu speichernde Binärdatei verwendet wird. Der zweite Parameter namens Graph vom Typ IML_Graph ist die zu speichernde Graphinstanz. Als Erstes wird die Graphinstanz markiert, d. h. alle erreichbaren Knoten werden in die neue Generation überführt. Anschließend werden alle Instanzen[5] vom Typ Storable gelöscht, die nicht der neusten Generation angehören. Dies wurde dadurch gelöst, dass im SKilL-Zustand durch den *Storage Pool* des Typs Storable iteriert wird und dabei jede Instanz, die einer alten Generation angehört, gelöscht wird. Als Nächstes wird der Dateiname für den SKilL-Zustand neu gesetzt. Anschließend

[5] Davon ausgenommen sind die Instanzen des nicht direkt in IML spezifizierten Typs Identifier.

wird geprüft, ob eine Instanz des Typs `IML_Graph` im SKilL-Zustand existiert. Falls nicht, wird eine neue Instanz vom Typ `IML_Graph` erstellt, wobei der Wurzelknoten als Argument übergeben wird. Ansonsten wird die vorhandene Instanz geladen und der Wurzelknoten gesetzt. Zum Schluss wird der SKilL-Zustand geschlossen. Beim Schließen eines SKilL-Zustands wird dieser in die angegebene Binärdatei übertragen.

6.5 Lesen einer SKilL-Binärdatei

Das Ada-Paket `IML.IO` stellt eine `Load`-Funktion zur Verfügung, um einen IML-Graphen aus einer Binärdatei zu laden. Die Implementierung dieser Funktion wird nun so ausgetauscht, dass eine Datei vom SKilL-Binärformat eingelesen werden kann. Der Austausch der Implementierung wird in Listing 6.6 veranschaulicht.

Listing 6.6: Laden einer SKilL-Binärdatei

```
package body IML.IO is
   ...
   function Load
     (Filename : in String)
      return IML_Graphs.IML_Graph is
      Graph : IML_Graphs.IML_Graph;
   begin
      Graph := new IML_Graphs.IML_Graph_Class;
      Graph.Sf := Siml.Api.Open (Filename, Skill.Files.Read, Skill.Files.Write);
      Graph.Root_Node := IML_Roots.IML_Root (Graph.Sf.IML_Graphs.Get (1).Get_Root_Node);
      return Graph;
   end Load;
   ...
end IML.IO;
```

Die `Load`-Funktion akzeptiert einen Parameter als Eingabe. Dieser Parameter namens `Filename` vom Typ `string` ist der Dateiname der einzulesenden Binärdatei. Als Nächstes wird eine neue Instanz vom Typ `IML_Graph_Class` erstellt. Das Feld `Sf` zeigt auf den eingelesenen SKilL-Zustand. Das Feld `Root_Node` zeigt auf den Wurzelknoten des geladenen Graphen. Anschließend wird die geladene Graphinstanz zurückgegeben.

6.6 Austausch der `Collect_Nodes`-Funktion

Die originale Implementierung der `Collect_Nodes`-Funktion im Ada-Paket `Storables` sammelt die Instanzen vom Benutzertyp *`T`* im IML-Graphen mithilfe des Markierungsalgorithmus ein. Das Ada-Binding verwaltet die Instanzen eines Benutzertyps *`T`* in dem entsprechenden *Storage Pool*, durch das effizient iteriert werden kann. Daher wird die Implementierung der `Collect_Nodes`-Funktion so angepasst, dass die Instanzen mithilfe des Iterators vom entsprechenden *Storage Pool* anstatt mithilfe des Markierungsalgorithmus eingesammelt werden. Da hierfür eine Abhängigkeit mit dem Ada-Paket `IML_Graphs` eingegangen werden muss, damit auf die Graphinstanz zugegriffen werden kann, wird

die `Collect_Nodes`-Funktion in das Hilfspaket `Skill_Storables_Helper` ausgelagert, sodass ein Zyklus in der *Elaboration Order* verhindert wird. Deshalb muss ein Werkzeug, das die originale `Collect_Nodes`-Funktion verwendet, diese durch die entsprechende Funktion aus dem Ada-Paket `Skill_Storables_Helper` ersetzen.

6.7 Hilfspaket für den SKilL-Zustand

In den handgeschriebenen Dateien befinden sich unter anderem auch die nicht direkt in IML spezifizierten Typen. Zwei dieser nicht direkt in IML spezifizierten Typen sind die Typen namens `Identifier` und `SLoc`. Ihre Ada-Pakete `Identifiers` und `SLocs` benötigen Zugriff auf die Graphinstanz, die den SKilL-Zustand hält, sodass Instanzen von ihnen erstellt werden können. Sobald die Ada-Pakete `Identifiers` und `SLocs` eine Abhängigkeit mit dem Ada-Paket `IML_Graphs` eingehen, entsteht in der *Elaboration Order* ein Zyklus. Um diesen Zyklus zu durchbrechen, kommt das Hilfspaket `Skill_State` zum Einsatz, das den SKilL-Zustand wie in Listing 6.7 veranschaulicht zur Verfügung stellt. Alle Methoden im Ada-Paket `IML_Graphs` wurden so angepasst, dass beim Setzen einer neuen Graphinstanz automatisch der dazugehörige SKilL-Zustand im Ada-Paket `Skill_State` gesetzt wird. Die Ada-Pakete `Identifiers` und `SLocs` können nun auf den SKilL-Zustand mithilfe des Ada-Pakets `Skill_State` zugreifen, ohne einen Zyklus in der *Elaboration Order* zu verursachen.

Listing 6.7: `Skill_State` als Hilfspaket für den SKilL-Zustand

```
package Skill_State is
  procedure Set_Constructor_Graph (Sf : Siml.Api.File);
  function Get_Constructor_Graph return Siml.Api.File;
end Skill_State;
```

6.8 SLoc

Der nicht direkt in IML spezifizierte Typ namens `SLoc` ist der einzige Typ, der nicht vom Benutzertyp `Storable` erbt. Außerdem ist er nicht wie die anderen Typen ein klassenweiter Zeigertyp, sondern ein Verbund (englisch record). Die Konstante `No_SLoc` zeigt im Gegensatz zu Listing 5.1 nicht auf `null`, sondern auf ein Verbund mit Standardwerten. Die erste Maßnahme war daher die Anpassung des Typs `SLoc` an das Schema in Listing 5.1. Dies wird in Listing 6.8 veranschaulicht.

Listing 6.8: SLoc nach der Anpassung

```
package SLocs is
  subtype SLoc is Siml.Sloc_Dyn;
  No_SLoc : constant SLoc := null;
  ...
end SLocs;
```

Aufgrund dieser Anpassung kann der Typ `SLoc` nun ein Nullzeiger werden. Damit es aufgrund des Nullzeigers zu keiner Ausnahme kommt, müssen die Methoden im Ada-Paket `SLocs` um eine

Nullzeigerprüfung erweitert werden und gegebenenfalls den passenden Standardwert zurückgeben. In Listing 6.9 wird diese Erweiterung anhand der `Get_Line`-Funktion veranschaulicht.

Listing 6.9: Nullzeigerprüfung anhand der `Get_Line`-Funktion

```
package body SLocs is
   ...
   function Get_Line
    (A_SLoc : in SLoc)
     return Natural is
   begin
     if Siml.Equals (null, Siml.Sloc (A_SLoc)) then
       return 0;
     else
       return Natural (A_SLoc.Get_Pos.Get_Line);
     end if;
   end Get_Line;
   ...
end SLocs;
```

Wie in Listing 6.10 veranschaulicht, wird der Gleichheitsoperator im Ada-Paket `SLocs` überladen. Um eine Rekursion bei der Nullzeigerprüfung zu verhindern, wird die Funktion zur Überprüfung der Gleichheit des Benutzertyps `SLoc` aus dem Ada-Binding verwendet. Ist das Argument für die `Get_Line`-Funktion ein Nullzeiger, so wird als Standardwert die Ganzzahl 0 zurückgegeben.

Listing 6.10: Überladung des Gleichheitsoperators im Ada-Paket `SLocs`

```
package SLocs is
   ...
   function "=" (Left, Right : in SLoc) return Boolean;
   ...
end SLocs;
```

7 Test

In diesem Kapitel wird die SKilL-basierte IML-Implementierung an verschiedenen Quelltextdateien getestet. Laut Aufgabenstellung dieser Arbeit soll die Funktionsfähigkeit einer durch das Werkzeug `cafeCC` generierten Datei im SKilL-Binärformat durch das Wiedereinlesen und Verwenden der Daten an einem weiteren Werkzeug aus dem Projekt Bauhaus gezeigt werden. Dafür wurde das Werkzeug namens `ccdiml` ausgesucht, weil es bis auf die Typen `Identifier` und `SLoc` keine weiteren nicht direkt in IML spezifizierten Typen verwendet und eine Testsuite mitliefert. Das Problem der Testsuite war jedoch, dass bereits in der originalen IML-Implementierung alle Tests fehlschlugen. Dies konnte behoben werden, indem die erwarteten Testergebnisse durch die in einer Neuausführung der originalen IML-Implementierung erzeugten Ergebnisse aktualisiert wurden. Danach konnte die SKilL-basierte IML-Implementierung die Testsuite erfolgreich durchlaufen, d. h. beide IML-Implementierungen verhalten sich gleich.

Auf der Suche nach weiteren Werkzeugen, die bis auf die Typen `Identifier` und `SLoc` keine weiteren nicht direkt in IML spezifizierten Typen verwenden, konnten die Werkzeuge namens `imlmetrics` und `imlstat` ermittelt werden. Nachdem die ersten manuellen Tests mit dem Werkzeug `imlmetrics` durchgeführt und die Ergebnisse anhand der Ergebnisse aus der originalen IML-Implementierung verglichen wurden, konnten vereinzelt Differenzen entdeckt werden. Diese Differenzen konnten im Laufe dieser Arbeit beseitigt werden.

7.1 Vorgehensweise

Da der vorhandene Linker direkt auf dem Binärstrom des IML-Binärformats arbeitet und daher für die SKilL-basierte IML-Implementierung nicht verwendet werden kann, können für diesen Test nur einzelne Quelltextdateien verwendet werden. Deshalb werden in diesem Test nur Projekte verwendet, die aus einer einzelnen Kompilierungseinheit bestehen.

Für diesen Test werden zwei separate Bauhaus-Umgebungen verwendet. Die erste Bauhaus-Umgebung verwendet die in dieser Arbeit entwickelte SKilL-basierte IML-Implementierung. Die zweite Bauhaus-Umgebung verwendet die originale IML-Implementierung.

Eine zu testende Quelltextdatei wird auf einer Bauhaus-Umgebung mit dem Werkzeug `cafeCC` aufgerufen. Dieses Werkzeug generiert aus der Quelltextdatei einen IML-Graphen, der anschließend in einer Binärdatei abgespeichert wird. Diese Binärdatei wird dann von einem Werkzeug eingelesen und verarbeitet. Das dabei zurückgelieferte Ergebnis wird gespeichert. Dieser Prozess wird auf der anderen Bauhaus-Umgebung wiederholt, wobei das gleiche Werkzeug verwendet wird. Die Ergebnisse für jeweils eine Quelltextdatei müssen beim Vergleich identisch sein, d. h. die Differenzmenge

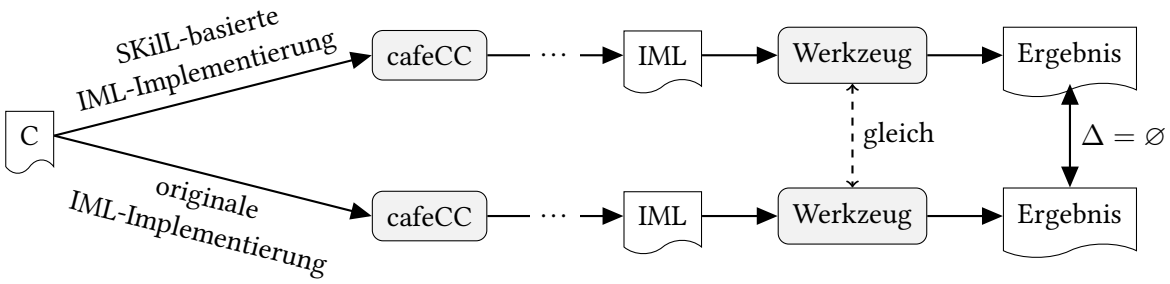


Abbildung 7.1: Der Testablauf für eine in der Programmiersprache C geschriebenen Quelltextdatei. In beiden Bauhaus-Umgebungen wird das gleiche Werkzeug verwendet. Die zurückgelieferten Ergebnisse müssen übereinstimmen

eines Ergebnispaares ist die leere Menge. In Abbildung 7.1 wird der Testablauf anhand einer in der Programmiersprache C geschriebenen Quelltextdatei veranschaulicht.

7.2 Testumgebung

Die Tests wurden auf dem universitätseigenen Rechner namens *pslx0* mit folgendem Betriebssystem und Compiler ausgeführt.

Betriebssystem Debian 8.6 (Linux version 3.16.0-4-amd64)

Compiler GNAT Pro 7.2.2 (20140525-47)

Das Werkzeug `cafeCC` wurde mit den Parametern `-B‘pwd‘ -c` aufgerufen. Der erste Parameter `-B` mit dem Argument `‘pwd‘` gibt an, dass anstatt des ganzen Dateipfades nur der Dateiname der Quelltextdatei verwendet werden soll. Der zweite Parameter `-c` gibt an, dass der Linker nicht aufgerufen werden soll.

7.3 Testdaten

Als Testdaten wurden zusätzlich zu den vierzehn Testdateien aus dem Werkzeug `ccdiml` und den drei Testdateien aus dem Werkzeug `imlmetrics` die Testdateien aus der Tabelle 7.1 verwendet. Dabei wurde darauf geachtet, dass nicht nur einfache und kleine Projekte verwendet werden. Bei der größten Datei handelt es sich um das Projekt SQLite, das ein einbettbares, relationales Datenbankmanagementsystem ist. Die dazugehörige Testdatei *sqlite3.c* ist eine aus mehr als 100 verschiedenen Quelltextdateien verkettete Datei.

Dateiname	Dateigröße	Beschreibung
bzip2.c	200,13 kB	Komprimierungsprogramm

Dateiname	Dateigröße	Beschreibung
duktape.c	3,08 **MB**	Einbettbare JavaScript-Engine
easyzlib.c	324,22 kB	Wrapper für die zlib-Komprimierungsbibliothek
empty.c	0 B	Leere Datei
fibonacci.c	561 B	Berechnung der Fibonacci-Zahlen
floats.c	306 B	Gleitkommaarithmetik in binären Ausdrücken
gzip.c	265,87 kB	Komprimierungsprogramm
hello.c	71 B	Simples „Hello World“-Programm
levenshtein.c	1,61 kB	Implementierung der Levenshtein-Distanz
matrices.c	1,77 kB	Multiplikation von zwei Matrizen
mpc.c	104,19 kB	Parserkombinatoren-Bibliothek
oggenc.c	1,64 **MB**	Ogg-Vorbis Encoder
parg.c	5,63 kB	Parser für `argv`
ph7.c	1,88 **MB**	Einbettbare Implementierung von PHP
sqlite3.c	6,68 **MB**	Einbettbares, relationales Datenbanksystem
stmr.c	15,21 kB	Implementierung des Porter-Stemmer-Algorithmus
tweetnacl.c	16,25 kB	Kryptografie-Bibliothek
unqlite.c	1,80 **MB**	Einbettbare NoSQL-Datenbank
ylog.c	5,62 kB	Protokollierungssystem
yuri.c	7,62 kB	Parsing- und Validierungsbibliothek für URI
zlib_amalg.c	346,74 kB	Komprimierungsbibliothek
jo_mpeg.cpp	8,69 kB	MPEG Writer
lightmapper.h	61,31 kB	Lightmap-Bibliothek
sched.h	34,68 kB	Multithread-Task-Scheduler
utf8.h	27,46 kB	String-Funktionen für UTF-8

Tabelle 7.1: Die zusätzlich verwendeten Testdateien für den Test. Die Einheit Megabyte ist fett gedruckt, sodass große Dateien leichter erkennbar sind. Die Tabelle ist zuerst nach der Dateiendung und dann nach dem Dateinamen sortiert

Listing 7.1: Ablauf des Testwerkzeugs in Pseudocode

```
-- generate.sh
for $file in $files loop
  cafeCC -B‘pwd‘ -c $file; -- der Dateiname für die Ausgabe ist $file.o
  java -jar recode.jar $file.o $file.o; -- nur in der SKilL-basierten IML-Implementierung
  ccdiml $file.o > /tmp/ccdiml.$file.1; -- die andere Bauhaus-Umgebung verwendet die Zahl 2
  imlmetrics $file.o > /tmp/imlmetrics.$file.1;
end loop;

-- check.sh
for $file in $files loop
  diff -s /tmp/ccdiml.$file.*;
  diff -s /tmp/imlmetrics.$file.*;
end loop;
```

7.4 Testwerkzeug

Das Testwerkzeug besteht aus den zwei Bash-Skripten mit den Dateinamen *generate.sh* und *check.sh*. In Listing 7.1 wird der Ablauf des Testwerkzeugs in vereinfachter Weise veranschaulicht.

Das erste Bash-Skript mit dem Dateinamen *generate.sh* generiert die zu vergleichenden Ergebnisse für eine gegebene Testdatei und wird daher auf beiden Bauhaus-Umgebungen separat ausgeführt. Dabei werden alle Quelltextdateien, die in dem Ordner des Bash-Skriptes liegen und die Dateiendung *.c* oder *.cpp* besitzen, als Testdateien verwendet. Sollen noch zusätzlich einzelne Headerdateien mit der Dateiendung *.h* getestet werden, müssen ihre Dateinamen manuell zum Bash-Skript hinzugefügt werden. Die Testdateien werden nacheinander mithilfe des Werkzeugs `cafeCC` übersetzt. Die erzeugte Binärdatei wird bei der SKilL-basierten IML-Implementierung zusätzlich durch ein Java-Binding, das die generierte SKilL-Spezifikation unterstützt, recodiert. Nach jeder Übersetzung einer Testdatei wird die erzeugte und gegebenenfalls recodierte Binärdatei an die Werkzeuge[6] `ccdiml` und `imlmetrics` übergeben. Deren Ergebnisse werden anschließend in den im Bash-Skript definierten Ordner abgespeichert, wobei der Ordner von beiden Bauhaus-Umgebungen erreichbar sein muss.

Das zweite Bash-Skript mit dem Dateinamen *check.sh* überprüft für jede Testdatei das entsprechende Ergebnispaar auf Differenzen.

7.5 Ergebnisse

Für jede Testdatei aus den Testdaten waren für jeweils beide Werkzeuge die dazugehörigen zwei Ergebnisse identisch, d. h. die Differenzmenge aller Ergebnispaare war die leere Menge. Der Test war somit erfolgreich.

[6] Das Werkzeug `imlstat` wurde nicht getestet, weil die Ergebnisse aufgrund des nicht direkt in IML spezifizierten Typs `Identifier` nicht vergleichbar sind.

8 Offene Punkte

In diesem Kapitel werden die offenen Punkte besprochen, die aufgrund ihres Umfangs nicht im Rahmen dieser Arbeit gelöst wurden sowie weitere Hinweise gegeben.

8.1 Ada-Paket `Storable_Lists`

Das Ada-Paket `Storable_Lists` ist eine eigene Implementierung einer doppelt verketteten Liste, die für die Programmiersprache C exportiert wird. Jedoch wird sie vom Werkzeug `cafe++` in allen Fällen nur temporär verwendet (siehe Listing 8.1). Daher wurde entschieden, die Liste nicht an SKilL anzupassen. Sollte ein Werkzeug die `Storable_Lists` benötigen, deren Elemente vom SKilL-Zustand erfasst werden sollen, so muss diese durch das Ada-Paket `Storables.Lists` ersetzt werden.

8.2 Benutzertyp `Identifier`

Der nicht direkt in IML spezifizierte Typ `Identifier` erbt vom Benutzertyp `Storable`. Vor dem Schreiben einer SKilL-Binärdatei werden alle nicht im IML-Graphen erreichbaren Instanzen vom Typ `Storable` gelöscht. Jedoch werden durch den Markierungsalgorithmus nicht alle Instanzen vom Typ `Identifier` erfasst. Aus diesem Grund sind die Instanzen vom Typ `Identifier` von der Löschung ausgenommen.

8.3 Fork

Zu Beginn dieser Arbeit wurde angenommen, dass nur die generierte API sowie die Lese- und Schreibmethoden einer Binärdatei angepasst werden müssen. Nach und nach hat sich herauskristallisiert, dass die Bibliothek `libIML` stark angepasst werden muss. Daher wurde das Projekt Bauhaus für

Listing 8.1: Temporäre Verwendung der `Storable_Lists` in `cafe++`

```
Storable_List tmp_items = storable_lists_make();
storable_lists_append (tmp_items, ...);
..._append_list_to_... (..., tmp_items);
storable_lists_destroy(&tmp_items);
```

die SKilL-basierte IML-Implementierung von der originalen IML-Implementierung abgespalten. Es existieren somit zurzeit zwei Bauhaus-Umgebungen.

Des Weiteren mussten einige Werkzeuge für diese Arbeit leicht angepasst werden. Beim Werkzeug `cafe++` wurde bei zwei Methoden jeweils ein toter Parameter entfernt, weil das dazugehörige Feld in der angepassten IML-Spezifikation nicht vorhanden ist. Weiterhin wurden die `SLoc`-*Assertions* entfernt, weil eine Instanz vom Benutzertyp `SLoc` ein Nullzeiger werden kann. Bei den Werkzeugen `iml_strip`, `imlmetrics` und `imlstat` wurde jeweils die `Collect_Nodes-Funktion` durch die entsprechende Funktion aus dem Ada-Paket `Skill_Storables_Helper` ersetzt.

8.4 Linker

Der im Projekt Bauhaus bereitgestellte Linker wurde vom Autor der Doktorarbeit [Sta09, § 11.2.1] entwickelt. Dieser Linker arbeitet direkt auf dem Binärstrom des IML-Binärformats und kann aus diesem Grund nicht für die SKilL-basierte IML-Implementierung verwendet werden. Zwar kann das Werkzeug `cafeCC` Projekte übersetzen, die aus mehreren Quelltextdateien bestehen, aber die dabei generierten einzelnen IML-Graphen können nicht zu einem gesamten IML-Graphen zusammengefügt werden. Weiterhin beinhaltet das Ada-Paket `IML_Graphs` einigen Linker-spezifischen Code.

Mithilfe des Werkzeugs `iml2sf` [FW16, § 2.1] kann ein Projekt, das bereits als IML-Graph im IML-Binärformat vorliegt, in einen semantisch äquivalenten IML-Graphen im SKilL-Binärformat transformiert werden. Somit ist es über den Umweg der originalen IML-Implementierung möglich, Projekte, die aus mehreren Quelltextdateien bestehen, auch in der SKilL-basierten IML-Implementierung zu verwenden.

8.5 `singleton`-Restriktion

Der Benutzertyp `IML_Graph` wird in der SKilL-Spezifikation mit der `singleton`-Restriktion annotiert. Der SKilL-Codegenerator für das Ada-Binding generiert jedoch keinen spezialisierten Code für *Storage Pool*s, die nur eine Instanz verwalten dürfen. Die SKilL-basierte IML-Implementierung geht jedoch davon aus, dass es vom Benutzertyp `IML_Graph` nur eine Instanz gibt und greift daher immer auf den *Storage Pool* mit dem Indexwert 1 zu.

8.6 Sporadische Abstürze von `cafe++`

Während dem Testen des Werkzeugs `cafeCC` ist aufgefallen, dass manche Quelltextdateien das aufgerufene Werkzeug `cafe++` sporadisch mit der Fehlermeldung *internal error: assertion failed: …* zum Absturz bringen. Eine diesen Fehler verursachende Quelltextdatei konnte auf zwei Codezeilen reduziert werden (siehe Listing 8.2).

Um herauszufinden, wie oft die Quelltextdatei in Listing 8.2 einen Kompiliervorgang zum Absturz bringen kann, wurde sie tausendmal mit dem Werkzeug `cafeCC` übersetzt. Dieser Vorgang wurde

Listing 8.2: Fehler verursachende Quelltextdatei

```
#include<stdio.h>
float a=1.0;
```

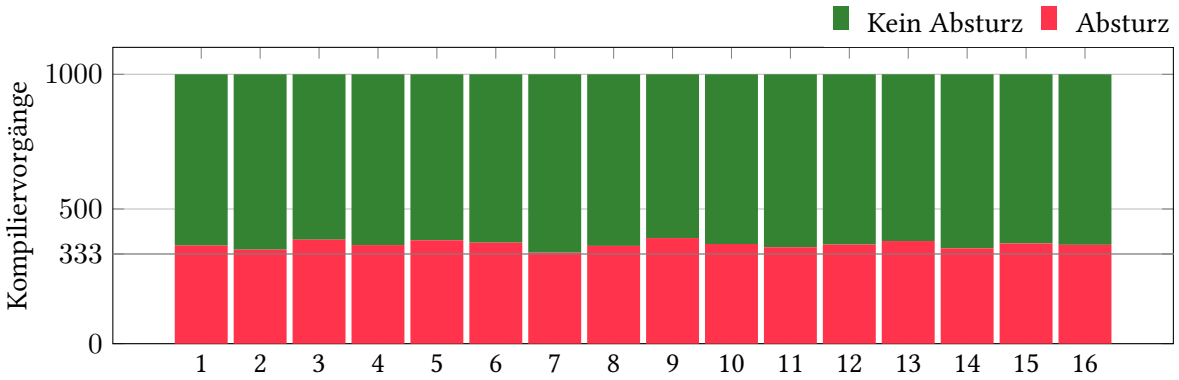


Abbildung 8.1: Die den Fehler verursachende Quelltextdatei wurde tausendmal mit dem Werkzeug `cafeCC` in 16 Runden übersetzt

fünfzehnmal wiederholt. Das Ergebnis, das in Abbildung 8.1 visualisiert wird, zeigt, dass jeder dritte Kompiliervorgang fehlschlägt.

8.7 Werkzeug `cobra`

Neben dem Werkzeug `cafeCC` soll die Funktionsfähigkeit der SKilL-basierten IML-Implementierung an einem weiteren Werkzeug aus dem Projekt Bauhaus gezeigt werden, indem die vom Werkzeug `cafeCC` generierte Binärdatei wieder eingelesen und verwendet wird. Die initiale Planung war, dies anhand des Werkzeugs `cobra` zu zeigen. Es stellt einen Browser für den IML-Graphen zur Verfügung, sodass durch ihn mithilfe von Hyperlinks navigiert werden kann. Jedoch hat das Werkzeug `cobra` Abhängigkeiten zu Bibliotheken, die einige nicht direkt in IML spezifizierte Typen verwenden. Daher wurde als Ersatz das Werkzeug `ccdiml` ausgesucht, weil es bis auf die Typen `Identifier` und `SLoc` keine weiteren nicht direkt in IML spezifizierten Typen verwendet. Des Weiteren konnten noch die Werkzeuge `imlmetrics` und `imlstat` lauffähig gemacht werden.

9 Performance-Evaluation

In diesem Kapitel wird die SKilL-basierte IML-Implementierung und die originale IML-Implementierung bezüglich ihrer Performance an drei Werkzeugen verglichen.

Die Performance-Evaluation besteht aus drei verschiedenen Tests und testet jeweils die Werkzeuge `ccdiml`, `imlmetrics` und `imlstat` bezüglich ihrer Laufzeiten. Jeder Test wurde zehnmal wiederholt und absolvierte somit insgesamt elf Runden.

9.1 Testumgebung

Die Tests wurden auf dem universitätseigenen Rechner namens *pslx0* mit folgender Ausstattung ausgeführt.

CPU 4x AMD Opteron 6174, je 12x 2,2 GHz

RAM 256 GB 1333 MHz DDR3

HDD Eurostor ES-6600D (Raid 5), 7x Toshiba MG03SCA200

Betriebssystem Debian 8.6 (Linux version 3.16.0-4-amd64)

Compiler GNAT Pro 7.2.2 (20140525-47) mit dem Optimierungsflag `-O2`

9.2 Testdaten

Als Testdaten wurden Projekte verwendet, die hauptsächlich in der Programmiersprache C entwickelt sind. Für diese Projekte liegen bereits IML-Graphen im IML-Binärformat vor. Mithilfe des Werkzeugs `iml2sf` [FW16, § 2.1] wurden die IML-Graphen vom IML-Binärformat (*.iml*) in das SKilL-Binärformat (*.iml.sf*) transformiert. Jedes Projekt liegt somit in beiden Binärformaten vor, die jeweils den semantisch äquivalenten IML-Graphen speichern. Die Tabelle 9.1 gibt für alle getesteten Projekte Auskunft über die Dateigrößen beider Binärformate.

	Dateigröße in MB			Dateigröße in MB	
Name	**.iml**	**.iml.sf**	**Name**	**.iml**	**.iml.sf**
Astro	1,09	0,94	aget	0,27	0,28
bash	15,17	14,69	bash43	19,97	18,81

Name	Dateigröße in MB		Name	Dateigröße in MB	
	.iml	**.iml.sf**		**.iml**	**.iml.sf**
bashversion	0,04	0,06	bf_test	0,13	0,14
bison	5,73	5,33	bluefish	17,95	17,14
concepts	0,76	0,75	cook	6,77	6,84
cu	3,07	3,00	darkhttpd	0,43	0,44
doc2gih	0,53	0,53	empty	0,00	0,02
gethost	0,12	0,14	gnugo	51,76	47,85
gnuplot	16,84	16,25	gnuplot_x11	1,30	1,24
gqview	24,61	23,77	grep	2,61	2,33
joseki	1,63	1,44	make.new	2,45	2,32
mkbuiltins	0,22	0,22	mkeyes	0,11	0,13
mkpat	1,27	1,21	mksignames	0,06	0,08
mksyntax	0,06	0,07	nano	1,84	1,76
pgen	2,78	2,88	php-cgi	300,40	280,87
php	302,06	281,97	php7-cgi	428,27	378,50
php7	429,85	379,93	php7dbg	450,82	397,43
psize.aux	0,05	0,07	screen	7,45	7,33
sed	2,66	2,50	sgfgen	0,02	0,04
simple	0,01	0,02	smtprc	1,16	1,13
tcsh	8,07	7,99	time	0,16	0,16
trueprint	1,50	1,47	units	0,66	0,66
uuchk	1,49	1,44	uucico	5,90	5,98
uuconv	2,05	1,99	uucp	2,15	2,11
uulog	1,37	1,34	uuname	0,77	0,78
uupick	1,53	1,50	uustat	2,37	2,33
uux	2,17	2,13	uuxqt	2,54	2,50

Tabelle 9.1: Die verwendeten Projekte für alle drei Tests. Die Spalte *.iml* gibt Auskunft über die Dateigrößen für das IML-Binärformat. Die Spalte *.iml.sf* gibt Auskunft über die Dateigrößen für das SKilL-Binärformat

Die bereitgestellten Projekte namens `_freeze_importlib`, `_testembed`, `bc`, `dc`, `python` und `tstuu` wurden nicht verwendet. Die ersten fünf Projekte weisen aus bisher unbekannten Gründen leichte Differenzen bei den Ergebnispaaren für das Werkzeug `ccdiml` oder das Werkzeug `imlmetrics` auf.

Listing 9.1: Benötigte gemeinsam genutzte Bibliotheken für SKilL

```
$ objdump -p ccdiml | grep NEEDED
  NEEDED          libIML.so # IML: ~50 MB; SKilL: ~200 MB
  NEEDED          libSiml.so # SKilL: ~150 MB
  NEEDED          libSkill_Common.so # SKilL: ~2 MB
```

Listing 9.2: Benötigte Taktzyklen des Laufzeit-Linkers beim Werkzeug `ccdiml`

```
$ LD_DEBUG=statistics ccdiml 2>&1 | grep total
    total startup time in dynamic loader: 170054121 clock cycles # IML (∅ aus 1000 Runden)
    total startup time in dynamic loader: 675849973 clock cycles # SKilL (∅ aus 1000 Runden)
```

Beim letzteren Projekt terminiert das Werkzeug `imlmetrics` für beide IML-Implementierungen nicht.

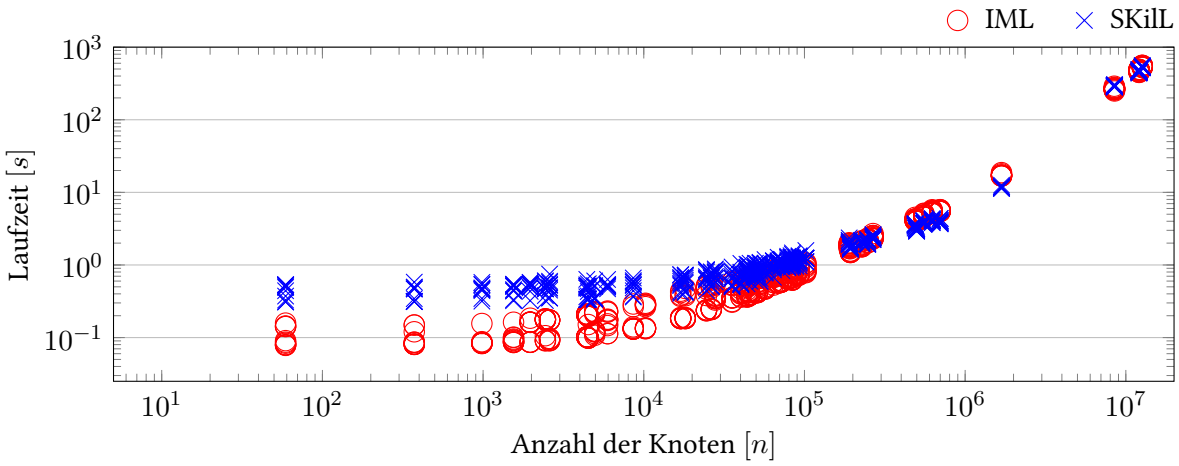
Das kleinste Projekt namens *empty* besteht aus einer leeren Quelltextdatei. Das IML-Binärformat erreicht bei der Serialisierung des IML-Graphen eine Dateigröße von 2 kB, die in der Tabelle 9.1 auf 0 MB abgerundet wird. Das SKilL-Binärformat erreicht bei der Serialisierung des IML-Graphen eine Dateigröße von 17 kB, die in der Tabelle 9.1 auf 0,02 MB aufgerundet wird. Die Differenz von 15 kB kommt durch das Typsystem von SKilL zustande, das mitserialisiert werden muss.

Das größte Projekt namens *php7dbg* besteht aus der Fehlerbeseitigungsplattform für PHP 7. Das IML-Binärformat erreicht bei der Serialisierung des IML-Graphen eine Dateigröße von 450,82 MB. Das SKilL-Binärformat erreicht bei der Serialisierung des IML-Graphen eine Dateigröße von 397,43 MB. Damit benötigt das SKilL-Binärformat bei diesem Projekt circa 12 % weniger Speicherplatz als das IML-Binärformat.

Die Abbildungen 9.1, 9.2 und 9.3 visualisieren jeweils für die Werkzeuge `ccdiml`, `imlmetrics` und `imlstat` die Laufzeiten für die Projekte aus der Tabelle 9.1. Damit eine vertikale Gerade durch die Messpunkte beider Binärformate für dasselbe Projekt gelegt werden kann, ist die unabhängige Variable anstatt der Dateigröße die Anzahl der Knoten im IML-Graphen.

9.3 Anlaufzeit der Werkzeuge

In allen drei Abbildungen ist deutlich erkennbar, dass bei der SKilL-basierten IML-Implementierung am Anfang die Anlaufzeit der Werkzeuge dominiert. Das kommt unter anderem dadurch zustande, dass der Laufzeit-Linker in der SKilL-basierten IML-Implementierung zwei weitere Bibliotheken laden muss (siehe Listing 9.1) und die Datei *libIML.so* viermal so groß ist. Die beim Werkzeug `ccdiml` benötigten Taktzyklen für den Laufzeit-Linker veranschaulicht das Listing 9.2 für beide IML-Implementierungen. Bei einem Prozessortakt von 2,2 GHz liegt die Grundlast für den Laufzeit-Linker bei der SKilL-basierten IML-Implementierung bei 0,31 s. Im Gegensatz dazu liegt die Grundlast für den Laufzeit-Linker bei der originalen IML-Implementierung bei 0,08 s. Die SKilL-basierte IML-Implementierung hat für den Laufzeit-Linker somit eine ungefähr viermal größere Grundlast.



	IML	SKilL
‘mega nodes‘	38.580***	38.598***
	(0.212)	(0.169)
Constant	−4.812***	−4.431***
	(0.708)	(0.564)
Observations	594	594
R^2	0.982	0.989
F Statistic (df = 1; 592)	33’026.610***	52’090.170***
Note:		*p<0.1; **p<0.05; ***p<0.01

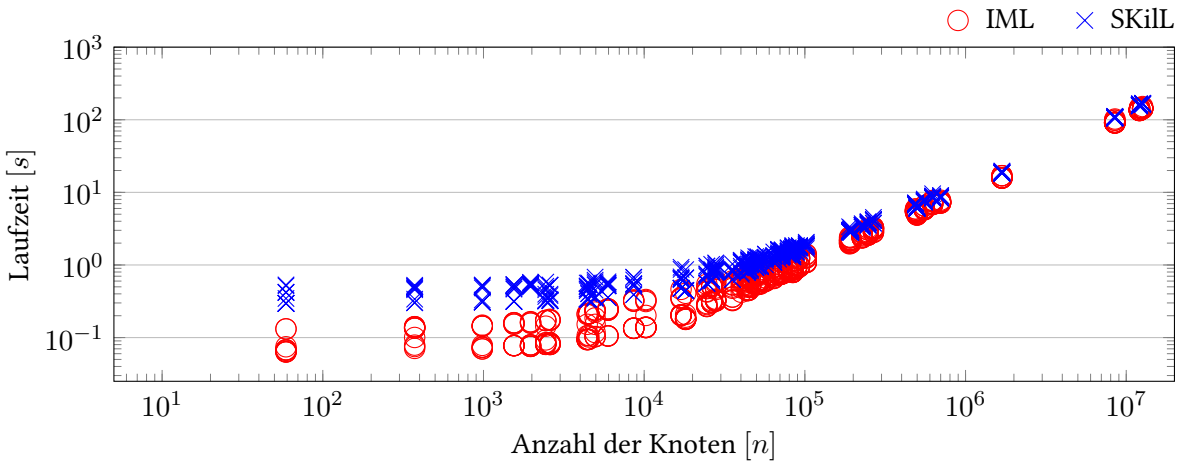
Abbildung 9.1: Das Diagramm veranschaulicht die gemessene Laufzeit und die Tabelle [Hla15] die dazugehörige lineare Regression für das Werkzeug `ccdiml`

9.4 Werkzeug `ccdiml`

Das erste untersuchte Werkzeug `ccdiml` identifiziert Codeklone im IML-Graphen nach dem AST-basierten Verfahren von Baxter [BYM+98].

In Abbildung 9.1 ist zu erkennen, dass ab $10^{5.25}$ Knoten[7] die Laufzeit beider Binärformate ungefähr gleich ist. Dies beschreiben auch die Regressionsgeraden. Die Steigungen und die konstanten Kosten sind ungefähr gleich groß. Die konstanten Kosten liegen unter Beachtung des jeweiligen Standardfehlers auf der negativen Y-Achse, sodass die Regressionsgerade auch negative Laufzeiten beschreibt. Das ist natürlich nicht möglich, weil die Laufzeit positiv sein muss.

[7] $10^{5.25} \approx 177828$



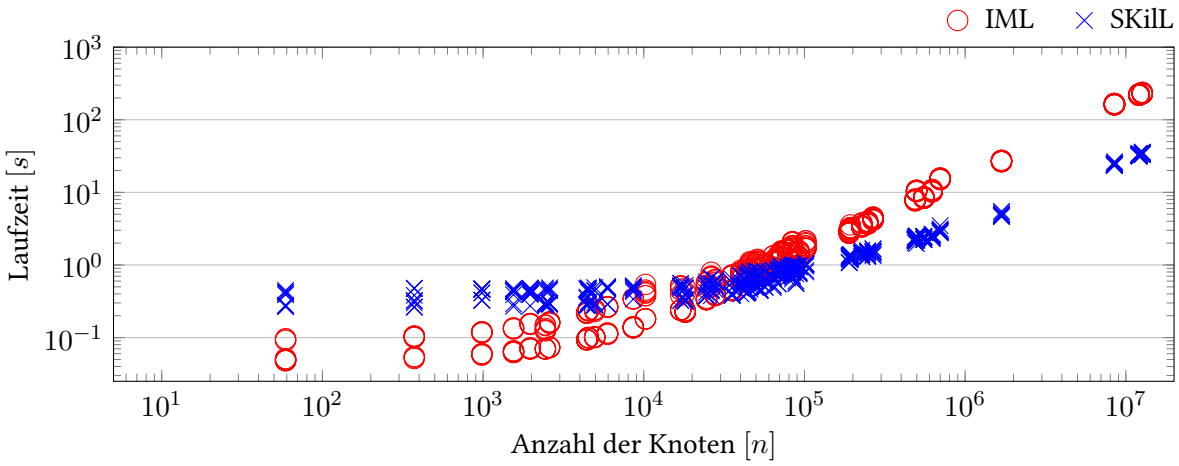
	IML	SKilL
'mega nodes'	11.255***	13.005***
	(0.017)	(0.014)
Constant	−0.064	0.320***
	(0.056)	(0.048)
Observations	594	594
R^2	0.999	0.999
F Statistic (df = 1; 592)	457'155.500***	834'000.800***
Note:		*p<0.1; **p<0.05; ***p<0.01

Abbildung 9.2: Das Diagramm veranschaulicht die gemessene Laufzeit und die Tabelle [Hla15] die dazugehörige lineare Regression für das Werkzeug `imlmetrics`

9.5 Werkzeug `imlmetrics`

Das zweite untersuchte Werkzeug `imlmetrics` berechnet aus einem IML-Graphen verschiedene Metriken, die unter anderem die Anzahl der Anweisungen, Codezeilen, Methodenaufrufe und -parameter sowie die zyklomatische Komplexität umfassen.

In Abbildung 9.2 ist zu erkennen, dass die Laufzeit der SKilL-basierten IML-Implementierung sich nur der originalen IML-Implementierung annähert. Dass es zu keinem Schnittpunkt kommt, beschreiben auch die Regressionsgeraden. Die Regressionsgerade der SKilL-basierten IML-Implementierung liegt über der Regressionsgeraden der originalen IML-Implementierung. Weiterhin ist die Steigung bei der SKilL-basierten IML-Implementierung größer als bei der originalen IML-Implementierung. Laut der Steigungen ist die SKilL-basierte IML-Implementierung circa 16 % teurer.



	IML	SKilL
'mega nodes'	18.519***	2.699***
	(0.023)	(0.005)
Constant	−0.080	0.567***
	(0.075)	(0.017)
Observations	594	594
R^2	0.999	0.998
F Statistic (df = 1; 592)	674'875.500***	281'626.400***
Note:		*p<0.1; **p<0.05; ***p<0.01

Abbildung 9.3: Das Diagramm veranschaulicht die gemessene Laufzeit und die Tabelle [Hla15] die dazugehörige lineare Regression für das Werkzeug `imlstat`

9.6 Werkzeug `imlstat`

Das dritte untersuchte Werkzeug `imlstat` zählt alle Knoten in einem IML-Graphen. Dieses Werkzeug ist unter anderem interessant, weil es nahezu vollständig durch den IML-Graphen iteriert.

In Abbildung 9.3 ist zu erkennen, dass ab $10^{4.5}$ Knoten[8] die Laufzeit der SKilL-basierten IML-Implementierung besser ist. Dies beschreiben auch die Regressionsgeraden. Laut der Steigungen ist die SKilL-basierte IML-Implementierung circa 85 % günstiger. Des Weiteren kann die Grundlast des Werkzeugs für die SKilL-basierte IML-Implementierung bei den konstanten Kosten abgelesen werden.

[8] $10^{4.5} \approx 31623$

9.7 Zusammenfassung

Zusammenfassend lässt sich hervorheben, dass durch den Austausch der IML-Implementierung die Laufzeit der getesteten Werkzeuge für große IML-Graphen entweder gleich, nicht wesentlich schlechter oder wesentlich besser geworden ist. Alle Regressionsgeraden haben erwartungsgemäß eine positive Steigung. Das Bestimmtheitsmaß ist bei allen Regressionsgeraden sehr hoch, sodass davon ausgegangen werden kann, dass sich die Laufzeit aller Werkzeuge hauptsächlich durch die Anzahl der Knoten erklären lässt. Das Werkzeug `imlstat` hat bei der SKilL-basierten IML-Implementierung ab $10^{4.5}$ Knoten eine wesentlich bessere Laufzeit, weil durch die *Storage Pool*s des SKilL-Zustands iteriert wird anstatt mithilfe des Markierungsalgorithmus die Instanzen im IML-Graphen einzusammeln.

10 Unspezifizierte Typen

In diesem Kapitel werden die wesentlichen Punkte bei der Übersetzung von nicht direkt in IML spezifizierten Typen nach SKilL erörtert. Oft kann eine Eins-zu-eins-Übersetzung vorgenommen werden, wobei bei der Übersetzung nach SKilL folgende Punkte beachtet werden müssen.

10.1 Ada-Paket `Storable_Lists`

Der Typ `Storable_Lists` wird nach SKilL als Liste mit dem Typparameter `Storable` übersetzt. Weiterhin muss das Werkzeug angepasst werden, sodass anstatt des Ada-Pakets `Storable_Lists` das Ada-Paket `Storables.Lists` verwendet wird. Mit dieser Maßnahme wird außerdem die Anzahl der Implementierungen von Listen im Projekt Bauhaus reduziert.

10.2 *Variant Records*

In der Programmiersprache Ada werden sogenannte *Variant Records* [TDB+12, § 3.8.1] bereitgestellt. Ein *Variant Record* erlaubt es mithilfe einer Diskriminante, die aktuelle Variante von Feldern zu bestimmen. In Listing 10.1 wird dies an einem beispielhaften *Variant Record* veranschaulicht. Der Typ `A` besitzt eine Diskriminate namens `D` vom Typ `Boolean` und stellt somit zwei Varianten von Feldern bereit. Je nach dem Wahrheitswert stellt der Verbund entweder das Feld namens `A` vom Typ `Natural` oder das Feld namens `B` vom Typ `Boolean` zur Verfügung. Der Name eines Feldes ist eindeutig und kann nicht über mehrere Varianten hinweg wiederverwendet werden. Da SKilL keine Benutzertypen mit Varianten unterstützt, wird ein *Variant Record* nach SKilL als Benutzertyp übersetzt, der die Felder aller Varianten vereint.

Listing 10.1: Beispielhafter *Variant Record*

```
type A (D : Boolean) is record
  case D is
    when True => A : Natural;
    when False => B : Boolean;
  end case;
end record;
B : A := (D => True, A => 0);
C : A := (D => False, B => True);
```

Listing 10.2: Beispielhafte Hashmap mit einem Array als Elementtyp

```
B {}
Flat_Type { B[] data; }
/** map<v64, B[]> nicht möglich! */
A { map<v64, Flat_Type> data; }
```

10.3 Zusammengesetzter Typ als Typparameter

Wenn ein zusammengesetzter Typ als Typparameter eines anderen zusammengesetzten Typs verwendet werden soll, muss ein flacher Benutzertyp eingeführt werden, weil SKilL nur simple Typen als Typparameter eines zusammengesetzten Typs erlaubt. In Listing 10.2 wird dies an einer beispielhaften Hashmap veranschaulicht, die als Elementtyp ein Array verwenden soll. Der Benutzertyp namens `A` besitzt ein Feld namens `data`, wobei als Typ eine Hashmap mit dem Schlüsseltyp `v64` verwendet wird. Als Elementtyp soll ein Array mit dem Typparameter `B` verwendet werden. Da dies nicht möglich ist, wird ein flacher Benutzertyp namens `Flat_Type` eingeführt, der das Array mit dem Typparameter `B` als Feld besitzt. Dieser flache Benutzertyp wird anschließend als Elementtyp der Hashmap verwendet.

11 Zusammenfassung

Der Zweck dieser Arbeit ist der Austausch der originalen IML-Implementierung durch die SKilL-basierte IML-Implementierung im Projekt Bauhaus. Hierfür wurden die zwei Werkzeuge namens Spezifikationsgenerator (SpecGen) und Codegenerator (CodeGen) entwickelt. Das erste Werkzeug SpecGen generiert aus einer gegebenen IML-Spezifikation eine SKilL-Spezifikation. Das zweite Werkzeug CodeGen generiert aus einer gegebenen SKilL-Spezifikation die angepasste IML-Implementierung. Mithilfe der angepassten IML-Implementierung wird bei der Serialisierung des IML-Graphen das SKilL-Binärformat verwendet. Somit kann der IML-Graph in eine Datei im SKilL-Binärformat geschrieben beziehungsweise aus einer Datei im SKilL-Binärformat gelesen werden.

Nach einer kurzen Einführung über die Gründe des Austauschs der IML-Implementierung vermittelt Kapitel 2 die relevanten Grundlagen für die nachfolgenden Kapitel. Dabei werden das Projekt Bauhaus und die dazugehörige Bibliothek `libIML` sowie die Serialisierungssprache SKilL abgedeckt. Kapitel 3 erfasst den Istzustand der IML-Implementierung und beschreibt den Sollzustand der IML-Implementierung. Dabei konnte ermittelt werden, dass für die erfolgreiche Überführung vom Istzustand in den Sollzustand die zwei oben genannten Werkzeuge benötigt werden. Die nachfolgenden zwei Kapitel beschreiben jeweils die zwei Werkzeuge.

Kapitel 4 beschreibt die Vorgehensweise bei der Entwicklung und die Funktionsweise des ersten Werkzeugs SpecGen. Mithilfe der Parserkombinatoren wird aus einer gegebenen IML-Spezifikation ein Abstract Syntax Tree (AST) aufgebaut, aus dem dann die SKilL-Spezifikation generiert wird. Weiterhin werden die Übersetzungen von Deklarationen und Feldern sowie einzelnen *Builtin*s nach SKilL an Beispielen veranschaulicht.

Kapitel 5 beschreibt die Vorgehensweise bei der Entwicklung und die Funktionsweise des zweiten Werkzeugs CodeGen. Mithilfe des Parsers und der Intermediate Representation (IR), die beide bereits zu Beginn der Arbeit zur Verfügung standen, wird die angepasste IML-Implementierung für eine gegebene SKilL-Spezifikation generiert. Für den CodeGen wurde die IR um weitere Funktionalitäten wie beispielsweise die Implementierung der `default`-Restriktionen für die Felder erweitert. Die Parameter einer Instanziierungsmethode eines konkreten Benutzertyps müssen zudem nach der Reihenfolge ihrer Spezifizierung sortiert sein und daher anhand der Tiefensuche eingesammelt werden, damit diese für das Werkzeug `cafe++` korrekt exportiert werden können.

Damit die angepasste IML-Implementierung kompiliert werden kann, müssen die handgeschriebenen Dateien in der Bibliothek `libIML` angepasst werden. Kapitel 6 beschreibt daher die signifikanten Änderungen an den handgeschriebenen Dateien. Unter anderem wurde ein Hilfspaket für den SKilL-Zustand eingeführt, weil die Ada-Pakete der nicht direkt in IML spezifizierten Typen `Identifiers` und `SLocs` keine Abhängigkeit mit dem Ada-Paket `IML_Graphs` eingehen können, da ansonsten ein Zyklus in der *Elaboration Order* entsteht.

Anschließend wird die SKilL-basierte IML-Implementierung in Kapitel 7 getestet. Da der Linker direkt auf dem Binärstrom des IML-Binärformats arbeitet, konnten nur einzelne Quelltextdateien als Testdateien verwendet werden. Dabei wird jede Testdatei mit dem Werkzeug `cafe++` übersetzt und einem Werkzeug übergeben, das die Binärdatei wieder einliest und verarbeitet. Das zurückgelieferte Ergebnis wird gespeichert. Dieser Vorgang wird mit der originalen IML-Implementierung wiederholt, wobei das Werkzeug, das die erzeugte Binärdatei wieder einliest und verarbeitet, gleich ist. Die Ergebnispaare werden für jede Testdatei auf Differenzen untersucht. Der Test wurde mit den Werkzeugen `ccdiml` und `imlmetrics` durchgeführt und war für beide Werkzeuge erfolgreich, weil die Differenzmenge aller Ergebnispaare die leere Menge war.

Kapitel 8 bespricht die offenen Punkte, die aufgrund ihres Umfangs nicht im Rahmen dieser Arbeit gelöst wurden und gibt weitere Hinweise. Unter anderem hat sich herausgestellt, dass die Bibliothek `libIML` stark angepasst werden muss. Daher wurde das Projekt Bauhaus für die SKilL-basierte IML-Implementierung von der originalen IML-Implementierung abgespalten. Des Weiteren wurden im Laufe dieser Arbeit die Werkzeuge `cafe++`, `iml_strip`, `imlmetrics` und `imlstat` leicht angepasst.

Kapitel 9 evaluiert die SKilL-basierte IML-Implementierung und die originale IML-Implementierung bezüglich ihrer Performance an den drei Werkzeugen `ccdiml`, `imlmetrics` und `imlstat`. Hierfür wurden Projekte verwendet, die bereits im IML-Binärformat vorlagen und mithilfe des Werkzeugs `iml2sf` in das SKilL-Binärformat transformiert wurden. Die Performance-Evaluation hat ergeben, dass die Performance für die SKilL-basierte IML-Implementierung im Vergleich zu der originalen IML-Implementierung für das Werkzeug `ccdiml` gleich, für das Werkzeug `imlmetrics` etwas schlechter und für das Werkzeug `imlstat` wesentlich besser ist.

Abschließend erörtert Kapitel 10 die wesentlichen Punkte bei der Übersetzung von nicht direkt in IML spezifizierten Typen nach SKilL. Bei der Übersetzung nach SKilL ist zu beachten, dass der Typ `Storable_Lists` durch den Typ `list<Storable>` ersetzt wird. Bei einem *Variant Record* werden unabhängig der Diskriminante alle Felder in den Benutzertyp übernommen. Ein zusammengesetzter Typ als Typparameter wird nach SKilL mithilfe eines flachen Typs übersetzt.

11.1 Ausblick

Im Zuge der langfristigen Umstellung auf die SKilL-basierte IML-Implementierung empfiehlt es sich, die Bibliothek `libIML` einer Refaktorisierung zu unterziehen. Dabei sollten Ada-Pakete, die Teil eines Werkzeugs sind, verschoben und nicht verwendete Ada-Pakete entfernt werden. Weiterhin sollte die Anzahl der Implementierungen von Listen und Sets reduziert werden.

An dieser Stelle empfiehlt sich eine weitere Untersuchung, ob der nicht direkt in IML spezifizierte Typ `Identifier` nicht doch als Typdefinition gelöst werden kann. Die Instanziierungsmethode vom Typ `Identifier` prüft bei jeder Instanziierung, ob ein Identifier mit dem anzulegenden String existiert und gibt diesen gegebenenfalls zurück. Diese Überprüfung ist aufgrund des *String Pool*s in SKilL nicht mehr notwendig.

Abkürzungsverzeichnis

API Application Programming Interface.

AST Abstract Syntax Tree.

CodeGen Codegenerator.

CPU Central Processing Unit.

GNAT GNU NYU Ada Translator.

HDD Hard Disk Drive.

IML Bauhaus Intermediate Language.

IO Input/Output.

IR Intermediate Representation.

MPEG Moving Picture Experts Group.

PHP PHP: Hypertext Preprocessor.

RAM Random-Access Memory.

SKiIL Serialization Killer Language.

SpecGen Spezifikationsgenerator.

SQL Structured Query Language.

UCS Universal Character Set.

URI Uniform Resource Identifier.

UTF UCS Transformation Format.

Literaturverzeichnis

[BYM+98] I. D. Baxter, A. Yahin, L. Moura, M. Sant'Anna, L. Bier. „Clone Detection Using Abstract Syntax Trees". In: *Proceedings of the International Conference on Software Maintenance*. ICSM '98. Washington, DC, USA: IEEE Computer Society, 1998, S. 368–. ISBN: 0-8186-8779-7. URL: http://dl.acm.org/citation.cfm?id=850947.853341 (zitiert auf S. 64).

[Cze06] J. Czeranski. „The IML generator imlgen". English. Rev. 19785. Unveröffentlicht. University of Stuttgart, Institute of Software Technology, Programming Languages und Compilers, Juli 2006 (zitiert auf S. 11, 34, 41).

[EKP+99] T. Eisenbarth, R. Koschke, E. Plödereder, J.-F. Girard, M. Würthner. „Interaktive und inkrementelle Wiedergewinnung von SW-Architekturen". German. In: *1. Workshop Software Reengeneering*. Bd. 7 - 99. Fachberichte Informatik. Bad Honnef: Universität Koblenz-Landau, Mai 1999, S. 17–26. URL: http://www2.informatik.uni-stuttgart.de/cgi-bin/NCSTRL/NCSTRL_view.pl?id=INPROC-1999-57&engl=1 (zitiert auf S. 7, 9, 10).

[Fel13] T. Felden. *The SKilL Language*. English. Technical Report Computer Science 2013/06. University of Stuttgart, Institute of Software Technology, Programming Languages und Compilers: University of Stuttgart, Faculty of Computer Science, Electrical Engineering, und Information Technology, Germany, Sep. 2013, S. 43. URL: http://www2.informatik.uni-stuttgart.de/cgi-bin/NCSTRL/NCSTRL_view.pl?id=TR-2013-06&engl=1 (zitiert auf S. 7, 12–15).

[FelXX] T. Felden. „The SKilL Language — Version 1.0". English. Technical Report Computer Science. Effective July 26, 2016. Bisher unveröffentlicht. University of Stuttgart, Institute of Software Technology, Programming Languages und Compilers (zitiert auf S. 12–14).

[FW16] T. Felden, M. Wittiger. „Migrating Bauhaus from IML to SKilL". In: *Softwaretechnik-Trends* 36.2 (2016). URL: http://pi.informatik.uni-siegen.de/stt/36_2/./01_Fachgruppenberichte/WSRE2016/WSRE2016_21_paper_1.pdf (zitiert auf S. 7, 21, 58, 61).

[Gai16] S. Gaiser. *Automatisierter Vergleich von Codeklonerkennungsergebnissen*. Deutsch. Bachelorarbeit: Universität Stuttgart, Institut für Softwaretechnologie, Programmiersprachen und Übersetzerbau. Bachelorarbeit. Apr. 2016. URL: http://www2.informatik.uni-stuttgart.de/cgi-bin/NCSTRL/NCSTRL_view.pl?id=BCLR-0262&engl=0 (zitiert auf S. 7).

[Har14] F. Harth. „Plattform- und sprachunabhängige Serialisierung mit SKilL". Deutsch. Diplomarbeit. Universität Stuttgart, Fakultät Informatik, Elektrotechnik und Informationstechnik, Germany, Nov. 2014, S. 55. URL: http://www2.informatik.uni-stuttgart.de/cgi-bin/NCSTRL/NCSTRL_view.pl?id=DIP-3665&engl=0 (zitiert auf S. 16).

[has] „Anbindung von SKilL an Haskell". Bachelorarbeit. Bisher unveröffentlicht. Universität Stuttgart, Fakultät Informatik, Elektrotechnik und Informationstechnik, Germany (zitiert auf S. 16).

[Hla15] M. Hlavac. *stargazer: Well-Formatted Regression and Summary Statistics Tables*. R package version 5.2. Harvard University. Cambridge, USA, 2015. URL: http://CRAN.R-project.org/package=stargazer (zitiert auf S. 64–66).

[IEE08] „IEEE Standard for Floating-Point Arithmetic“. In: *IEEE Std 754-2008* (Aug. 2008), S. 1–70. DOI: 10.1109/IEEESTD.2008.4610935 (zitiert auf S. 13).

[KGW98] R. Koschke, J. F. Girard, M. Wurthner. „An Intermediate Representation for Integrating Reverse Engineering Analyses“. In: *Reverse Engineering, 1998. Proceedings. Fifth Working Conference on*. Okt. 1998, S. 241–250. DOI: 10.1109/WCRE.1998.723194 (zitiert auf S. 10).

[LP15] V. Layka, D. Pollak. *Beginning Scala*. 2nd. Berkely, CA, USA: Apress, 2015. ISBN: 9781484202333 (zitiert auf S. 23).

[MPO08] A. Moors, F. Piessens, M. Odersky. *Parser combinators in Scala*. CW Reports CW491. Department of Computer Science, K.U.Leuven, Feb. 2008. URL: https://lirias.kuleuven.be/handle/123456789/164870 (zitiert auf S. 23).

[Ode+] M. Odersky et al. *Scala Language Specification*. URL: http://www.scala-lang.org/files/archive/spec/ (zitiert auf S. 16).

[Ode14] M. Odersky. *Scala By Example*. Juni 2014. URL: http://www.scala-lang.org/docu/files/ScalaByExample.pdf (zitiert auf S. 23).

[pcl] *Scala Standard Parser Combinator Library*. URL: http://www.scala-lang.org/files/archive/api/2.11.2/scala-parser-combinators/#scala.util.parsing.combinator.Parsers (zitiert auf S. 23).

[Prz14] D. Przytarski. *Performance-Evaluation einer sprach- und plattformunabhängigen Serialisierungssprache*. Deutsch. Bachelorarbeit: Universität Stuttgart, Institut für Softwaretechnologie, Programmiersprachen und Übersetzerbau. Bachelorarbeit. Juni 2014. URL: http://www2.informatik.uni-stuttgart.de/cgi-bin/NCSTRL/NCSTRL_view.pl?id=BCLR-0106&engl=0 (zitiert auf S. 16).

[Rot15] J. Roth. „Reduktion des Speicherverbrauchs generierter SKilL-Zustände“. Deutsch. Masterarbeit. Universität Stuttgart, Fakultät Informatik, Elektrotechnik und Informationstechnik, Germany, Mai 2015, S. 82. URL: http://www2.informatik.uni-stuttgart.de/cgi-bin/NCSTRL/NCSTRL_view.pl?id=MSTR-0019&engl=0 (zitiert auf S. 16).

[RVP06] A. Raza, G. Vogel, E. Plödereder. „Bauhaus – A Tool Suite for Program Analysis and Reverse Engineering“. In: *Reliable Software Technologies – Ada-Europe 2006: 11th Ada-Europe International Conference on Reliable Software Technologies, Porto, Portugal, June 5-9, 2006. Proceedings*. Hrsg. von L. M. Pinho, M. González Harbour. Berlin, Heidelberg: Springer Berlin Heidelberg, 2006, S. 71–82. ISBN: 978-3-540-34664-7. DOI: 10.1007/11767077_6. URL: http://dx.doi.org/10.1007/11767077_6 (zitiert auf S. 7, 9, 10).

[SK03] S. Setzer, T. Karaca. „Erweiterung und Generierung der Zwischendarstellung IML für C++ Programme“. Deutsch. Diplomarbeit. Universität Stuttgart, Fakultät Informatik, Germany, März 2003, S. 191. URL: http://www2.informatik.uni-stuttgart.de/cgi-bin/NCSTRL/NCSTRL_view.pl?id=DIP-2048&engl=0 (zitiert auf S. 9, 10).

[ski] *SKilL on Github*. URL: https://github.com/skill-lang/skill (zitiert auf S. 16).

[Sta09] S. Staiger-Stöhr. „Kombinierte statische Ermittlung von Zeigerzielen, Kontroll- und Datenfluss“. Deutsch. Dissertation. Universität Stuttgart, Fakultät Informatik, Elektrotechnik und Informationstechnik, Germany, Dezember 2009, S. 431. URL: http://www2.informatik.uni-stuttgart.de/cgi-bin/NCSTRL/NCSTRL_view.pl?id=DIS-2009-06&engl=0 (zitiert auf S. 9, 58).

[TDB+12] S. T. Taft, R. A. Duff, R. L. Brukardt, E. Ploedereder, P. Leroy. *Ada 2012 Reference Manual. Language and Standard Libraries - International Standard ISO/IEC 8652:2012(E).* 2012. URL: http://www.ada-auth.org/standards/12rm/RM-Final.pdf (zitiert auf S. 69).

[Ung14] W. Ungur. „Nutzbarkeitsevaluation einer sprach- und plattformunabhängigen Serialisierungssprache“. Deutsch. Diplomarbeit. Universität Stuttgart, Fakultät Informatik, Elektrotechnik und Informationstechnik, Germany, Juli 2014, S. 66. URL: http://www2.informatik.uni-stuttgart.de/cgi-bin/NCSTRL/NCSTRL_view.pl?id=DIP-3603&engl=0 (zitiert auf S. 16).

Alle URLs wurden zuletzt am 30. 11. 2016 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, Unterschrift