Studiengang:	Softwaretechnik bzw. Informatik
Prüfer:	Prof. Dr. rer. nat. Erhard Plödereder
Betreuer:	Dr. Rainer Koschke
Beginn am:	11. September 2002
Beendet am:	11. März 2003
CR-Klassifikation:	D.3.4

Diplomarbeit Nr. 2048

Erweiterung und Generierung der Zwischendarstellung IML für C++ Programme

Tahir Karaca Sebastian Setzer

Institut für Informatik
Universität Stuttgart
Breitwiesenstraße 20-22
D-70565 Stuttgart

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Kapitel 1

Einleitung

T.K.

Dieses Dokument befasst sich mit der Diplomarbeit „Erweiterung und Generierung der Zwischendarstellung IML für C++ Programme". In diesem Kapitel soll zu Beginn das Bauhaus-Projekt kurz vorgestellt werden, um die Rahmenbedingungen zu veranschaulichen und um das Thema der Arbeit einordnen zu können. Darauf folgt eine Beschreibung der Aufgabenstellung. Abschließend wird ein Überblick über den Aufbau der Arbeit gegeben.

1.1 Bauhaus

T.K.

Bauhaus ist ein Projekt des Instituts für Softwaretechnologie der Universität Stuttgart, Abteilung Programmiersprachen und Übersetzerbau, und der Fraunhofer Einrichtung für Experimentelles Software-Engineering (FhG IESE), Kaiserslautern. Das Bauhaus-Projekt dient der Unterstützung der Wartung von Software, insbesondere im Bereich des Programmverstehens.

Wenn Software im dauerhaften Einsatz ist, entsteht fast zwangsläufig der Bedarf einer Wartung. Genauer gesagt gilt eine Software dann als veraltet, wenn sie den an sie gestellten Anforderungen nicht mehr genügt. H. Sneed hat diese Tatsache passend mit dem Satz

> *„Software veraltet in dem Maße, wie sie mit der Wirklichkeit nicht Schritt hält."*

zusammengefasst. Die Gründe für eine Alterung von Software können in die drei Gruppen untergliedert werden: Fehler, geänderte Umweltbedingungen und geänderte Kundenanforderungen. Aus diesen drei Gründen lassen sich die drei Wartungsarten ableiten, mit denen sich die Wartung definieren lässt:

- korrektive,
- anpassende und
- perfektionierende Wartung.

Unabhängig vom Wartungsgrund ist es die Aufgabe des Wartungsingenieurs, die Wartung der Software durchzuführen. Dieser steht damit aber oft vor einer nicht-trivialen Aufgabe, da:

- die Wartung nicht unbedingt vom Entwickler der Software durchgeführt wird.
- eine Dokumentation der Software entweder überhaupt nicht existiert oder veraltet sein könnte.
- die Architektur der Software durch bereits durchgeführte Wartungsarbeiten verwässert worden sein könnte.

In jedem Fall fehlen dem Wartungsingenieur Informationen über den aktuellen Stand der zu wartenden Software. In der Praxis ist es sehr unwahrscheinlich, dass der Wartungsingenieur perfekt über die Software informiert ist. So muss er sich diese Informationen auf irgendeine Weise beschaffen, was oft den Hauptanteil des Wartungsaufwands ausmacht. Im Vergleich mit dem Entwicklungsaufwand beträgt der Wartungsaufwand während der gesamten Lebensdauer der Software 67% bis 80% des Gesamtaufwands [3].

An dieser Stelle soll das Bauhaus-Projekt den Wartungsingenieur unterstützen. Das genaue Ziel des Bauhaus-Projektes ist die Entwicklung von

- Beschreibungsmitteln,
- Analysemethoden und
- zugehörigen Werkzeugen,

die es dem Wartungsingenieur erleichtern, semi-automatisch

- die Soll- und Ist-Architektur von Altsystemen herzuleiten,
- die Architektur bei Änderungen am System nachzuführen und
- geplante Auswirkungen von Programmänderungen abzuschätzen.

Um dieses Ziel zu erreichen werden Beschreibungsmittel sowie Analysemethoden und -werkzeuge erforscht, die es ermöglichen, Architekturen zu extrahieren, darzustellen und zu manipulieren[32].

IML

Das Projekt Bauhaus baut auf einer universellen Zwischendarstellung des zu analysierenden Programms auf, auch *Intermediate Language* (IML) genannt. Die IML repräsentiert syntaktische und semantische Informationen über das Programm.

Der Aufbau der IML ist vergleichbar mit dem AST eines Übersetzers. Es handelt sich um einen Graphen, der die Programminformationen in seinen Knoten speichert. Die IML vereint jedoch zwei zentrale Eigenschaften, die ein gewöhnlicher AST nicht hat:

1. *Abstraktion von der Programmiersprache*

   Die Darstellung von Programmen in der IML geschieht unabhängig von der Programmiersprache, in der das Programm implementiert ist. D.h. es existiert eine Version der IML, die mächtig genug ist, verschiedene Programmiersprachen einheitlich zu umfassen.

2. *Quellennahe Darstellung*

   Die Darstellung in der IML enthält gleichzeitig genug Informationen über den ursprünglichen Quelltext, damit Meldungen an den Wartungsingenieur in einer nachvollziehbaren und nicht verwirrenden Form ausgegeben werden können.

In Abbildung 1.1 wird als Skizze dargestellt, wie die IML generiert und weiterverarbeitet wird. Ausgehend von den Quelltextdateien des zu analysierenden Programms wird mit Hilfe der Bauhaus-Werkzeuge cafe und jafe eine IML-Darstellung der einzelnen Übersetzungseinheiten generiert. Um eine Darstellung eines gesamten Programms zu erhalten, werden danach die einzelnen IML-Dateien mit dem dem sog. IML-Linker imllink zu einer umfassenden Darstellung zusammengefasst. Ist die IML soweit generiert, können die Basisanalysen von Bauhaus beginnen, das Programm zu analysieren. Momentan stehen z.B. eine Kontrollflussanalyse, eine Points-To-Analyse oder eine Seiteneffekt-Analyse zur Verfügung.

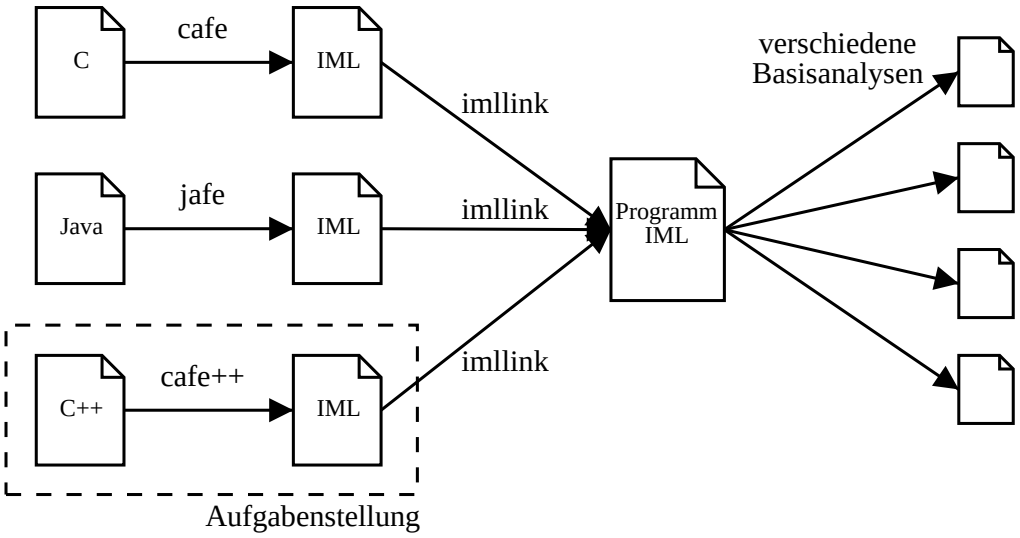


Abbildung 1.1: Generieren und Verarbeitung der IML

Bisher ist in der IML die Möglichkeit vorgesehen, Programme in ANSI-C und Java darzustellen. Für diese Programmiersprachen existieren in Bauhaus bereits die Front-Ends cafe und jafe, die die IML-Darstellung von den jeweiligen Quelltextdateien erstellen können. Der Name cafe steht dabei für „C Analysing Front-End" und jafe für „Java Analysing Front-End".

1.2 Aufgabenstellung

T.K.

Die Aufgabenstellung des Projekts ist es, einen Übersetzer zu entwickeln, der aus einem C++ Programm dessen Repräsentation in IML generiert. Diese Aufgabe kann in zwei Schwerpunkte unterteilt werden. Erstens ist die IML-Darstellung so zu erweitern, dass auch die C++ spezifischen Sprachelemente in

der IML ausgedrückt werden können und zweitens ist ein Übersetzer zu Entwickeln, der die IML-Darstellung für beliebige C++ Programme generiert. In Abbildung 1.1 auf der vorherigen Seite ist im gestrichelten Kasten angedeutet, wo diese Diplomarbeit innerhalb des Bauhaus-Projekts einzuordnen ist.

Wie schon anfangs beschrieben, ist die IML zum Zeitpunkt des Projektbeginns fähig, ANSI C Programme darzustellen. Die Konzeption der IML ist jedoch auch dafür ausgelegt, sowohl prozedurale, als auch objektorientierte Programmiersprachen darstellen zu können. Eine Erweiterung der IML, um auch Java Programme repräsentieren zu können, war bei Projektbeginn Thema einer parallel laufenden Diplomarbeit, die von Markus Knauß durchgeführt wurde. Diese Diplomarbeit wurde am 1. Oktober 2002 abgeschlossen. Da die C++ Erweiterung der IML auf der von Java aufbauen soll, ist dieser Termin auch für dieses Projekt wichtig. Weiterhin sind folgende funktionale und nicht-funktionale Anforderungen bei der Erweiterung der IML gegeben:

- Die Erweiterung der IML soll die bisherigen Konzepte, die hinter der Darstellung stehen, berücksichtigen und in deren Sinne fortgeführt werden. Z.B. soll die C++ Erweiterung eine spätere Erweiterung um andere Programmiersprachen wie Ada 95 nicht blockieren.
- Allgemein strebt die IML primär Quellennähe und Analysierbarkeit an; eine Eignung für die Code-Generierung ist sekundär.
- Die Integration der Konzepte zur Modellierung von C++ soll so vorgenommen werden, dass möglichst wenig Änderungen in den nachfolgenden, bereits existierenden Analysen notwendig werden.
- Es wird großer Wert auf die Qualität der Dokumentation gelegt.

Die Entwicklung eines Übersetzers von C++ nach IML ist der zweite Schwerpunkt dieser Diplomarbeit. Auch hier sind einige Anforderungen an das Produkt vorgegeben:

- Der Übersetzer soll von C++ Quelldateien ausgehend die IML eines kompletten Programms erzeugen. Wenn die Übersetzungseinheiten in einzelne IML-Dateien übersetzt werden, können sie später mit dem bereits existierenden IML-Linker aus dem Bauhaus-Projekt zu einer gesamten Programmrepräsentation kombiniert werden.
- Da die syntaktische und semantische Analyse von C++ Quelltext sehr aufwändig ist und den zeitlichen Rahmen einer Diplomarbeit bei weitem überschreiten würde, soll ein bereits existierendes C++ Analysewerkzeug dafür genutzt werden. Beispielsweise kommt für diese Aufgabe das Front-End des C++ Übersetzers aus der GNU Compiler Collection in Frage.
- Die Implementierung des Ada Codes für die IML-Konstrukte muss mit Hilfe des in Bauhaus existierenden Generators erfolgen. Das hat den Hintergrund, dass die IML-Darstellung effizient gespeichert und geladen werden soll.

- Es wird großer Wert auf die Qualität des Entwurfs und der Implementierung gelegt.
- Es ist ein umfangreicher Test gefordert, um die Produktqualität zu sichern.
- Sämtliche Programmdokumentationen sind in Englisch zu verfassen.

Der Name des Projekts ist „cafe++" in Anlehnung an die Front-Ends „cafe" und „jafe" und seht für „C++ Analysing Front-End".

Aufgrund der besonders großen Implementierungslastigkeit dieser Aufgabenstellung ist die Diplomarbeit für zwei Autoren ausgeschrieben worden. Beide Autoren sollen die Aufgabenstellung gemeinsam bearbeiten.

1.3 Aufbau der Arbeit

T.K.

Das zweite Kapitel widmet sich der IML und beschreibt diese näher. Auch der erste Schwerpunkt der Diplomarbeit, die Erweiterung der IML, wird im zweiten Kapitel behandelt. Kapitel 3 geht auf das verwendete C++ Front-End ein, indem es dessen Auswahl beschreibt und dann die Eigenschaften des ausgewählten Front-Ends vorstellt. In Kapitel 4 wird der Entwurf und die Realisierung von cafe++ erläutert, bevor sich Kapitel 5 mit dem Test von cafe++ befasst. Kapitel 6 enthält die Bedienungsanleitung von cafe++. Abschließend wird in Kapitel 7 der Projektverlauf und die Ergebnisse zusammengefasst und ein Ausblick auf die Zukunft von cafe++ gegeben. In den Anhängen findet sich letztendlich ein Glossar und ein kleines Beispiel eines vollständigen IML-Graphen.

1.3.1 Anforderungen an den Leser

T.K.

Vom Leser wird erwartet, dass er grundlegende Kenntnisse über den Übersetzerbau besitzt. Zusätzlich sollten die Syntax, die Semantik und die Konzepte hinter den Programmiersprachen C, C++ und Java bekannt sein. Einzelheiten der Programmiersprachen, wie beispielsweise das Konzept der Ausnahmen, werden in dieser Diplomarbeit nicht genau vorgestellt.

1.3.2 Notation

S.S.

In dieser Ausarbeitung werden einige Notationsstile eingesetzt, die hier erläutert werden sollen:

- Quelltexte in Programmiersprachen, Kommandozeileneingaben und Dateinamen sind in Schreibmaschinenschrift gesetzt, z.B. `void`.
- Jede Erwähnung von IML-Knotentypen und deren Attribute ist in serifenloser Schrift gesetzt, z.B. IML_Root.
- Alle Begriffe wurden aus dem Deutschen gewählt, soweit dies möglich war. Die Übersetzung der C++ Sprachelemente wurde an die deutsche

Übersetzung des C++ Standardwerks von Bjarne Stroustrup [31] angelehnt. Die entsprechenden englischen Begriffe können dem Glossar auf Seite 181 entnommen werden.

- Die Abbildungen sind in der UML-Notation verfasst, soweit nichts anderes angegeben wird.
- Für Diplomarbeiten, die in Form einer Gruppenarbeit zugelassen werden, schreibt die Prüfungsordnung vor, dass der zu bewertende Beitrag des einzelnen Kandidaten aufgrund objektiver Kriterien, die eine eindeutige Abgrenzung ermöglichen, deutlich unterscheidbar und bewertbar sein muss [26, 27]. In dieser Diplomarbeit wird bei jedem Kapitel und bei jedem Unterkapitel als Marginalie das Namenszeichen des Autoren angegeben, womit eine klare Zuordnung möglich ist.
- Referenzen auf Abschnitte oder Paragraphen des C++ Standards [14] erfolgen in der Form [14, Abschnitt (kapitel.abschnitt)-paragraph], wobei „kapitel.abschnitt" für das Kürzel des Abschnitts steht, das im Standard am Beginn jedes Abschnitts in eckigen Klammern steht. „paragraph" ist die Nummer des relevanten Paragraphen in dem Abschnitt und wird weggelassen, falls der ganze Abschnitt relevant ist.

Kapitel 2

IML für C++

T.K.

Dieses Kapitel behandelt das Thema der IML-Erweiterung, die ein Hauptbestandteil der Diplomarbeit darstellt. Zu Beginn wird dazu eine Einführung in die bisherige IML gegeben, bevor das Vorgehen bei der Erweiterung der IML besprochen wird. Im Anschluss daran folgt eine Diskussion über die verschiedenen Aspekte der Modellierung und eine vollständige Spezifikation der endgültigen IML.

2.1 Bisherige IML

T.K.

An erster Stelle sollen hier die grundlegenden Eigenschaften der IML beschrieben werden. Kapitel 2.1.1 gibt dazu eine Einführung in die IML und behandelt die Fragen:

- Aus welchen Elementen ist die IML aufgebaut?
- Wie sehen diese aus und wie sind sie angeordnet?

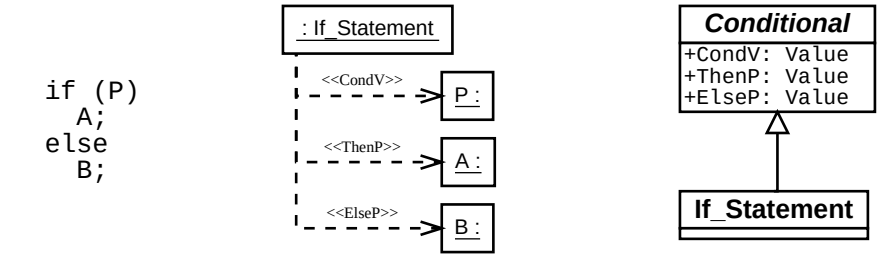
Danach geht Kapitel 2.1.2 auf die Entwicklungsgeschichte der IML ein.

2.1.1 Aufbau der IML

T.K.

Wie in der Einleitung bereits angedeutet wurde, handelt es sich bei der IML um eine universelle Zwischendarstellung von Quelltexten [32]. Die Zwischendarstellung selbst ist ein Graph, der aus Knoten und Kanten besteht. Die Knoten enthalten Attribute, welche entweder Verweise auf andere Knoten, also Kanten, oder konkrete Werte enthalten können. Falls ein Attribut einen konkreten Wert enthalten soll, stehen für dessen Darstellung eine Reihe von vorgegebenen Datentypen zur Verfügung. Im Wesentlichen sind dies Zeichenketten, Ganzzahl-, Fließzahl- und boolesche Typen. Verweist ein Knoten auf einen anderen IML-Knoten, wird dafür eine Kante eingesetzt. Die Attribute können zusätzlich nicht nur einfache Objekte enthalten, sondern auch geordnete Listen („list of") und ungeordnete Mengen („set of") von Objekten. Die Kanten in der IML sind alle gerichtet und enthalten selbst keine Attribute.

Jeder Knoten in der IML ist von einem IML-Knotentyp instantiiert, welcher genau dessen Semantik festlegt. Beispielsweise stellen alle Knoten vom Typ



(a) C-Quelltext (b) IML-Graph (c) IML-Typhierarchie

Abbildung 2.1: Darstellung der `if`-Anweisung in der IML

If_Statement (siehe Abbildung 2.1) bedingte Anweisungen mit der Semantik des „if"-Sprachelements in der Programmiersprache Ada oder C dar. Der Knotentyp definiert weiterhin auch die genaue Anzahl, die Namen und die Typen der Attribute eines Knotens. Das If_Statement hat beispielsweise ein Attribut CondV, welches auf den Knoten verweist, der die Bedingung modelliert. Das Attribut ThenP modelliert den Code, der ausgeführt werden soll, falls die Bedingung wahr ist, wogegen ElseP den Code modelliert, der bei einer unwahren Bedingung auszuführen ist. Alle diese Attribute des If_Statements sind vom Typ Value, einem weiteren IML-Knotentyp. Da Value kein eingebauter IML-Knotentyp ist, zeigen von jedem If_Statement-Knoten drei gerichtete Kanten hin zu drei Value-Knoten.

2.1.1.1 IML-Typhierarchie

T.K.

Das Typsystem in der IML ist nach dem objektorientierten Paradigma aufgebaut. Es können also IML-Typen von bestehenden IML-Typen abgeleitet werden, was einen wesentlichen Aspekt der Modellierung in der IML darstellt, da mit Hilfe genau dieses Modellierungsmerkmals zum einen eine quellennahe Darstellung des Quelltextes und zum anderen eine vereinheitlichte Darstellung über verschiedene Programmiersprachen hinweg realisiert wird. Für das oben erwähnte If_Statement bedeutet das, dass es einen IML-Knotentyp Conditional gibt, der ganz allgemein für eine bedingte Verzweigung im Quelltext steht und von dem das If_Statement abgeleitet ist. Das If_Statement stellt also eine Spezialisierung einer allgemeinen Verzweigung in der IML dar. Die Attribute CondV, ThenP und ElseP erbt das If_Statement ebenfalls vom Conditional-Typ.

Das Typsystem mit der Objektorientierung, die der IML-Modellierung zu Grunde liegt, kann folgendermaßen genauer definiert werden:

- Jeder Knotentyp außer IML_Root ist von genau einer Oberklasse abgeleitet, es existieren somit keine Mehrfachableitungen.
- Alle IML-Knoten sind in einer Typhierarchie integriert. Das bedeutet, dass es eine gemeinsame Oberklasse für alle IML-Knoten gibt, welche die Klasse IML_Root ist.

- Als einzige Ausnahmen davon sind die eingebauten Datentypen (Zeichenketten, Ganzzahl-, Fließzahl-, boolesche und weitere Typen) von der Objekthierarchie ausgeschlossen.
- Es wird zwischen abstrakten und konkreten Typen unterschieden. Der Unterschied ist, dass abstrakte Typen Obertypen definieren, von denen keine Objekte instantiiert werden können.
- Die Vererbungshierarchie der IML-Klassen hat einen reinen Spezialisierungscharakter, d.h. jede Ableitung erweitert nur die Schnittstelle eines Knotentyps oder konkretisiert variable Attribute. Damit ist jedes Objekt eines Obertyps immer ersetzbar durch Objekte ihrer Untertypen, ohne dass Probleme beim Zugriff auf das Objekt auftauchen können.
- Im Unterschied zur üblichen Objektorientierung verwendet die IML-Modellierung keine Nachrichten, die zwischen den Objekten oder Klassen ausgetauscht werden. Als Konsequenz verfügen weder Klassen noch Objekte der IML über Methoden. Einzige Ausnahme sind die bestehenden `set`- und `get`-Methoden der IML-Klassen, die einfache Zugriffsfunktionen implementieren, die Werte von Attributen lesen und schreiben können.
- Das Überschreiben von Attributen ist nicht möglich.
- Die Information, ob ein Attribut eine Assoziation oder eine Komposition darstellt, wird durch „semantische" und „syntaktische" Kanten ausgedrückt.

Abbildung 2.2 zeigt die IML-Typhierarchie, wie sie für die IML bei Beginn dieser Diplomarbeit aufgebaut war. Die Abbildung soll dabei einen groben Überblick über die Typen vermitteln, weswegen die Hierarchie der Übersichtlichkeit wegen an manchen Stellen beschnitten wurde.

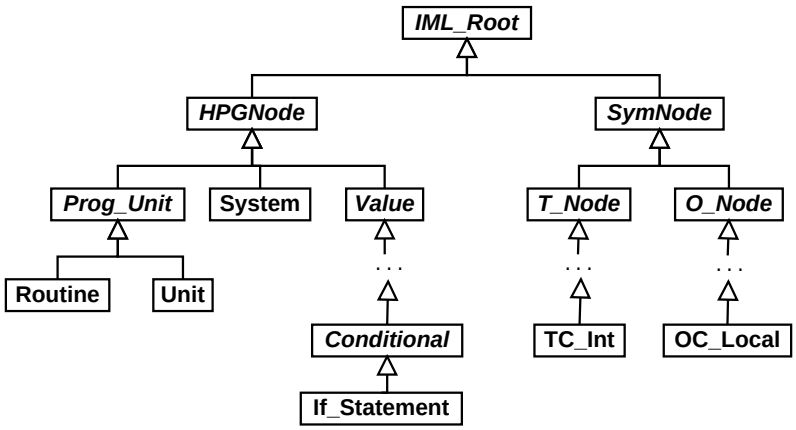


Abbildung 2.2: Übersicht über die IML-Typhierarchie für die Modellierung von C-Programmen

T.K.

2.1.1.2 HPG-Knoten

Der Hierarchische Programm Graph (HPG) ist ein Teilgraph eines IML-Graphen, der den ausführbaren Teil eines Programms beschreibt [30]. Alle Knoten dieses Teilgraphs sind vom Knotentyp HPGNode abgeleitet (siehe Abbildung 2.2). Die direkt von HPGNode abgeleiteten Klassen teilen die Klassenhierarchie unterhalb von HPGNode nochmals genauer auf:

- Prog_Unit
  Alle Knotentypen unter Prog_Unit modellieren entweder Übersetzungseinheiten, Klassen oder ausführbare Einheiten im Quelltext. Ausführbare Einheiten sind z.B. Funktionen in C.
- System
  Objekte vom Typ System bilden die Wurzelknoten von vollständigen IML-Graphen. Erzeugt werden Objekte vom Typ System lediglich vom IML-Linker (siehe Kapitel 2.1.1.4).
- Value
  Value ist die abstrakte Oberklasse aller Ausdrücke, die zu einem Wert ausgewertet werden können. Der Typ des Wertes ist im Ausdruck definiert durch das Attribut „EType: T_Node" von Value.

Innerhalb des HPG existieren sowohl semantische als auch syntaktische Kanten. Die syntaktischen Kanten im HPG spannen eine Baumstruktur auf, die einem Abstrakten Syntaxbaum (AST) entspricht. Wurzelknoten dieser Bäume sind entweder Unit-Knoten oder ein System-Knoten. Die semantischen Kanten werden dagegen verwendet, um im HPG zusätzliche Querverweise darstellen zu können.

T.K.

2.1.1.3 Symboltabelle

Ein IML-Graph beinhaltet außer den ausführbaren Anteilen eines Programms durch den HPG auch eine Symboltabelle. Die Symboltabelle besteht, wie der Rest des IML-Graphen, ebenfalls aus IML-Knoten. Die Knoten der Symboltabelle sind in der Typhierarchie alle von SymNode abgeleitet (siehe Abbildung 2.2 auf der vorherigen Seite). Wie direkt aus der Abbildung ersichtlich ist, umfasst die Symboltabelle zwei Bereiche. Zum einen sind dies die Typen, modelliert durch Knoten mit der Oberklasse T_Node, und zum anderen die Deklarationen im Quelltext, welche durch Knoten mit der Oberklasse O_Node modelliert werden. Ein Beispiel für einen modellierten Typ ist der Knotentyp TC_Int, der die Ganzzahltypen im Quelltext darstellt. OC_Local ist dagegen ein Beispiel für Deklarationen in der IML. Ein OC_Local-Knoten steht für die Deklaration einer lokalen Variable im Quelltext.

Die Namen der Knotentypen der Symboltabelle richten sich nach einer Konvention, die den Präfix des Namens festlegt. Der erste Buchstabe legt fest, ob es sich um einen Typ („T") oder eine Deklaration („O") handelt. Danach folgt ein Kürzel für die Programmiersprache und ein „_", gefolgt von dem eigentlichen Namen. Für die Programmiersprachen existieren bisher „C" und „Ada" für die gleichnamigen Programmiersprachen und „J" für Java.

Die Kanten vom HPG zur Symboltabelle, von der Symboltabelle zurück zum HPG und innerhalb der Symboltabelle sind ausschließlich semantische Kanten.

2.1.1.4 IML-Linker

S.S.

Wenn der IML-Graph einer einzelnen Übersetzungseinheit erzeugt wird, bildet ein Objekt vom Typ Unit den Wurzelknoten. Ein solcher Graph kann unvollständig sein, da er deklarierte, aber nicht definierte Sprachelemente, z.B. Funktionen, enthalten und verwenden kann. Es ist die Aufgabe des IML-Linkers, mehrere unvollständige IML-Graphen zu einem vollständigen zusammenzufügen und dabei die nicht definierten Abhängigkeiten aufzulösen. Erst wenn dies geschehen ist, wird vom IML-Linker ein System-Knoten eingesetzt, welcher das gesamte Programm repräsentiert. Es gibt also genau ein Objekt vom Typ System in der IML-Darstellung eines Programms.

Damit der Linker unterscheiden kann, welche Deklarationen definiert wurden und welche noch undefiniert sind, enthalten Unit-Knoten zwei Attribute. Das Attribut „Provided_Definitions: set of OC_Entity" enthält die definierten und exportierten Deklarationen. Alle undefinierten Deklarationen sind dagegen im Attribut „Unresolved_Declarations: set of OC_Entity" aufgeführt. Die hauptsächliche Arbeit des Linkers besteht nun darin, diese beiden Mengen über mehrere Übersetzungseinheiten hinweg ineinander aufzulösen. Dazu werden die Deklarationen anhand ihres Namens miteinander verglichen.

Die Deklaration „`extern int i;`" bedeutet nach dem C Standard [12] beispielsweise noch nicht, dass eine entsprechende Variable existieren muss. Erst nachdem `i` benutzt wird, ist es in den Unresolved_Declarations enthalten und wird gelinkt. Dazu muss sie jedoch in einer anderen Übersetzungseinheit z.B. als „`int i = 12345;`" definiert sein. Eine solche Definition würde dann in den Provided_Definitions der definierenden Übersetzungseinheit auftauchen.

2.1.1.5 Kontrollfluss

T.K.

Der Kontrollfluss wird in der IML nicht vom Front-End modelliert, sondern von einer IML-Analyse bestimmt. Diese Analyse bringt die bestehenden IML-Knoten im HPG in eine Ausführungsreihenfolge. Dazu müssen im HPG bereits alle Knoten vorhanden sein, die für die Ausführung benötigt werden, da die Kontrollflussanalyse lediglich eine Reihenfolge festlegt aber keine neuen Knoten einfügt.

2.1.2 Entwicklung der IML

T.K.

Die grundlegende Idee zur IML ist in [7] dokumentiert. Im Laufe der Zeit wurde die IML ständig weiterentwickelt und um die Unterstützung zusätzlicher Sprachen erweitert. Abbildung 2.3 auf der nächsten Seite zeigt die Entwicklung der IML über den Verlauf der Zeit. Die erste Version der IML wurde 1996 in der Studienarbeit von Martin Württhner entwickelt, mit dem Hintergrund, Ada 83-Programme darzustellen [32]. Jürgen Rohrbach hat daraufhin

1998 die IML für die Darstellung von C Programmen erweitert [30]. Die Unterstützung von Ada 83-Programmen wurde nach diesem Schritt jedoch aufgegeben. Danach wurde die IML von der Abteilung Programmiersprachen und Übersetzerbau der Universität Stuttgart bis zum aktuellen Zeitpunkt gewartet und weiterentwickelt. Markus Knauß erweiterte schließlich die IML 2002 für die Darstellung von Java Programmen [19]. Im aktuellen Entwicklungsschritt wurde nun in dieser Diplomarbeit die IML zusätzlich für die Darstellung von C++ Programmen erweitert und unterstützt letztendlich die Darstellung von C, C++ und Java Programmen.

Die Planungen für die Zukunft sehen eine Weiterentwicklung der bisherigen IML, auch IML1 genannt, in Richtung der IML2 vor. Die IML2 ist eine grundlegende Neugestaltung der IML im Hinblick auf eine bessere Darstellung und orientiert sich eher an den Bedürfnissen von Werkzeugen für das Programmverstehen als an der internen Darstellung eines Übersetzers. Die Umsetzung der IML2 soll aufgrund des Aufwands für das Anpassen der bestehenden Bauhaus-Software durch eine schrittweise Migration geschehen. Die notwendigen Änderungen hierfür sind in der IML1 teilweise bereits durchgeführt aber noch nicht vollständig abgeschlossen worden. Besonders der HPG enthält bereits viele Konzepte der IML2.

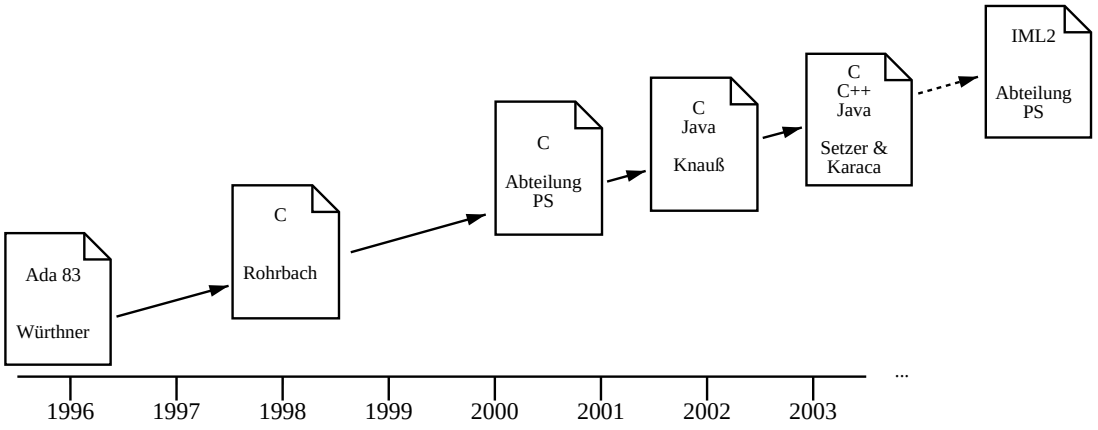


Abbildung 2.3: Entwicklung der IML vom Beginn bis heute

Während der Weiterentwicklung wurde die Implementierung der IML aus dem konkreten Quelltext von Bauhaus herausgelöst und wird nun in einer allgemeinen Form in der so genannten IML-Spezifikation definiert. Diese IML-Spezifikation ermöglicht es, zentral in einer Datei, die IML zu spezifizieren und die Quelltexte für die IML-Konstrukte in Bauhaus mit Hilfe eines Generators erzeugen zu lassen. Im weiteren Text dieser Diplomarbeit ist von dieser Datei die Rede, wenn von der IML-Spezifikation gesprochen wird.

2.2 Vorgehen bei der Erweiterung

T.K.

Die Erweiterung der IML bedeutete eine Erweiterung der bestehenden Bauhaus-Software, insbesondere der IML Spezifikation. Damit ergab sich die Si-

tuation einer typischen Wartungsarbeit mit den damit verbundenen Eigenschaften. Wie diese Aufgabe angegangen wurde, beschreiben die nachfolgenden Kapitel.

2.2.1 Ausgangsbasis

T.K.

Zu Beginn der IML-Erweiterung waren zwei IML-Spezifikationen verfügbar. Zum einen existierte die IML für C, die auch in der aktuellen Bauhaus-Software im Einsatz war. Diese IML-Spezifikation bildete die grundlegende Ausgangsbasis, da sie auf dem aktuellen Stand war und auch von den Mitarbeitern der Abteilung weiterentwickelt wurde. Verwendet wurde die Revision 1.206 der Spezifikation.

Die zweite IML-Spezifikation war die IML für Java und C, die aus dem jafe-Projekt entstanden ist [19]. Diese IML-Version baute auf einer älteren Version der IML für C auf und erweiterte diese um die Modellierung von Java Programmen. Alle Erweiterungen der IML für Java sind detailliert beschrieben für die aktuelle Version 1.1.

2.2.2 Anforderungen an die Erweiterung

T.K.

Die Erweiterung der IML für die Darstellung von C++ Programmen ist an folgende funktionale und nicht-funktionale Anforderungen gebunden. Diese sind zum Teil in der Aufgabenstellung genannt, zum Teil wurden sie von den Bearbeitern der Diplomarbeit ausgearbeitet, um die Erweiterung in eine sinnvolle Richtung zu lenken.

- Die Erweiterung der IML soll in einer Weise durchgeführt werden, die die vollständige Darstellung von C++ Programmen ermöglicht. Zudem sollen weiterhin auch noch C und Java Programme dargestellt werden können. Auch soll die Erweiterung eine spätere Erweiterung um andere Programmiersprachen, wie Ada 95, nicht blockieren.

- Die Darstellung der IML strebt primär Quellennähe und Analysierbarkeit an. Eine weitere Eignung für die Quelltext-Generierung ist zu berücksichtigen, jedoch ein sekundäres Ziel.

- Die Darstellung semantisch gleicher Sprachelemente soll in der IML auch durch gleiche Elemente dargestellt werden, es wird also eine vereinheitlichte Darstellung über verschiedene Programmiersprachen angestrebt. Diese Anforderung steht im Konflikt zur Quellennähe und ist im Zweifelsfall zweitrangig.

- Die Darstellung von C++ Programmen soll sich am ISO Standard für C++ [14] orientieren. Weiterhin ist die Unterstützung für Dialekte von verbreiteten C++ Compilern eine gewünschte, aber nicht kritische Anforderung. Diese sind im Wesentlichen die Microsoft Erweiterungen, eingesetzt im Microsoft Visual C++ Compiler, die Borland Erweiterungen, eingesetzt im Borland C++ Builder, die IBM Erweiterungen, eingesetzt im IBM VisualAge C++ Compiler und die GNU Erweiterungen, eingesetzt im GNU C++ Compiler.

- Die Integration der Konzepte zur Modellierung von C++ soll so vorgenommen werden, dass möglichst wenig Änderungen in den nachfolgenden, bereits existierenden Analysen und Werkzeugen notwendig werden.
- Die Erweiterung der IML soll die bisherigen Konzepte, die hinter der Darstellung stehen, berücksichtigen und in deren Sinne fortgeführt werden. Dazu zählen folgende Konzepte:
  - Die IML besitzt eine Ausführungssemantik, d.h. die IML sollte so gestaltet werden, dass potentiell eine Ausführung auf einer (virtuellen) IML-Maschine möglich ist.
  - Semantische Analysen und Namensauflösungen sollten in der IML-Darstellung von Programmen nicht mehr notwendig sein.
  - Die Eigenschaften der IML-Knotentyphierarchie müssen auch weiterhin eingehalten werden.
  - Attribute von IML-Klassen sollen möglichst konkret sein, d.h. die Typhierarchie ausnutzen und möglichst konkrete Typen bestimmen. Damit kann die IML-Repräsentation möglichst sicher, d.h. ohne Typkonvertierung, traversiert werden.

2.2.3 Durchführung

T.K.

Die erste Frage, die bei der IML-Erweiterung zu klären war, betraf die Wahl der Version der IML-Spezifikation. Einerseits war die aktuelle Version der IML für C als Vorgabe vorhanden, andererseits sollten auch die Erweiterungen für Java übernommen werden, die jedoch auf einer älteren Version der IML für C basierten. Für dieses Projekt wurde dann entschieden, auf der IML für C aufbauend, die Java Erweiterungen nochmals an der aktuellen Version der IML für C umzusetzen. Dieses Vorgehen erwies sich zusätzlich als vorteilhaft, da die Java Erweiterungen der IML in diesem Projekt nochmals überarbeitet und erweitert wurden, dabei die bestehende IML für C jedoch weitestgehend unverändert blieb.

Als problematisch erwies sich weiterhin die Dokumentation der IML für C. Leider waren nicht ausreichend Informationen vorhanden, wie C Sprachelemente in IML übersetzt werden und wodurch sich einige IML-Knoten und Teile der IML-Vererbungshierarchie auszeichneten. Die vorhandene Dokumentation der IML für C war vorhanden in Form der Studienarbeit für die ursprüngliche C IML [30] und in Form von Kommentaren in der Spezifikationsdatei. Die Studienarbeit beschreibt ausführlich die Konzepte und grundlegenden Ideen, die die Modellierung für C Programme in der IML festlegen, ist aber leider nicht mehr auf dem aktuellen Stand, da in der Zwischenzeit die IML für C stark weiterentwickelt wurde. Die Kommentare in der IML Spezifikationsdatei sind hingegen auf dem aktuellen Stand der IML-Version, beschreiben die IML allerdings nur sehr grob.

Um dieses Defizit an Informationen zu bewältigen, wurden zuerst kleine Beispielprogramme in C in IML übersetzt und das Ergebnis mit dem Werkzeug iml2html aus dem Bauhaus-Projekt inspiziert, um ein Verständnis für die

aktuelle IML-Modellierung von C Programmen zu erlangen. Als danach die Erweiterung der IML für C++ zunehmend weitere Fragen bezüglich der Bedeutung verschiedener IML-Knoten und der IML-Vererbungshierarchie hervorbrachte, wurden diese Fragen in Gesprächen mit Hilfe der Mitarbeiter der Abteilung geklärt. Besonders schwere Fragen und Fragen mit widersprüchlichen Antworten wurden mit Hilfe eines Fragebogens geklärt, der ebenfalls von den jeweiligen Mitarbeitern der Abteilung beantwortet wurde.

Als weitere qualitätssichernde Maßnahme wurden zusätzlich zwei Reviews durchgeführt. Die C++ Erweiterungen der IML wurden darin von den Mitarbeitern der Abteilung und Herrn Prof. Plödereder begutachtet. Ziel dieser Reviews war es, mögliche Probleme in der Modellierung zu identifizieren und ungeklärte Anforderungen zu klären. Die Reviews wurden in Form von Structured Walkthroughs abgehalten, was sich als sehr geeignet erwies.

2.3 Aspekte der C++ Modellierung

T.K.

An dieser Stelle sollen die Aspekte der IML-Erweiterung vorgestellt werden, zu denen Entscheidungen getroffen werden mussten. Dabei werden die Entscheidungsgrundlagen und die getroffenen Entscheidungen angegeben. In Kapitel 2.4 werden dagegen alle Erweiterungen vorgestellt, ohne den Entscheidungsprozess nochmals zu dokumentieren. Diese Aufteilung in ein „Warum" und ein „Wie" soll das Verständnis für die IML erleichtern, da je nach Anwendung (z.B. für die Wartung von cafe++) meist nur eines der beiden Kapitel von Interesse ist.

In Kapitel 2.2.2 wurden bereits die Anforderungen genannt, die bei der Erweiterung zu erfüllen sind. Durch diese Anforderungen sind die Rahmenbedingungen im Voraus bereits recht genau vorgegeben. Allerdings bleibt trotzdem noch Spielraum für die genaue Auslegung der Erweiterungen. Die maßgeblichen Gründe hierfür können in drei Punkten zusammengefasst werden:

- *Unvollständige Spezifikation der Anforderungen*

  Die Anforderungen an die Erweiterung der IML sind nicht vollständig genug, um jede mögliche Gestaltung der IML bewerten zu können. Besonders in Detailfragen fehlen dadurch teilweise Entscheidungsgrundlagen. Der Grund für diese Unklarheiten liegt in der fehlenden Relevanz mancher untergeordneten Kriterien, wie z.B. dem Speicherbedarf der IML-Repräsentation.

  In solchen Fällen wurden die genauen Vor- und Nachteile der jeweiligen Modellierung ausgearbeitet, um aufgrund dessen eine Entscheidung treffen zu können.

- *Konfliktäre Anforderungen*

  Wie oft beim Bearbeiten von Spezifikationen, sind sind auch bei der IML-Erweiterung nicht alle Anforderungen orthogonal zueinander, sondern teilweise konfliktär. Beispielsweise wird gleichzeitig eine quellennahe Darstellung und eine vereinheitlichte Darstellung über verschiedene Pro-

grammiersprachen hinweg angestrebt. Diese beiden Anforderungen sind offensichtlich nicht gleichzeitig im vollem Umfang umsetzbar.

Um im Falle solcher konfliktären Anforderungen Entscheidungen treffen zu können, ist es notwendig, die konfliktären Anforderungen einer Gewichtung zu unterziehen. Eine solche Gewichtung wurde bereits grob bei der Vorstellung der Anforderungen vorgenommen. Allerdings konnte die Gewichtung nicht immer genau genug durchgeführt werden, so dass teilweise Entscheidungen in Grenzgebieten entstanden sind. Durch die jeweilige Entscheidung der Mitarbeiter der Abteilung Programmiersprachen und Übersetzerbau in den Reviews wurde in solchen Fällen eine genaue Gewichtung implizit vorgenommen. Um diese Entscheidungen für die weitere Wartung der IML verständlich zu machen, werden sie in den nachfolgenden Unterkapiteln dokumentiert.

- *Mehrdeutigkeiten in der bestehenden IML*

  Durch die Auslegung der bisherigen Konzepte in der IML bei der Erweiterung für C++ kam es zu Mehrdeutigkeiten, die eine genaue Zuordnung neuer IML-Knoten in der Vererbungshierarchie nicht möglich machten. Beispielsweise werden von O_Node direkt OC_Entity und OC_User_Type_Declaration abgeleitet, die jeweils Entitäten im Speicher und benutzerdefinierte Elemente modellieren. An dieser Stelle hat die Vererbung zwei unterschiedliche Diskriminatoren, da zum einen nach dem Kriterium der Repräsentation im Speicher (OC_Entity) und zum anderen nach der Typdeklaration (OC_User_Type_Declaration) abgeleitet wird. Unter Umständen können diese beiden Kriterien jedoch gleichzeitig auf ein Element zutreffen, wie es zum Beispiel bei einer Klasse mit statischen Attributen der Fall ist.

In den weiteren Kapiteln wird die Modellierung der IML bezüglich der Erweiterung um C++ erörtert. Wenn es bei der Modellierung mehrere Vorgehensweisen gegeben hat, die den Anforderungen entsprechen, werden diese möglichst umfassend beschrieben und jeweils mit einer Entscheidung für eine Alternative abgeschlossen.
Beschrieben werden nur Themen, die sich entweder gegenüber der IML für C oder für Java geändert haben oder neu hinzugekommen sind. Themen, die ohne Änderungen von der IML für Java übernommen wurden, werden nicht nochmals erwähnt. Ebenfalls unerwähnt bleiben kleinere Änderungen in der IML, wie das Verschieben von Attributen in IML-Klassen oder Typänderungen bei Attributen. Änderungen dieser Art sind detailliert in Kapitel 2.4 aufgeführt. Dort ist auch die vollständige Spezifikation der Änderungen zu finden, während in diesem Kapitel nur die grundsätzliche Modellierung behandelt und nicht immer auf die Details eingegangen wird.

2.3.1 Verwendete Notation

T.K.

Die in den folgenden Graphiken verwendete Notation ist ein Dialekt der Unified Modelling Language (UML) in der Version 1.4 [29]. Erweitert wurde die

UML-Darstellung in den Objektdiagrammen durch gerichtete Assoziationskanten, die vom Objekt mit dem entsprechenden Attribut hin zum Wert des Attributs zeigen. Der Name des jeweiligen Attributs ist dabei als Stereotype der Kante angegeben. Teilweise sind die Namen der Attribute auch in Zahlen ausgedrückt, um mehr Übersichtlichkeit zu erreichen. In diesen Fällen ist der konkrete Name in der dazugehörigen Legende nachzulesen. Um semantische von syntaktischen Kanten unterscheiden zu können werden semantische Attribute weiterhin in kursivem Text gesetzt, wohingegen syntaktische nichtkursiv gesetzt sind.
Eine andere hier angewendete Notation für Klassenhierarchien stammt von dem imlgen Werkzeug aus dem Bauhaus-Projekt. Das imlgen Werkzeug generiert den Quelltext der IML-Klassen und eine grafische Darstellung der Klassenhierarchie aus der IML-Spezifikation. Die Details der Notation für die Klassenhierarchie ist in der Dokumentation des Werkzeuges nachzulesen.

2.3.2 Erweiterte Namen/name mangling

S.S.

C++ und Java unterstützen das Überladen von Funktionen. Das heißt, verschiedene Funktionen sind unter dem gleichen Namen mit verschiedenen Parameterprofilen bekannt. Weiterhin kann eine Methode oder ein Attribut in mehreren Klassen und C++ Namensbereichen gleich definiert sein.

Mit diesen Spracherweiterungen (Überladen und Namensbereiche) ist jedoch der Name eines Symbols nicht mehr ausreichend, um das Symbol in der Symboltabelle eindeutig zu identifizieren. Um dieses Problem zu lösen, wurde in der IML für Java das Attribut Mangled_Name parallel zu den Namen in der Symboltabelle hinzugefügt. Dieses Attribut enthält den so genannten erweiterten Namen (engl. mangled name). Die Eindeutigkeit des erweiterten Namens wird dadurch erreicht, dass die Signatur der Funktion in den erweiterten Namen hineinkodiert wird.

Bei dem Zwischenvortrag dieser Diplomarbeit wurde vorgeschlagen, die komplette Symboltabelle auf den erweiterten Namen abzuändern, so dass das bisherige Name Attribut der Klasse O_Node den erweiterten Namen beinhaltet. Dies entspricht der üblichen Vorgehensweise bei Java und C++ Übersetzern und erleichtert das Anpassen der auf der IML aufbauenden Analysen und Werkzeuge, da die Symboltabelle wieder eindeutige Namen beinhaltet. Der ursprüngliche Name, wie er im Programmtext notiert ist, wird in dem Attribut Unmangled_Name in der Klasse O_Node aufbewahrt.

Nach einer Diskussion während des Zwischenvortrags wurde entschieden, das Attribut Name ganz zu entfernen. Es gibt jetzt nur noch die beiden Attribute Mangled_Name und Unmangled_Name. Dadurch wird eine Überprüfung aller Werkzeuge, die das Attribut Name verwenden, erzwungen. Für jedes Werkzeug muss entschieden werden, ob der eindeutige Mangled_Name oder der gut lesbare Unmangled_Name der benötigte ist.

Die Namen der formalen Parameter einer Funktion müssen nicht unbedingt erweitert werden, da sie nicht vom Linker verarbeitet werden müssen. Es gibt folgende Möglichkeiten, das Attribut Mangled_Name für formale Parameter zu setzen:

1. Null, also kein eindeutiger Name.
2. Gleich wie Unmangled_Name, ebenfalls nicht eindeutig.
3. Zusammengesetzt aus dem erweiterten Namen der Funktion (oder des Typs der Funktion) und dem Namen des formalen Parameters. Um bei unbenannten formalen Parametern Eindeutigkeit zu erreichen, kann die Position des Parameters, zum Beispiel als Dezimalzahl, angehängt werden.

Die Eindeutigkeit der Namen ist nur im IML-Linker relevant. Alle nachfolgenden Analysen können auf die IML-Knoten-Identität zurückgreifen. Deshalb sind auch Alternative 1 und 2 vorstellbar.

Da Alternative 1 die einfachste ist, weniger Speicher benötigt und trotzdem alle benötigten Informationen enthält, wurde diese Alternative ausgewählt.

2.3.3 Ausnahmebehandlung

T.K.

Das Sprachelement der Exceptions wurde bereits in der IML für Java umgesetzt. Aufgrund der starken semantischen Äquivalenz der Ausnahmebehandlung in Java und C++ wird hierfür in der IML dieselbe Darstellung für beide Programmiersprachen verwendet.

Um die Modellierung konsistenter zu der bestehenden IML zu gestalten, wurden jedoch zwei Änderungen gegenüber der IML für Java in der Vererbungshierarchie durchgeführt. Zum einen hat jetzt die Klasse Throw_Statement Unconditional_Branch als Oberklasse und nicht mehr Sequence_Item. Damit kann jetzt auch das Ende der Laufzeit von lokalen Variablen modelliert werden. Zum anderen hat die Klasse Try_Catch_Finally_Statement Sequence_Item als Oberklasse, was vorher Value gewesen ist. Damit wird ausgedrückt, dass das Try_Catch_Finally_Statement auch zu den Ausdrücken gehört, die einen Ablauf bilden können. Die geänderte Klassenhierarchie ist in Abbildung 2.4 dargestellt.

Abbildung 2.4: Klassenhierarchie für die Ausnahmebehandlung

2.3.3.1 Funktionsaufrufe und eingebaute Operatoren

S.S

Ausnahmen, die von einer Funktion nach außen propagiert werden, bewirken bei der aufrufenden Funktion unter Umständen das Verlassen des aktuellen Gültigkeitsbereiches von Variablen. Das gleiche gilt für Ausnahmen, die von eingebauten Operatoren ausgelöst werden, z.B. bei Division durch Null, Überlauf oder Dereferenzieren eines ungültigen Zeigers. Die erzeugten Variablen aus Gültigkeitsbereichen, die verlassen werden, müssen in diesen Fällen

zerstört werden. Folgende Möglichkeiten zur Beschreibung der Variablenzerstörung wurden diskutiert:

1. Jeder Funktionsaufruf (Routine_Call) und jeder eingebaute Operator, der eine Ausnahme auslösen kann, bekommt eine Statement_Sequence mit den Destruktoraufrufen (Direct_Call) und End_Of_Lifetimes, die im Fall einer ausgelösten Ausnahme ausgeführt wird. Das entspricht der Modellierung von Sprüngen bei Unconditional_Branches.

2. Anstatt einer Statement_Sequence mit den Destruktoraufrufen könnte auch ein Sprung an die richtige Stelle einer Statement_Sequence erzeugt werden, in der alle in der Funktion initialisierten Variablen in umgekehrter Reihenfolge zerstört werden.

3. Funktionsaufrufe und Unconditional_Branches besitzen keine Statement_Sequence, sondern eine *list* of class OC_Stack_Object, anstatt wie seither *set* of class OC_Stack_Object. Jedes OC_Stack_Object kennt seinen O_Destructor. Das Problem hierbei ist, dass OC_Component keine Unterklasse von OC_Stack_Object ist. Konstruktoraufrufe können aber die automatische Freigabe von OC_Components zur Folge haben.

4. Der Kontrollfluss wird explizit modelliert. Dies könnte beim Auslösen von Ausnahmen durch ein If_Statement nach jedem Routine_Call und einem zusätzlichen Artificial-Parameter in jeder Routine mit C++ Bindekonvention geschehen. Bei der Erweiterung der IML für Java wurde jedoch gegen diese Modellierung entschieden.

   Anstatt eines zusätzlichen Parameters könnte auch die globale Exception-Variable benutzt werden. Siehe auch Kapitel 2.3.3.3.

5. Destruktoraufrufe werden nicht modelliert. Eine noch zu entwickelnde Analyse bzw. die virtuelle IML-Maschine zerstört beim Verlassen eines Blocks genau die Objekte umgekehrter Reihenfolge, die vorher erzeugt wurden.

6. Ignorieren der Tatsache, dass im Falle einer Ausnahme bei Funktionsaufrufen und eingebauten Operatoren Destruktoren aufgerufen werden müssen.

Gegen Alternative 3. und 5. spricht, dass für alles, was ausgeführt werden muss, HPG-Knoten existieren sollen, um sie in den Kontrollflussgraphen einbinden zu können.

Da durch die große Anzahl an bedingten Sprüngen, die durch Ausnahmen entstehen können, eine sinnvolle Analyse praktisch unmöglich wird, wurde zugunsten der letzten Alternative entschieden. Bei Throw_Statements gibt es jedoch, wie bei allen Unconditional_Branches, ein End_Lifetime-Attribut mit Destruktoraufrufen.

2.3.3.2 Initialisierung zusammengesetzter Objekte

S.S

Wenn während der Initialisierung von Klassenobjekten oder Arrays Ausnahmen auftreten, müssen die bereits initialisierten Teile wieder aufgeräumt werden. Folgende Möglichkeiten der Behandlung von Ausnahmen bei Array-Initialisierungen sind möglich:

1. Array-Initialisierungen könnten von einer künstlichen Funktion „`construct_Array`" abgearbeitet werden, wodurch auftretende Exceptions durch die normale Ausnahmebehandlung behandelt und bereits erzeugte Objekte korrekt zerstört werden. Diese künstliche Funktion könnte in einem Modul `Runtime`, das zum Programm gelinkt wird, implementiert sein.

```
typedef void (*funcPtr)(void* this_Pointer);
void Runtime::construct_Array(
    void* array, int element_Size, int count,
    funcPtr constructor, funcPtr destructor)
{
    for (void* it = array,
         void* end = array+(count*element_Size);
         it != end;
         it += element_Size) {
        try {
            constructor(it);
        }
        catch(...) {
            for (it = it - element_Size,
                 void* rEnd = array-element_Size;
                 it != rEnd;
                 it -= element_Size) {
                destructor(it);
            }
            throw;
        }
    }
}
```

   Bei dieser Modellierung braucht man eine künstliche Funktion, die der Funktionszeigersignatur entspricht, falls ein Konstruktor nur durch Standard-Parameter zum Standard-Konstruktor wird.

   Der this-Parameter des Konstruktors und des Destruktors ist nicht vom Typ `void*`. Um eine Typkonversion einzufügen, braucht man für jeden Konstruktor eine künstliche Funktion, da die Konversion des Funktionszeigertyps nicht erlaubt ist.

2. Im Prinzip wird die Initialisierung von Array-Initialisierungen auf dieselbe Weise modelliert wie bei Alternative 1, nur erfolgt die Modellie-

rung des IML-Graphen „inline", also wird für jede Initialisierung eine neue IML-Modellierung erzeugt.

3. Die Funktion `Runtime::construct_Array` wird als generische Funktion implementiert. Das ist aber problematisch, da sie dann nicht einfach als fertiges Runtime-Modul zum Programm gelinkt werden kann.

Dieselbe Argumentation wie bei Funktionsaufrufen führte dazu, dass Ausnahmen bei Array-Initialisierungen nicht gesondert in der IML modelliert werden. Siehe hierzu Kapitel 2.3.3.1.

2.3.3.3 Mehrere Ausnahmebehandlungen

S.S

Der C++ Standard [14, Abschnitt (except.terminate)] schreibt für die Ausnahmebehandlung vor, dass die Funktion „`void terminate();`" in den folgenden Fällen aufgerufen wird:

- „when the exception handling mechanism, after completing evaluation of the expression to be thrown but before the exception is caught (15.1), calls a user function that exits via an uncaught exception (for example, if the object being thrown is of a class with a copy constructor, terminate() will be called if that copy constructor exits with an exception during a throw), or
- when the destruction of an object during stack unwinding (15.2) exits using an exception, or"
- in weiteren, hier nicht relevante Fällen.

Das heißt, wenn eine Ausnahme nicht rechtzeitig behandelt wird, wird das Programm abgebrochen.

Zwischen dem Auslösen einer Ausnahme und ihrer Behandlung geschehen zwei Dinge, bei denen wieder eine Ausnahme auftreten kann:

1. Kopie des Ausnahme-Objekts, möglicherweise durch einen Copy-Konstruktor;
2. Aufräumen von Ressourcen durch Destruktoren;

Innerhalb von Destruktoren und Copy-Konstruktoren dürfen in diesem Fall Ausnahmen ausgelöst werden, falls sie rechtzeitig behandelt werden. Falls das geschieht, existieren kurzzeitig mehrere zu behandelnde Ausnahmen. Eine globale Variable für die zu behandelnde Ausnahme reicht jedoch aus, falls sie bei einem `try`-Block zwischengespeichert wird. Damit die Zwischenspeicherung ohne Aufruf eines Copy-Konstruktors möglich ist (ein Copy-Konstruktor darf nur einmal, beim Auslösen der Ausnahme, aufgerufen werden), muss die globale Exception-Variable ein Zeiger auf das Ausnahme-Objekt sein.

Die Tatsache, dass die ausgelöste Ausnahme einen beliebigen Typ haben kann, ist ein weiterer Grund, warum die globale Ausnahme-Variable ein Zeiger sein muss.

Mehrere Threads werden in der IML zur Zeit nicht modelliert. Falls Threads in der IML modelliert würden, müsste jeder Thread eine eigene globale Ausnahme-Variable haben, da in jedem Thread Ausnahmen auftreten können.

2.3.4 Klassen, Metaklassen und Objekte

T.K.

Klassen wurden in der IML bereits durch die Erweiterung für Java modelliert. Da das Konzept der Klassen in C++ dem von Java sehr ähnlich ist, ist es sinnvoll, diese in der IML mit derselben Repräsentation darzustellen. Die Modellierung von Klassen hat dabei insbesondere folgende Aspekte zu berücksichtigen:

- Darstellung von Objekten,
- Metaklassen und
- statischen Anteilen, d.h. statischen Methoden und Attributen.

Die grundlegende Modellierung eines Klassentyps geschieht weiterhin durch T_Class-Knoten. Einzelne Attribute dieses Knotens und die Vererbungshierarchie wurden jedoch angepasst und erweitert, um auch C++ Klassentypen darstellen zu können.

2.3.4.1 Objekte

T.K.

Die Darstellung von Objekten von Klassen wurde analog zur Vorgehensweise bei `struct`-Elementen aus der IML für C durchgeführt. Damit werden Instanzen von Klassen auf die gleiche Weise dargestellt wie die Instanzen von allen anderen Typen, also durch O_Local und O_Global-Knoten. Dynamisch erzeugte Objekte werden durch einen `new`-Operator und einer Zuweisung an eine Pointervariable modelliert. Die gesonderte Darstellung durch O_Instance-Knoten in der IML für Java ist dadurch nicht mehr nötig.

2.3.4.2 Metaklassen

T.K. und S.S.

Metaklassen existieren zwar nicht direkt in C++, sind jedoch für die IML von Java von Bedeutung, um Klassenobjekte darstellen zu können. In C++ ist das Konzept der Metaklassen weiterhin für die Behandlung der Typinformation (`class type_info`) anwendbar.

Metaklassen werden in derselben Art dargestellt, wie die Klassen selbst, also durch T_Class-Knoten. Dies entspricht der Vorgehensweise vieler objektorientierter Programmiersprachen, wie Smalltalk, Java und C++. Der einzige Unterschied zu den benutzerdefinierten Klassen ist, dass die Metaklassen automatisch vom Übersetzer des Programms definiert werden bzw. von der Programmiersprache vorgeben sind. In Java und C++ gibt es nur eine Metaklasse (java.lang.Class bzw. std::type_info), von der alle Klassenobjekte Instanzen sind (für jede Klasse eine). In Smalltalk gibt es dagegen für jede Klasse eine Metaklasse.

Klassenobjekte, also Instanzen der Metaklassen, existieren in Java und in Form der Typinformation auch in C++. Bei den beiden Programmiersprachen werden diese Klassenobjekte automatisch vom Übersetzer des Programms angelegt und sind im globalen Sichtbarkeitsbereich, haben eine globale Lebenszeit und besitzen eine eindeutige Identität. Aus diesem Grund werden sie in

der IML für Java und C++ durch implizit angelegte globale Variablen modelliert.

Ein Beispiel dafür, wie eine solche Modellierung in IML aussieht, ist in Abbildung 2.5 für das folgende C++ Programm aufgeführt:

```
class Basis {
};
class A : public Basis {
};
class B : public Basis {
};
```

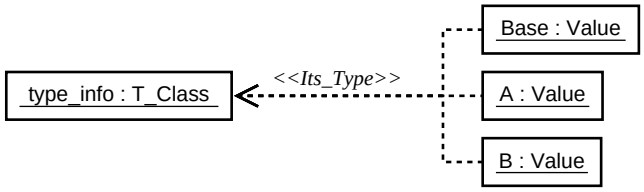


Abbildung 2.5: Objektdiagramm der IML Darstellung der Metaklassen und Klassenobjekte zum Beispiel

Die Vererbungsbeziehungen zwischen den Klassen sind in der IML-Darstellung nicht sichtbar, weil bei C++, im Gegensatz zu Smalltalk, jede Klasse dieselbe Metaklasse hat – `type_info`. Die Vererbungsbeziehungen können in C++ nicht zur Laufzeit ermittelt werden. Falls die Vererbungsbeziehungen zur Laufzeit ermittelt werden *könnten*, wären sie auch nicht in der IML-Darstellung sichtbar. Sie wäre in den Klassenobjekten modelliert, von denen jedoch nur die Deklarationen in der IML enthalten sind.

Genaugenommen existieren nicht einmal die Deklarationen der Klassenobjekte in der IML. Die Klassenobjekte werden nur vom `typeid`-Operator zurückgegeben, der in der IML als eingebauter Operator, Typeid_Operator, dargestellt wird.

2.3.4.3 Statische Anteile

T.K.

Die Modellierung statischer Methoden und Attribute einer Klasse ist auf mehrere Arten möglich. Im Folgenden werden drei Alternativen vorgestellt, die alle den Anforderungen entsprechen und somit umsetzbar wären. In den angefügten Beispielen zu den Beschreibungen ist jeweils nur der interessante Teil der IML-Modellierung zu sehen. Teilweise fehlen dabei Klassen, die in der Klassenhierarchie unter O_Node oder T_Node zu finden wären.

1. Die erste Alternative ist stark angelehnt an die Deklaration von Klassen in Java und C++. Die Typbeschreibung einer Klasse enthält dabei die dynamischen und statischen Komponenten der Klasse. Statische Methoden werden durch O_Static_Method-Knoten dargestellt, welche von O_Routine abgeleitet sind, da sie statisch gebunden werden können und

keinen `this`-Zeiger haben. Statische Attribute werden durch O_Static-_Component-Knoten dargestellt, welche von O_Global abgeleitet sind, da sie eine globale Lebensdauer und keinen Offset wie die dynamischen Attribute haben. Die hierfür relevante Klassenhierarchie in der IML ist in Abbildung 2.6 dargestellt.

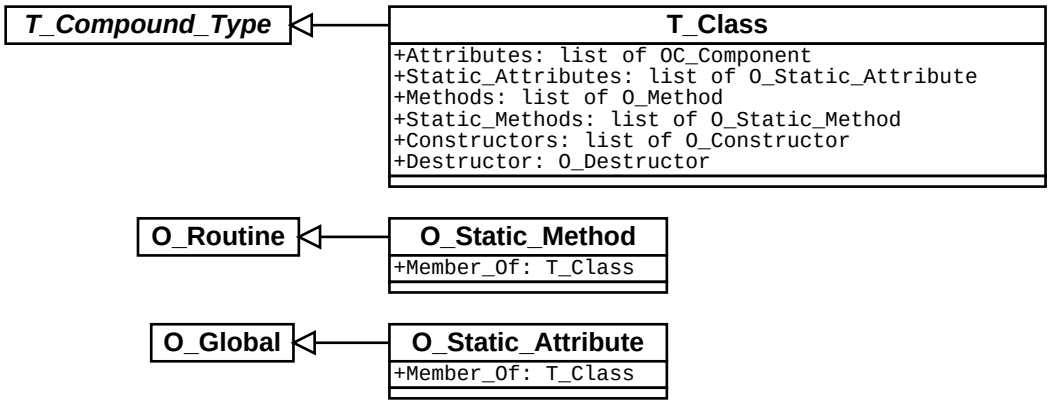


Abbildung 2.6: Modellierung statischer Komponenten – Alternative 1

Diese Vorgehensweise hat den Vorteil einer kompletten Typbeschreibung, die auch statische Teile umfasst. Weiterhin ist diese Typbeschreibung angelehnt an die Notation statischer Komponenten in Java und C++, was eine hohe Quelltextnähe bedeutet. Da die statischen Attribute einer C++ Klasse sogar auf verschiedene Übersetzungseinheiten verteilt sein können, kann mit dieser Vorgehensweise die Semantik von C++ sehr gut modelliert werden.

2. In der zweiten Alternative wird die Typbeschreibung einer Klasse aufgeteilt in einen dynamischen und einen statischen Anteil. Vom statischen Anteil der Typbeschreibung wird dabei vom Übersetzer automatisch eine globale Instanz erzeugt. Alleine der dynamische Anteil bildet die übliche Instanz einer Klasse in der IML ab. Abbildung 2.7 auf der nächsten Seite veranschaulicht die relevante IML-Klassenhierarchie hierfür.

   Der Vorteil dieser Modellierung ist eine klare Trennung zwischen den dynamischen und den statischen Anteilen einer Klasse. Gleichzeitig enthält die Typbeschreibung einer Klasse die Beschreibung beider, der dynamischen und der statischen Anteile.

   Nachteilig würde sich diese Modellierung auf den IML-Linker auswirken, der statische Komponenten einer Klasse aus mehreren Übersetzungseinheiten vereinen können müsste. Dazu wäre eine zusätzliche Erweiterung des Linkers notwendig.

3. Die dritte Möglichkeit ist angelehnt an die Modellierung statischer Anteile in Smalltalk. Dabei enthält die Typbeschreibung einer Klasse ausschließlich dynamische Anteile. Alle statischen Anteile einer Java oder C++ Klasse sind Bestandteil der Metaklasse der Klasse in IML. Diese

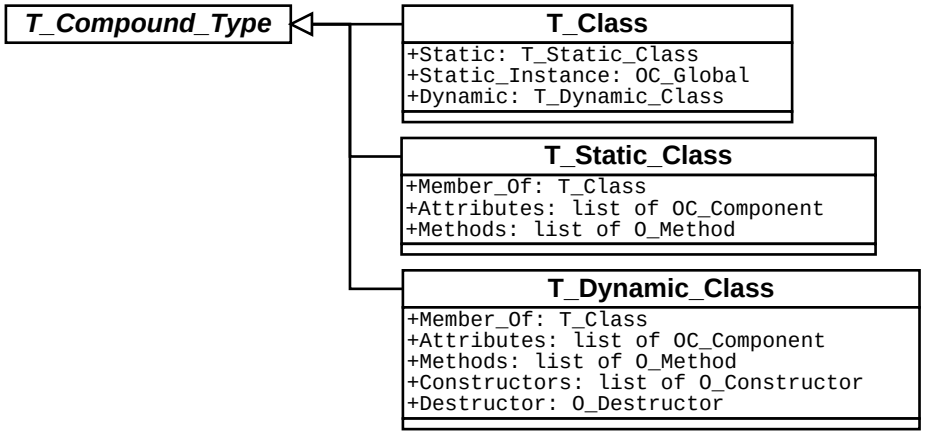


Abbildung 2.7: Modellierung statischer Komponenten – Alternative 2

Metaklassen würden dann automatisch vom Übersetzer definiert werden und es würde auch automatisch eine globale Instanz von jeder Metaklasse erzeugt werden. Abbildung 2.8 stellt die IML-Klassenhierarchie für diese Alternative dar.

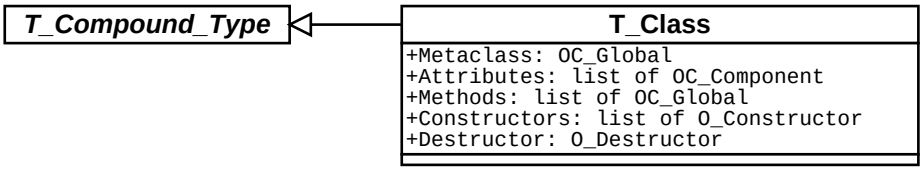


Abbildung 2.8: Modellierung statischer Komponenten – Alternative 3

Für diese Alternative spricht die einfache Semantik der Modellierung, da es keiner gesonderten Behandlung statischer Anteile bedarf.

Nachteilig ist, dass damit die Quelltextnähe zu Java und C++ verloren geht. Außerdem gelten ebenfalls die Nachteile bezüglich des IML-Linkers, die bereits bei Alternative 2 genannt wurden.

Aufgrund der überwiegenden Vorteile wird in dieser Diplomarbeit die erste Alternative umgesetzt. Sie hat weiterhin den Vorteil, verständlicher zu sein, was sich im Review über die IML-Spezifikation mit den Mitarbeitern der Abteilung herausgestellt hat.

2.3.4.4 `struct`-Sprachelemente

T.K. und S.S.

Das `struct` Sprachelement in C++ wird in der für C++ erweiterten IML-Klassenhierarchie von `T_Class` abgeleitet. Der Grund für diese Vorgehensweise liegt in der besonderen Semantik, die bei C++, im Vergleich zu C, stark erweitert wurde. Tatsächlich unterscheiden sich die `struct` und `class` Sprachelemente in C++ nur durch den vorgegebenen Sichtbarkeitsbereich. Bei `struct`-Deklarationen ist der vorgegebene Sichtbarkeitsbereich für die Komponen-

ten und Basisklassen einer Klasse `public`. Bei `class`-Deklarationen ist der vorgegebene Sichtbarkeitsbereich `private`.

Der Name des IML-Knotens für C++ `struct` Typbeschreibungen lautet TCPP_Struct, um dieses für C++ spezielle Sprachkonstrukt deutlich zu markieren. Record-Datentypen aus Pascal-ähnlichen Programmiersprachen, wie z.B. `struct` Sprachelemente in C oder `Record` Sprachelemente in Ada, werden weiterhin mit TC_Struct dargestellt.

Ein vermeintliches Problem entsteht durch diese Modellierung, wenn Übersetzungseinheiten unterschiedlicher Programmiersprachen mit C++ Übersetzungseinheiten zu einem Programm zusammengesetzt werden. Beispielsweise könnte eine C oder Ada Übersetzungseinheit durch eine Headerdatei einen `struct`-Datentyp definieren, auf den von einer C++ Übersetzungseinheit aus zugegriffen wird. In diesem Falle erzeugen beide Implementierungen unterschiedliche Datentypen (TC_Struct und TCPP_Struct) für denselben Quelltext.

Eine Lösung des Problems könnte darin bestehen, `struct`-Datentypen aus C Übersetzungseinheiten auch im C++ Übersetzer als TC_Struct in der IML abzubilden. Leider ist das nicht möglich, da es keine Möglichkeit gibt, den Ursprung als C Übersetzungseinheit zu erkennen. Eine „`extern "C"`" Deklaration beeinflusst z.B. nur die Bindekonventionen und trifft damit keine Aussagen über Typen, da diese nicht gebunden werden. Weiterhin wäre es denkbar, C `struct` Deklarationen daran zu erkennen, dass diese keine Methoden und Basisklassen haben.

Es kann nicht erkannt werden, ob in anderen Übersetzungseinheiten Unterklassen eines `struct`s definiert werden. Deshalb müssen Knoten vom Typ TC_Struct als Basisklassen zugelassen werden, falls in C++ Übersetzungseinheiten Knoten vom Typ TC_Struct erzeugt werden. Bei Varianten stellt sich diese Frage nicht, da dort keine Unterklassen erlaubt sind.

Alternativ zur Erkennung von `struct`-Datentypen könnte auch die IML so verändert werden, dass TC_Struct eine Unterklasse von T_Class_Type ist. Dies wurde nicht gemacht, da bei der Erweiterung der IML für Java entschieden wurde, Klassen und Records separat zu behandeln.

Letztendlich ist das Problem jedoch nicht relevant, da im schlimmsten Fall beim Aufruf einer Funktion in einer anderen Übersetzungseinheit Zeiger auf einen TC_Struct implizit auf einen TCPP_Struct oder anders herum konvertiert werden. Bei gegebenem korrekten Quelltext sollte aber nur der Übersetzer diese Zeiger dereferenzieren müssen, der auch die referenzierten Objekte erzeugt hat. Dies entspricht der üblichen Vorgehensweise beim Zusammenlinken von Übersetzungseinheiten verschiedener Programmiersprachen.

2.3.5 Varianten

S.S.

Im Vergleich zu C können Varianten in C++ auch Methoden, Konstruktoren und Destruktoren haben. Wie bei Klassen kann auch die Sichtbarkeit von Elementen definiert werden. Varianten dürfen jedoch nicht in Vererbungshierarchien auftreten. Da Varianten nicht in Vererbungshierarchien auftreten dürfen, ergeben auch virtuelle Methoden keinen Sinn und sind deshalb nicht erlaubt.

C++ Varianten werden in der IML gleich wie Klassen und C++ Strukturen dargestellt. Parallel zu den Klassen ...Class und ...CPP_Struct gibt es Klassen ...CPP_Union. Die Attribute zur Modellierung von Vererbungshierarchien (z.B. Extends) und virtuellen Methoden (z.B. Virtual_Call_Table) sind in den gemeinsamen Oberklassen enthalten. Sie werden bei Varianten nicht benutzt. Diese Attribute hätten auch in die Klassen für Klassen und Strukturen verschoben oder in einer neuen gemeinsamen Oberklasse dieser Klassen zusammengefasst werden können. Dadurch wäre aber die IML-Spezifikation komplizierter geworden. Es ist ohne schwerwiegende Änderungen an cafe++ möglich, zu einer der beiden Alternativen zu wechseln.

2.3.6 Methoden und Attribute

S.S.

Die Modellierung statischer Methoden und Attribute wurde bereits in Kapitel 2.3.4 beschrieben und soll hier nicht noch einmal behandelt werden.

Die Modellierung der nicht-statischen Attribute wurde jedoch gegenüber der Modellierung in der IML für Java geändert. Nicht-statische Attribute werden jetzt nicht mehr durch OC_Local-Knoten, sondern durch OC_Component-Knoten dargestellt. Diese Darstellung entspricht der Darstellung von Attributen von `struct`-Sprachelementen und kann den Offset des Attributs zum Klassenobjekt darstellen. Weiterhin geschieht die Darstellung von Bitfeldern durch OC_Bitfield-Knoten auf diese Weise analog zu den üblichen Attributen.

Es gibt unterschiedliche Ansichten darüber, was Methoden von normalen Funktionen unterscheidet:

1. Methoden und Funktionen unterscheiden sich dadurch, dass Methoden einen `this`-Zeiger haben.

2. Methoden und Funktionen unterscheiden sich dadurch, dass alle Methoden `virtual` sind und Funktionen nie `virtual` sind.

3. Methoden und Funktionen unterscheiden sich dadurch, dass Methoden zu Klassen gehören (also auch statische Methoden).

In der IML sind Funktionen diejenigen Unterprogramme, die nicht zu Klassen gehören und werden durch Routine-Knoten dargestellt. Methoden haben `this`-Zeiger. Static_Methoden gehören zur Klasse, haben aber keinen `this`-Zeiger. Virtuelle Methoden sind ebenfalls Methoden und werden durch ein Attribut im Typ gekennzeichnet.

2.3.7 Friends

S.S.

Das Sprachelement der `friend`-Deklarationen ist in der IML für C++ neu hinzugekommen. `friend`-Deklarationen öffnen den Sichtbarkeitsbereich auf Attribute von Klassen für entweder andere Klassen oder Funktionen.

Da Sichtbarkeit und Zugriffsrechte vollständig in der semantischen Analyse eines Programms behandelt werden können und für die Ausführungssemantik der IML nicht von Bedeutung sind, könnte die Modellierung von

`friend`-Deklarationen in der IML auch ausgelassen werden. `friend`-Deklarationen werden aber trotzdem in der IML modelliert, da zum einen Quelltextnähe und Quelltext-Generierung angestrebt werden und zum anderen diese Information für eventuell folgende Analysen interessant sein könnte.

`friend`-Deklarationen werden in T_Class_Type modelliert durch folgende zusätzliche Attribute:

- Friend_Routines: set of OC_Routine

  Dadurch, dass eine Funktion als `friend` deklariert wird, wird auch die Funktion selbst implizit deklariert. Wenn Funktionsdeklarationen durch OC_Routine-Knoten dargestellt werden, müssen auch `friend`-Deklarationen mit IML-Knoten dieses Typs dargestellt werden.

- Friend_Classes: set of O_Class_Type

  Analog zu `friend`-Funktionen müssen `friend`-Klassen-Deklarationen als O_Class-Knoten dargestellt werden.

O_Class_Type erhält das neue Attribut

- Friend_Of: set of T_Class_Type

  Dieses Attribut enthält die Rückwärtsverweise zu Friend_Classes.

OC_Routine wird ebenfalls erweitert:

- Friend_Of: set of T_Class_Type

  Dieses Attribut enthält die Rückwärtsverweise zu Friend_Routines.

Verschiedene Übersetzungseinheiten

Die Definitionen der `friend`-Klassen einer Funktion sind im Normalfall in derselben Übersetzungseinheit enthalten, in der auch die Definition der Funktion enthalten ist. Im anderen Fall könnten die Funktionen nicht auf die Interna der `friend`-Klassen zugreifen, den `friend`-Status also nicht ausnutzen. Das ist in C++ zwar erlaubt, wird aber in der IML nicht dargestellt, da hierfür eine Änderung des IML-Linkers nötig wäre.

2.3.8 Inline-Funktionen und -Methoden

S.S.

Inline-Funktionen müssen in jeder Übersetzungseinheit definiert werden, in der sie aufgerufen werden. Es kann also in einem System mehrere (gleiche) Definitionen einer inline-Funktion geben. Die Adresse einer inline-Funktion muss immer dieselbe sein. Statische Variablen und Zeichenkettenliterale innerhalb von inline-Funktionen müssen auch immer dieselben sein. Deshalb wird von cafe++ für *jede* Deklaration einer externen inline-Funktion ein Unresolved_Declaration-Eintrag und für jede Definition ein Provided_Definition-Eintrag erzeugt. Sogar die Definition-Kante einer Deklaration, die selbst eine Definition ist, wird nicht vor dem Linken gesetzt. Der Linker wählt dann eine

Definition aus und setzt diese als Definition *aller* Deklarationen ein. In der aktuellen Version gibt der Linker noch eine Warnung aus, dass eine mehrfache Definition vorliegt. Der Linker sollte so erweitert werden, dass diese Warnung bei inline-Funktionen nicht ausgegeben wird.

2.3.9 Überladene Funktionen und Operatoren

S.S.

Überladene Funktionen werden in der IML als verschiedene Funktionen mit dem gleichen Unmangled_Name dargestellt. Falls eine überladene Funktion als *eine* Funktion betrachtet werden soll, kann dazu das Attribut Unmangled_Name verwendet werden.

Überladene Operatoren werden in der IML wie normale Funktionen dargestellt, wobei das Attribut Unmangled_Name den Namen des Operators enthält, z.B. „`operator+`". Zusätzlich wird bei dem OC_Routine-Knoten, der die Deklaration des Operators repräsentiert, das Attribut Is_Operator auf „wahr" gesetzt.

In Templates kann manchmal nicht entschieden werden, ob ein Überladener oder ein eingebauter Operator aufgerufen werden soll, weil der Typ der aktuellen Parameter noch unbekannt ist. In diesem Fall wird in der IML-Darstellung des Templates ein eingebauter Operator verwendet. Bei der Instantiierung des Templates wird immer der korrekte IML-Graph erzeugt. Die Instantiierung kann sich in diesem Punkt also von ihrem Template unterscheiden.

2.3.10 Templates

S.S.

Im Folgenden wird das C++ Sprachkonstrukt für Templates kurz beschrieben. Diese Beschreibung fasst die zu modellierenden Eigenschaften von Templates zusammen. Darauf aufbauend wird ab Kapitel 2.3.10.3 die Modellierung in der IML beschrieben. Templates sind Makro-ähnliche Schablonen für Klassen (ein Beispiel ist in Abschnitt „Verschachtelte Template-Deklaration" auf Seite 50 aufgeführt) und für Funktionen (ein Beispiel ist in Abschnitt „Template Funktion mit Typ- und Konstantentemplateparameter" auf Seite 50 aufgeführt). Nicht-virtuelle Methoden und Konstruktoren können auch Template-Funktionen sein, Destruktoren jedoch nicht.

Die Instantiierung des Templates entspricht der Expansion eines Makros. Bei der Instantiierung werden alle Parameter belegt und die dabei entstehende Instanz kann wie eine normale Klasse bzw. Funktion verwendet werden. Falls ein Template mehrmals mit den gleichen Parametern instantiiert wird, muss nur eine Instanz erzeugt werden, während ein Makro jedesmal neu expandiert werden müsste.

Allgemein werden Templates für Klassen auf folgende Weise deklariert: „**`template`**`<...>` **`class`** *Klasse*" und Templates für Funktionen folgendermaßen: „**`template`**`<...>` *Rückgabe_Typ funktion*`(`*Parameterliste*`)`", wobei jeweils „`...`" eine Liste von Templateparametern ist, wie sie in Kapitel 2.3.10.1 beschrieben wird.

2.3.10.1 Arten von Templateparametern

S.S.

Die verschiedenen, im Folgenden beschriebenen Arten von Template-Parametern können beliebig in der Template-Signatur kombiniert werden. Die Template-Signatur ist eine Liste aus Deklarationen von Templateparametern. Sie wird dem entsprechend in der IML als Liste aus Deklarationen von Templateparametern modelliert (siehe Kapitel 2.3.10.4).

Konstanten-Templateparameter Ein Beispiel für eine Konstanten-Template-Signatur ist: „`template<int i, int j>`".

Bei der Instantiierung müssen Konstanten angegeben werden. Die Parameter können im Template wie normale konstante Variablen verwendet werden.

Siehe auch „Template Funktion mit Typ- und Konstantentemplateparameter" auf Seite 50.

Typ-Templateparameter Ein Beispiel für eine Typ-Template-Signatur ist: „`template <class S, typename T>`". Die Schlüsselworte „`class`" und „`typename`" sind in diesem, und nur in diesem Zusammenhang synonym – sie unterscheiden sich nur in ihrem Dokumentationswert.

Bei der Instantiierung müssen Typen angegeben werden. Die Parameter können im Template wie normale, vollständig definierte Typen verwendet werden. Beim Aufruf von Template-Funktionen kann der Übersetzer die Template-Argumente anhand der für den Funktionsaufruf erforderlichen Signatur oft auch automatisch bestimmen.

Siehe auch „Template Funktion mit Typ- und Konstantentemplateparameter" auf Seite 50 und „Verschachtelte Template-Deklaration" auf Seite 50.

Verschachtelte Template-Deklarationen Ein Beispiel für eine verschachtelte Template-Signatur ist: „`template< template<...> class Nested > class C;`".

Bei der Instantiierung müssen Template-Klassen angegeben werden, die der angegebenen Template-Signatur entsprechen. Die Parameter können im Template wie normale Template-Klassen benutzt werden.

Siehe auch „Verschachtelte Template-Deklaration" auf Seite 50.

2.3.10.2 Template-Spezialisierungen

S.S.

Templates können spezialisiert werden. Dabei wird für bestimmte Template-Signaturen eines Templates, die mit der Signatur eines Haupt-Templates verträglich sind, ein spezieller Template-Inhalt angegeben. Jedes normale Template kann spezialisiert werden und wird dadurch zum Haupt-Template der Spezialisierungen.

Beispiel aus [14, Abschnitt (temp.class.spec)]:

```
template<class T1, class T2, int I>
                        class A             { }; // #1
template<class T, int I>
                        class A<T, T*, I>   { }; // #2
```

```
template<class T1, class T2, int I>
                    class A<T1*, T2, I> { }; // #3
template<class T>
                    class A<int, T*, 5> { }; // #4
template<class T1, class T2, int I>
                    class A<T1, T2*, I> { }; // #5
```

Die erste Deklaration des Beispiels deklariert das Haupt-Template. Die nachfolgenden Deklarationen deklarieren partielle Spezialisierungen des Haupt-Templates.

Auf die Modellierung von Template-Spezialisierungen in der IML wird in Kapitel 2.3.10.5 eingegangen.

2.3.10.3 Modellierung der Instantiierung von Templates

S.S.

In den vorangehenden Kapiteln wurde das Template-Sprachelement von C++ erklärt. In den folgenden Kapiteln wird die Modellierung von Templates in der IML diskutiert. In diesem Kapitel wird mit zwei verschiedenen Ansätzen zur Modellierung der Template-Instantiierung begonnen, da die Entscheidung zwischen den beiden Ansätzen grundlegend für weitere Entscheidungen bezüglich der Modellierung von Templates ist.

Die Instantiierung von Templates könnte durch einen Knoten modelliert werden, der die aktuellen Templateparameter und einen Verweis auf das Template enthält. IML-Analysen müssten dann das Template selbst instantiieren, also die formalen Templateparameter mit den aktuellen Templateparametern belegen.

Es wurde jedoch entschieden, die Instantiierung von Templates durch Kopieren von IML-Teilgraphen zu modellieren. Die Instantiierung wird also vom Front-End durchgeführt und das Ergebnis im IML-Graph abgelegt. Bei der Instantiierung eines Templates wird der Teilgraph, der das Template beschreibt, kopiert und die formalen Parameter der Kopie werden mit den aktuellen Parametern der Instantiierung belegt. Bei weiteren Instantiierungen mit genau den gleichen Argumenten wird keine neue Kopie des Teilgraphen angelegt. Bei Instantiierungen mit anderen Argumenten werden neue Kopien angelegt.

Diese Vorgehensweise hat folgenden *Vorteil*:

- Große Vereinfachung für die Analysen, da diese die Instantiierung von Templates außer Betracht lassen können, falls sie nicht von Interesse sind.

und folgende *Nachteile*:

- Aufwändige Kopieroperationen beim Generieren des IML-Graphen.
- Redundante Information und große Datenmasse.
- Weniger Quelltextnähe.

Die Vereinfachung für die Analysen durch das Kopieren überwiegt die Nachteile, so dass Instantiierungen von Templates in der IML als instantiierte Kopien gebildet werden.

2.3.10.4 Modellierung von Templateparametern

S.S.

In Kapitel 2.3.10.1 wurde erklärt, was für Arten von Templateparametern es gibt. In diesem Kapitel wird beschrieben, wie die Templateparameter und ihre Verwendung innerhalb des Templates in der IML modelliert werden. Die dabei verwendeten neuen IML-Klassen werden im nächsten Kapitel nochmals aufgeführt.

Deklarationen von Typ-Templateparametern werden ähnlich modelliert, wie Typdefinitionen durch `typedef`. Die Deklaration eines Templateparameters innerhalb der Template-Signatur wird durch einen O_Template_Type_Parameter-Knoten dargestellt. Die Verwendung eines Parameters wird, analog zu `typedef`s, über einen T_Template_Type_Parameter_Name realisiert.

Deklarationen von Konstantenparameter werden als O_Template_Const_Parameter modelliert. O_Template_Const_Parameter ist eine Unterklasse von OC_Variable, die Verwendung wird also gleich behandelt wie die Verwendung anderer konstanter Variablen.

Bei der Instantiierung des Templates kann die Belegung des Parameters auf zwei Arten geschehen. Die erste Möglichkeit ist, den Knoten, der den formalen Parameter repräsentiert, auf den Wert des aktuellen Parameters verweisen zu lassen. Die zweite Möglichkeit ist, den aktuellen Parameter direkt an jeder Stelle, an der der Parameter verwendet wird, einzusetzen. Zuerst wurde eine Entscheidung zugunsten der ersten Möglichkeit getroffen, weil die Verwendung von Templateparametern in der Instatiierung dann noch als solche erkannt werden können. Beim Entwurf von cafe++ stellte sich jedoch heraus, dass nur die zweite Möglichkeit umsetzbar ist (siehe Kapitel 4.4).

2.3.10.5 IML-Erweiterungen für Templates

S.S.

In diesem Kapitel wird erklärt, wie sich die Darstellung von Templates in der IML von der von normalen Klassen bzw. Funktionen unterscheidet. Zuerst wird die Modellierung mit Hilfe eines Attributs Is_Template, für die wir uns entschieden haben, erläutert. Anschließend werden vier Alternativen mit ihren Vor- und Nachteilen angeführt.

Erkennung von Templates durch das Attribut Is_Template Eine Anforderung war, dass man bei jedem Knoten schnell erkennen kann, ob es sich um ein Template handelt oder um einen normal benutzbaren Knoten. Es darf dazu z.B. nicht notwendig sein, mehrerer Kanten zu traversieren, um festzustellen, ob nicht belegte Templateparameter existieren, und somit der zu überprüfende Knoten ein Template ist.

Ob ein IML-Knoten ein Template repräsentiert ist, je nach Modellierung, entweder durch ein Attribut Is_Template oder durch den Typ des Knotens erkennbar. In Kapitel 2.3.10.6 werden verschiedene Möglichkeiten zur Erkennung durch den Typ des Knotens beschrieben. Die Modellierung durch das Attribut Is_Template, die tatsächlich umgesetzt wurde, wird zuerst erklärt, so dass bei den Alternativen nur noch die Unterschiede genannt werden müssen.

Es wurde jedoch entschieden, ein Attribut zu benutzen, anstatt die IML-Klassenhierarchie komplizierter zu machen. Falls sich diese Entscheidung als

unvorteilhaft herausstellt, kann sie später mit relativ geringem Aufwand geändert werden. Eine Korrektur der Entscheidung für das Vorgehen in „Alternative 4“ auf Seite 49 wäre wesentlich schwieriger.

Die Verwendung eines Attributs anstelle verschiedener Typen hat den Nachteil, dass die Überprüfung des Attributs bei der IML-Analyse vergessen werden kann. Allerdings können Analysen, die Templates nicht beachten, niemals auf Templates stoßen. Um ein Template im IML-Graph zu erreichen, muss das weiter unten beschriebene Attribut The_Template ausgewertet werden.

Der SymNode-Teil der IML-Knotentyphierarchie nach der Erweiterung für Templates ist in Abbildung 2.9 dargestellt.

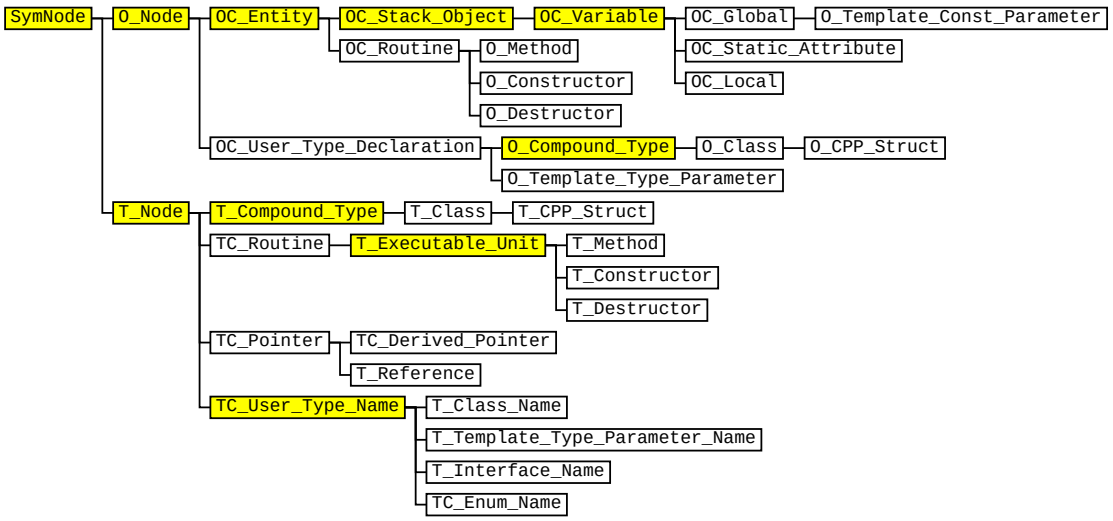


Abbildung 2.9: IML-Knotenhierarchie-Ausschnitt bei der Erkennung von Templates durch ein Attribut

Im Folgenden werden alle Attribute beschrieben, die der IML hinzugefügt wurden, um Templates beschreiben zu können.

Neue Attribute in SymNode:

Is_Template: boolean
Is_Exported: boolean

Is_Template ist „wahr”, wenn der Knoten ein Template modelliert. Is_Template ist auch „wahr”, wenn es sich um ein teilweise instantiiertes Template handelt. Is_Template ist „unwahr”, wenn es sich um ein instantiiertes Template oder kein Template handelt. Für Knoten, bei denen Is_Template „unwahr” ist, müssen auf der IML arbeitende Analysen die Besonderheiten der Modellierung von Templates nicht berücksichtigen.

Da die Anzahl von SymNodes relativ klein ist im Vergleich zur Anzahl der HPGNodes, ist der Platzbedarf für dieses Attribut nicht zu groß, auch wenn es so weit oben in der Knotentyphierarchie eingefügt wird.

Is_Exported wurde zum Zeitpunkt der Fertigstellung dieser Diplomarbeit noch nicht in die IML eingefügt, könnte aber eine interessante Zusatzinformation für Analysen sein, die die Schnittstelle zwischen Übersetzungseinheiten analysieren. Es gibt an, ob ein Template bzw. die Instantiierung eines Templates aus der Übersetzungseinheit exportiert wird.

Neue Attribute von Klassendeklarationen (O_Class)

```
The_Template: O_Class
```

Das Attribut The_Template zeigt auf das Template, aus dem die Instanz gebildet wurde. Es ist Null, falls der Knoten nicht aus einem Template instantiiert wurde, also bei Templates, die nicht aus anderen Templates durch Spezialisierung entstanden sind und bei Knoten, die nie Templates waren.

```
Template_Type_Parameters: list of O_Template_Type_Parameter
Template_Const_Parameters: list of O_Template_Const_Parameter
```

Diese beiden Attribute modellieren die Template-Signatur, also eine Liste aus Deklarationen von Templateparametern. In der Template-Signatur im Quelltext können Typ- und Konstanten-Templateparameter beliebig gemischt auftreten. Die genaue Reihenfolge der Parameter geht durch die Aufteilung in zwei Listen verloren, kann jedoch durch die Quelltextpositionsinformation rekonstruiert werden. Die Aufteilung in zwei Listen geschieht, um eine höhere Typsicherheit zu erreichen.

Alle drei Attribute mussten bei der IML-Klasse für Deklarationen, also bei O_Class und nicht bei T_Class, hinzugefügt werden, weil sie auch für Forward-Deklarationen gebraucht werden. Falls die Information über Template-Parameter auch für den Typ verfügbar sein muss, kann in T_Class ein Attribut Definition eingefügt werden, das auf die Definition der Klasse verweist. Die Definition der Klasse ist in jeder Übersetzungseinheit, in der ein entsprechender T_Class-Knoten vorkommt, vorhanden.

Neue Attribute von Funktionen (OC_Routine)

```
The_Template: OC_Routine
Template_Type_Parameters: list of O_Template_Type_Parameter
Template_Const_Parameters: list of O_Template_Const_Parameter
```

Die Bedeutung der Attribute entspricht der der gleichnamigen Attribute von O_Class.

Diese Attribute müssen zu OC_Routine, nicht zu TC_Routine, weil sie nicht zur Signatur gehören. Das ist daran erkennbar, dass im Typ von Funktionszeigern nie Templateparameter auftreten, weil Funktionszeiger nur auf *echte* Routinen zeigen können.

Die Attribute dürfen auch nicht zu Routine, weil sie auch für Forward-Deklarationen gebraucht werden.

Attribute von T_Template_Type_Parameter_Name

keine

Knoten vom Typ T_Template_Type_Parameter_Name modellieren die Verwendung von Typ-Templateparametern (siehe Kapitel 2.3.10.4).

Attribute von O_Template_Type_Parameter

The_Template: O_Template_Type_Parameter
Default: T_Node
Parameter_Of: class O_Node

Knoten vom Typ O_Template_Type_Parameter modellieren die Deklaration von Typ-Templateparametern (siehe Kapitel 2.3.10.4). Die Attribute werden in Kapitel 2.4.11 detailiert erklärt.

Attribute von O_Template_Const_Parameter

Default: Value
Parameter_Of: class O_Node

Knoten vom Typ O_Template_Const_Parameter modellieren die Deklaration von Konstanten-Templateparametern (siehe Kapitel 2.3.10.4). Die Verwendung geschieht über Entity_L_Value-Knoten, wie bei normalen konstanten Variablen.

Wenn Standardwerte (defaults) für Templateparameter vorhanden sind, sind die Typen/Konstanten der Parameter Null und Is_Template ist „wahr". Deshalb wird auch eine Kopie gemacht, wenn alle Templateparameter Standardwerte haben und eine Instantiierung diese nicht überschreibt.

Das Attribut „Default" von O_Template_Const_Parameter ist ein semantisches Attribut, da der Parameter selbst durch einen SymNode-Knoten dargestellt wird. Der Ausdruck für den Standardwert ist jedoch ein HPG-Knoten, muss also im HPG eingehängt werden. Deshalb wird der Standardwert in der Initialisierungsfunktion der Übersetzungseinheit ausgewertet, der Wert wird dort aber nicht verwendet.

Template-Spezialisierungen Wenn spezialisierte Templates instantiiert werden, zeigt das The_Template-Attribut der Instantiierung auf das spezialisierte Template. Erst das The_Template-Attribut der Spezialisierung zeigt dann auf das Haupt-Template.

Damit Spezialisierungen sicher von normalen Instantiierungen unterschieden werden können, wird bei Instantiierungen auch von vollständig spezialisierten Templates eine Kopie erzeugt. Is_Template von Spezialisierungen ist immer „wahr".

2.3.10.6 Alternative IML-Erweiterungen für Templates

S.S.

In diesem Kapitel werden Alternativen zu der tatsächlich umgesetzten Modellierung aus Kapitel 2.3.10.5 vorgestellt. Bei jeder Alternative wird begründet, warum sie nicht ausgewählt wurde.

Alternative 1 Alternative 1 ist eine der Alternativen zur Erkennung von Templates durch das Attribut Is_Template. Hier werden Templates von normalen Klassen oder Funktionen unterschieden, indem sie durch Knoten eines anderen Typs modelliert werden, als normale Klassen oder Funktionen. Es werden nur die Unterschiede zu „Erkennung von Templates durch das Attribut Is_Template“ auf Seite 42 beschrieben.

In dieser Modellierung sind IML-Klassen für Templates Oberklassen der entsprechenden normalen Klassen. Dies ist in Abbildung 2.10 auf der nächsten Seite dargestellt.

Problem bei O_Template_Class und T_Template_Class-Knoten Es kann nicht unterschieden werden, ob ein Knoten eine Template-Klasse oder -Struktur repräsentiert, weil beide durch Knoten vom selben Typ dargestellt werden. Eine mögliche Lösung wäre, Klassen und Strukturen nicht durch verschiedene IML-Klassen sondern durch ein Attribut (vom Typ *Aufzählung*) zu unterschieden.

Problem bei O_Template_Routine-Knoten Analog zum Problem oben kann nicht unterschieden werden, ob ein Knoten eine Template-Routine, Template-Methode oder einen Template-Konstruktor repräsentiert. Destruktoren können nie Templates sein.

Mögliche Lösungen wären:

1. Teilweise Duplikation der IML-Klassenhierarchie – siehe Alternative 2 und 3.

   Nachteil: Es entstehen viele IML-Klassen, was zu einer komplizierten Typhierarchie führt.

2. Modellierung durch Attribute anstatt durch Vererbung:

   (a) Wie in „Erkennung von Templates durch das Attribut Is_Template“ auf Seite 42 beschrieben.

   (b) Zwischen Routinen, Methoden, Konstruktoren und Destruktoren wird durch ein oder mehrere Aufzählungsattribute unterschieden.

   Nachteil: Das ist eine große Änderung der vorhandenen Klassenhierarchie, da Attribute von T_Constructor nach T_Routine verschoben werden müssen, obwohl sie nur für Konstruktoren sinnvoll sind.

3. Im Fall von Templates wird nicht zwischen Routinen, Methoden und Konstruktoren unterschieden.

   Nachteil: Der IML-Graph enthält weniger Informationen. Deshalb ist ein Wechsel zu einer der anderen Möglichkeiten schwieriger als umgekehrt.

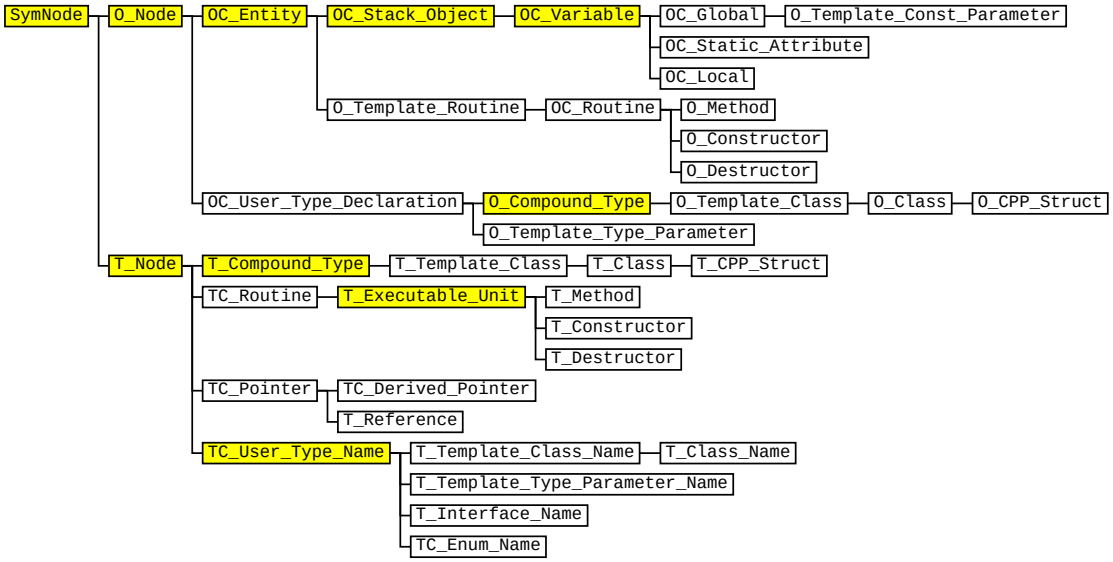


Abbildung 2.10: IML-Knotenhierarchie-Ausschnitt zur Alternative 1

Attribute von Template-Klassendeklarationen (O_Template_Class)

The_Template: O_Template_Class
Template_Type_Parameters: list of O_Template_Type_Parameter
Template_Const_Parameters: list of O_Template_Const_Parameter

Attribute von Funktionen (O_Template_Routine)

The_Template: O_Template_Routine
Template_Type_Parameters: list of O_Template_Type_Parameter
Template_Const_Parameters: list of O_Template_Const_Parameter

Mit dieser Modellierung werden zwar, im Gegensatz zu „Erkennung von Templates durch das Attribut Is_Template“ auf Seite 42, keine Attribute zu vorhandenen Klassen hinzugefügt. Die Konstruktoren für die IML-Knoten ändern sich aber trotzdem, weil neue Attribute geerbt werden.

Wegen der oben beschriebenen Probleme wurde diese Alternative nicht ausgewählt.

Alternative 2 Alternative 2 ist eine der Alternativen zur Erkennung von Templates durch das Attribut Is_Template. Wie in „Alternative 1“ auf der vorherigen Seite werden Templates von normalen Klassen oder Funktionen unterschieden, indem sie durch Knoten eines anderen Typs modelliert werden, als normale Klassen oder Funktionen. Es werden deshalb nur die Unterschiede zu „Alternative 1“ auf der vorherigen Seite beschrieben.

In der IML-Knotentyphierarchie, die in Abbildung 2.11 auf der nächsten Seite dargestellt ist, sind alle IML-Klassen für nicht-instantiierte Templates außerhalb von O_Node und T_Node. So können Analysen sicher sein, dass O_Nodes und T_Nodes keine Templates repräsentieren.

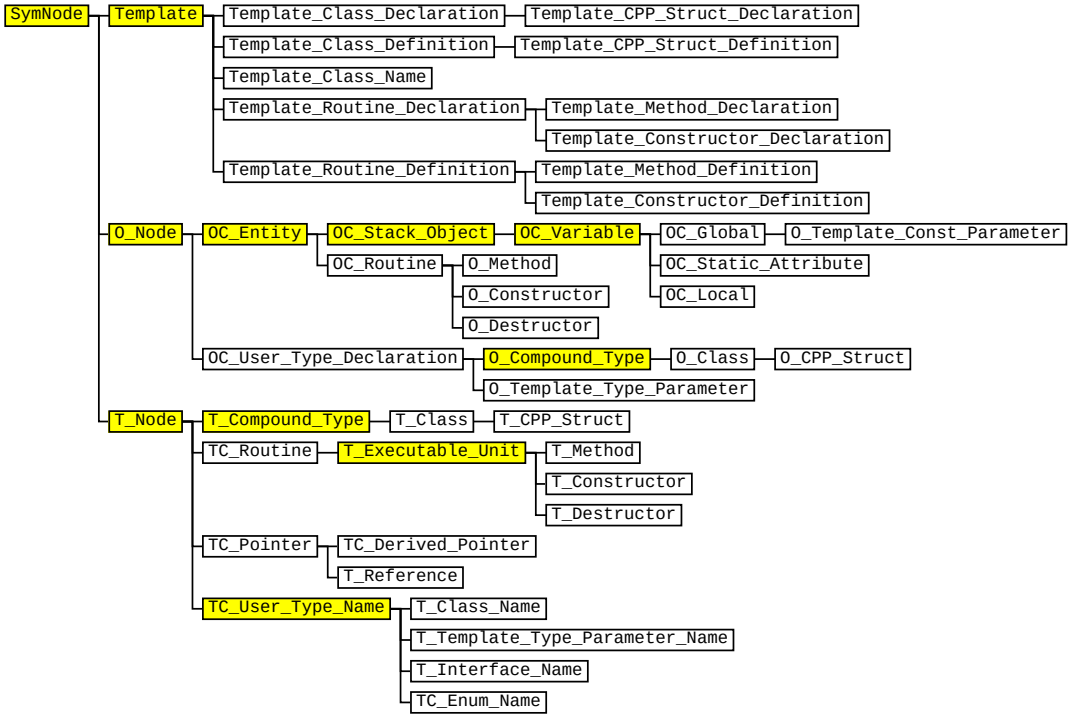


Abbildung 2.11: IML-Knotenhierarchie-Ausschnitt zur Alternative 2

Attribute von T_Template_Type_Parameter_Name

Parameter_Of: class Template

Attribute von O_Template_Type_Parameter

The_Template: O_Template_Type_Parameter
Default: T_Node
Parameter_Of: class Template

Attribute von O_Template_Const_Parameter

Default: OC_Variable
Parameter_Of: class Template

Diese Alternative wurde nicht ausgewählt, weil durch die Duplikation eines Teils der SymNode-Hierarchie ein erhöhter Wartungsaufwand für die IML entstehen würde. Änderungen in der IML-Spezifikation von O_Nodes und T_Nodes müssten sonst bei Templates nachgezogen werden und umgekehrt.

Alternative 3 Alternative 3 ist eine der Alternativen zur Erkennung von Templates durch das Attribut Is_Template. Wie in „Alternative 1“ auf Seite 46 werden Templates von normalen Klassen oder Funktionen unterschieden, indem sie durch Knoten eines anderen Typs modelliert werden als normale Klassen

oder Funktionen. Es werden deshalb nur die Unterschiede zu „Alternative 1“ auf Seite 46 beschrieben.

In der IML-Knotentyphierarchie, die in Abbildung 2.12 dargestellt ist, sind alle IML-Klassen für nicht-instantiierte Templates parallel zu den entsprechenden normalen Klassen angeordnet. Hierbei ist nicht ganz klar, wo die Templateklassen genau eingeordnet werden sollen; beispielsweise, ob T_Template_Class wirklich ein T_Compound_Type ist.

Auch diese Alternative wurde nicht ausgewählt, da die Klassenhierarchie unübersichtlicher ist als bei Alternative 1. Außerdem ist die Typsicherheit kaum höher als bei Alternative 1, da eine Analyse, die z.B. auf OC_User_Type_DeclaraFtion-Knoten arbeitet und den genaueren Typ nicht beachtet, nicht bemerkt, wenn sie ein Template vor sich hat. In Alternative 1 wäre ein Template in diesem Fall sogar einfacher zu erkennen.

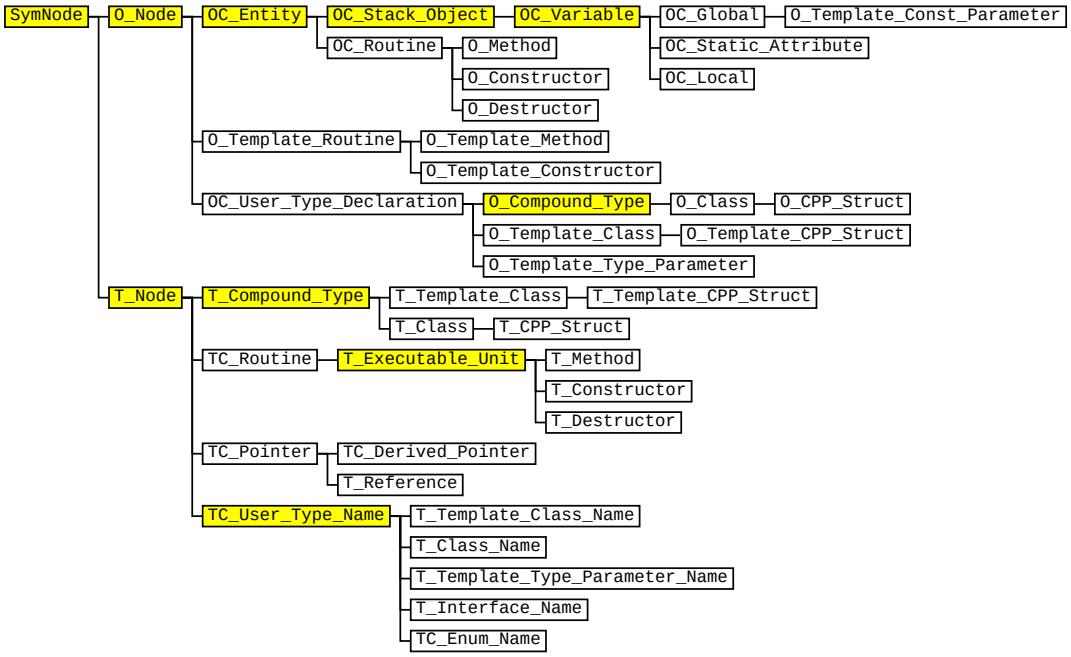


Abbildung 2.12: IML-Knotenhierarchie-Ausschnitt zur Alternative 3

Alternative 4 Alternative 4 beschreibt eine Modellierung von Templates, die so minimal gehalten wurde, dass sie gerade die Information bereitstellt, die für bestehende Analysewerkzeuge ausreicht.

Zur IML wird nur eine Klasse, Template, und das Attribut The_Template hinzugefügt. Alle Templates werden als Knoten vom Typ Template dargestellt. Instanztiierungen verweisen mit The_Template auf das entsprechende Template. Parameter werden nicht repräsentiert – in den Instantiierungen sind die aktuellen Parameter direkt an Stelle der formalen Parameter eingesetzt.

Diese Vorgehensweise ist ausreichend für Analysen, auf die Templates keinen Einfluss haben, wie Kontroll- und Datenflussanalysen. Sie reicht auch aus

für die Erstellung von RFGs, da RFGs nur Typen und Routinen enthalten und keine einzelnen Template-Parameter (siehe [18]).

Alternative 4 ist einfach zu implementieren und die IML-Erweiterung ist klein und gut verständlich. Allerdings könnten die Informationen für zukünftige Analysen zu wenig sein. Ein Umstieg auf eine der anderen Alternativen ist später aber nur mit sehr großem Aufwand möglich, weshalb auch diese Alternative nicht ausgewählt wurde, sondern die Modellierung mit Hilfe des Attributs Is_Template, die am Anfang des Kapitels beschrieben wurde.

2.3.10.7 Beispiele

S.S.

Die hier angeführten Beispiele beziehen sich auf „Erkennung von Templates durch das Attribut Is_Template“ auf Seite 42.

Template Funktion mit Typ- und Konstantentemplateparameter In Abbildung 2.13 auf Seite 87 ist ein Beispiel für eine Template-Funktion abgebildet. Der Graph über der gestrichelten Linie zeigt die Modellierung des Templates. Der Graph unter dem Strich zeigt die Instantiierung des Templates. Die Parameter `i` und `j` sind Konstanten-Templateparameter. Sie können innerhalb der Definition der Funktion wie normale Konstanten benutzt werden, im Beispiel zur Initialisierung von `t1`. Der Typ von `i` ist selbst ein Template-Parameter der Funktion, wird also erst bei der Instantiierung der Funktion festgelegt.

Das Zuweisen von Werten an die Konstanten-Templateparameter geschieht analog wie bei normalen Konstanten. Die Konstanten-Templateparameter des Templates werden bei der Instantiierung durch Initialize-Knoten mit den aktuellen Parametern belegt. Bei dem Typ-Templateparameter wird das User_Type Attribut gesetzt.

Die T_Template_Type_Parameter_Name-Knoten werden für jede Verwendung des Parameters neu angelegt. Dadurch kann für jede Benutzung die Quelltextposition (sloc) der Benutzung repräsentiert werden, wie das auch bei TC_Typedef_Name der Fall ist.

Verschachtelte Template-Deklaration In Abbildung 2.14 auf Seite 88 ist der Parameter `C` der Klasse `Ex` eine Klasse, die selbst einen Template-Typ-Parameter hat. Wie beim vorherigen Beispiel ist überhalb der gestrichelten Linie das Template modelliert und darunter die Instantiierung. Bei der Instantiierung wird der Parameter `C` mit der Klasse `myarray` belegt. Der Parameter `R` von `myarray` wird mit `S` belegt.

Verschachtelte Templates werden wie normale Templates behandelt, d.h. der User_Type von O_Template_Type_Parameter ist im Template Null. Bei Instantiierungen ergibt sich ein Problem: die Type_Declaration eines T_Template_Type_Parameter_Name zur Benutzung des Templates zeigt eigentlich auf die Deklaration des Templateparameters, der User_Type dieser Deklaration ist aber ein Template. Für die Analysen sollte der User_Type eine Instanz dieses Templates sein. Deshalb wird auch der Templateparameter instantiiert, also kopiert. Der Parameter selbst verweist weiter auf das Template, mit dem

er belegt wurde, die Instantiierung des Parameters verweist auf die Instantiierung des Templates.

2.3.11 L-Values

S.S.

L-Values sind Ausdrücke, die auf der linken Seite einer Zuweisung stehen können, also Werte, die verändert werden können. Dazu gehören Zugriffe auf Variablen und Dereferenzierungen von Zeigern. In der IML gibt es ein Interface „L_Value". Alle IML-Knotentypen, die L-Values sein können, implementieren dieses Interface.

In der IML müssen auch Ausdrücke mit Strukturtyp, auf die mit einem Field_Selection-Operator zugegriffen wird, L-Values sein. Das ist unabhängig davon, ob die Elementselektion als L-Value benutzt wird. Es reicht aus, dass eine Elementselektion als L-Value benutzt werden *könnte*. Obwohl das Ergebnis eines Funktionsaufrufs nicht verändert werden kann, muss es in der IML als L-Value dargestellt werden, weil eine Elementselektion möglich ist, falls das Ergebnis Strukturtyp hat.

In der IML für C wurde übersehen, dass das Ergebnis einer C_Comma_Sequence auch Strukturtyp haben kann und deshalb das Interface L_Value implementieren sollte.

In C++ können Zuweisungen (durch die Operatoren „=", „+=", „*=" usw.) auch L-Values sein, in C jedoch nicht. Die IML sollte nicht so geändert werden, dass Zuweisungen immer L-Values sind. Wenn zu viele Knoten in der IML L-Values sind, können Analysen keinen echten Vorteil mehr aus der Unterscheidung von L- und R-Values ziehen.

Um die Probleme der Modellierung durch ein Interface L_Value zu umgehen, werden L-Values in der IML zukünftig möglicherweise nicht mehr durch ein Interface, sondern durch ein boolesches Attribut gekennzeichnet. Da auch die IML für C und viele Analysen betroffen sind, wird eine Entscheidung dazu nicht im Rahmen dieser Diplomarbeit getroffen, sondern auf dem nächsten IML-Treffen der Abteilung Programmiersprachen und Übersetzerbau.

2.4 Spezifikation der IML für C, C++ und Java

T.K.

Dieses Kapitel beschreibt nun detailliert die Spezifikation der IML-Darstellung für C, C++ und Java Quelltexten. Im Gegensatz zu Kapitel 2.3 wird hier jedoch auf eine Argumentation der Vorgehensweise bei der IML-Erweiterung verzichtet.

Beschrieben werden in dieser Spezifikation alle Knotentypen mit ihren Attributen, soweit sie für die Modellierung von Java und C++ Programmen relevant sind. Knotentypen, die sich gegenüber der IML für C nicht geändert haben, sollen hier nicht genauer beschrieben werden.

Für alle Attribute von Knotentypen, die Listen enthalten, sind die Elemente der Listen nach deren Vorkommen im Quelltext sortiert. Bei der weiteren Beschreibung der Attribute wird auf diese Eigenschaft nicht mehr gesondert hingewiesen.

Die folgenden Unterkapitel sind thematisch geordnet und haben denselben Aufbau. Der Aufbau und die verwendetete Notation entspricht der in IML-Spezifikation für Java eingesetzten [19]. Zu Beginn wird die Thematik vorgestellt und meist ein Beispiel für die IML-Modellierung des Themas aufgeführt, um das Verständnis zu vereinfachen. Nach der Einführung veranschaulicht ein Ausschnitt aus der Typhierarchie die Position der relevanten Knoten in der IML-Typhierarchie, die zur Modellierung der Thematik zum Einsatz kommen. In dieser Typhierarchie ist der Übersichtlichkeit wegen immer die vollständige Vererbung bis zur gemeinsamen Oberklasse IML_Root angegeben. Nach der Typhierarchie werden daraufhin alle beteiligten Knoten mit ihren Attributen im Detail beschrieben.

Die Notation der IML-Typhierarchie ist in UML-Klassendiagrammen [29] verfasst, wobei zusätzlich alle gegenüber der IML für C neu hinzugekommenen Typen grau hinterlegt sind.

Die Notation der Typbeschreibungen ist ebenfalls an die Klassendiagramme der UML angelehnt. Wie bei der IML-Typhierarchie werden hier Knotentypen und Attribute, die sich gegenüber der IML für C geändert haben, grau hinterlegt. Um semantische von syntaktischen Kanten unterscheiden zu können, werden semantische Attribute weiterhin in kursivem Text gesetzt, wohingegen syntaktische nicht-kursiv gesetzt sind.

Bei den Beispielen, bei denen UML-Objektdiagramme zum Einsatz kommen, wurde die Notation ebenfalls erweitert. Um die Richtung der Kanten anzuzeigen, sind die Assoziationskanten in den Objektdiagrammen ebenfalls gerichtet. Gestrichelte Assoziationskanten repräsentieren weiterhin semantische und durchgezogene syntaktische Kanten. Die Stereotype einer Kante zeigt das Attribut des verweisenden Knotens an.

2.4.1 Oberer Teil der IML-Hierarchie

T.K.

Typhierarchie

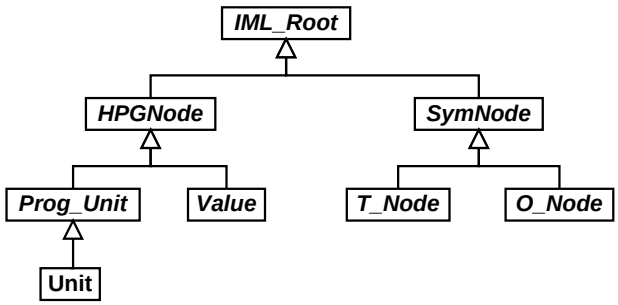


Abbildung 2.15: Oberer Teil der IML-Typhierarchie

Typbeschreibung

IML_Root	Oberklasse aller IML Klassen.
Parent: class IML_Root	Verweis auf den Elternknoten im hierarchischen Programmgraph. Rückwärtskante zu einer syntaktischen Kante. Dieses Attribut gehört eigentlich in die Klasse HPGNode und sollte den Typ HPGNode haben. Der IML-Generator weiß aber nicht, dass syntaktische Kanten nur zwischen HPGNodes verlaufen dürfen.
SLoc: builtin SLoc	Die Quelltextposition, zu der der IML-Knoten gehört.
Artificial: builtin Boolean	„Wahr" für Knoten, die kein Quelltextelement repräsentieren.

HPGNode	Knoten des hierarchischen Programmgraphs.

Prog_Unit	Knoten für logische Einheiten im Programm.
Subunits: list of class Prog_Unit	Liste der verschachtelten Einheiten.
Symbol_Table: set of class O_Node	Alle Definitionen direkt innerhalb der Einheit.
Declaration_Table: set of class O_Node	Alle Deklarationen direkt innerhalb dieser Einheit, die keine Definitionen (also nicht in Symbol_Table enthalten) sind. Dieses Attribut wurde aus Unit hierher verschoben, weil es auch für Klassen benötigt wird und weil in C++ auch innerhalb von Funktionen Deklarationen auftreten, die keine Definitionen sind.

Unit	Knoten für Übersetzungseinheiten.
Initialization_Code: class Routine	Verweis auf eine Routine, die alle Ressourcen der Übersetzungseinheit mit globaler Lebensdauer initialisiert. Dies kann durch Zuweisung oder Konstruktoraufruf geschehen.
Finalization_Code: class Routine	Verweis auf eine Routine, die alle Ressourcen der Übersetzungseinheit mit globaler Lebensdauer freigibt. Ressourcen werden durch Destruktoraufrufe freigegeben.
Provided_Definitions: set of class OC_Entity	Menge aller Definitionen der Übersetzungseinheit, die dem Linker zum Auflösen der Unresolved_Declarations anderer Übersetzungseinheiten zur Verfügung gestellt werden.
Unresolved_Declarations: set of class OC_Entity	Menge aller Deklarationen, zu denen vom Linker eine Definition gefunden werden muss.
Global_Lifetime_Definitions: list of class OC_Variable	Liste aller Variablen mit globaler Lebensdauer.
Uses: list of builtin Identifier	Liste aller Namensbereiche, deren Namen in dieser Übersetzungseinheit verfügbar sind.

Value	Knoten zur Darstellung von Ausdrücken.
CF_Next: class Value	Kontrollflussinformation. Dieses Attribut wird nicht vom Front-End, sondern erst eine später durchgeführte Kontrollflussanalyse gesetzt.
CF_Previous: class Value	Kontrollflussinformation. Dieses Attribut wird nicht vom Front-End, sondern erst eine später durchgeführte Kontrollflussanalyse gesetzt.
CF_Basic_Block: builtin Basic_Block	Kontrollflussinformation. Dieses Attribut wird nicht vom Front-End, sondern erst eine später durchgeführte Kontrollflussanalyse gesetzt.
EType: class T_Node	Verweis auf einen Knoten, der den Typ des Ausdrucks beschreibt.

SymNode	Knoten für Symboltabelleneinträge. Diese Knoten gehören nicht zum HPG. Kanten von und zu den Knoten der Symboltabelle sind ausschließlich semantische Kanten.
Is_Template: builtin Boolean	„Wahr" für Knoten von nicht instantiierten Templates. Wenn Is_Template „unwahr" ist, kann der Knoten unabhängig von jeglichen Templates betrachtet werden. Das gilt auch für Knoten in instantiierten Templates.

T_Node	Knoten zur Beschreibung von Typen.
Type_Size: builtin Natural	Größe eines Objekts mit dem Typ in Bytes.

O_Node	Knoten für Deklarationen. Definitionen sind Deklarationen und werden auch durch O_Nodes beschrieben.
Mangled_Name: builtin Identifier	Der eindeutige Name, der deklariert wird. In Java und C++ ist das der erweiterte Name. In C sind Mangled_Name und Unmangled_Name gleich.
Unmangled_Name: builtin Identifier	Der Name, wie er in der Deklaration im Quelltext auftritt. Unmangled_Name kann Leerzeichen enthalten, wie in „`operator new`".
Namespace_Name: builtin Identifier	Namensbereich, zu dem diese Deklaration gehört. Klassen werden in diesem Zusammenhang auch als Namensbereiche betrachtet.
Its_Type: class T_Node	Verweis auf den Typ der deklarierten Einheit bzw. den Typ, der durch diesen Knoten deklariert wird.
Visibility: builtin Visibility_Definition	Sichtbarkeit des Namens von außerhalb der Klasse, falls die Deklaration in einer Klasse ist. Default_Decl, falls die Deklaration nicht innerhalb einer Klasse ist.

2.4.2 Klassen, Strukturen und Varianten

T.K.

Klassen, Strukturen und Varianten werden im HPG durch Knoten des abstrakten Typs Class_Type modelliert. Von diesem abgeleitet sind die Knoten Class und CPP_Union, welche jeweils Klassen (C++ und Java) und C++ Varianten darstellen. Der Typ CPP_Struct, der Strukturen in C++ darstellt, ist aufgrund

der identischen Semantik von Class abgeleitet. In 2.3.4 ist eine detaillierte Argumentation für diese Modellierung zu finden.

Die Vererbungshierarchie in der Symboltabelle spiegelt die Vererbungshierarchie für Klassen, Strukturen und Varianten aus dem HPG wider. Für die Modellierung von Klassen-, Struktur- und Variantendeklarationen stehen die Knoten O_Class, OCPP_Struct und OCPP_Union zur Verfügung. Die Knoten T_Class, TCPP_Struct und TCPP_Union repräsentieren die Typen von Klassen, Strukturen und Varianten. Dagegen stehen die Knoten T_Class_Name, TCPP_Struct_Name und TCPP_Union_Name für die Verwendung von Klassen, Strukturen und Varianten.

Die Darstellung von Vererbungs- und Implementierungsbeziehungen für Klassen und Schnittstellen hat sich gegenüber der IML für C und Java [19] nicht grundlegend geändert. Wie bisher geschieht dies über Extends_Relation-Knoten, welche von der erbenden Klasse hin zur beerbten zeigen. Neu ist jedoch, dass auch Vererbungsbeziehungen zwischen Klassen und Strukturen modelliert werden können. Diese Extends_Relation-Knoten werden praktisch als attributierte Knoten eingesetzt, welche auch die Eigenschaften der Abhängigkeit anzeigen.

Nach dem folgenden Beispiel zeigt Abbildung 2.17 auf der nächsten Seite die vollständige Vererbungshierarchie der IML-Knotentypen. Daraufhin folgt eine detaillierte Beschreibung aller IML-Knotentypen mit deren Attributen.

Beispiel

Folgender Quelltext soll die Modellierung von Klassen, Strukturen und Varianten in C++ veranschaulichen. Abbildung 2.16 auf der nächsten Seite zeigt ausschnittsweise den IML-Graphen für die Modellierung des Beispiels.

```
union Variante {
    int Attribut1;
    void Methode1();
};
struct Struktur {
    static int Attribut2;
    static void Methode2();
};
class Klasse : public Struktur {
};
```

Im HPG dieses IML-Graphen werden `Variante`, `Struktur` und `Klasse` durch Knoten vom Typ CPP_Union, CPP_Struct und Class modelliert. Die Knoten der Symboltabelle sind in der Abbildung des IML-Graphen direkt unter den HPG-Knoten platziert. Über die Definition-Kante gelangt man von den HPG-Knoten zu den Deklarationen, welche durch Knoten vom Typ OCPP_Union, OCPP_Struct und O_Class modelliert sind. Die Typen sind weiterhin über die Its_Type-Kanten der Deklarationen erreichbar und werden durch Knoten vom Typ TCPP_Union, TCPP_Struct und T_Class dargestellt.

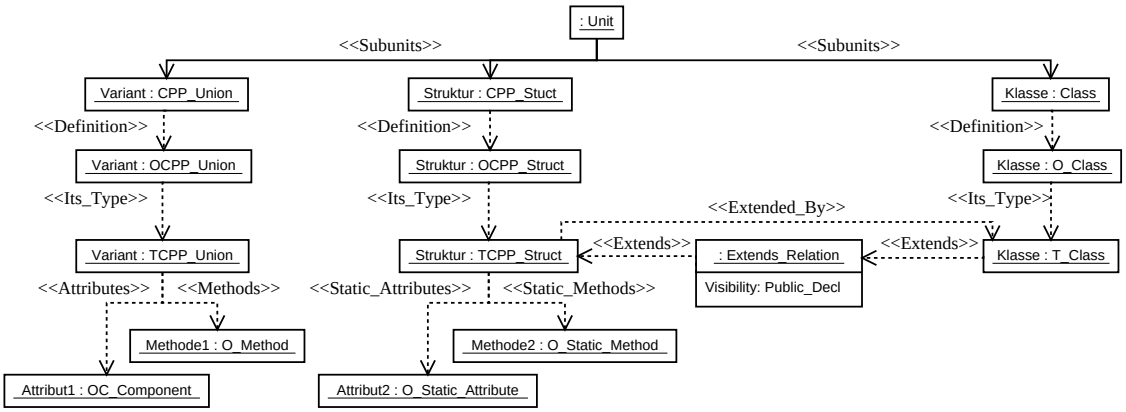


Abbildung 2.16: IML-Graph für das Beispiel Klassen-, Strukturen- und Variantenmodellierung

Das Beispiel zeigt weiterhin zwei Besonderheiten der IML für C++. Zum einen enthalten `Variante` und `Struktur` Methoden, wie aus deren Typknoten ersichtlich wird. Die Attribute von `Struktur` sind zusätzlich `static`, um die besondere Modellierung durch O_Static_Attribute- und O_Static_Method-Knoten veranschaulichen zu können. Außerdem besteht eine Vererbungsbeziehung zwischen `Klasse` und `Struktur`, was durch den Extends_Relation-Knoten dargestellt wird, welcher auch die Attribute der Vererbung trägt.

Typhierarchie

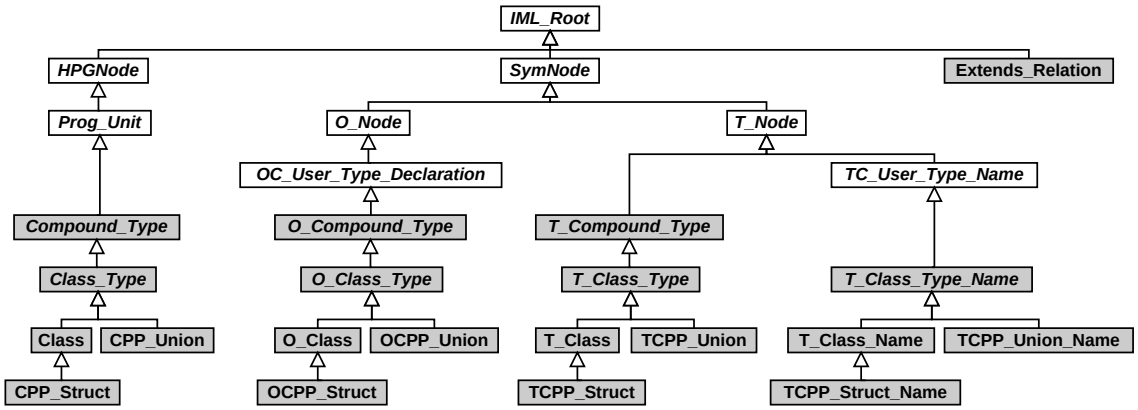


Abbildung 2.17: IML-Typhierarchie für die Modellierung von Klassen, Strukturen und Varianten

Typbeschreibung

O_Compound_Type	Abstrakter Deklarations-Knoten für die Gruppierung von Klassen- und Schnittstellendeklarationen.
HPG_Compound_Type: class Compound_Type	Verweis auf den HPG-Knoten, der diese Klasse oder diese Schnittstelle definiert.
Is_Anonymous: builtin Boolean	„Wahr", wenn diese Deklaration keinen Namen im Quelltext besitzt, d.h. ein anonymer Typ ist. „Unwahr" anderenfalls.

O_Class_Type	Abstrakter Deklarations-Knoten für definierte Klassen.
The_Template: class O_Class_Type	Verweis auf den generischen Typ, falls es sich um eine teilweise oder vollständige Instantiierung einer generischen Klasse handelt. Im Falle einer vollständigen Instantiierung ist dieser Knoten eine automatisch erzeugte Kopie der instantiierten Klasse. Null wenn es sich um eine nicht-generische Klasse oder eine nicht-instantiierte generische Klasse handelt.
Template_Type_Parameters: list of class O_Template_Type_Parameter	Liste der generischen Typparameter von Klassen. Die Liste enthält auch bereits instantiierte Parameter.
Template_Const_Parameters: list of class O_Template_Const_Parameter	Liste der generischen Wertparameter von Klassen. Die Liste enthält auch bereits instantiierte Parameter.
Friend_Of: set of class T_Compound_Type	Menge von Klassen, zu denen diese Klasse eine Freund-Beziehung hat, also auf die Attribute diese Klasse zugreifen kann.

O_Class	Deklarations-Knoten für Klassen (Java und C++).

OCPP_Struct	Deklarations-Knoten für C++ Strukturen.

OCPP_Union	Deklarations-Knoten für C++ Varianten.

T_Compound_Type	Abstrakter Typ-Knoten für die Gruppierung von Klassen- und Schnittstellentypen.
Is_Strictfp: builtin Boolean	„Wahr", wenn das Attribut `strictfp` für den modellierten Typ angegeben wurde. „Unwahr" anderenfalls und in allen Sprachen außer Java.
Extends: list of class Extends_Relation	Liste aller Typen, von denen der modellierte Typ erbt.
Extended_By: set of class T_Node	Menge aller Typen, die vom modellierten Typ erben.
Methods: list of class O_Method	Liste aller nicht-statischen Methoden, die für den modellierten Typ definiert oder deklariert wurden.
Nested_Types: list of class O_Compound_Type	Liste aller Klassen- oder Schnittstellentypen, die innerhalb des modellierten Typs deklariert oder definiert wurden.
Outer_Type: class T_Compound_Type	Verweis auf den umschließenden Typ, falls der modellierte Typ innerhalb eines anderen Typs definiert wurde. Null ansonsten.

T_Class_Type	Abstrakter Typ-Knoten für definierte Klassen, C++ Strukturen und C++ Varianten.
Is_Abstract: builtin Boolean	„Wahr" wenn der modellierte Typ abstrakt ist und damit nicht instantiiert werden kann. „Unwahr" ansonsten.
Is_Final: builtin Boolean	„Wahr" wenn der modellierte Typ mit `final` definiert wurde. „Unwahr" ansonsten und in allen Sprachen außer Java.
Friend_Routines: set of class OC_Routine	Menge aller Funktionen, die eine Freund-Beziehung zum modellierten Typ haben, also auf dessen Attribute zugreifen können.
Friend_Classes: set of class O_Class_Type	Menge aller Klassen, die eine Freund-Beziehung zum modellierten Typ haben, also auf dessen Attribute zugreifen können.
Virtual_Call_Table: list of class OC_Routine	Liste aller virtuellen Methoden, die in dem modellierten Typ deklariert, definiert oder geerbt wurden. Den Anfang der Liste bilden die geerbten Methoden, wonach die deklarierten und definierten aufgeführt sind.
Block_Initializers: list of class Routine	Liste aller Block-Initialisierer, die für den modellierten Typ definierten wurden (Java).
Attributes: list of class OC_Component	Liste aller nicht-statischen Attribute, die für den modellierten Typ deklariert wurden.
Static_Attributes: list of class O_Static_Attribute	Liste aller statischen Attribute, die für den modellierten Typ deklariert wurden.
Constructors: list of class O_Constructor	Liste aller Konstruktoren, die für den modellierten Typ deklariert wurden.
Finalizer: class O_Destructor	Verweis auf einen Destruktor, falls für den modellierten Typ ein Destruktor deklariert wurde. Null ansonsten.
Static_Methods: list of class O_Static_Method	Liste aller statischen Methoden, die für den modellierten Typ deklariert wurden.

T_Class	Typ-Knoten für Java- oder C++ Klassen.
Implements: list of class Extends_Relation	Liste aller Java-Schnittstellen, die diese Klasse implementiert.

TCPP_Struct	Typ-Knoten für C++ Strukturen.

TCPP_Union	Typ-Knoten für C++ Varianten.
T_Class_Type_Name	Abstrakter IML-Typ für die Verwendung von Klassen, C++ Strukturen und C++ Varianten.
T_Class_Name	Modelliert die Verwendung von Klassen.
TCPP_Struct_Name	Modelliert die Verwendung von C++ Strukturen.
TCPP_Union_Name	Modelliert die Verwendung von C++ Varianten.
Compound_Type	Abstrakter IML-Typ für die Gruppierung von Klassen und Schnittstellen im HPG. Klassen haben Attribute und ausführbare Methoden, die auf den Attributen arbeiten. Schnittstellen deklarieren lediglich Methoden, definieren diese aber nicht.
Class_Type	Abstrakter IML-Typ, der alle Klassen beinhaltet, die definiert sind.
Definition: class O_Class_Type	Verweis auf die Deklaration, die den Typ definiert.
Class	Modelliert Java- und C++ Klassen im HPG.
CPP_Struct	Modelliert C++ Strukturen im HPG.
CPP_Union	Modelliert C++ Varianten im HPG.
Extends_Relation	Modelliert eine Vererbungs- oder Implementierungsbeziehung.
Extends: class T_Compound_Type *Visibility: builtin Visibility_Definition* *Is_Virtual: builtin Boolean*	Verweis auf den beerbten Typ. Sichtbarkeit, die für die Vererbungsbeziehung gilt. „Wahr" wenn die Vererbung als `virtual` definiert wurde. „Unwahr" ansonsten und in Java.

2.4.3 Funktionen und Methoden

Zum Unterschied zwischen Methoden (...Method) und Funktionen (...Routine) siehe „Methoden und Attribute " auf Seite 37.

S.S.

Typhierarchie

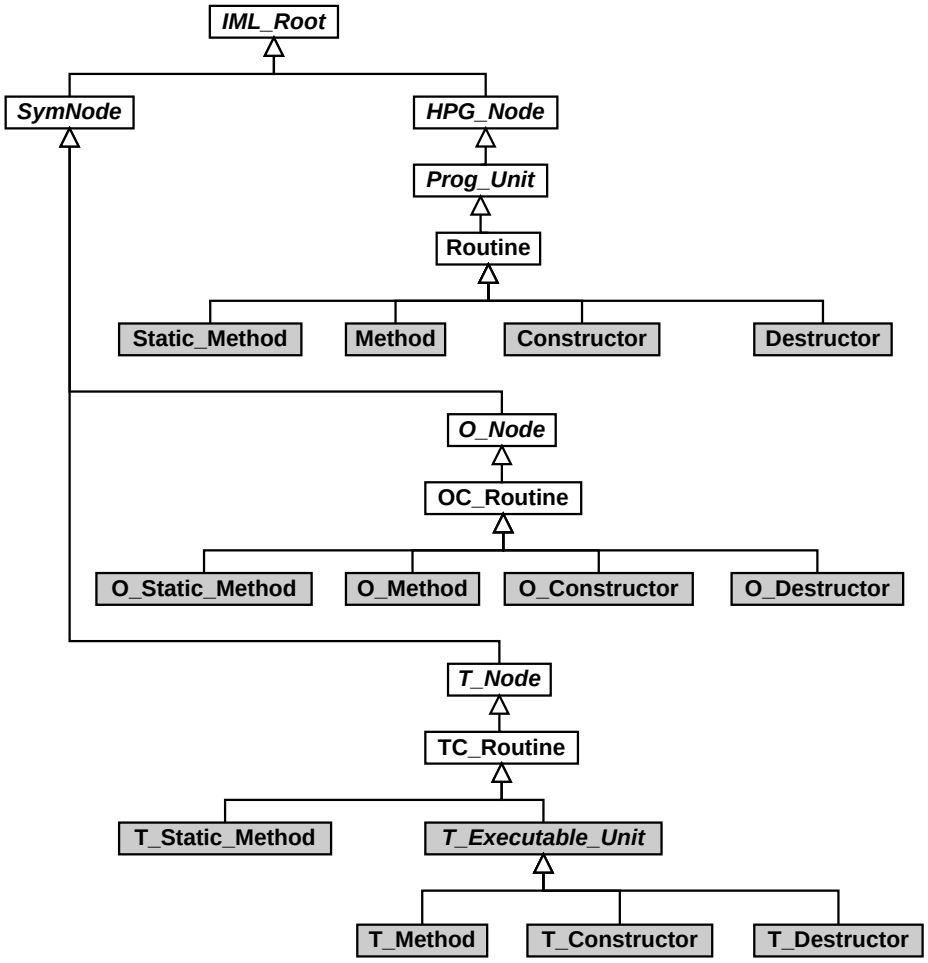


Abbildung 2.18: IML-Typhierarchie für die Modellierung von Funktionen und Methoden

Typbeschreibung

Method	Knoten für den ausführbaren Teil von Methoden.

Static_Method	Knoten für den ausführbaren Teil von statischen Methoden.

Constructor	Knoten für den ausführbaren Teil von Konstruktoren.
Base_Initialization: list of class Sequence_Item	Initialisierungen der Basisklassen. Enthalten sind die explizit angegebenen und die implizit erzeugten Initialisierungen.
Field_Initialization: list of class Sequence_Item	Initialisierungen der Komponenten. Enthalten sind die explizit angegebenen und die implizit erzeugten Initialisierungen.

Destructor	Knoten für den ausführbaren Teil von Destruktoren.

Das Attribut Its_Type von OC_Routine-Knoten ist immer vom Typ TC_Routine. In C++ sind die Attribute Return_Type, Params und Ellipsys von OC_Routine-Knoten immer genau gleich wie die gleichnamigen Attribute des entsprechenden TC_Routine-Knotens. Bei K&R C können sie unterschiedlich sein. Durch die Duplikation dieser Attribute kann in Analysen eine Dereferenzierung und eine Typkonvertierung eingespart werden.

OC_Routine	Deklarations-Knoten für Funktionen.
Params: list of class OC_Parameter	Liste der deklarierten Funktionsparameter.
Storage_Class: builtin C_Storage_Class	Storage Class, wie in der Deklaration angegeben.
Definition: class OC_Routine	Verweis auf die Definition dieser Deklaration. Verweist auf sich selbst, falls dieser Knoten die (eindeutige) Definition ist. Wird vom Linker gesetzt, falls die Definition in einer anderen Übersetzungseinheit ist als dieser Knoten und für inline Funktionen.
Ellipsis: builtin Boolean	„Wahr", wenn die Funktion eine unbekannte Anzahl Parameter hat.
HPG_Routine: class Routine	Verweis auf die Beschreibung des ausführbaren Teils der Funktion bei Definitionen. Null bei allen anderen Deklarationen.
PTFs: storable list of builtin Storable	Attribut, das für die Points-To-Analyse reserviert ist (PTF = Partial-Transfer-Function).
Return_Type: class T_Node	Verweis auf den Rückgabetyp der Funktion.
Allocate_Result: builtin Boolean	Dieses Attribut ist für spätere Analysen reserviert. Es zeigt an, ob für den Rückgabewert Speicher reserviert wird.
Unsure_Result: builtin Boolean	Dieses Attribut ist für spätere Analysen reserviert. Es zeigt an, ob es unsicher ist, ob für den Rückgabewert Speicher reserviert werden muss.
Friend_Of: set of class T_Class_Type	Menge aller Klassen, von denen die Funktion ein Freund ist.
Is_Operator: builtin Boolean	„Wahr" für Operatoren.
The_Template: class OC_Routine	Verweis auf das Template, von dem diese Funktion eine Instanz ist. Null, falls diese Funktion keine (partielle) Instanz eines Templates ist.
Template_Type_Parameters: list of class O_Template_Type_Parameter	Die Template-Typparameter dieser Funktion. Die Liste ist nur leer, falls es keine Parameter gibt. Bereits instantiierte Parameter sind auch aufgeführt.
Template_Const_Parameters: list of class O_Template_Const_Parameter	Die Template-Konstanten-Parameter dieser Funktion. Die Liste ist nur leer, falls es keine Parameter gibt. Bereits instantiierte Parameter sind auch aufgeführt.

O_Method	Deklarations-Knoten für Methoden. Der erste Eintrag in dem geerbten Attribut Params ist der implizite `this`-Zeiger.

O_Static_Method	Deklarations-Knoten für statische Methoden. Statische Methoden haben keinen `this`-Zeiger.

O_Constructor	Deklarations-Knoten für Konstruktoren. Der erste Eintrag in dem geerbten Attribut Params ist der implizite `this`-Zeiger.
Is_Explicit: builtin Boolean	„Wahr", wenn der Konstruktor explizit aufgerufen werden muss.

O_Destructor	Deklarations-Knoten für Destruktoren. Der erste Eintrag in dem geerbten Attribut Params ist der implizite `this`-Zeiger.

TC_Routine	Typ-Knoten für Funktionen und für den Ziel-Typ von Funktionszeigern.
Params: list of class OC_Parameter	Siehe OC_Routine oben.
Return_Type: class T_Node	Siehe OC_Routine oben.
Ellipsis: builtin Boolean	Siehe OC_Routine oben.
Throws: list of class T_Node	Typen von Ausnahmen, die von dieser Funktion geworfen werden können. Diese Liste enthält nur die Typen, die in der Deklaration stehen. Die Liste ist leer, wenn keine oder eine leere Liste im Quelltext angegeben wurde.
Is_Synchronized: builtin Boolean	Wird nur für Java verwendet [19].
Is_Native: builtin Boolean	Wird nur für Java verwendet [19].
Is_Strictfp: builtin Boolean	Wird nur für Java verwendet [19].
Is_Inline: builtin Boolean	„Wahr" für Typen von inline-Funktionen im Sinne von C++ (d.h. es ist eine Empfehlung an den Übersetzer).

T_Static_Method	Typ-Knoten für statische Methoden.
Its_Class: class O_Compound_Type	Verweis auf die Klasse, zu der die statische Methode gehört.

T_Executable_Unit	Typ-Knoten für nicht-statische, ausführbare Einheiten in Klassen, in den folgenden Attributbeschreibungen Methoden genannt. Der erste Parameter ist der `this`-Zeiger.
This_Offset: builtin Natural	Position des `this`-Zeigers in der Parameterliste. Immer „1" für C++.
Is_Abstract: builtin Boolean	„Wahr", wenn die Methode als abstrakt deklariert wurde. In C++ kann die Methode trotzdem definiert und aufgerufen werden. Kann nur „wahr" sein, wenn Is_Virtual auch „wahr" ist.
Is_Virtual: builtin Boolean	„Wahr", wenn die Methode virtuell ist, d.h. wenn eine dynamische Bindung über den `this`-Zeiger beim Aufruf stattfindet.
Its_Class: class O_Compound_Type	Verweis auf die Klasse, zu der die Methode gehört.

T_Method	Typ-Knoten für Methoden.
Is_Constant: builtin Boolean	„Wahr", wenn die Methode das Objekt nicht verändert, mit Ausnahme von `mutable`-Attributen. D.h. der `this`-Zeiger zeigt auf ein konstantes Objekt.

T_Constructor	Typ-Knoten für Konstruktoren.

T_Destructor	Typ-Knoten für Destruktoren.

T.K.

2.4.4 Virtuelle Methodenaufrufe

Die Modellierung von virtuellen Methodenaufrufen hat sich gegenüber der IML-Spezifikation für C und Java [19] nicht grundlegend geändert. Der einzige Unterschied besteht darin, dass virtuelle Methoden in C++ auch auf Instanzen von Strukturen aufgerufen werden können. Außerdem sind Methoden in C++ primär nicht-virtuell, es sei denn, sie sind mit dem Schlüsselwort `virtual` gekennzeichnet.

Knoten vom Typ Virtual_Call stellen einen virtuellen Methodenaufruf im HPG dar. Entscheidendes Merkmal eines virtuellen Methodenaufrufs gegenüber einem direkten Aufruf ist, dass die aufgerufene Methode erst zur Laufzeit ermittelt werden kann, da sie von der Instanz abhängt, auf der sie aufgerufen wird. Die Menge der Methoden, die bei einem virtuellen Methodenaufruf zur Verfügung stehen, setzt sich zusammen aus der Methode im definierten Typ der Instanz oder einer Unterklasse des Typs, in der die Methode überschrieben wurde. Knoten des Typs Virtual_Field_Selection modellieren genau diese Auswahl aus der gegebenen Menge von Methoden zur Laufzeit.

Abbildung 2.19 zeigt die Knotentyphierarchie für die Darstellung von virtuellen Methodenaufrufen, wonach eine genaue Beschreibung der Knotentypen folgt.

Typhierarchie

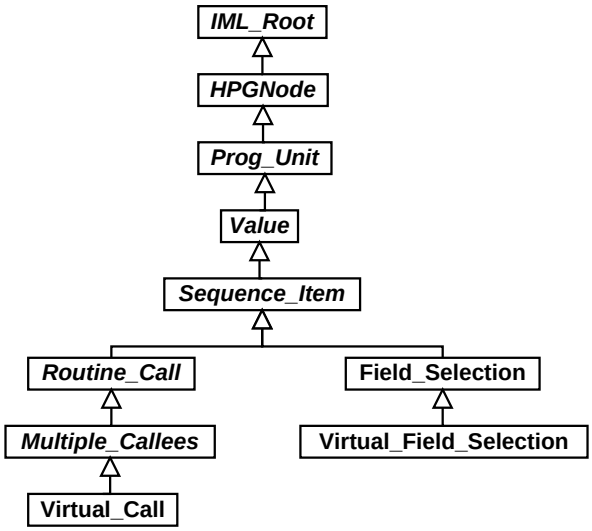


Abbildung 2.19: IML-Typhierarchie für die Modellierung von virtuellen Methodenaufrufen

Typbeschreibung

Virtual_Call	Modelliert den Aufruf einer virtuellen Methoden im HPG. Das von Routine_Call geerbte Attribut Expr ist bei Knoten dieses Typs immer vom Typ Virtual_Field_Selection.

Virtual_Field_Selection	Modelliert die Auswahl der konkreten Methode bei einem virtuellen Methodenaufruf über die virtuelle Funktionstabelle (Attribut Virtual_Call_Table in T_Class_Type). Das von Field_Selection geerbte Attribute Component verweist hier immer auf den definierten Methodentyp (OC_Routine). Die dynamische Instanz wird durch das Attribut Operand von Field_Selection bestimmt, welches einen Ausdruck enthält, der zum konkreten Typ evaluiert.

2.4.5 Statische Elemente

2.4.5.1 Darstellung statischer Elemente

S.S.

Static_Method, O_Static_Method und T_Static_Method werden in Kapitel 2.4.3 beschrieben. Siehe auch Kapitel 2.3.4.3.

Typhierarchie

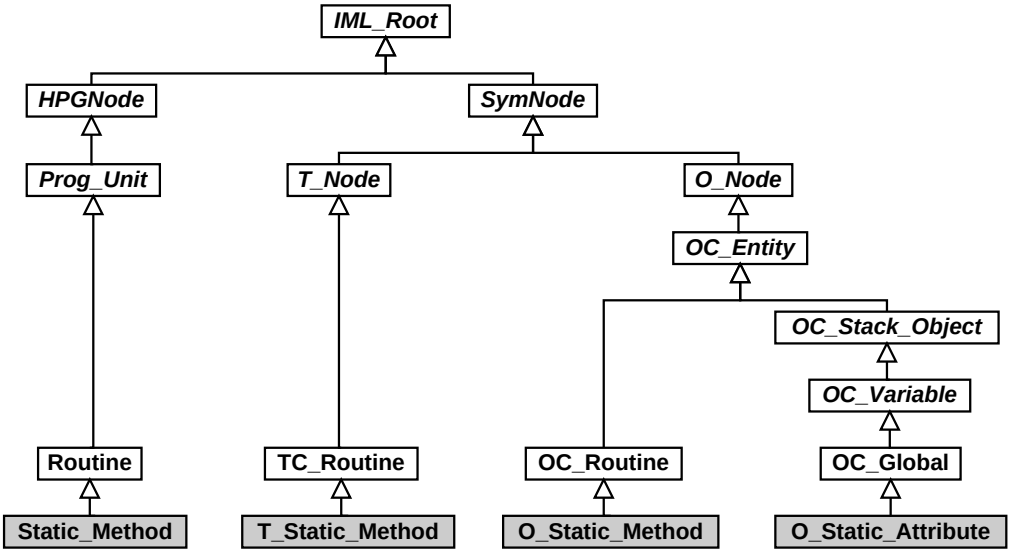


Abbildung 2.20: IML-Typhierarchie für die Modellierung von statischen Elementen

Typbeschreibung

O_Static_Attribute	Deklarations-Knoten für statischen Attribute von Klassen.
Member_Of: class T_Class_Type	Verweis auf die Klasse, zu der das statische Attribut gehört.

2.4.5.2 Selektion statischer Elemente

S.S.

Der Zugriff auf ein statisches Element einer Klasse von außen kann in C++ durch Bereichsauflösung und durch Elementselektion geschehen.

Beispiel:

```
struct C {
    static Element s;
} Objekt_Ausdruck, *Zeiger_Ausdruck;
C::s                 // Bereichsauflösung
Objekt_Ausdruck.s    // Elementselektion
Zeiger_Ausdruck->s   // Elementselektion
```

Der Zugriff über Bereichsauflösung wird in der IML wie jeder andere Zugriff auf eine Variable modelliert, also durch einen Knoten vom Typ Entity_L_Value.

Beim Zugriff über Elementselektion muss der Ausdruck, über den zugegriffen wird, ausgewertet werden. Der Wert des Ausdrucks wird verworfen. Die Auswertung ist nötig, weil der Ausdruck Seiteneffekte haben kann. Deshalb wurde die neue IML-Klasse Static_Field_Selection eingeführt.

Für die Elementselektion mit einem Zeiger wurde der Knotentyp Dereference_Static_Field_Selection eingeführt. Im Gegensatz zur Modellierung des Attributs Operand bei Dereference_Field_Selection wird hier kein künstlich erzeugter Dereference-Knoten eingefügt, da das Ergebnis des Ausdrucks nicht verwendet wird.

Typhierarchie

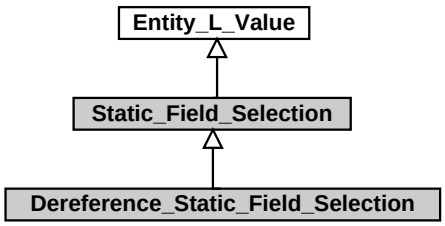


Abbildung 2.21: IML-Typhierarchie für die Modellierung zur Selektion statischer Elemente

Typbeschreibung

Static_Field_Selection	Knoten für die Selektion statischer Elemente über einen Ausdruck, der sich zu einem Objekt auswertet.
Operand: class Value	Verweis auf den auszuwertenden Ausdruck. Entspricht dem Operand-Attribut von Field_Selection, der Wert wird jedoch verworfen.
Dereference_Static_Field_Selection	Knoten für die Selektion statischer Elemente über einen Ausdruck, der sich zu einem Zeiger auf ein Objekt auswertet.

2.4.6 Ausnahmen (Exceptions)

S.S.

Die Modellierung von Ausnahmen in der IML wurde fast unverändert von der IML für Java übernommen. Eine ausführlichere Beschreibung ist deshalb in [19] zu finden. Hier wird nur ein kurzer Überblick über die Vorgänge bei der Ausnahmebehandlung gegeben.

Beim Auslösen einer Ausnahme wird das geworfene Objekt in einen neu angelegten Speicherbereich kopiert. Eine globale Ausnahmevariable zeigt auf

diesen Speicherbereich. Danach erfolgt die Zerstörung aller automatischen Variablen durch die IML-Maschine. Die Zerstörung beginnt bei End_Lifetime des Throw_Statements, das die Ausnahme ausgelöst hat und geht die bisher aufgebauten Blöcke und den Aufrufstack zurück bis zum nächsten `try`-Block (Guarded_Block von Try_Catch_Finally_Statement). Dort wird in dem Handling_Block durch eine Kaskade von Verzweigungen (ähnlich wie bei `if-else-if`...) überprüft, ob die Ausnahme behandelt werden kann oder ob der nächste `try`-Block auf dem Aufrufstack gesucht werden muss. Falls die Ausnahme behandelt wird, wird der Parameter des behandelnden `catch`-Blocks mit dem Ausnahme-Objekt, auf das die globale Ausnahme-Variable zeigt, initialisiert. Anschließend werden die Anweisungen im `catch`-Block ausgeführt und dadurch die Ausnahme behandelt.

Typhierarchie

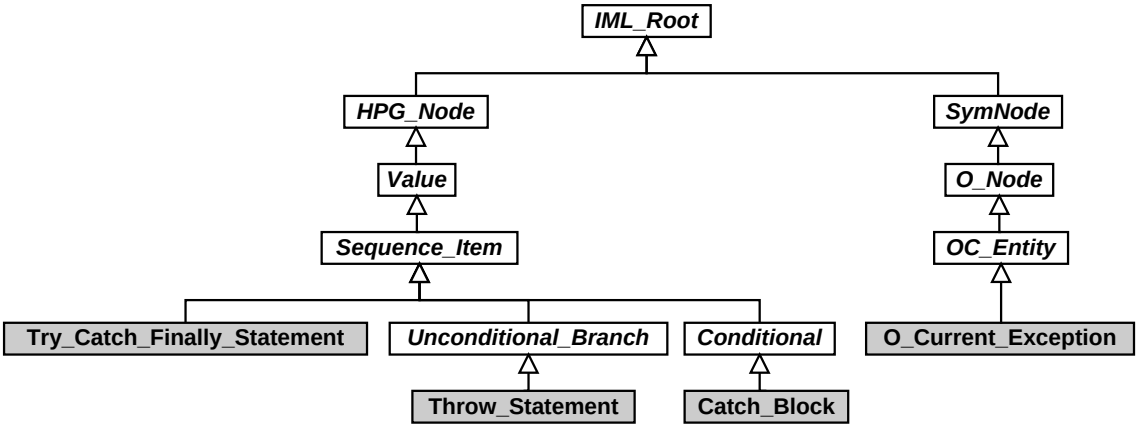


Abbildung 2.22: IML-Typhierarchie für die Modellierung von Ausnahmen

Typbeschreibung

Try_Catch_Finally_Statement	Knoten für die Behandlung von Ausnahmen. Zuerst wird der `try`-Block (Guarded_Block) ausgeführt. Falls darin eine Ausnahme ausgelöst wird, wird der passende Catch_Block gesucht, in dem der Handling_Block ausgeführt wird. Danach wird immer Finalization ausgeführt.
Guarded_Block: class Statement_Sequence	Verweis auf die Anweisungen, deren Ausnahmen behandelt werden sollen.
Handling_Block: class Catch_Block	Verweis auf den Behandlungs-Block, eine Kaskade von Catch_Blocks.
Finalization: class Statement_Sequence	Verweis auf den `finally`-Block. Nur für Java relevant, immer Null in C++.

Catch_Block	Knoten für die Behandlung von ausgelösten Ausnahmen. Die Ausnahmenbehandlung wird mit Hilfe einer if-Kaskade modelliert. Die Bedingung CondV der if-Anweisungen überprüft den Typ der ausgelösten Ausnahme. Falls der Typ passt, wird im ThenP-Zweig der Parameter des `catch`-Blocks initialisiert und die Ausnahmebehandlung durchgeführt. Andernfalls wird im elseP-Zweig der nächste Catch_Block ausgeführt, falls das Ende der Catch_Block-Kaskade noch nicht erreicht wurde.

Throw_Statement	Knoten für das Auslösen von Ausnahmen.
Exception_Type: class Value	Verweis auf den Ausdruck, der die globale Ausnahme-Variable initialisiert und dadurch den Typ der aktuellen Ausnahme bestimmt. Die Initialisierung mit einer Kopie der ausgelösten Ausnahme geschieht durch Aufruf des new-Operators. Falls Exception_Type Null ist, wird die in der globalen Ausnahmevariable enthaltene Ausnahme weitergeworfen. D.h. die Ausnahmebehandlung geht mit der alten Ausnahme weiter, ohne eine neue Kopie des Ausnahmeobjekts zu erzeugen.

O_Current_Exception	Deklarations-Knoten der globalen Ausnahme-Variable.

2.4.7 Zeiger auf Elemente (Pointer to member)

S.S.

In C++ wurden Zeiger auf Elemente von Strukturen oder Klassen eingeführt. Diese Zeiger können auf statische und nicht-statische Attribute und Methoden zeigen.

Beispiel

Folgende Beispiele zeigen die verschiedenen Möglichkeiten, auf Elemente einer Sturktur zuzugreifen. Die einzelnen Beispiele beziehen sich jeweils auf die Definition der Struktur S in den ersten Zeilen.

```
struct S {              // Struktur S
    int i;              // Element i
    void f();           // Methode f
} s, *pS;               // Objekt s, Zeiger pS
int S::* pi = &S::i;    // Zeiger pi auf Element i
                        // des Struktur-Typs S
s.*pi = 42;             // Zugriff auf Element i
                        // des Objekts s
pS->*pi = 42;           // Zugriff auf Element i
                        // des Objekts *pS
void (S::*pf)() =       // Zeiger pf auf Methode f
    &S::f;              // des Struktur-Typs S
(s.*pf)();              // Aufruf der Methode f
(pS->*pf)();            // Aufruf der Methode f
```

In Abbildung 2.23 wird der Teilgraph dargestellt, der den Methodenaufruf „`(pS->*pf)();`" modelliert. Der Aufruf selbst wird durch einen HPG-Knoten vom Typ Pointer_To_Member_Call repräsentiert. Die aufzurufende Methode wird durch einen Knoten vom Typ Pointer_To_Member_Field_Selection ermittelt.

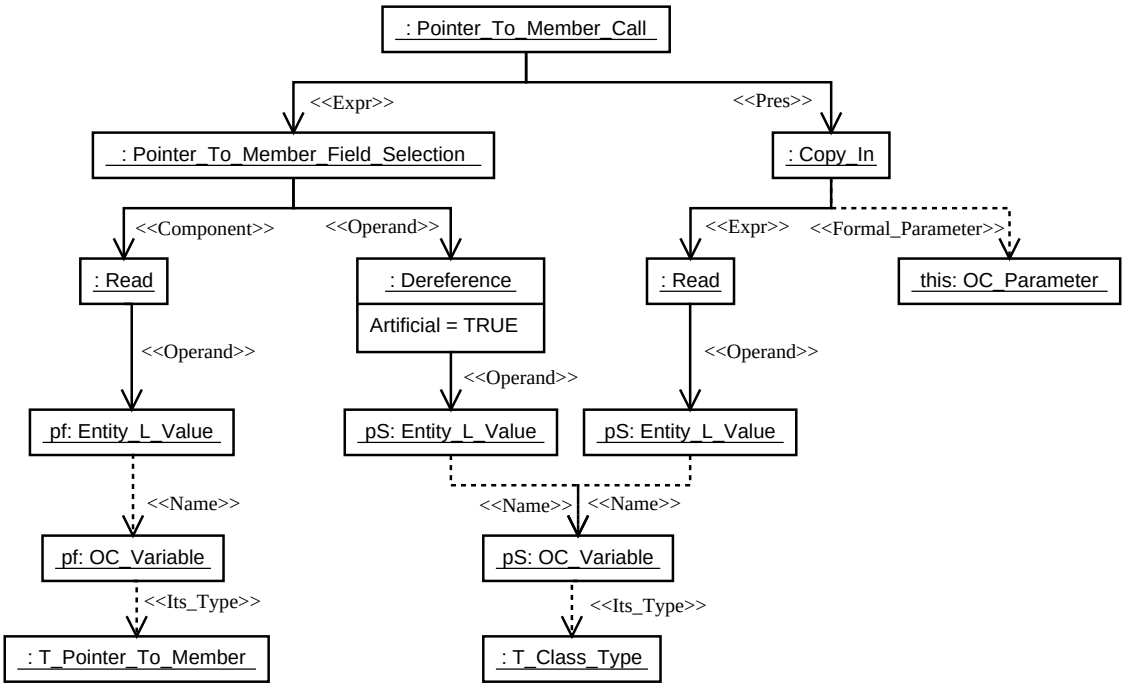


Abbildung 2.23: IML-Beispielgraph eines Methodenaufrufes über einen Elementzeiger

Typhierarchie

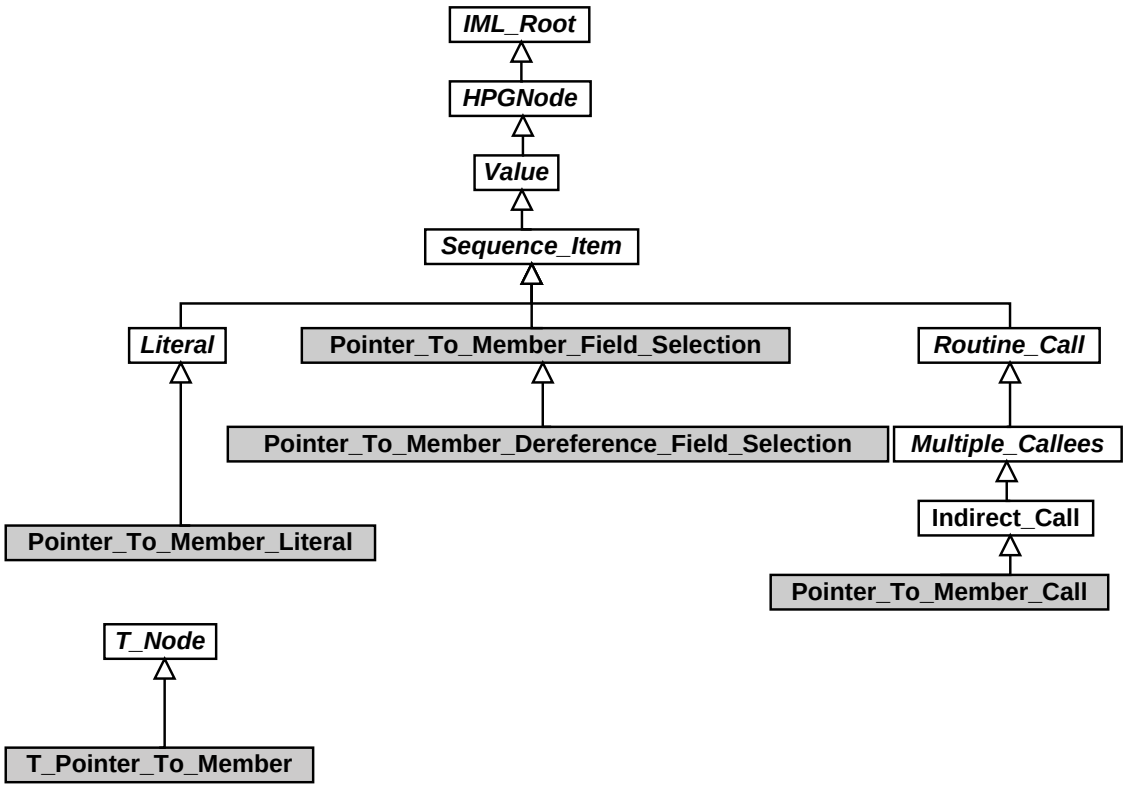


Abbildung 2.24: IML-Typhierarchie für die Modellierung von Zeigern auf Elemente

Typbeschreibung

T_Pointer_To_Member	Typ-Knoten zur Modellierung von Zeigern auf Elemente. Im Beispiel oben war dies „`int S::*`".
Pointed_To_Type: class T_Node *The_Class: class O_Class_Type*	Verweis auf den Typ der Komponente. Verweis auf die Klasse, zu der das Element gehört, auf das gezeigt wird.

Pointer_To_Member_Field_Selection	Knoten zur Modellierung der Benutzung von Zeigern auf Elemente mit einem Objekt. Im Beispiel „`s.*pi`".
Component: class Value	Verweis auf einen Ausdruck vom Typ T_Pointer_To_Member, der die Komponente auswählt. Im Beispiel „`pi`" oder „`pf`".
Operand: class Value	Verweis auf einen L_Value-Ausdruck vom Typ T_Class_Type_Name oder T_Class_Type. Im Beispiel „`s`".

Pointer_To_Member_Dereference_Field_Selection	Knoten zur Modellierung der Benutzung von Zeigern auf Elemente mit einem Zeiger auf ein Objekt. Im Beispiel „`pS->*pi`".
Pointer_To_Member_Call	Knoten zur Modellierung von Methodenaufrufen über Zeiger auf Methoden (Methoden sind auch Elemente). Im Beispiel „`(s.*pf)()`" und „`(pS->*pf)()`".
Pointer_To_Member_Literal	Knoten zur Modellierung des voll qualifizierten Namens eines Elements. Im Beispiel „`&S::i`".
Value: class O_Node	Verweis auf die Deklaration des Elements (z.B. ein OC_Component oder O_Method)

S.S.

2.4.8 `new` und `delete`

Die `new`- und `delete`-Operatoren sind in C++ für das Anlegen und Freigeben von Objekten auf der Halde und für das Initialisieren und Aufräumen dieser Objekte zuständig. Das Anlegen bzw. Freigeben der Objekte ist gleichbedeutend mit der Reservierung bzw. Freigabe des Speicherplatzes, den die Objekte einnehmen. Beide Operatoren gibt es in einer Version für einzelne Objekte und in einer Version für Arrays. Auch die überladbaren Funktionen, die von den Operatoren zum Reservieren bzw. Freigeben des Speichers aufgerufen werden, gibt es in Versionen für einzelne Objekte und für Arrays: `operator new` und `operator new[]` bzw. `operator delete` und `operator delete[]`. Parameter, die über die Platzierungs-Syntax angegeben werden, werden an die überladbaren Funktionen zur Speicherreservierung bzw. -freigabe weitergegeben.

Zur Initialisierung von Arrays aus Objekten einer Klasse mit Konstruktor ist eine statisch unbekannte Anzahl von Konstruktoraufrufen nötig. In der IML wird dafür eine künstliche `for`-Schleife erzeugt. Die Schleife benötigt zwei temporäre Zeiger-Variablen: einen Iterator und einen Zeiger hinter das Ende des Arrays, damit der Ausdruck für die Größe des Arrays nicht zu oft ausgewertet werden muss. Die künstlich erzeugte `for`-Schleife für

```
C array[count];
```

oder

```
C* array = new C[count];
```

sieht im Pseudo-Code wie folgt aus:

```
for(C* it = array, C* end = array + count;
    it != end;
    ++it) {
    it->C(); // Konstruktoraufruf
}
```

Die Variable `count` im Schleifenkopf ist der Parameter für die Größe des Arrays, die im Quelltext angegeben ist. Der Parameter wird genau ein einziges Mal ausgewertet, was wegen der möglichen Seiteneffekte wichtig ist.

Zum Aufräumen eines Arrays mittels mehrerer Destruktoraufrufe wird eine ähnliche Schleife erzeugt. Zu beachten ist die für die Zerstörung vorgeschriebene umgekehrte Reihenfolge der Iteration:

```
for(C* it = array + count - 1, C* end = array - 1;
    it != end;
    --it) {
    it->~C(); // Destruktoraufruf
}
```

Beispiel

```
class C {
    // Attribute, die 20 Bytes Speicher benötigen.
public:
    C();
};
```

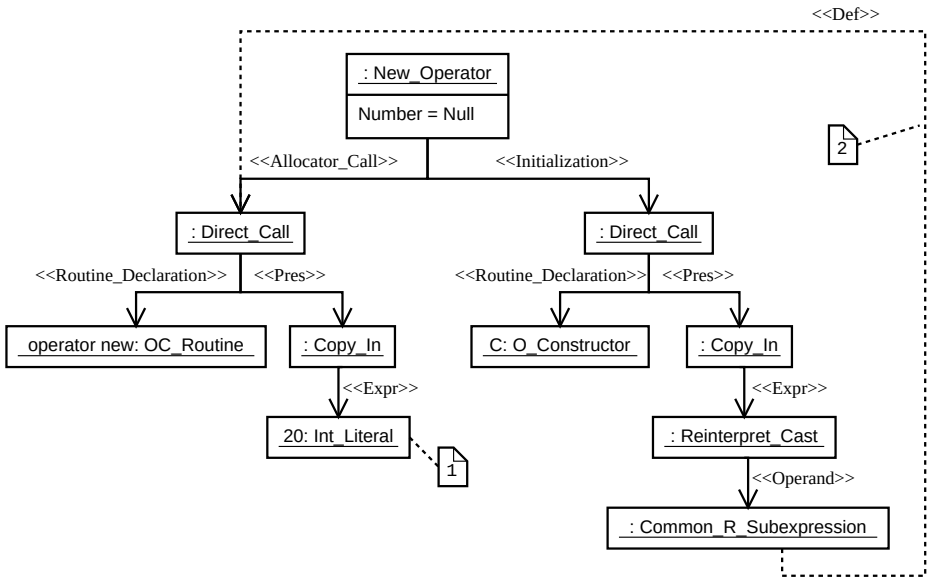


Abbildung 2.25: IML-Beispielgraph zum `new`-Operator

In Abbildung 2.25 ist der IML-Graph angegeben, für den Ausdruck „`new C`" angegeben.

Die Größe des zu reservierenden Speicherplatzes wird mit einem Int_Literal dargestellt und als erster Parameter an die Speicherreservierungsfunktion übergeben (1).

Das Ergebnis des Aufrufs der Speicherreservierungsfunktion ist der Zeiger auf den zu initialisierenden Speicherbereich (2).

Typhierarchie

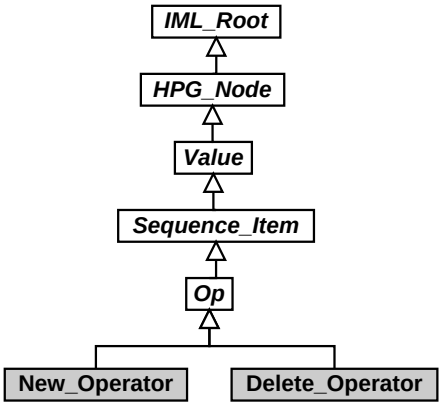


Abbildung 2.26: IML-Typhierarchie für die Modellierung von `new`- und `delete`-Operatoren

Typbeschreibung

Delete_Operator	Knoten für den `delete`-Operator (nicht zu verwechseln mit der überladbaren Funktion zur Speicherfreigabe, „`operator delete`").
Object: class Value	Verweis auf einen Ausdruck, dessen Ergebnis auf das zu zerstörende Objekt zeigt.
Finalization: class Sequence_Item	Verweis auf die Freigabe des Objekts. Ein Destruktoraufruf, wenn nur ein einzelnes Element zerstört werden muss. Wenn, wie bei Arrays, mehrere Objekte zerstört werden müssen, enthält dieses Attribut eine künstlich erzeugte `for`-Schleife, die die einzelnen Objekte zerstört. Null, falls kein Destruktoraufruf nötig ist.
Deallocator_Call: class Direct_Call	Verweis auf einen Aufruf der Funktion zur Speicherfreigabe, „`operator delete`" oder „`operator delete[]`". Der erste Parameter ist ein Zeiger auf den freizugebenden Speicherplatz. Falls die Freigabefunktion einen zweiten Parameter mit Typ `size_t` hat, wird darin die Größe des freizugebenden Bereichs übergeben.

New_Operator	Knoten für den `new`-Operator (nicht zu verwechseln mit der überladbaren Funktion zur Speicherreservierung, „`operator new`“).
Object_Type: class T_Node	Verweis auf den Typ des zu erzeugenden Objekts. Bei Arrays ist es ein Verweis auf den Typ eines einzelnen Objekts.
Allocator_Call: class Direct_Call	Verweis auf einen Aufruf der Funktion zur Speicherreservierung, „`operator new`“ oder „`operator new[]`“. Der erste Parameter ist die Größe des zu reservierenden Speicherplatzes, gefolgt von den beliebig vielen Platzierungsparametern des Operators (Platzierungs-Syntax). Der Rückgabewert ist ein Zeiger auf den reservierten Speicherplatz.
Initialization: class Sequence_Item	Verweis auf die Initialisierung des neu erzeugten Objekts. Wenn, wie bei Arrays, mehrere Objekte initialisiert werden müssen, enthält dieses Attribut eine künstlich erzeugte for-Schleife, die die einzelnen Objekte initialisiert.
Number: class Value	Verweis auf einen Ausdruck, der die Anzahl neu zu erzeugender und zu initialisierender Objekte angibt. Null, falls kein Array-`new` vorliegt.

2.4.9 Typinformation zur Laufzeit (RTTI)

S.S.

In C++ kann zu Instanzen von Klassen mit virtuellen Funktionen zur Laufzeit Typinformation ermittelt werden. In Java ist das bei Instanzen aller Klassen möglich.

Die Typinformation zu einem Objekt erhält man mit dem `typeid`-Operator. Der `typeid`-Operator kann auch auf Typen angewendet werden, zum Beispiel um das Ergebnis mit dem Ergebnis eines `typeid`-Operators auf einem Objekt zu vergleichen. Eine weitere, indirekte Anwendung der Laufzeit-Typinformation ist die Benutzung des `dynamic_cast<>`-Operators (siehe Kapitel 2.4.10).

Beispiel

In diesem Beispiel hat die Basisklasse einen virtuellen Destruktor, damit sie polymorph ist.

```
class Base {
public:
    virtual ~Base();
};
class Sub: public Base {
};
// ...
Base* p;
if (typeid(*p) == typeid(Sub)) {
   // ...
}
```

In Abbildung 2.27 ist der IML-Graph des Ausdrucks

```
(typeid(*p) == typeid(Sub))
```

aus dem obrigen Beispiel dargestellt.

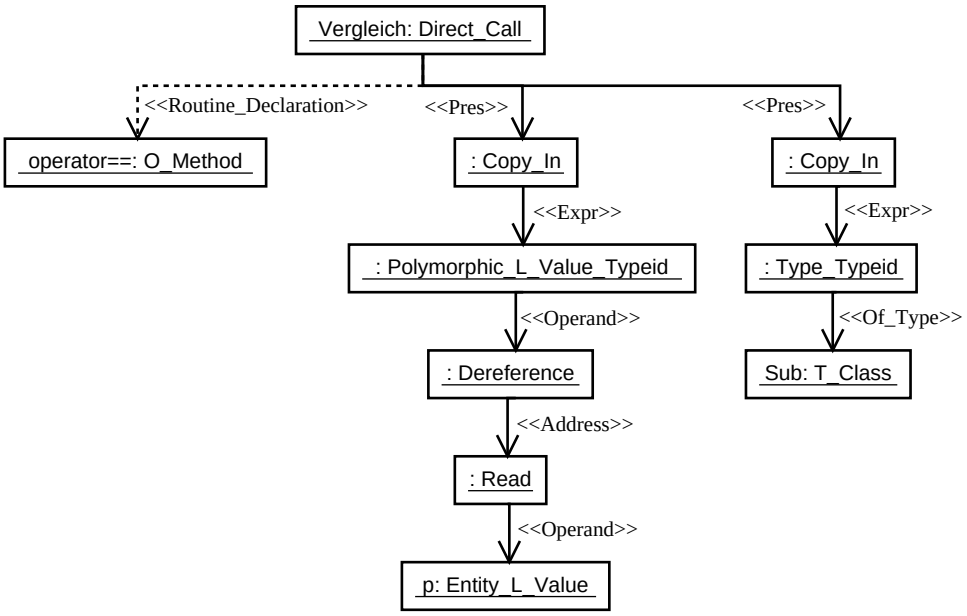


Abbildung 2.27: IML-Beispielgraph zum `typeid`-Operator

Typhierarchie

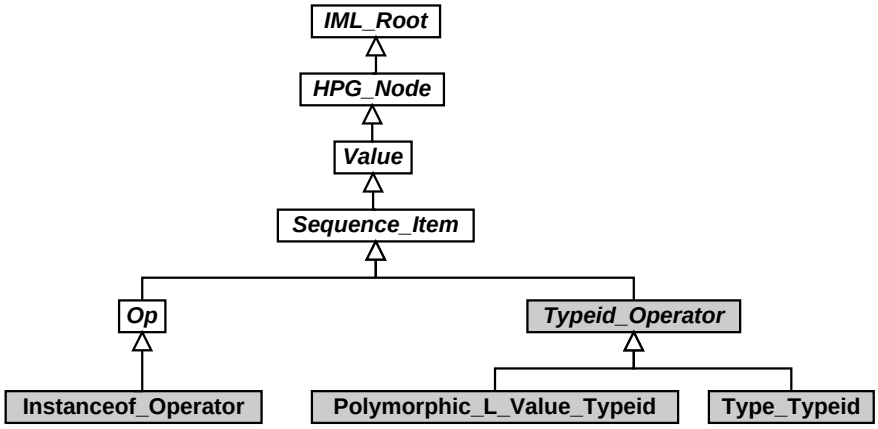


Abbildung 2.28: IML-Typhierarchie für die Modellierung von Laufzeit-Typinformation

Typbeschreibung

Instanceof_Operator	Knoten für den `instanceof`-Operator von Java. Wird in C++ für die Bedingung von `catch`-Blöcken verwendet. Überprüft, ob ein Objekt Instanz eines Typs ist.
Object: class Value	Verweis auf einen Ausdruck, dessen Ergebnis ein Zeiger auf das zu überprüfende Objekt ist.
Class_Type: class T_Node	Verweis auf den Typ, auf den das Objekt geprüft werden soll.

Typeid_Operator	Knoten für den `typeid`-Operator von C++.

Polymorphic_L_Value_Typeid	Knoten für den `typeid`-Operator mit einem polymorphen L-Value (also Dereferenzierung eines Zeigers oder einer Referenz) als Operand.
Operand: class Value	Verweis auf den Operand.

Type_Typeid	Knoten für den `typeid`-Operator, wenn das Ergebnis zum Übersetzungszeitpunkt bekannt ist.
Of_Type: class T_Node	Verweis auf den Typ des Operanden, falls der Operand ein Objekt ist, oder der Operand selbst, falls der Operand ein Typ ist.

2.4.10 Typkonvertierungen

T.K.

In C++ wurden mehrere neue Operatoren zur Typumwandlung eingeführt. Sie dienen zur besseren Unterscheidung der Art einer Typumwandlung. Dadurch sollen Programme besser lesbar sein und somit Fehler vermieden werden. Beim `dynamic_cast<>`-Operator ist außerdem noch eine Überprüfung zur Laufzeit vorhanden; es wird überprüft, ob es sich um eine korrekte Typumwandlung von einer Oberklasse in eine Unterklasse handelt. Die Schreibweise der neuen Typumwandlungsoperatoren erinnert an Templates, und tatsächlich könnten diese Operatoren auch durch Template-Funktionen implementiert sein. Bei der Erweiterung der IML für Java wurde jedoch entschieden, den `Checked_Cast` sowie die neuen C++ Typumwandlungsoperatoren als eingebaute Operatoren darzustellen.

Typhierarchie

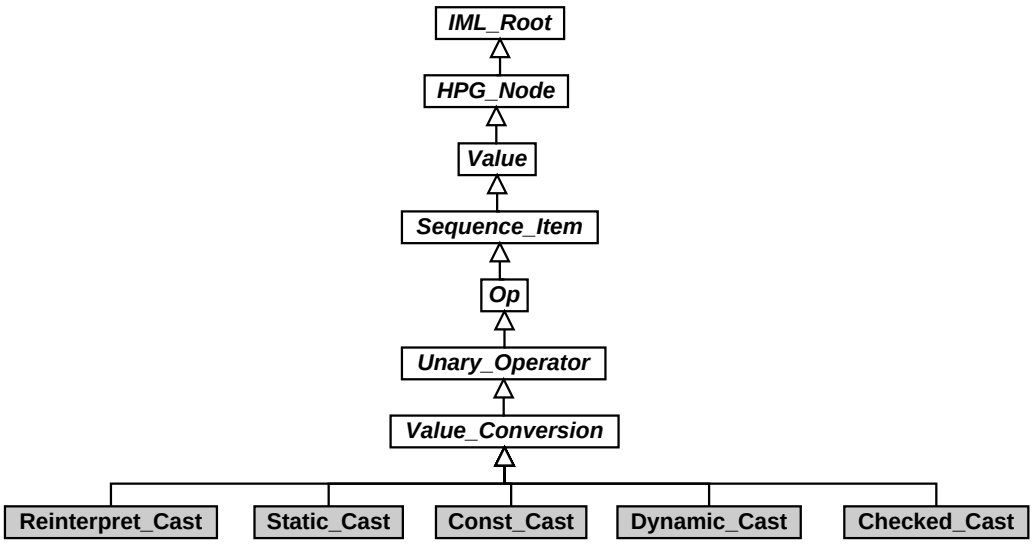


Abbildung 2.29: IML-Typhierarchie für die Modellierung von Typkonvertierungen

Typbeschreibung

Reinterpret_Cast	Knoten für den `reinterpret_cast<>`-Operator von C++, der nur den Typ eines Ausdrucks ändert, aber keine Änderung des Wertes vornimmt.
Static_Cast	Knoten für den `static_cast<>`-Operator von C++, der den Typ eines Objekts per Kopie umwandelt.
Const_Cast	Knoten für den Operator zum Entfernen des `const`-Typqualifizierers.

Dynamic_Cast	Knoten für den `dynamic_cast<>`-Operator von C++, der bei Fehlschlagen der Typumwandlung Null zurückgibt, wenn der Operand ein Zeiger ist, und eine Ausnahme auslöst, falls der Operand eine Referenz ist.

Checked_Cast	Knoten für den Typumwandlungsoperator von Java.

2.4.11 Templates

S.S.

Eine genaue Beschreibung von Templates und Alternativen zur Darstellung von Templates in der IML ist in Kapitel 2.3.10 zu finden.

Die Knotentypen zur Darstellung der Deklaration und Benutzung von Templateparametern sind neu in der IML für C++.

Typhierarchie

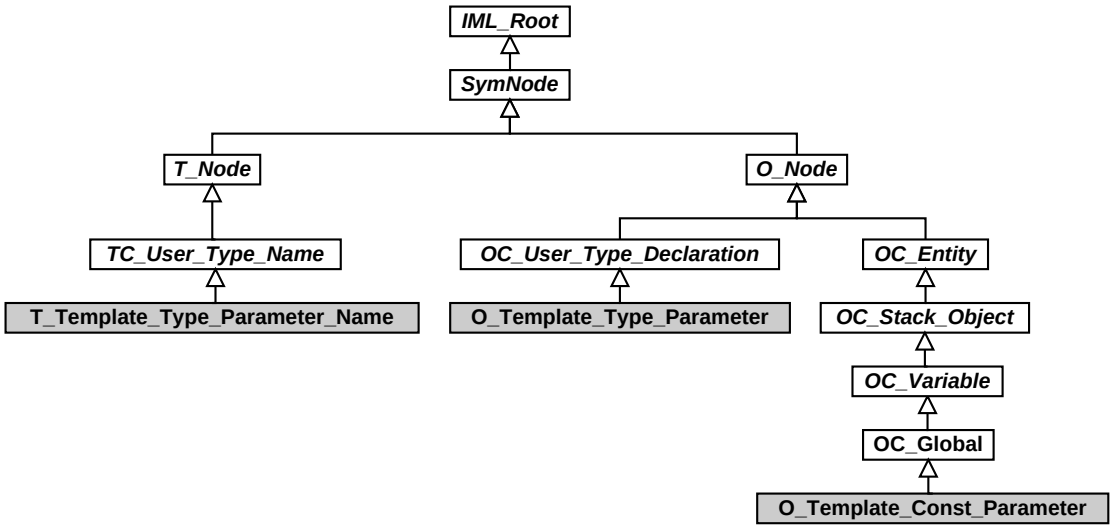


Abbildung 2.30: IML-Typhierarchie für die Modellierung von Template-Parametern

Typbeschreibung

T_Template_Type_Parameter_Name	Knoten für die Benutzung eines Template-Typparameters.

O_Template_Type_Parameter	Deklarations-Knoten für Template-Typparameter.
The_Template: class O_Template_Type_Parameter	Im Normalfall ist dieser Verweis Null. Nur falls der Template-Parameter selbst wieder ein Template ist und dieser Knoten eine Instantiierung dieses Templates repräsentiert, zeigt der Verweis auf den eigenlichen Parameter (siehe „Verschachtelte Template-Deklaration“ auf Seite 50).
Parameter_Of: class O_Node	Verweis auf das O_Class oder OC_Routine-Template, zu dem dieser Parameter gehört.
Default_Type: class T_Node	Verweis auf den Standardwert für den Typ.

O_Template_Const_Parameter	Deklarations-Knoten für Konstanten-template-Parameter.
Parameter_Of: class O_Node	Verweis auf das O_Class oder OC_Routine-Template, zu dem dieser Parameter gehört.
Default_Value: class Value	Verweis auf den Standardwert des Parameters. Null, falls keiner angegeben wurde. Da das Attribut semantisch ist, wird es in der Initialisierungsfunktion eingehängt, sein Wert wird dort jedoch nicht verwendet.

Zur Darstellung von Template-Klassen und -Funktionen wurden die IML-Klassen für Klassen- und Funktionsdeklarationen um einige Attribute erweitert. Die kompletten Beschreibungen der IML-Typen für Klassen sind in Kapitel 2.3.4 und für Funktionen und Methoden in Kapitel 2.4.3 zu finden.

2.4.12 Typqualifizierer

T.K.

Typqualifizierer werden in der IML durch zusätzliche Knoten in der Symboltabelle dargestellt. Jeweils ein Typqualifizierer ändert einen bestehenden Typ in der vorgegebenen Weise. Der Knotentyp T_Const_Qualifier bewirkt, dass die Instanzen des Typs als konstant angesehen werden, d.h. ihren Wert nach der Initialisierung nicht mehr ändern. T_Volatile_Qualifier dagegen besagt, dass Instanzen eines Typs ihren Wert unabhängig vom Compiler im Speicher ändern können. Für eine Übersetzung von Quelltexten in nativen Code bedeutet eine `volatile`-Qualifikation normalerweise eine Einschränkung der Optimierungsmöglichkeiten des Übersetzers. In IML-Graphen, wo keine Optimierungen durchgeführt werden, wird dieses Attribut ausschließlich wegen der Quelltextnähe modelliert.

Alternativ zu der Modellierung durch zusätzliche Knoten in der IML-Symboltabelle wäre auch die Modellierung durch die zwei zusätzlichen Attribute

Is_Constant und Is_Volatile denkbar gewesen. Die gewählte Vorgehensweise ist jedoch vorteilhafter für nachfolgende Analysen, die auf dem IML-Graphen arbeiten, da dort Typqualifizierer teilweise nicht von Interesse sind. Bei der Modellierung durch zusätzliche Knoten können solche Analysen z.B. einfacher Typen vergleichen, da jeder Typ in der Symboltabelle durch genau einen Knoten dargestellt wird, unabhängig von den Typqualifizierern. Darüberhinaus wurde die Modellierung der IML2, dem geplanten Nachfolger dieser IML1, nachempfunden, wodurch die spätere Weiterentwicklung der Bauhaus-Software vereinfacht wird.

Nach den einleitenden Beispielen wird in Abbildung 2.32 auf der nächsten Seite die Typhierarchie für die Modellierung von Typqualifizierern veranschaulicht, gefolgt von einer detaillierten Beschreibung der neuen Knotentypen.

Beispiele

Folgender Quelltext soll die Modellierung von Typqualifizierern anhand von drei kleinen Beispielen veranschaulichen. In Abbildung 2.31 sind passend dazu drei Typen einer IML-Symboltabelle modelliert; auf eine Modellierung der Variablen wurde an dieser Stelle verzichtet.

```
const char* pc           = "Beispiel 1";
char *const cp           = "Beispiel 2";
volatile const char* vcp = "Beispiel 3";
```

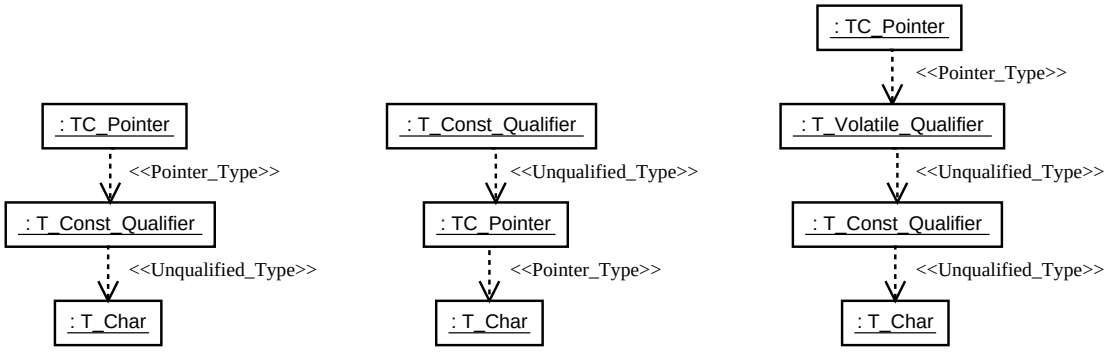


Abbildung 2.31: IML-Beispielgraphen zu Typqualifizierern

Beispiel 1 ist eine Deklaration eines Zeigers auf einen konstanten `char`-Typ. Beispiel 2 dagegen deklariert einen konstanten Zeiger auf eine `char`-Variable, erkennbar an der Reihenfolge im IML-Graphen. In Beispiel 3 werden zwei Typqualifizierer auf einen Typ angewendet, womit ein Zeiger auf einen `volatile` und `const char`-Typ entsteht.

Typhierarchie

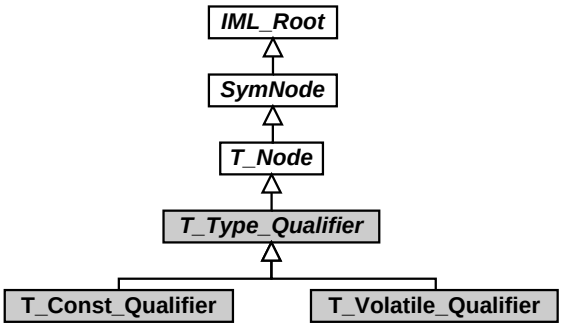


Abbildung 2.32: IML-Typhierarchie für die Modellierung von Typqualifizierern

Typbeschreibung

T_Type_Qualifier	Abstrakter Knotentyp für die Darstellung von Typqualifizierern, die einen modellierten Typ weiter spezifizieren.
Unqualified_Type: class T_Node	Verweis auf den Typknoten, der näher beschrieben werden soll. Ein Verweis auf einen weiteren Typqualifizierer ist auch möglich.

T_Const_Qualifier	Modelliert den Typqualifizierer `const`, der vorschreibt, dass die Werte aller Instanzen dieses Typs nach der Initialisierung nicht mehr geändert werden dürfen.

T_Volatile_Qualifier	Modelliert den Typqualifizierer `volatile`, der anzeigt, dass sich die Werte der Instanzen dieses Typs auch ohne den Einfluss des Compilers im Speicher ändern können.

2.4.13 Sonstiges

S.S.

In diesem Kapitel werden IML-Erweiterungen beschrieben, die so einfach sind, dass sie kein eigenes Kapitel benötigen.

Typhierarchie

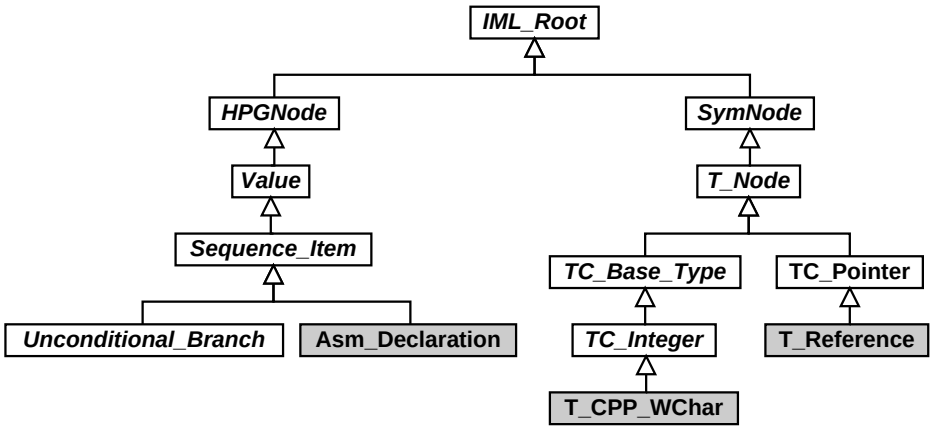


Abbildung 2.33: IML-Typhierarchie für die Modellierung der sonstigen neuen Typen

Typbeschreibung

Unconditional_Branch	Knoten für die Darstellung von unbedingten Verzweigungen.
End_Lifetime: class Statement_Sequence	Verweis auf die Anweisungen zur Zerstörung von Objekten, deren Gültigkeitbereich durch den Sprung verlassen wird. Dieses Attribut war bisher vom Typ „set of class OC_Stack_Object". Der Typ wurde geändert, damit auch Destruktoraufrufe modelliert werden können.
Asm_Declaration	Knoten für Assembler-Deklarationen der Form `asm("...");` Das angegebene Zeichenkettenliteral wird systemabhängig interpretiert.
Value: class String_Literal	Verweis auf das Zeichenkettenliteral. Es enthält üblicherweise Assembler-Quelltext.
TCPP_WChar	Typ-Knoten für den wide-character Typ. Wide-character ist ein neuer fundamentaler Typ von C++ zur Darstellung von Zeichen eines größeren Zeichensatzes, wie z.B. Unicode. Die Größe des Typs ist implementierungsabhängig.

T_Reference	Typ-Knoten für Referenzen. Referenzen verhalten sich wie konstante Zeiger, können aber nie Null sein. Die Syntax zur Benutzung und zur Initialisierung unterscheidet sich von der für Zeiger.

```
template<class T, T i, int j>
void f() {
    T t1 = i;
  // ...
}
// ...
f<char, 2, 5>();
```

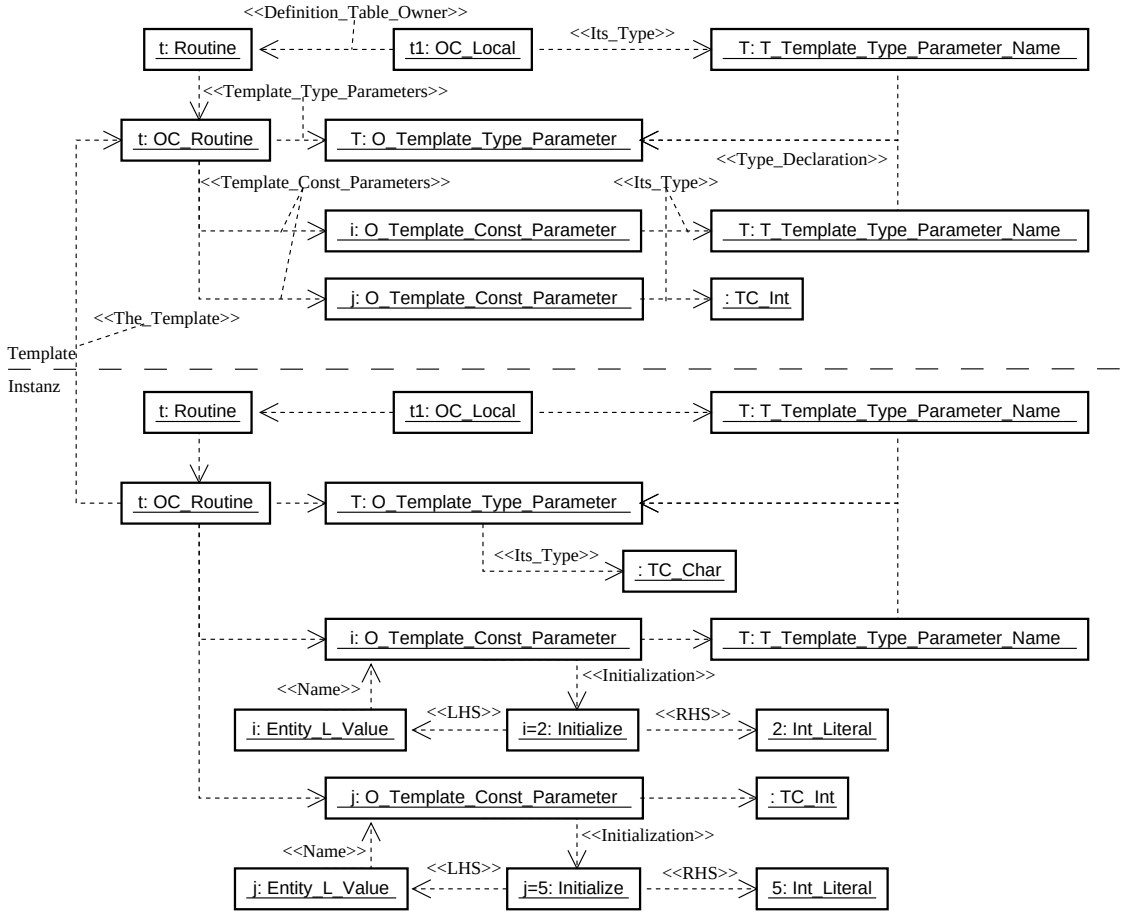


Abbildung 2.13: Beispiel mit IML-Graph: Template Funktion mit Typ- und Konstantentemplates

```
template<class S, template<class T> class C>
class Ex { // Example - Class
  C<S> values;
};
template<class R> class myarray { /* ... */ };
Ex<int, myarray> ex;
```

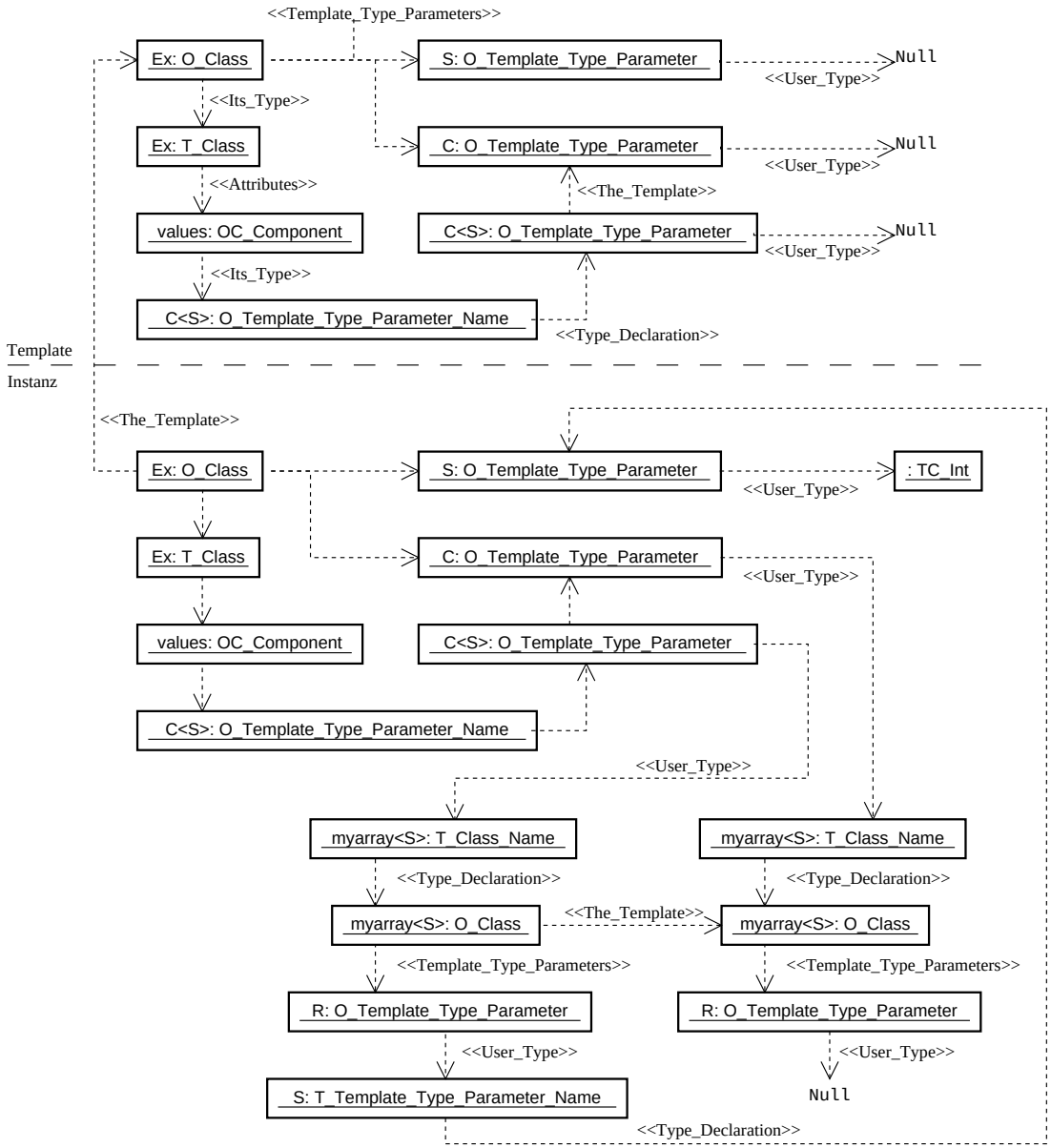


Abbildung 2.14: Beispiel mit IML-Graph: Verschachtelte Template-Deklaration

Kapitel 3

Das C++ Front-End

T.K.

Eine direkte Übersetzung von C++ Quelltext in die IML wäre eine sehr schwere und aufwändige Aufgabe. Aus diesem Grund wird die Übersetzung in mindestens zwei Teilschritte aufgeteilt. Im ersten Schritt wird der C++ Quelltext eingelesen und einer Analyse unterzogen. Ergebnis dieser Analyse ist eine Zwischendarstellung, oft in Form eines ASTs und einer Symboltabelle. Auf diesem Zwischenergebnis aufbauend kann dann, in einem zweiten Schritt der Übersetzung, die IML erzeugt werden.

Dieses Kapitel widmet sich dem ersten Übersetzungsschritt, dem so genannten C++ Front-End. Das C++ Front-End liest C++ Quelltexte ein und führt syntaktische und semantische Analysen durch. Im Falle einer korrekten Übersetzungseinheit wird dann eine Zwischendarstellung des Quelltextes erzeugt. Da allein die Realisierung dieses ersten Teilschritts den Rahmen einer Diplomarbeit bei weitem überschreiten würde, enthält bereits die Aufgabenstellung die Vorgabe, auf ein bestehendes Front-End aufzubauen. Dazu musste allerdings zuerst eines der zur Verfügung stehenden Front-Ends ausgewählt werden, was im nächsten Unterkapitel beschrieben ist. Danach wird auf das ausgewählte C++ Front-End näher eingegangen.

3.1 Auswahl eines C++ Front-Ends

T.K.

Die Auswahl des C++ Front-Ends, das bei dieser Diplomarbeit zum Einsatz kommen soll, wurde durch eine Evaluierung festgelegt. Für die Evaluierung werden zunächst die Anforderungen an das Front-End genannt und die zur Verfügung stehenden Produkte aufgeführt. Danach wird die eigentliche Evaluierung durchgeführt, um zuletzt zu einem Ergebnis in Form einer positiven Entscheidung für eines der vorgestellten Front-Ends zu gelangen.

3.1.1 Anforderungen

T.K.

Für die Realisierung des Übersetzers stehen in dieser Diplomarbeit mehrere C++ Front-Ends zur Auswahl. Um nun zu entscheiden, welches Front-End am besten für diese Aufgabe geeignet ist, werden an dieser Stelle die Anforderungen an die Front-Ends gestellt. Diese Anforderungen sollen die Auswahl an Front-Ends einschränken und den Auswahlprozess lenken. Einige der hier

genannten Anforderungen entstammen direkt der Aufgabenstellung der Diplomarbeit, andere wurden von diesen abgeleitet. Wie schon vorher erwähnt wurde, ist eine Anforderung die Verwendung eines bestehenden C++ Analysewerkzeuges.

Die folgende Liste fasst alle relevanten funktionalen und nicht-funktionalen Anforderungen zusammen und beschreibt diese näher. Darüberhinaus werden auch untergeordnete Anforderungen genannt, die nicht zwingend umgesetzt werden müssen. Diese sind als „zweitrangig" gekennzeichnet.

1. *Ausreichende Information*

   Die wichtigste Anforderung an das Front-End ist, dass es ausreichend Informationen bereitstellt, um daraus eine IML-Repräsentation aufzubauen. Welche Informationen als ausreichend gelten, ist im Detail der IML-Spezifikation zu entnehmen. Als grobe Richtlinie kann jedoch gesagt werden, dass es möglich sein sollte, ein semantisch äquivalentes Programm aus der Zwischendarstellung des Front-Ends wiederzugewinnen.

2. *Plattformen*

   Das Front-End sollte zumindest Solaris und Linux als Ausführungsplattform unterstützen. Weiterhin ist eine Unterstützung der HP-UX und MS Windows Plattformen eine zweitrangige Anforderung.

3. *Einfache Anwendung*

   Die Anwendung des IML-Übersetzers sollte für den Benutzer einfach und mit wenigen Handgriffen möglich sein.

4. *Einfache Entwicklung*

   Dieser Punkt bezieht sich hauptsächlich auf den Entwicklungsaufwand des IML-Übersetzers, da die Diplomarbeit zeitlich auf maximal sechs Monate beschränkt ist. Zwei Faktoren bestimmen dabei die Entwicklungszeit: die Struktur der Zwischendarstellung des Front-Ends und der Übersetzungsvorgang des IML-Übersetzers selbst. Der erste Faktor könnte das Traversieren eines ASTs eines C++ Übersetzers oder das Übersetzen einer weiteren Zwischensprache, z.B. Datrix [4], in die IML bedeuten. Wichtig ist in allen Fällen die Verfügbarkeit von ausreichender Dokumentation über die vorhandene Zwischendarstellung. Der zweite Faktor beinhaltet z.B. die Programmiersprache, in der das Front-End implementiert ist, den Buildprozess oder die Kopplung an die Funktionen zur Generierung der IML, welche in Ada 95 implementiert sind.

5. *Minimale Wartung*

   Nach Abschluss der Diplomarbeit ist es sehr wahrscheinlich, dass eine Wartung der Implementierung notwendig wird. Die zwei wahrscheinlichsten Wartungsgründe sind die Anpassung an neue Versionen der IML-Spezifikation und die Anpassung an neue Versionen des verwendeten Front-Ends. Der letzte Punkt impliziert, dass der Umfang der Änderungen, die am Front-End vorgenommen werden müssen, so gering wie möglich gehalten werden sollte.

6. *Unterstützung für C++ Dialekte*

   Die Konformität zum C++ Sprachstandard [14] ist die wichtigste sprachenbezogene Anforderung. Eine Unterstützung für populäre C++ Dialekte ist eine weitere, zweitrangige Anforderung. Die wichtigsten Dialekte sind hierbei die GNU C++ Erweiterungen, eingesetzt im C++ Übersetzer der GNU Compiler Collection, die Microsoft Erweiterungen, eingesetzt im Microsoft VisualC++ Übersetzer, der IBM VisualAge C++ Dialekt und die Borland Erweiterungen, welche im Borland C++ Builder zum Einsatz kommen.

7. *Lizenzierung*

   Der Einsatz des bestehenden Front-Ends darf nicht mit der Bauhaus-Lizenz kollidieren. Eine Implikation dieser Bedingung ist, dass der Bauhaus-Quelltext in seiner bisherigen Lizenz verbleibt. Ein Wechsel zu GPL, der GNU General Public License, ist beispielsweise nicht erwünscht. Einzige Ausnahme dieser Regel ist der Quelltext des IML-Generators zum Laden, Speichern und Bearbeiten von IML-Graphen. Für diesen wäre eine freie Lizenzierung denkbar.

8. *Effizienz*

   Die Ausführungsgeschwindigkeit des IML-Übersetzers ist wichtig, wenn große Programme übersetzt werden sollen. Die Zeit für die gesamte Übersetzung wird natürlich auch stark vom Front-End bestimmt. Als Richtlinie für den IML-Übersetzer sollte gelten, dass er nicht viel langsamer sein sollte als cafe, der bestehende C nach IML-Übersetzer des Bauhaus-Projekts.

3.1.2 Produkte

T.K.

Dieses Kapitel stellt die verschiedenen Front-Ends vor, welche in dieser Evaluierung betrachtet wurden. Jedes Unterkapitel befasst sich mit einem einzelnen Front-End und beginnt mit einer Übersicht über die Details. Danach folgt eine kurze Vorstellung des Produkts und eine Beschreibung der Einbindungsmöglichkeit in Bauhaus. Die Produkte sind in alphabetischer Reihenfolge aufgeführt. Bei jedem Produkt wurde die letzte verfügbare, stabile Version ausgewählt.

3.1.2.1 CodeStore

T.K.

Produktname:	CodeStore
Produktversion:	Unbekannt
Hersteller:	IBM Research Division
Internetadresse:	`http://www.research.ibm.com/softwaretechnology/`
Lizenz:	Unbekannt

Die offizielle Internetseite von CodeStore besagt:

> CodeStore is a class library that supports the writing of C++-knowledgable tools. Such tools include incremental compilers, incremental linkers, development environments, browsers, and such. CodeStore provides a representation for C++ programs, as well as C++-specific processing, like parsing and semantic analysis. CodeStore is dynamically extensible in a variety of ways. Tool-writers can use this extension mechanism to fundamentally change the way in which programs are compiled. We are interested in programming tools and development environments.

Unglücklicherweise wird das Projekt nicht mehr unterstützt und wird, nach Aussage von Michael Karasick von IBM, von der Internetseite entfernt werden. Darum ist die CodeStore-Bibliothek nicht mehr Bestand weiterer Untersuchungen und wird in der weiteren Evaluierung nicht mehr aufgeführt, obwohl das Produkt geeignet gewesen wäre.

T.K.

3.1.2.2 Columbus

Produktname:	Columbus/CAN
Produktversion:	3.1, 07.06.2002
Hersteller:	FrontEndART Ltd.
Internetadresse:	`http://www.frontendart.com/`
Lizenz:	Columbus/CAN Academic License

Columbus/CAN ist ein Softwaresystem, welches entwickelt wurde, um Informationen aus C++ Quelltexten zu extrahieren. Das Produkt besteht aus einem allgemeinen Reengineering Front-End für die Analyse, interne Zwischendarstellung, Filterung und Export von Informationen über Quelltextdateien.

Der Analyseprozess beginnt mit dem Entfernen aller Präprozessoranweisungen durch einen C++ Präprozessor, der nicht Bestandteil von Columbus ist. Zu diesem Zweck kann jeder existierende Präprozessor eingesetzt werden, z.B. auch der GNU C++ Präprozessor oder der MS VisualC++ Präprozessor. Der pure C++ Quelltext wird dann vom CAN Kommandozeilenwerkzeug eingelesen, welches davon eine interne Darstellung, „Columbus Schema for C++" [8] genannt, erzeugt. Eine vollständige Repräsentation eines Programmes wird durch das Zusammenlinken der einzelnen Columbus-Dateien erzeugt. Die extrahierten Daten können dann interaktiv gefiltert und nach verschiedenen Formaten exportiert werden. Columbus unterstützt in der angegebenen Version CPPML (XML), GXL (XML), HTML und RSF (rigi).

Eine Generierung von IML könnte auf zwei Wegen erfolgen. Eine Möglichkeit wäre, einen Übersetzer zu entwickeln, der eines der von Columbus exportierten Formate in IML übersetzt. Eine andere Möglichkeit wäre, ein Export Plug-in für Columbus zu entwickeln, welches durch das Columbus API auf die interne Darstellung zugreift und direkt IML erzeugt. Die zweite Möglichkeit wäre hier zu bevorzugen, da sie einige Vorteile gegenüber der ersten bietet:

- Der gesamte Übersetzungsprozess vom Quelltext bis zur IML-Repräsentation ist schneller, da das Laden und Speichern der Zwischendarstellung und ein zweiter Übersetzungsvorgang entfällt.

- Es stehen mehr Informationen für die IML-Generierung zur Verfügung, da die Informationen in den exportierten Formaten eine Untermenge der Informationen ist, die durch das Columbus API verfügbar sind.

Aus diesen Gründen wird in der weiteren Diskussion nur noch die zweite Vorgehensweise, die Realisierung als Export Plug-in, berücksichtigt.

3.1.2.3 CPPX

T.K.

Produktname:	CPPX
Produktversion:	1.1, 02.10.2001
Hersteller:	Tom Dean, Andrew Malton und Ric Holt
Internetadresse:	http://swag.uwaterloo.ca/~cppx/
Lizenz:	GNU General Public License Version 2

Das CPPX Projekt beabsichtigt, ein Open Source Reengineering-Werkzeug aufzubauen, welches C++ Quelltext in die Datenformate GXL, TA und VCG übersetzt. Eine genaue Beschreibung der Produktdetails, der Projektziele und des Entwicklungsprozesses ist in [4] zu finden.

Der gesamte Übersetzungsvorgang ist bei CPPX hauptsächlich in zwei Teilschritte aufgeteilt. Im ersten Schritt wird eine geänderte Version des GNU C++ Übersetzers verwendet, um den AST des Übersetzers in ein proprietäres Binärformat zu schreiben. Danach wird in einem zweiten Schritt dieses Binärformat in das gewünschte Ausgabeformat konvertiert. Diese Konvertierung geschieht mittels einer Sammlung kleiner Programme, wobei jedes für einen Teil des AST zuständig ist.

Um mit CPPX einen IML-Übersetzer zu realisieren, würde eines der Ausgabeformate von CPPX, z.B. GXL, in IML übersetzt werden müssen. Ein eigenständiges Werkzeug, das allein von der Ausgabe von CPPX abhängt, würde für diese Aufgabe genügen.

3.1.2.4 GCC-XML

T.K.

Produktname:	GCC-XML
Produktversion:	0.2, 03.04.2002
Hersteller:	Brad King von Kitware, Inc.
Internetadresse:	http://www.gccxml.org
Lizenz:	GNU General Public License Version 2

GCC-XML ist ebenfalls ein Open Source Reengineering-Werkzeug, welches eine XML-Darstellung von C++ Quelltexten erzeugt. Unglücklicherweise ist das generierte XML-Format weder standardisiert noch dokumentiert.

Ähnlich wie CPPX verwendet GCC-XML den GNU C++ Compiler zur syntaktischen und semantischen Analyse der Quelltexte. Bei GCC-XML wird dies realisiert durch den zusätzlichen Kommandozeilenschalter des GNU C++ Compilers „-fxml = <generierte XML-Datei>". Diese weitere Übersetzeroption traversiert den AST des Übersetzers und generiert daraus direkt eine XML-Darstellung, so dass kein internes Darstellungsformat nötig ist.

Eine Verwendung von GCC-XML zur Generierung einer IML-Repräsentation würde die Entwicklung eines eigenständigen Übersetzers bedeuten, der

das Ausgabeformat von GCC-XML in IML übersetzt. Ähnlich zu CPPX würde auch hier das zu entwickelnde Werkzeug ausschließlich von der Ausgabe von GCC-XML abhängen.

3.1.2.5 GEN++

T.K.

Produktname:	Genoa - GEN++
Produktversion:	1.4
Hersteller:	Prem Devanbu und Laura Eaves bei Lucent Technologies/Bell Laboratories
Internetadresse:	http://www.cs.ucdavis.edu/~devanbu/genp/
Lizenz:	"Non-exclusive Binary Code License" von Bell Labs und Cfront Lizenz

GEN++ ist eine Entwicklungsumgebung, gedacht um Analysewerkzeuge für C++ zu erzeugen. Die mit GEN++ implementierten Analysewerkzeuge werden in einer „high-level domain-specific language" (DSL) spezifiziert, die speziell für die einfache Implementierung von Analysewerkzeugen entworfen wurde.

GEN++ basiert auf dem Cfront 3.0.3 Übersetzer für die Analyse der C++ Quelltexte. Eine Abfrageschnittstelle stellt die so gewonnenen Informationen zur Verfügung. Diese Eigenheit von GEN++ macht es für die Anwender von GEN++ nötig, auch eine Lizenz des Cfront-Übersetzers zu erwerben.

Da GEN++ eine Entwicklungsumgebung für Analysewerkzeuge ist, würde ein C++ nach IML-Übersetzer als ein in der GEN++ Sprache implementiertes Werkzeug realisiert werden. Dieses würde auf die von GEN++ bereitgestellten Informationen zugreifen und daraus eine IML-Darstellung erzeugen.

3.1.2.6 GNU C++ Compiler

T.K.

Produktname:	GNU Compiler Collection (GCC)
Produktversion:	3.2, 14.08.2002
Hersteller:	Free Software Foundation (FSF)
Internetadresse:	http://gcc.gnu.org/
Lizenz:	GNU General Public License Version 2

Die GNU Compiler Collection ist eine Sammlung mehrerer Open Source Crosscompiler. Es existieren momentan Front-Ends für die Programmiersprachen C, C++, Objective C, Fortran, Java und Ada 95, zusammen mit Laufzeitbibliotheken für diese.

Die GCC übersetzt eine Übersetzungseinheit in mehreren Schritten. Die ersten Schritte beinhalten das Einlesen der Quelldatei, das Parsen, eine Optimierung des ASTs und die Generierung der „register transfer langugage" (RTL) [25]. Grob gesagt liest der Parseprozess die Quelldatei ein, parst deren Inhalt, führt alle nötigen semantischen Analysen durch und konstruiert eine abstrakte Darstellung der Übersetzungseinheit. Die nachfolgenden Verarbeitungsschritte führen Optimierungen auf dieser Interndarstellung durch, bevor die gesamte Interndarstellung in die RTL Struktur konvertiert wird. Die meisten der nachfolgenden Schritte arbeiten auf dieser RTL.

Die zentrale Idee, den GCC für die Generierung einer IML-Repräsentation zu nutzen, besteht darin, den C++ Übersetzer im GCC so zu erweitern, dass direkt IML erzeugt werden kann. Dazu müsste eine Übersetzungsfunktion implementiert werden, welche direkt nach dem Parsen die interne abstrakte Darstellung in IML umformt. Diese Vorgehensweise für die Einbindung ist sehr ähnlich zu der, welche beim GCC-XML Projekt angewendet wird.

3.1.2.7 EDG C++ Front-End

S.S.

Produktname:	EDG C++ Front-End
Produktversion:	3.0
Hersteller:	Edison Design Group (EDG)
Internetadresse:	http://www.edg.com/cpp.html
Lizenz:	„Scientific Evaluation Licence" oder eine proprietäre kommerzielle Lizenz

Das C++ Front-End von EDG ist ein Front-End, welches konzipiert wurde, um darauf aufbauend C++ Übersetzer zu entwickeln. Es wird bisher für Nativcode-Übersetzer, ebenso wie für Analysewerkzeuge eingesetzt.

Das EDG-Front-End übernimmt, nach dem Vorbild des klassischen Übersetzerbaus, die vollständige syntaktische und semantische Analyse des Quelltextes und übergibt dem Back-End einen fertigen AST. Dieser AST enthält bei EDG alle Informationen des ursprünglichen Quelltextes, Zeilen- und Spaltennummern und die ursprünglichen Bezeichner mit eingeschlossen. Da das gesamte Front-End in ANSI C implementiert wurde, ist auch der AST des Front-Ends ein stark verzeigerter Graph mit Strukturen als Knoten. Der Zugriff auf diese Strukturen geschieht direkt oder unter Zuhilfenahme von vorgegebenen Funktionen und Macros.

Back-Ends, die auf den AST des Front-Ends zugreifen wollen, haben dazu verschiedene Möglichkeiten. Zum einen kann der AST (komplett oder stückweise in Funktionen) im Speicher an das Back-End übergeben werden. Zum anderen kann das Front-End den AST in eine Datei schreiben, von der das Back-End den AST wieder einlesen kann.

Um mit Hilfe des C++ Front-Ends von EDG einen Übersetzer für IML zu realisieren, müsste ein geeignetes Back-End entwickelt werden. Das Back-End würde die im AST enthaltenen Informationen direkt in IML übersetzen.

3.1.2.8 Introspector

T.K.

Produktname:	GCC XML Introspector Project
Produktversion:	0.4 Beta, 06.04.2002
Hersteller:	James Michael DuPont
Internetadresse:	http://introspector.sourceforge.net/
Lizenz:	GNU General Public License Version 2

Introspector ist ein Werkzeug, um Metadaten aus einem Programm zu extrahieren. Die Metadaten beinhalten bei Introspector alle Daten des Übersetzers, des Buildprozesses, des CVS-Änderungsprotokolls und der Mailinglisten. Alle verfügbaren Metadaten werden in einer Datenbank abgelegt und sind von dort aus für alle Werkzeuge zugreifbar, die sie benötigen.

In der aktuellen Implementierung von Introspector ist der GNU C++ Übersetzer die einzige Informationsquelle. Auf dessen Daten kann über eine Schnittstelle im XML-Format aus zugegriffen werden. Das Format der XML-Ausgabe ist in der aktuellen Version noch in der Entwicklung und wird höchstwahrscheinlich noch viele Änderungen erfahren. In der aktuellen Version orientiert sich die IML-Ausgabe noch sehr stark am AST des GCC.

Ein Zugriff auf die Datenbank von Introspector würde in diesem Fall die Grundlage einer geeigneten Realisierung eines IML-Übersetzers darstellen. Beim aktuellen Stand des Projektes würde das die Entwicklung eines alleinstehenden Übersetzers bedeuten, der ausgehend von XML-Dateien aus der Datenbank von Introspector eine IML-Repräsentation des Programmes erstellt. Die Vorgehensweise ist damit ähnlich zu GCC-XML und CPPX.

3.1.2.9 OpenC++

T.K.

Produktname:	OpenC++
Produktversion:	2.5.12, Oktober 2001
Hersteller:	Shigeru Chiba von XEROX Corporation
Internetadresse:	`http://www.csg.is.titech.ac.jp/~chiba/openc++.html`
Lizenz:	Proprietäre Lizenz

OpenC++ ist eine Werkzeugsammlung von C++ Übersetzern und Analysewerkzeugen. Es existieren bereits einige Werkzeuge, die auf der Basis von OpenC++ aufbauen. Beispielsweise implementiert ein Projekt eine Spracherweiterung zu ISO C++. Ein anderes Projekt erzeugt Vererbungsgraphen ausgehend vom C++ Quelltext.

Die Verarbeitung von (spracherweiterten) C++ Übersetzungseinheiten ist bei OpenC++ in drei Teilschritte aufteilbar. Zuerst wird das so genannte „base level" Programm, welches möglicherweise C++ Spracherweiterungen enthält, von einem gewöhnlichen C++ Präprozessor verarbeitet. Danach übersetzt der OpenC++ Übersetzer das Resultat in reinen C++ Quelltext. Im letzten Schritt wird der erzeugte C++ Quelltext von einem gewöhnlichen C++ Übersetzer in nativen Code übersetzt.

Um einen Übersetzer oder eine Analyse für C++ Quelltext zu realisieren, muss ein Plug-in für OpenC++ implementiert werden. Dazu ist eine festgelegte Schnittstelle, „Metaobject Protocol" (MOP) genannt, verfügbar, die Zugriff auf die interne Darstellung gewährt. Die durch das MOP verfügbaren Daten entsprechen praktisch der Ausgabe des OpenC++ Parsers. Die interne Darstellung ist eine verknüpfte Liste von Tokens, welche stark der Syntax von Lisp gleicht. Basierend auf dieser internen Darstellung müsste ein Übersetzer nach IML realisiert werden.

3.1.3 Evaluierung

T.K.

Nachdem alle zur Verfügung stehenden Produkte vorgestellt wurden, werden diese nun nach den Kriterien bewertet, die in Kapitel 3.1.1 genannt wurden. Jedes Unterkapitel beschäftigt sich mit einem dieser Kriterien und beschreibt, wie gut die Produkte das Kriterium erfüllen.

3.1.3.1 Ausreichende Information

T.K.

Die IML-Darstellung von C und Java Programmen ist bereits durch die aktuelle IML-Spezifikation festgelegt. Da die IML-Spezifikation für C++ abwärtskompatibel sein sollte, müssen die Produkte mindestens diesen Umfang an Informationen bereitstellen. Darüberhinaus sollten ebenfalls Informationen über spezielle C++ Sprachelemente wie Templates verfügbar sein. Eine genaue Beschreibung über die Darstellung von C++ Programmen in der IML ist in Kapitel 2.4 zu finden.

Eine Gemeinsamkeit aller vorgestellten Produkte ist, dass sie nur reinen C++ Quelltext analysieren. Das bedeutet, dass auf Präprozessoranweisungen nicht mehr zugegriffen werden kann, da sie bereits durch einen implizit oder explizit vorgeschalteten Präprozessor entfernt wurden. Die einzige Ausnahme hiervon ist das EDG-Front-End, welches auch Informationen über den Ursprung von durch Macros expandiertem Quelltext bereitstellt.

Die Columbus-Analyse stellt Quelltextinformationen sehr detailliert zur Verfügung, vor allem, da sie für diese spezielle Aufgabe realisiert wurde. Unglücklicherweise hat die Analyse einige Mängel, welche in [28, Kapitel 9 „Deficiencies“] aufgeführt sind. Um das Verständnis für das Resultat der Evaluierung zu vereinfachen, sollen die wichtigsten Mängel an dieser Stelle nochmals genannt werden:

- Funktionsrümpfe werden überhaupt nicht analysiert. Diese Einschränkung impliziert, dass keine Informationen über Ausdrücke, Anweisungen und Ausnahmebehandlungen vorhanden sind.
- Es sind keine Informationen über überladene Funktionen vorhanden.
- Die Behandlung von Templates ist weder zuverlässig, noch vollständig. Das Verwenden von Templates kann zu unerwarteten Ergebnissen führen.

Einige dieser Punkte werden in der nächsten freigegebenen Version der Software behoben werden. Diese erschien jedoch erst Mitte Februar 2003, also in der Endphase dieser Diplomarbeit. Daher gelten die aufgeführten Mängel als KO-Kriterium für das Produkt Columbus.

Da GEN++ auf dem Cfront 3.0.3 Übersetzer aufbaut, erbt es auch dessen Nachteile. Wichtig für diese Anforderung ist dabei, dass nur eine sehr frühe Version des C++ Standards unterstützt wird. Zusätzlich reduziert GEN++ selbst die vorhandene Menge an Informationen: „The genii specification in gen++ is not meant to be a complete, consistent, well-defined representation of C++.“ [5, Seite 27]. Beispielsweise sind nur die Namen der Definitionen und der Parameter von Templates verfügbar. Wie bei Columbus stellen sich auch die Einschränkungen von GEN++ als KO-Kriterium heraus.

OpenC++ verfügt über keine semantischen Details, da nur eine syntaktische Analyse der Quelltexte vorgenommen wird. Informationen, wie die Genauigkeit von eingebauten C++ Datentypen, sind damit nicht vorhanden. Eine Implementierung einer semantischen Analyse würde jedoch den Rahmen

dieser Diplomarbeit bei weitem überschreiten, weswegen auch diese Anforderung ein KO-Kriterium für OpenC++ darstellt.

Das C++ Front-End von EDG ist als Front-End für Übersetzer und Analysewerkzeuge konzipiert und führt bereits eine vollständige semantische Analyse durch. Dadurch sind alle benötigten Informationen für eine IML-Repräsentation vorhanden. Zusätzlich sind Quelltextverweise vorhanden, die ebenfalls für die IML-Generierung benötigt werden.

Alle anderen Front-Ends basieren auf dem GNU C++ Übersetzer, so dass sie alle unter den gleichen Mängeln leiden:

- Einige Informationen gehen im AST des Übersetzers verloren. Dadurch wird

  ```
  enum Colour { WHITE = 3, GREEN, BLACK,
      BLANC = 3, VERT, NOIR };
  Colour x = WHITE;
  ```

  in derselben Form dargestellt werden wie

  ```
  enum Colour { WHITE = 3, GREEN, BLACK = 5,
      BLANC = 3, VERT = 4, NOIR };
  Colour x = BLANC;
  ```

  oder sogar

  ```
  enum Colour { WHITE = 3, GREEN = 4 , BLACK,
      BLANC = WHITE, VERT = GREEN, NOIR = BLACK };
  Colour x = (Colour)3;
  ```

- Einige Typkonvertierungen gehen im AST verloren. Es gibt in C++ zwei Arten von Typkonvertierungen: elementare und konstruktive. Elementare Typkonvertierungen, die auf Literale angewendet werden, gehen verloren, wie oben bereits beschrieben wurde. Konstruktive Typkonvertierungen werden durch Konstruktoraufrufe ersetzt, so dass mit der Definition:

  ```
  class Blat {
      float y;
      Blat(int x) { y = x; }
  };
  ```

  die folgenden Zeilen nicht unterscheidbar sind:

  ```
  int foo(int y) { Blat x = (Blat)y; }
  int foo(int y) { Blat x = Blat(y); }
  int foo(int y) { Blat x = y;       }
  ```

- Zeilen- und Spalteninformationen von Literalen sind nicht vorhanden.

Außer diesen genannten Einschränkungen baut der GNU C++ Compiler garantiert eine vollständige Repräsentation des Quelltextes auf, da die nachfolgenden Phasen des Übersetzers nur auf den AST aufbauen, um Nativcode zu erzeugen. Die meisten Optimierungen werden auch ausgespart, wenn auf den AST zugegriffen wird, da die Optimierungen erst später im Verarbeitungsprozess durchgeführt werden.

Den GNU C++ Compiler direkt einzusetzen verspricht, ausreichend Informationen zur Verfügung zu haben, um eine IML-Repräsentation zu erzeugen. Die Projekte, die auf den GCC aufbauen, werden nur noch kurz durch ihre zusätzlichen Mängel charakterisiert. Der Ausgabe von CPPX fehlen Genauigkeitsangaben für die eingebauten Datentypen. Außerdem bereitet die Behandlung von Templates Probleme. GCC-XML stellt dagegen keine Informationen über Funktionsrümpfe zur Verfügung, was ebenfalls ein KO-Kriterium darstellt. Introspector stellt im Prinzip den kompletten AST des GCC zur Verfügung, jedoch konnte mit der aktuellen Implementierung nicht einmal eine einfache Klassendefinition übersetzt werden.

3.1.3.2 Plattformen

T.K.

Wie vorher schon erwähnt wurde, sollte der IML-Übersetzer auf Solaris und Linux lauffähig sein. Eine weitere Unterstützung von HP-UX und MS Windows ist eine zweitrangige Anforderung. Die folgende Liste beschreibt die von den Front-Ends unterstützten Plattformen:

Columbus Nur MS Windows wird unterstützt, was es auch in diesem Punkt unbrauchbar macht.

GEN++ Solaris, SunOS 4.x und HP-UX werden unterstützt. Da Linux als Plattform nicht unterstützt wird und wahrscheinlich auch in Zukunft nicht unterstützt werden wird, werden hier die Anforderungen nicht erfüllt.

OpenC++ MS Windows und POSIX-kompatible Plattformen werden unterstützt, was alle geforderten Plattformen mit einbezieht.

GNU C++ Es wird eine große Menge von Plattformen unterstützt, inklusive Solaris, Linux, HP-UX und MS Windows. Die vollständige Liste der unterstützten Plattformen ist unter `http://gcc.gnu.org/install/specific.html` nachzulesen.

CPPX Die aktuelle Version unterstützt nur Linux. Die ältere Version 1.0 unterstützt zusätzlich auch Solaris als Plattform. Prinzipiell sollte aber CPPX alle Plattformen unterstützen, die auch der GCC unterstützt, da CPPX als Patch für den GCC implementiert ist.

EDG C++ Front-End Es werden alle Plattformen mit einem ANSI C Übersetzer unterstützt.

GCC-XML Dieselben Plattformen wie der GNU C++ Compiler.

Introspector Alle geforderten Plattformen, HP-UX und MS Windows mit eingeschlossen.

3.1.3.3 Einfache Anwendung

T.K.

Hier wird die Anwendung des IML-Übersetzers aus Benutzersicht dargestellt. Die beschriebenen Szenarien beziehen sich auf ein mögliches fertiges Produkt und schließen so die Generierung der IML-Repräsentation mit ein.

Columbus Wenn Projektdateien von MS VisualC++ oder Borland C++ Builder vorliegen, können diese direkt verwendet werden, was die Anwendung in solchen Fällen sehr komfortabel macht. Anderenfalls muss der Benutzer die Übersetzungseinheiten von Hand mittels einer GUI auswählen oder es wird ein Kommandozeilenwerkzeug eingesetzt.

GEN++ Für jede Übersetzungseinheit muss die GEN++ Applikation einmal aufgerufen werden. Der Benutzer könnte diesen Schritt automatisieren, indem ein Shellscript eine von Hand erzeugte Liste von Übersetzungseinheiten automatisch abarbeitet. Alternativ könnte in einem leicht abgeänderten Makefile GEN++ anstatt des üblichen Übersetzers aufgerufen werden.

OpenC++ Hier gelten dieselben Bedingungen wie bei GEN++, jedoch muss zusätzlich der IML-Übersetzer aufgerufen werden.

GNU C++ Eine Anwendung würde einfach das Hinzufügen eines weiteren Übersetzerschalters bedeuten. Zudem könnte die Generierung von Nativcode durch einen bereits vorhandenen Übersetzerschalter unterdrückt werden. Das macht die Anwendung sehr einfach, da ein bestehender Makeprozess mit minimalen Änderungen für die Generierung von IML herangezogen werden kann. Ein weiterer Vorteil besteht darin, dass der GNU C++ Compiler bereits in einer großen Anzahl von Projekten eingesetzt wird.

CPPX, EDG C++ Front-End, GCC-XML und Introspector Hier gelten dieselben Bedingungen wie beim GNU C++ Compiler, jedoch muss ein separater IML-Übersetzer aufgerufen werden, was automatisiert werden könnte.

3.1.3.4 Einfache Entwicklung

T.K.

Die Details des Entwicklungsprozesses sollen in diesem Kapitel erläutert werden. Mit inbegriffen sind die Schwierigkeiten der Implementierung, die Programmiersprache und die Integration der Funktionen zur Generierung der IML. Die IML-Funktionen sind in Ada 95 implementiert, können zusätzlich aber auch über eine C Schnittstelle aufgerufen werden.

Columbus Unglücklicherweise war das Columbus API zum Zeitpunkt der Evaluierung nicht verfügbar, so dass hier keine präzisen Aussagen gemacht werden können. Dadurch, dass aber die Schnittstelle von Columbus speziell für Export Plug-ins gestaltet wurde, kann von einer einfachen Implementierung ausgegangen werden. Die von Columbus verwendete Programmiersprache ist MS VisualC++, wodurch über die C

Schnittstelle auf die IML-Funktionen zugegriffen werden kann. Es ist weiterhin nicht bekannt, ob eine Dokumentation über das Columbus API erhältlich ist.

GEN++ Um GEN++ für die Implementierung des IML-Übersetzer zu verwenden, wäre es nötig, die spezielle Programmiersprache von GEN++ zu erlernen. Die Schnittstelle von GEN++ ist vollständig dokumentiert. Der Zugriff auf die IML-Funktionen ist nicht möglich, was ein KO-Kriterium für GEN++ darstellt.

OpenC++ Das Konvertierungs-Plug-in müsste in derselben Programmiersprache implementiert werden, in der auch OpenC++ implementiert ist, also C oder C++. Die Schnittstelle zu OpenC++ ist dokumentiert. Da jedoch eine semantische Analyse implementiert werden müsste, würde die Entwicklung sehr viel Zeit in Anspruch nehmen.

GNU C++ Der größte Aufwand müsste bei einer Entwicklung mit dem GNU C++ Compiler sicherlich in die Informationsextraktion aus dem AST gesteckt werden. Dokumentation über den AST ist momentan im Entstehen und noch nicht vollständig vorhanden. Das Verständnis für den Aufbau des ASTs könnte dadurch aufwändig werden. Die Implementierung müsste in C oder C++ durchgeführt werden, da der AST in C implementiert ist.
Wie die Entwicklung eines Prototyps zeigte, ist hier bei der Entwicklung relativ viel Aufwand für den Buildprozess nötig. Es muss zum einen der neue Quelltext in die bestehende Software eingebaut werden, was nicht ohne einen minimalen Satz an Änderungen der GCC-Quelltexte und Makefiles zu machen ist. Als schwierig hat sich auch das Einbinden von Ada 95-Quelltext erwiesen, da dies mit dem Ada Übersetzer, -Linker und -Binder geschehen muss. Der GCC-Buildprozess sieht eine solche Erweiterung aber nicht vor.

CPPX Für das Einlesen der XML-Dateien könnten bestehende Bibliotheken verwendet werden. Der eigenständige IML-Übersetzer könnte also in einer beliebigen Programmiersprache implementiert werden.

EDG C++ Front-End Hier gilt das, was bereits für den GNU C++ Compiler gesagt wurde, nur dass bereits eine gute und vollständige Dokumentation über den Aufbau des ASTs vorhanden ist. Eine Übersicht ist vorhanden in Form eines separaten Dokuments und eine detaillierte Beschreibung der Knoten und Kanten existiert in einer zentralen Quelltextdatei. Die Implementierung müsste entweder in Ada, C oder C++ geschehen, da der AST in C implementiert ist.

GCC-XML und Introspector Hier gilt das, was bereits für CPPX gesagt wurde, außer dass das Ausgabeformat nicht dokumentiert ist und sich wahrscheinlich noch ändert.

Eine Aufwandsabschätzung ist leider nur für die Implementierung auf Basis des GNU C++ Compilers möglich. Da die Autoren von CPPX vor einem sehr

ähnlichen Problem standen [4], kann angenommen werden, dass die Entwicklungszeiten ähnlich liegen werden. Die Autoren von CPPX haben acht Mannmonate für die Implementierung benötigt, was den Rahmen dieser Diplomarbeit leicht übersteigt. Dadurch, dass die Diplomarbeit von zwei Autoren durchgeführt wird, stehen zwar insgesamt 12 Mannmonate zur Verfügung, jedoch kann nicht die gesamte Zeit für die Implementierung verwendet werden.

Eine Abschätzung für die anderen Projekte ist mangels Bewertungsgrundlage nicht präzise möglich.

3.1.3.5 Minimale Wartung

T.K.

Die spezifischen Eigenschaften der Wartung durch die Verwendung der Front-Ends sollen in diesem Kapitel genannt werden.

Columbus Der hauptsächliche Wartungsaufwand würde durch eine Änderung des Columbus API entstehen. Eine Änderung des API ist glücklicherweise unwahrscheinlich, weil bereits mehrere Werkzeuge von dieser Schnittstelle abhängen.

GEN++ Die Wartung sollte keinen nennenswerten Aufwand erzeugen, da die Entwicklung von GEN++ 1998 gestoppt wurde. Das bedeutet aber auch, dass beim GEN++ Projekt selbst keine Wartung mehr betrieben wird. Eine eigenständige Wartung ist auch nicht möglich, da von GEN++ keine Quelltexte vertrieben werden.

OpenC++ Ähnlich wie bei Columbus wird auch hauptsächlich hier Wartungsaufwand bei Änderungen der OpenC++ API notwendig. Das ist aber durch den stabilen Zustand des Projektes ebenfalls unwahrscheinlich. Aufgrund der Tatsache, dass die Schnittstelle auch nur syntaktische Informationen bereit stellt, sind Änderungen zusätzlich unwahrscheinlich, da die Syntax von C++ standardisiert ist.

GNU C++ Änderungen des ASTs sind zu erwarten. Wahrscheinlich haben diese Änderungen aber nur einen kleineren Umfang. Außerdem muss der Patch des GCC für jede Version dessen neu erstellt werden. Es wird aber nicht nötig sein, jede Version des GNU C++ Compilers zu unterstützen.

CPPX Da die Realisierung von cafe++ in Form eines eigenständigen Werkzeuges geschieht, würde zusätzlicher Wartungsaufwand nur durch Änderungen des Ausgabeformats des Front-Ends entstehen.

EDG C++ Front-End Da das Front-End schon in vielen Anwendungen eingesetzt wird, wird EDG Änderungen am AST so gering wie möglich halten.

GCC-XML und Introspector Analog zu CPPX, aber Änderungen am Ausgabeformat sind hier wahrscheinlich.

3.1.3.6 Unterstützung für C++ Dialekte

T.K.

Da bei der Evaluierung nicht genügend Zeit für einen ausführlichen Test vorhanden war, sind Aussagen über die Standardkonformität nicht vollständig verfügbar.

Columbus zielt darauf ab, den C++ Standard [14] möglichst genau umzusetzen. GEN++ hält sich überhaupt nicht an den Standard, da es auf dem Cfront-Übersetzer basiert, welcher in einer Zeit entwickelt wurde, als noch kein Standard existierte. Bei OpenC++ waren keine Informationen über die Standardkonformität vorhanden. Der GNU C++ Compiler entspricht dem Standard bereits sehr gut [22]. Zusätzlich ist in der Entwicklung des GNU C++ Compilers eine weitere Verbesserung der Sprachkonformität sichtbar. Die drei Front-Ends CPPX, GCC-XML und Introspector, die auf dem GCC basieren, erben dessen Eigenschaften. EDG sagt aus, den Sprachstandard vollständig umzusetzen.

Weitere C++ Sprachdialekte werden nur von einigen Front-Ends unterstützt. An dieser Stelle sollte angemerkt werden, dass die verschiedenen Erweiterungen in keinem Fall vollständig dokumentiert sind. Aus diesem Grund behauptet auch kein Front-End Hersteller, den jeweiligen Dialekt im gesamten Umfang umzusetzen.

Columbus Die Microsoft Erweiterungen des MS VisualC++ 6.0 Übersetzers und die Borland Erweiterungen des C++ Builders 5 werden unterstützt.

GEN++ Die Erweiterungen des Cfront 3.0.3 Übersetzers werden unterstützt.

OpenC++ Für dieses Produkt sind keine Informationen verfügbar.

GNU C++, CPPX, GCC-XML und Introspector Es werden die GNU C++ Erweiterungen und einige der Microsoft Erweiterungen, zum Beispiel spezielle `pragma`-Anweisungen, unterstützt.
Eine Besonderheit des C Übersetzers im GCC ist dessen AST. Dieser ist, bis auf verschachtelte Funktionen, eine echte Untermenge des C++ ASTs [25]. Bedauerlicherweise baut der C Übersetzer keinen vollständigen AST währen der Übersetzung auf, wie es der C++ Übersetzer macht. Abgesehen davon könnte ein auf dem GCC basierender IML-Übersetzer auch so erweitert werden, dass er auch C Übersetzungseinheiten in IML umwandeln kann.

EDG C++ Front-End Das Front-End von EDG bietet Kompatibilitätsmodi für Cfront, Microsoft VisualC++ und den Sun Übersetzer an. Es stehen auch Kommandozeilenschalter zur Verfügung, die die Standardkonformität abschwächen, um so mehr Quelltexte verarbeiten zu können.
Ähnlich wie der AST beim GNU C++ Compiler hat auch die interne Darstellung von EDG die Möglichkeit, C Quelltexte darzustellen. Ohne die C99-Erweiterungen ist die interne Darstellung von C ebenfalls eine echte Untermenge der C++ Darstellung. So könnte auch hier das C++ Front-End für die Übersetzung von C Übersetzungseinheiten in die IML verwendet werden.

3.1.3.7 Lizenzierung

T.K.

In diesem Kapitel werden die Einflüsse bezüglich der Lizenzierung durch die Front-Ends beschrieben. Das Hauptaugenmerk wird auf den Einfluss von bestehender Bauhaus-Software und auf einen möglichen kommerziellen Einsatz von Bauhaus gelegt.

Columbus Ein Zugriff auf das Columbus API und die Entwicklung eines Export Plug-ins ist nicht durch die „Columbus/CAN Academic License" möglich und benötigt eine andere Lizenzierung. Nach Tibor Gyimonthy, einem Columbus-Entwickler, sollte eine spezielle Lizenzierung für das Bauhaus-Projekt möglich sein.

GEN++ GEN++ wird unter einer sehr restriktiven Lizenz freigegeben. Das Produkt darf nur für nicht-kommerzielle Zwecke eingesetzt werden und nur eine Instanz der Software darf zu einem Zeitpunkt auf einem Rechner ausgeführt werden. Dabei wird GEN++ nur in einer binären Version angeboten und darf nicht weiter verbreitetet werden. Zusätzlich ist es nötig, eine Lizenz des Cfront 3.0.3 Übersetzers zu erstehen. Die Firma HP, die momentan die Rechte an diesem Übersetzer besitzt, hat aber am 1. August 1999 aufgehört, Lizenzen für Cfront zu vertreiben.

OpenC++ OpenC++ wird unter einer sehr freizügigen Lizenz vertrieben. Es wird erlaubt, die Software selbst und darauf aufbauende Software beliebig zu verwenden, zu kopieren und weiterzugeben, ohne dass Lizenzgebühren anfallen. Das einzige Zugeständnis ist die Erwähnung folgender Copyright-Erklärung in der Dokumentation:

> Copyright (c) 1995, 1996 Xerox Corporation. All Rights Reserved.
>
> Use and copying of this software and preparation of derivative works based upon this software are permitted. Any copy of this software or of any derivative work must include the above copyright notice of Xerox Corporation, this paragraph and the one after it. Any distribution of this software or derivative works must comply with all applicable United States export control laws.

Die Auswirkungen auf die Bauhaus-Software sollten ohne Nachteile annehmbar sein.

GNU C++ Die komplette GNU Compiler Collection ist unter der „GNU General Public License version 2" (GPL) freigegeben. Diese Lizenz erlaubt es, die Software selbst und darauf basierende Software beliebig zu verwenden und weiterzugeben. Gleichzeitig wird das Copyright des ursprünglichen Autors geschützt. Der Hauptnachteil hier ist, dass jede auf GPL-Software basierende Software ebenfalls unter der GPL freigegeben werden muss. Das impliziert, dass die Quelltexte auf dieselbe Weise angeboten werden müssen wie die Binärform der Software.
Wenn diese Regeln auf die Bauhaus-Software angewendet werden, bedeutet dies, dass der IML-Übersetzer selbst unter der GPL veröffentlicht

werden muss. Da der IML-Übersetzer auf die IML-Funktionen zugreifen müsste, müssen diese auch unter der GPL freigegeben werden. An dieser Stelle könnte ein Problem entstehen, denn auch große Teile der bestehenden Bauhaus-Software greifen auf die IML-Funktionen zu. Ohne weitere Maßnahmen müssten also große Teile des Bauhaus-Projektes unter der GPL freigegeben werden. Der einzige Weg, diese unerwünschten Auswirkungen zu vermeiden, wäre, die IML-Funktionen unter zwei Lizenzen (der bisherigen und der GPL) freizugeben. Das ist möglich, da die Universität Stuttgart als Copyright-Inhaber die Rechte dazu hat. Dieses Verfahren der Merhfachlizenzierung wird auch bei anderen Projekten, wie z.B. OpenOffice und Mozilla, angewandt.
Nicht genau definiert in der GPL ist weiterhin die Rechtslage bezüglich von GPL-Software abgeleiteter Software. Es ist nicht ganz klar, wann eine Software als abgeleitete Software gilt. Konkret ist in diesem Fall unklar, ob nicht die gesamte Bauhaus-Software unter der GPL freigegeben werden muss, wenn sie auch nur auf den Ergebnissen des GCC aufbaut. Dieser Punkt kann ohne genaue Untersuchung der Rechtslage leider nicht vollständig geklärt werden.

CPPX, GCC-XML und Introspector Alle Front-Ends, die auf dem GNU C++ Compiler aufbauen, müssen ebenfalls unter die GPL gestellt werden. Da der IML-Übersetzer in diesen Fällen in einem eigenständigen Prozess ausgeführt wird und nur von der XML-Ausgabe der Front-Ends abhängt, ist die Lage hier, gegenüber einer direkten Verwendung des GCC, entschärft. Die Frage, ob allein die Abhängigkeit von der XML-Ausgabe ausreicht, um den IML-Generator und die übrige Bauhaus-Software unter die GPL stellen zu müssen, ist jedoch nicht klar. Hier gilt das Gleiche, was beim GNU C++ Compiler schon gesagt wurde.

EDG C++ Front-End Das EDG-Front-End ist in einer für wissenschaftliche Zwecke freien Lizenz verfügbar. Diese ist aber für das Bauhaus-Projekt nicht ausreichend, da diese auch kommerziell zum Einsatz kommt. Daher müssen für die kommerziellen Einsätze des Front-Ends Lizenzen von EDG erworben werden. Form und Preis der Lizenzen wurden mit EDG abgeklärt, so dass das Front-End eingesetzt werden kann.

3.1.3.8 Effizienz

T.K.

Aufgrund der unterschiedlichen Plattformen und Anwendung war es nicht möglich, eine absolute und vergleichbare Messung der Laufzeiten durchzuführen. Zusätzlich konnte die Laufzeit für die Übersetzungsvorschrift in IML auch nur grob abgeschätzt werden. Aus diesen Gründen wird hier nur eine qualitative Abschätzung der Laufzeiten gegeben.

Columbus Für dieses Produkt wurde keine Untersuchung vorgenommen.

GEN++ Beim Einsatz von GEN++ kann eine geringe Effizienz erwartet werden, da bereits im ersten Übersetzungsschritt ein Übersetzer (Cfront) ausgeführt werden muss. Der IML-Übersetzer, der danach ausgeführt

werden würde, würde in einer interpretierten Programmiersprache ausgeführt werden. Eine Übersetzung in (schnelleren) Nativcode ist hier nicht möglich. Zusätzlich müssten die Informationen über eine Abfrageschnittstelle bezogen werden, was sehr wahrscheinlich langsamer sein wird als das direkte Traversieren einer Datenstruktur.

OpenC++ Wenn man eine semantische Analyse aus der Betrachtung herauslässt, kann von einer hohen Ausführungsgeschwindigkeit ausgegangen werden. Auf die Informationen ist über eine Callback-Schnittstelle aus zugreifbar, und der IML-Übersetzer würde in Nativcode-Form ausgeführt werden. Eine Zwischenspeicherung von Daten in Dateien entfällt.

GNU C++ Wenn direkt auf den AST des GNU C++ Compilers zugegriffen wird, kann ebenfalls eine hohe Ausführungsgeschwindigkeit erwartet werden. Die nachfolgenden Schritte des GCC könnten ausgeschaltet werden. Der IML-Übersetzer würde, wie auch der GCC selbst, in Nativcode-Form ausgeführt werden.

CPPX, GCC-XML und Introspector Die Ausführungsgeschwindigkeit würde auf jeden Fall langsamer sein als der direkte Zugriff auf den GCC, da die Daten in Dateien zwischengespeichert werden müssten, bevor sie vom IML-Übersetzer wieder eingelesen und interpretiert werden. Der IML-Übersetzer würde, da er ein eigenständiges Programm ist, in Nativcode-Form ausgeführt werden.

EDG C++ Front-End Hier gilt das, was auch beim direkten Zugriff auf den AST des GCC gesagt wurde. Wenn zusätzlich nur einzelne Funktionen im Speicher gehalten werden, was mit dem Front-End möglich ist, würden die Speicheranforderungen sinken.

3.1.4 Ergebnis

T.K.

In Tabelle 3.1 auf der nächsten Seite soll nochmals ein Überblick über die Ergebnisse der Evaluierung gegeben werden. Es werden die verschiedenen Produkte den Anforderungen gegenübergestellt. Die Spalten führen die Produkte auf und die Zeilen zeigen, wie gut diese die Anforderungen erfüllen.

Die einzigen beiden Front-Ends, die alle Anforderungen erfüllen, sind zum einen der GNU C++ Compiler und zum anderen das C++ Front-End von EDG. CPPX könnte ebenfalls noch mit in die Auswahl aufgenommen werden, wenn die Nachteile bezüglich der fehlenden Informationen akzeptierbar wären.

Im direkten Vergleich von CPPX mit dem GNU C++ Compiler hat der GCC einige Vorteile mehr zu bieten. Die Ausführungsgeschwindigkeit des GCC ist vergleichsweise besser, und auch die Anwendung des GCC mit der IML-Erweiterung wäre für den Benutzer einfacher und komfortabler.

Wenn der GNU C++ Compiler mit dem C++ Front-End von EDG verglichen wird, hat eine Implementierung basierend auf dem EDG-Front-End mehr Vorteile zu bieten. Einerseits ist die interne Darstellung des EDG-Front-Ends weit besser dokumentiert. Gleichzeitig stellt sie mehr Informationen über den Quelltext, z.B. über Macros, bereit. Auf der anderen Seite ist aber die Lizenzierung des EDG-Front-Ends bei weitem teurer als die des GNU C++ Compilers.

	Columbus	GEN++	GNU C++	OpenC++	CPPX	EDG	GCC-XML	Introspector
Ausreichende Information	⊖	⊖	⊕	⊖	⊖	⊕⊕	⊖	⊖
Plattformen	⊖	⊖	⊕⊕	⊕⊕	⊕⊕	⊕⊕	⊕⊕	⊕⊕
Einfache Anwendung	⊕	⊕⊕	⊕⊕	⊕	⊕⊕	⊕	⊕⊕	⊕⊕
Einfache Entwicklung	⊕⊕	⊖	⊕	⊖	⊕	⊕	⊖	⊖
Minimale Wartung	⊕⊕	⊕⊕	⊕	⊕⊕	⊕	⊕⊕	⊖	⊖
C++ Dialekte	⊕⊕	⊖	⊕	?	⊕	⊕⊕	⊕	⊕
Lizenzierung	⊕	⊖	⊕	⊕⊕	⊕	⊕	⊕	⊕
Effizienz	?	⊖	⊕⊕	⊕⊕	⊕	⊕⊕	⊕	⊕

Legende:
⊖ Die Anforderungen wurden nicht erfüllt
⊕ Die Anforderungen wurden erfüllt.
⊕⊕ Die Anforderungen wurden übererfüllt.
? Das Ergebnis ist nicht bekannt.

Tabelle 3.1: Übersicht über das Ergebnis der Front-End Evaluierung

Dafür ist jedoch die Rechtslage bezüglich der Lizenzierung klar.

Aus den oben genannten Gründen ist es die eindeutige Empfehlung der Autoren, das EDG-Front-End für die Entwicklung des IML-Übersetzers einzusetzen. Die zweitbeste Wahl fällt auf den C++ Compiler aus der GNU Compiler Collection.

Da die Wahl des Front-Ends weitreichende Konsequenzen für das Bauhaus-Projekt haben kann, besonders im Hinblick auf die Lizenzierung, war es die Entscheidung der Abteilung Programmiersprachen und Übersetzerbau, das C++ Front-End von EDG in dieser Diplomarbeit einzusetzen.

3.2 Das EDG-Front-End

S.S.

Das C++ Front-End der Firma EDG wurde als Front-End für die als Teil dieser Diplomarbeit zu erstellende Software ausgewählt. Die Gründe für die Auswahl sind in Kapitel 3.1 beschrieben. In diesem Kapitel wird das Front-End

genauer vorgestellt und die wichtigsten Eigenschaften des Front-Ends werden beschrieben. Diese Eigenschaften wirken sich auf den Entwurf der cafe++ Software in Kapitel 4 aus.

Das EDG-Front-End führt eine komplette syntaktische und semantische Analyse von C++ Quelltext durch. Das Ergebnis der syntaktischen Analyse wird als abstrakter Syntaxbaum dargestellt. Durch die semantische Analyse, also z.B. Namensbindung, werden in diesen Baum weitere Kanten eingefügt, wodurch ein allgemeiner gerichteter Graph entsteht. Dieser Graph, von EDG „IL" genannt, wird in Kapitel 3.2.1 beschrieben. Templates werden in instantiierter Form in der IL dargestellt. Für die korrekte Instantiierung von exportierten Templates, die in anderen Übersetzungseinheiten benutzt werden, sind mehrere Übersetzungsdurchläufe der einzelnen Übersetzungseinheiten notwendig. Ein Shellscript, das diese Übersetzungsdurchläufe in der richtigen Reihenfolge durchführt, wird mitgeliefert. Bei Bedarf werden auch Beschreibungen der Templates selbst im IL-Graphen erzeugt.

In das Front-End ist ein Präprozessor eingebaut, der vor der syntaktischen Analyse ausgeführt wird. Dadurch kann das Front-End Quelltextpositionen mit korrekter Spalteninformation zur Verfügung stellen. Fehler im Quelltext werden erkannt und mit der genauen Quelltextposition ausgegeben. Es werden auch „Precompiled Headers" unterstützt, um den Analysevorgang großer Systeme zu beschleunigen.

Als Beispiele zur Implementierung von Back-Ends werden ein C und ein C++ generierendes Back-End mitgeliefert. Beide Back-Ends generieren zur Eingabe semantisch äquivalenten Quelltext. Die Eingabe kann ein beliebiger C++ Quelltext sein, die Ausgabe ist C bzw. C++ Quelltext. Das C generierende Back-End ist mit Hilfe der „IL-lowering" Phase implementiert. Die „IL-lowering" Phase transformiert den IL-Graphen von C++ Programmen in einen IL-Graphen, der ein äquivalentes C Programm beschreibt. Die „IL-lowering" Phase kann benutzt werden, um ein Back-End zu vereinfachen – das Back-End muss dann nur noch C Sprachkonstrukte beherrschen. Das mitgelieferte C++ generierende Back-End ist als Beispiel für Back-Ends gedacht, die die C++ Sprachkonstrukte selbst implementieren. Das Front-End und die Beispiel-Back-Ends sind in ISO C implementiert.

Zusätzlich zu den Kommentaren im Quelltext des Front-Ends gibt es etwa 500 Seiten Dokumentation in ausdruckbarer Form [6]. Ein Kapitel davon, *„External Interface"*, kann im Internet kostenlos heruntergeladen werden (`http://www.edg.com`). Es ist interessant für Benutzer von Übersetzern, die mit Hilfe des EDG-Front-Ends implementiert wurden und die Schnittstelle des Hauptprogramms übernehmen.

Außer der kompletten Sprache C++ nach ISO/IEC 14882:1998 [14] werden bei Bedarf verschiedene Dialekte von C++, unter anderem von Microsoft, Sun und GNU, sowie verschiedene C Dialekte inklusive ANSI/ISO C89 und C99 unterstützt. Die Unterstützung dieser Dialekte hat jedoch Auswirkungen auf die Darstellung des IL-Graphen und muss deshalb auch vom Back-End implementiert werden, wenn sie aktiviert wird.

3.2.1 Der IL-Graph

S.S.

Das EDG-Front-End bietet alle Programminformationen über die analysierte Übersetzungseinheit zentral in einer verzeigerten Datenstruktur an. Diese Datenstruktur wird *intermediate language* (IL) genannt. Die Datenstruktur besteht im Wesentlichen aus Strukturen (`struct`) und Zeigern in den Strukturen, die auf andere Strukturen zeigen. Dadurch ergibt sich ein gerichteter Graph, mit den Strukturen als Knoten und den Zeigern als Kanten. Dieser Graph wird IL-Graph genannt.

Der IL-Graph ist ein abstrakter Syntaxbaum des Quelltextes mit zusätzlichen Kanten aus der semantischen Analyse und mit zusätzlichen Knoten für implizite Anweisungen und andere Zusatzinformationen. Die Mehrdeutigkeiten, die z.B. durch überladene Funktionen und Operatoren entstehen können, sind bereits aufgelöst, soweit das statisch möglich ist. Weiterhin enthält der IL-Graph detaillierte Informationen über Lebenszeiten und Sichtbarkeit von C++ Instanzen. Die Expansion von Templates wird ebenfalls bereits in einer geeigneten Weise vom Front-End durchgeführt. Obwohl diese erweiterten Informationen im IL-Graphen angeboten werden, werden vom Front-End keine Optimierungen am IL-Graphen durchgeführt, so dass dieser den ursprünglichen Quelltext sehr genau abbildet. Die IL ist keine maschinennahe oder gar von einer bestimmten Maschine abhängige Zwischendarstellung zur Beschreibung der Ausführungssemantik. Die enthaltenen Informationen sind jedoch ausreichend zur Beschreibung der Ausführungssemantik.

Um alle benötigten Informationen zu erlangen, muss das Front-End per Makrodefinitionen entsprechend konfiguriert werden (siehe [6, Kapitel 3] und Kapitel 3.2.3). Die Erzeugung von Quelltext-Informationen, die für ein normales Übersetzer-Back-End nicht relevant sind, kann durch Konfigurationsparameter deaktiviert werden.

Dadurch, dass der IL-Graph alle Programminformationen enthält, reicht es für ein Back-End wie den IML-Übersetzer aus, diesen IL-Graphen als Eingabedatum zu erhalten. Die Schnittstelle zwischen dem Front-End und dem Back-End besteht also nur aus der Übergabe des IL-Graphen. Der IL-Graph kann Funktion für Funktion oder für eine ganze Übersetzungseinheit übergeben werden. Näheres dazu ist in Kapitel 4.2.3.2 beschrieben.

Der Zugriff des Back-Ends auf die Knoten des IL-Graphen kann direkt erfolgen. Die Definitionen der Strukturen und damit der interne Aufbau der Knoten sind für das Back-End zugänglich. Zusätzlich zum direkten Zugriff auf die Attribute der Knoten kann über den Aufruf von Hilfsfunktionen oder -makros Information erhalten werden, die nicht direkt erreichbar ist. Ein Überblick über den Aufbau des IL-Graphen ist in der Dokumentation [6] des Front-Ends enthalten. Detailliertere Informationen können den Kommentaren in der Datei `il_def.h` entnommen werden (siehe Kapitel 3.2.1.1). Dort befinden sich die Definitionen der Knoten und Kanten des IL-Graphen. Informationen zum IL-Graphen eines bestimmten Quelltextes können dem IL-dump entnommen werden (siehe Kapitel 3.2.1.3).

3.2.1.1 Die Typen der IL-Graph-Knoten

S.S.

Die Definitionen zu den Typen der IL-Knoten im IL-Graphen sind in der Datei `il_def.h` enthalten. Alle Typen werden einheitlich `a_XXX` oder `an_XXX` genannt.

In ISO C können verschiedene Ausprägungen eines Typs nicht durch Vererbung dargestellt werden. Deshalb werden IL-Knoten oft mit einem „`kind`" Attribut mit Aufzählungstyp versehen. Verschiedene Ausprägungen eines Typs werden also nicht durch Zugehörigkeit von Objekten zu verschiedenen Unterklassen des Typs, sondern durch verschiedene Werte von `kind` dargestellt.

Falls `kind` mehr als einen Wert annehmen kann, also mehr als ein Typ dargestellt wird, werden die verschiedenen Typen in einer Variante untergebracht. Der Typ dieser Variante ist unbenannt, die Variante selbst heißt „`variant`". Durch Kommentare ist gekennzeichnet, zu welchem Wert von `kind` welches Feld von `variant` gehört. Falls zu einem Wert von `kind` mehrere Attribute gehören, werden diese in einem Feld von `variant` mit anonymem Struktur-Typ untergebracht. Polymorphe Zeiger werden nicht nur als Zeiger auf Strukturen mit einem `kind`-Attribut, sondern oft auch als verschiedene Felder von `variant` mit unterschiedlichem Zeigertyp dargestellt.

3.2.1.2 Traversieren des IL-Graphen

S.S.

Der IL-Graph kann traversiert werden, indem man die Kanten des Graphen verfolgt. Als Startpunkt muss dabei die globale Variable `il_header` verwendet werden. Diese Variable stellt die Übergabe des IL-Graphen vom Front-End an das Back-End dar. Dabei ist sicherzustellen, dass die Traversierung abbricht, auch wenn im IL-Graphen Zyklen vorhanden sind. Kanten, die in den Teilgraphen des ausführbaren Teils einer Funktion zeigen, gibt es nicht.

Die oberste Ebene des ausführbaren Teils einer Funktion wird durch Knoten von Typ „`a_scope`" dargestellt. Der einzige Zugang zu dem Teilgraphen von außerhalb ist seine Nummer. Die Nummer indiziert ein Array aus Zeigern auf `a_scope`-Knoten. Dieses Array enthält im Fall einer funktionenweisen Übergabe des IL-Graphen an das Back-End fast nur Null-Zeiger. Es ist in der globalen Struktur `il_header` enthalten.

Alternativ zum direkten Traversieren kann ein von EDG implementiertes Besuchermuster verwendet werden. Die Schnittstelle dazu befindet sich in `il_walk.h`. Bei Verwendung des Besuchermusters ist durch Markierung von Knoten sichergestellt, dass jeder Knoten nur einmal besucht wird.

3.2.1.3 IL-dump

S.S.

Im EDG-Front-End sind Funktionen zur textuellen Darstellung des IL-Graphen vorhanden. Diese Darstellung wird IL-dump genannt. Schon zu Beginn der Implementierung von cafe++ wurde die Ausgabe des IL-dumps mit Hilfe dieser Funktionen implementiert.

Der IL-dump enthält durch Leerzeilen getrennte Beschreibungen der Knoten. In der ersten Zeile jeder Knotenbeschreibung steht der Typ des Knotens und eine eindeutige Nummer. In den folgenden Zeilen sind die relevanten At-

tribute (auch die aus `variant` – siehe Kapitel 3.2.1.1) ausgegeben. Verweise auf andere Knoten erfolgen mittels deren eindeutigen Nummern.

Da die Graphenstruktur aus der textuellen Darstellung mit Verweisen als eindeutige Nummern nicht immer einfach ersichtlich ist, wurde das Werkzeug `il2dot` entwickelt. Die Werkzeugkette `il2dot` – `dot` – `psmulti` erzeugt aus IL-dumps, oder Ausschnitten daraus, eine ausdruckbare grafische Darstellung des IL-Graphen (-Ausschnitts). `il2dot` befindet sich im `tools`-Verzeichnis des cafe++ Projekts. `dot` ist ein Graphvisualisierungswerkzeug [2]. `psmulti` ist ein Werkzeug der Abteilung Programmiersprachen und Übersetzerbau, das große PostScript-Grafiken auf mehrere DIN A4-Blätter verteilt.

Neben den Kommentaren und dem Quelltext in der Datei `il_def.h` waren IL-dumps die wichtigste Informationsquelle dieser Diplomarbeit über die IL. Sie sind jedoch leider nicht immer vollständig. Es kann z.B. vorkommen, dass auf Knoten verwiesen wird, die nicht im IL-dump enthalten sind. Unter welchen Bedingungen der IL-dump unvollständig ist, ist unbekannt. Der Fall tritt aber äußerst selten auf. Beim Test von cafe++ (siehe Kapitel 5) ist das Fehlen von Knoten bei einem von 200 Testdateien aufgefallen.

3.2.1.4 Lebenszeit von Objekten

T.K.

In der IL werden Informationen über die Lebenszeit von Objekten detailliert zur Verfügung gestellt [6, Kapitel 5.18 „Object Lifetime“]. Mit Hilfe der so genannten „object lifetime”-Knoten ist es zu nahezu jedem Zeitpunkt möglich, die Objekte zu ermitteln, die zerstört werden müssen. Informationen dieser Art sind zum Beispiel besonders wertvoll beim Erreichen eines Blockendes oder von `goto`-Anweisungen.

Leider sieht die IL eine solch genaue Modellierung nur für Instanzen von Klassen vor, deren Destruktoren am Ende der Lebenszeit aufgerufen werden müssen. Das hat den Hintergrund, dass bei den übrigen Objekten ohne Destruktoren die Lebenszeit nicht interessant ist. Deren Lebenszeit ist identisch mit der Zeit, die sie im Speicher verweilen. Aus Gründen der Speichereffizienz der IL werden für Objekte dieser Art also keine „object lifetime”-Knoten erzeugt. Bei Objekten mit Destruktoren sind die erzeugten „object lifetime”-Knoten von großem praktischen Nutzen, da das EDG-Front-End implizite Destruktoraufrufe nicht explizit in der IL modelliert. Ein Back-End kann die „object lifetime” Information des Front-Ends verwenden, um die nötigen Destruktoraufrufe zu ermitteln. Benötigt ein Back-End auch detaillierte Informationen über die Lebenszeit aller Objekte, muss es sich diese selbst berechnen.

3.2.1.5 Exportierte Templates

S.S.

Exportierte Templates sind Templates, die mit dem Schlüsselwort `export` deklariert wurden. Templates werden in instantiierter Form in der IL dargestellt. Für die korrekte Instantiierung von exportierten Templates, die in anderen Übersetzungseinheiten benutzt werden, sind mehrere Übersetzungsdurchläufe der einzelnen Übersetzungseinheiten notwendig. Ein Shellscript, das diese Übersetzungsdurchläufe in der richtigen Reihenfolge durchführt, wird mitgeliefert. Es heißt `eccp`. Das Shellscript `eccp` muss möglicherweise an die

Bedürfnisse des Back-Ends, z.B. neue Kommandozeilenparameter, angepasst werden. Das Shellscript ruft einen „Prelinker" auf, der dafür sorgt, dass die Templates instantiiert werden. Dazu wird der Compiler nochmals aufgerufen, nachdem der Prelinker die nötige Information in Dateien mit der Endung `.ii` abgelegt hat. Das Front-End hat mehrere mögliche Einstellungen, wie die Instantiierung von exportierten Templates geschehen soll. Beispielsweise kann man das Front-End so einstellen, dass für jede Template-Instantiierung eine eigene `.o`-Datei erstellt wird.[6, Kapitel 2.11.1 „Automatic Instantiation", Kapitel 17.9 „One-instantiation-per-object mode"]

3.2.1.6 Symboltabelle

S.S.

Die Symboltabelle des Front-Ends wird nicht an das Back-End weitergegeben [6, S. 172].

Eine herkömmliche Symboltabelle, die Namen (in Form von Zeichenketten) auf andere Objekte abbildet, wird in einem Back-End in der Regel nicht benötigt. Alle nötigen Informationen sind in der IL vorhanden. Das gilt auch für cafe++. Wenn im Entwurf von cafe++ von einer Symboltabelle die Rede ist, ist eine Abbildung von IL-Knoten auf IML-Knoten gemeint. Siehe Kapitel 4.2.3.5.

3.2.1.7 Zweitrangige Deklarationen

S.S.

Knoten vom Typ `a_src_seq_secondary_decl` werden erzeugt, wenn die Einstellung `GENERATE_SOURCE_SEQUENCE_LISTS` auf `TRUE` gesetzt ist.

Der Kommentar in `il_def.h` dazu lautet: *„A 'source sequence secondary declaration entry' is pointed to from the source sequence list to identify a declaration that is not a 'primary' declaration – e.g., a function declaration that is not a definition."*

Tatsächlich werden Knoten vom Typ `a_src_seq_secondary_decl` *genau* für die Deklarationen erzeugt, die keine Definitionen sind. Auch wenn es für eine Deklaration innerhalb einer Übersetzungseinheit gar keine Definition und keine andere Deklaration gibt, wird diese dadurch nicht zu einer *„'primary' declaration"*, sondern sie wird durch einen `a_src_seq_secondary_decl`-Knoten dargestellt.

Eine Ausnahme von dieser Regel sind Namensbereiche (`namespace`). Diese können mehrfach geöffnet, also mehrfach definiert werden. Für alle weiteren Definitionen nach der ersten werden `a_src_seq_secondary_decl`-Knoten erzeugt. Eine weitere Ausnahme sind nicht-instantiierte Methoden von Template-Klassen, falls man diese als „nicht definiert" betrachtet.

3.2.1.8 Erweiterte Namen (name mangling)

S.S.

Das EDG-Front-End erzeugt während der IL-lowering Phase erweiterte Namen für überladene Funktionen und Variablen mit globaler Lebensdauer innerhalb von Namensbereichen (und Klassen). Die Signaturen von überladenen Funktionen sind in C++ eindeutig, der Name allein jedoch nicht. Die vom EDG-Front-End realisierte Erzeugung erweiterter Namen entspricht der Umsetzung, wie sie im Cfront-Übersetzer realisiert ist und auch der Beschreibung

in „The Annotated C++ Reference Manual" [6, Kapitel 6.2.1]. Die erweiterten Namen bestehen aus mehreren, durch Unterstriche („_") getrennten Teilen. Dazu gehören: Der erweiterte Name des Namensbereichs, der nicht erweiterte Name und der erweiterte Name des Typs. Aus diesem Grund müssen auch für Typen erweiterte Namen erzeugt werden. Im erweiterten Namen von Funktionstypen sind die erweiterten Namen der Parametertypen enthalten. Dadurch werden die erweiterten Namen von überladenen Funktionen eindeutig.

3.2.1.9 Beispiel eines IL-Graphen

S.S.

In Abbildung 3.1 ist der IL-Graph zu dem Programm

```
int main(){
}
```

abgebildet. Er wurde, wie in Kapitel 3.2.1.3 beschrieben, über die Werkzeugkette „cafe++ – il2dot – dot" erstellt. Die gelben Knoten, von denen zwei durch einen Rahmen zusammengefasst sind und eine verkettete Liste bilden, sind Knoten vom Typ *source-sequence-entry*. Die grünen Knoten sind vom Typ *statement*. Die eindeutigen Nummern der Knoten werden auch ausgegeben, damit in der textuellen Version des IL-dumps Details zu den Knoten gefunden werden können.

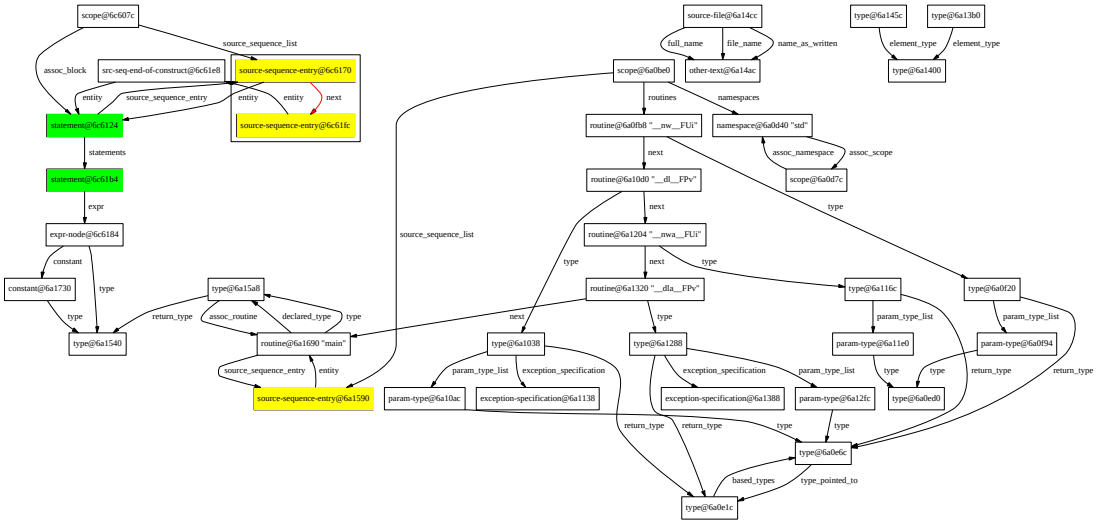


Abbildung 3.1: Beispiel IL-Graph zu „`int main(){}`"

Die Abbildung soll nur eine Vorstellung von der Struktur eines IL-Graphen vermitteln. Die Lesbarkeit der Beschriftungen ist für das Verständnis des Textes nicht relevant.

3.2.2 Einbindung des Front-Ends in ein Programm

S.S.

In der Datei `cfe.c` des Front-End-Quelltextes ist ein Hauptprogramm enthalten. Dieses Hauptprogramm hat als Namen den Wert des Makros `EDG_MAIN`.

Falls dieses Makro nicht während der Konfiguration gesetzt wird, wird sein Wert automatisch auf „`main`" gesetzt. Das Hauptprogramm in `cfe.c` ruft die Funktion `back_end()` auf. Diese Funktion wird im Quelltext von EDG nur definiert, falls eines der beiden C bzw. C++ generierenden Beispiel-Back-Ends aktiviert ist.

Um ein eigenes Back-End von dem Hauptprogramm in `cfe.c` aufrufen zu lassen, muss man nur die Funktion `back_end()` definieren. Die Beispiel-Back-Ends werden durch Setzen der Makros `BACK_END_IS_C_GEN_BE` und `BACK_END_IS_CP_GEN_BE` auf `FALSE` deaktiviert.

Der Aufruf des Back-Ends durch das Hauptprogramm in `cfe.c` erfolgt, je nach Konfiguration, einmal für die ganze Übersetzungseinheit oder je einmal für jede Definition einer Funktion (siehe Kapitel 4.2.3.2).

Wenn die Funktionalität des Hauptprogramms in `cfe.c` nicht ausreicht, kann es aus einem eigenen Hauptprogramm heraus aufgerufen werden, indem es mit Hilfe des Makros `EDG_MAIN` umbenannt wird. Dies kann z.B. eingesetzt werden, um eigene Kommandozeilenparameter aus der Parameterliste zu filtern.

Die einzige Information, die von dem Front-End an das Back-End weitergegeben wird, ist der IL-Graph über die globale Variable `il_header`. Die Traversierung dieses Graphen wird in Kapitel 3.2.1.2 beschrieben. In allen bisher beschriebenen Vorgehensweisen, die die Datei `cfe.c` unverändert benutzen, erfolgt die Parameterübergabe an das Back-End (also an die Prozedur `back_end()`) über globale Variablen. Falls das nicht erwünscht ist, oder wenn die Benutzung des Hauptprogramms in `cfe.c` aus anderen Gründen nicht möglich ist, kann die Datei `cfe.c` auch kopiert und an eigene Bedürfnisse angepasst werden.

In Kapitel 4.2.3 ist beschrieben, wie das Front-End in cafe++ eingebunden wurde.

3.2.3 Konfiguration des Front-Ends

S.S.

Das EDG-Front-End kann vor der Übersetzung des Front-End-Quelltextes und beim Aufruf des Programms (zum Analysieren von C++ Quelltext) konfiguriert werden. Die Konfiguration beim Aufruf des Programms erfolgt durch Kommandozeilenparameter. Die Standardwerte für die Kommandozeilenparameter können bei der Konfiguration vor der Übersetzung festgelegt werden.

Die Konfiguration vor der Übersetzung erfolgt durch Belegen von Präprozessor-Makros mit Werten, in den meisten Fällen „`TRUE`" oder „`FALSE`", in der Datei `defines.h`. Diese Datei wird vom EDG-Quelltext mit `#include` eingebunden, gehört jedoch nicht zum EDG-Quelltext. Beispiel-Konfigurationsdateien, die als Vorlage für eine eigene Datei `defines.h` genommen werden können, werden für verschiedene Plattformen mitgeliefert. Konfigurationsparameter, die in `defines.h` nicht gesetzt werden, werden in verschiedenen Header-Dateien von EDG gesetzt. Werte, die in `defines.h` gesetzt werden, werden in diesen Header-Dateien im Normalfall nicht überschrieben. Dies wurde durch `#ifndef`-Anweisungen erreicht. Die Beschreibung aller Konfigurationsparameter ist in den Kommentaren dieser Header-Dateien und in [6, Kapitel 3] zu finden.

Die Konfigurationseinstellungen des Front-Ends für cafe++ sind in Kapitel 4.3.4 beschrieben.

Globale Analyse mehrerer Übersetzungseinheiten

Wenn das Makro `COMPILE_MULTIPLE_TRANSLATION_UNITS` (nicht zu verwechseln mit `COMPILE_MULTIPLE_SOURCE_FILES`) in der EDG Headerdatei `host_envir.h` auf `TRUE` gesetzt ist, baut das EDG-Front-End einen einzigen IL-Graphen für alle Übersetzungseinheiten, deren Quelltexte in der Kommandozeile angegeben werden. Bei Benutzung dieses Merkmals ergeben sich folgende Konsequenzen:

- Der IML-Linker wird nicht benötigt. Dadurch werden alle durch den IML-Linker hervorgerufenen Probleme, z.B. die in Kapitel 7.2.1.2 beschriebenen, gelöst.

  Ohne den IML-Linker kann jedoch kein System analysiert werden, in dem mehrere Programmiersprachen verwendet werden, beispielsweise Ada und C++.

- Bei großen Systemen kann der Speicherbedarf und die Übersetzungszeit zu groß werden.

Da auch mehrsprachige Systeme mit Bauhaus analysiert werden sollen, muss cafe++ mit dem IML-Linker zusammenarbeiten. Eine globale Analyse durch cafe++ ist zweitrangig, kann jedoch mit Hilfe der Front-End-Einstellung `COMPILE_MULTIPLE_TRANSLATION_UNITS` und relativ wenig Aufwand später implementiert werden.

Kapitel 4

Entwurf und Realisierung

T.K.

Im direkten Anschluss an die Vorbereitung des Tests wurde der Entwurf von cafe++ festgelegt, der in diesem Kapitel beschrieben wird. Die Durchführung des Entwurfs kann zeitlich in drei aufeinanderfolgende Schritte aufgeteilt werden. Zu Beginn der Entwurfsphase wurden mehrere Tage für die Einarbeitung in das EDG-Front-End investiert. Dabei wurden zum einen die Dokumentation und der Quelltext des EDG-Front-Ends näher betrachtet. Zum anderen wurde ein minimaler Prototyp erstellt, der den IL-Graphen des Front-Ends in Textform ausgibt (siehe Kapitel 3.2.1.3), um die Beschaffenheit des ASTs zu untersuchen. Die Einarbeitungszeit war notwendig, damit die Autoren über genügend Wissen verfügen, um im nächsten Schritt den Systementwurf zu erstellen. Von beiden Autoren wurden dazu mehrere Vorschläge erarbeitet, die Schritt für Schritt gemeinsam diskutiert wurden, bis sich relativ schnell eine umfassende Lösung herauskristallisiert hatte. Als der Systementwurf fertig gestellt war, wurde dieser dokumentiert und parallel in einem Schnittstellenentwurf konkretisiert. Das Ergebnis des Schnittstellenentwurfs waren übersetzbare Quelltextdateien von den definierten Funktionen und Klassen, realisiert als Stubs. Aufgrund dessen konnte die darauf folgende Implementierung stark parallelisiert werden.

In den weiteren Kapiteln zum Entwurf werden zuerst die Anforderungen angegeben, die durch den Entwurf angestrebt wurden. Danach wird der Systementwurf selbst vorgestellt. Im Anschluss an den Entwurf wird in Kapitel 4.3 seine Realisierung beschrieben. In Kapitel 4.4 werden die bekannten Einschränkungen des Entwurfs und seiner Realisierung genannt. Kapitel 4.5 behandelt äußerliche Eigenschaften der Realisierung.

4.1 Anforderungen

T.K.

Damit ein guter Entwurf gelingen kann, müssen zuerst die Anforderungen definiert sein, die im Entwurf umgesetzt werden sollen. Ansonsten ist es schwierig, bei der Erstellung des Entwurfs auf ein eindeutiges Ziel hinzuarbeiten.

Die für den Entwurf von cafe++ geltenden Anforderungen sollen an dieser Stelle genannt werden. Den Ursprung haben diese Anforderungen aus den direkten Anforderungen aus der Aufgabenstellung zu cafe++ [21] und indirekt aus den Anforderungen, die für alle Bestandteile des Bauhaus-Projektes

gelten. Teilweise werden auch abgeleitete Kriterien genannt, wenn sie für das Projekt von Relevanz sind. Die Anforderungen, die bereits durch die Auswahl des Front-Ends (siehe Kapitel 3.1) und durch die Spezifikation der erweiterten IML (siehe Kapitel 2.4) umgesetzt wurden, gelten natürlich auch für den Entwurf, sollen hier jedoch nicht noch einmal genannt werden.

- Der Übersetzer soll die IML-Darstellung von einzelnen Übersetzungseinheiten erzeugen. Ist ein Programm in mehrere Übersetzungseinheiten aufgeteilt, können die IML-Dateien mittels des IML-Linkers zu einer gesamten Programmrepräsentation kombiniert werden.

- Im Falle von (teilweise) fehlerhaften Übersetzungseinheiten darf der cafe++ Übersetzer keine Ausgabe generieren, auch nicht von der bis zur Fehlerstelle erfolgreich bearbeiteten Übersetzungseinheit.

- cafe++ muss mindestens die Plattformen Linux und Solaris als Laufzeitumgebung unterstützen. Eine Unterstützung für weitere Plattformen, wie Microsoft Windows oder HP-UX, würde einen praktischen Vorteil für die spätere Anwendung haben, ist jedoch zweitrangig.

- Die Realisierung der cafe++ Implementierung stützt sich bei der syntaktischen und semantischen Analyse auf das C++ Front-End von EDG in der Version 3.0.1 (siehe hierzu Kapitel 3.1).

- Die Implementierung der IML-Klassen muss mit Hilfe des in Bauhaus existierenden Generators erfolgen. Das hat den Hintergrund, dass die IML-Darstellung effizient gespeichert und geladen werden soll. Daraus folgt, dass die Implementierung der IML-Klassen in Ada 95 geschehen muss, da der Generator Quelltexte in Ada 95 erzeugt.

- Der cafe++ Übersetzer sollte effizient implementiert sein. Als Richtlinie für die Geschwindigkeit des Übersetzungsprozesses kann gelten, dass eine Übersetzung von C++ Quelltexten nach IML nicht viel länger dauern sollte, als eine Übersetzung in Maschinensprache. Durch die Verwendung des Generators aus Bauhaus ist der Handlungsspielraum bezüglich dieser Anforderung bereits im Voraus eingeschränkt. Weiterhin legt das verwendete C++ Front-End die Übersetzungszeit natürlich auch maßgeblich fest.

- Durch den späteren Einsatz von cafe++ ist es sehr wahrscheinlich, dass das Projekt auf irgendeine Weise gewartet werden muss. Beispielsweise könnte die Implementierung an eine neue Version der IML-Spezifikation oder des C++ Front-Ends angepasst werden müssen. Eine möglichst minimale Wartung impliziert folgende Punkte:

  - Allgemein, wie für jeden Entwurf geltend, sollte ein Entwurf einfach im Aufbau und damit auch leicht verständlich für den Wartungsingenieur sein. Gleichzeitig sollte ein Entwurf jedoch mächtig in seiner Leistungsfähigkeit und Erweiterbarkeit sein.

- Eventuelle Änderungen und Anpassungen des bereits bestehenden Front-Ends sollten so einfach und gering wie möglich ausfallen, damit sie auch in neueren Versionen des Front-Ends mit möglichst wenig Aufwand durchgeführt werden können. Idealerweise sollte der bestehende Quelltext des Front-Ends überhaupt nicht verändert werden müssen.
- An den vom Generator erstellten Dokumenten dürfen überhaupt keine Änderungen durchgeführt werden, da das die Anpassung an neue Versionen der IML-Spezifikation zu sehr behindern würde.

4.2 Systementwurf

T.K.

Anschließend soll der Systementwurf beschrieben werden. Außer dem endgültigen Entwurf werden in den folgenden Kapiteln weiterhin Alternativen diskutiert, damit die Entwurfsentscheidungen für die spätere Wartung verständlicher werden.

4.2.1 Programmiersprache

T.K.

Bei der Wahl der Programmiersprache für die Implementierung von cafe++ gelten im Wesentlichen dieselben Aussagen wie beim jafe-Projekt (siehe [19, Kapitel 4.1]) für nähere Informationen. Die Implementierung des C++ Front-Ends von EDG wurde komplett in der Programmiersprache C realisiert, wobei ein portabler Dialekt verwendet wurde, um die Übersetzung mit möglichst vielen C Übersetzern möglich zu machen. Der Generator von Bauhaus erzeugt im Gegensatz dazu die Implementierung der IML-Komponente in Ada 95. Zusätzlich wurde jedoch im Zuge des jafe-Projektes der Generator erweitert und kann nun ebenfalls eine C Schnittstelle für diese IML-Datenstrukturen erzeugen. Aus denselben Gründen, die auch beim jafe-Projekt entscheidend waren, wurde die Programmiersprache für cafe++ auf C oder C++ eingeschränkt.

C als Programmiersprache war wegen des verwendeten C++ Front-Ends von EDG mindestens erforderlich. Zudem stand noch die Option zur Verfügung, C++ für die Implementierung von cafe++ zu verwenden. Die Beschränkung auf C hat den Vorteil, dass die Implementierung des Projekts mit nur zwei Programmiersprachen auskommt. Wenn C++ eingesetzt wird, baut die Implementierung von cafe++ bereits auf drei Programmiersprachen auf. Als Vorteil ergeben sich aber alle zusätzlichen Vorteile von C++, was vor allem bedeutet:

- C++ ist restriktiver als C. Aufgrund dessen kann davon ausgegangen werden, dass bei der Implementierung weniger Fehler bedingt durch die Programmiersprache auftreten. Ein solcher Vorteil ist z.B. das „strong typing" bei Klassen in C++.
- C++ unterstützt (im Gegensatz zu C) Spracherweiterungen wie Objektorientierung, Ausnahmebehandlung und generische Typen. Der Einsatz dieser Spracherweiterungen kann die Implementierung vereinfachen und beschleunigen.

Die Verwendung von C++ als weitere Programmiersprache birgt für den Wartungsingenieur auch keine weitere Anforderung, weil dieser, auch ohne den Einsatz von C++ in der Implementierung, Kenntnisse in C++ benötigt. Diese Kenntnisse müssen zum Warten eines C++ Übersetzers wie cafe++ unbedingt vorhanden sein. Der Zugriff von C++ auf C Quelltexte ist ebenso problemlos möglich.

Aus den genannten Gründen wurde als Programmiersprache für den cafe++ Übersetzer C++ ausgewählt. Zusammengefasst besteht die Implementierung von cafe++ aus drei Komponenten, die in drei Programmiersprachen realisiert sind: das C++ Front-End von EDG (C), dem cafe++ Übersetzer (C++) und den IML-Quelltexten des Generators (Ada 95). Der Zugriff vom cafe++ Übersetzer auf die Implementierung der IML-Klassen geschieht mittels einer ebenfalls vom Generator erzeugten C Schnittstelle.

4.2.2 Entwicklungsplattform

T.K.

Die Entwicklungsplattform von cafe++ muss in der Lage sein, Quelltexte in Ada 95, C und C++ zu übersetzen. Außerdem muss dies mindestens auf den Laufzeitumgebungen Linux und Solaris möglich sein. Um diesen Anforderungen gerecht zu werden, wurde folgende Konfiguration als Entwicklungsplattform ausgewählt:

- Für alle geforderten Laufzeitumgebungen ist die POSIX-Umgebung [11] ausreichend. Damit kann als Laufzeitumgebung Linux, Solaris, HP-UX, MS Windows und noch mehr unterstützt werden. Für MS Windows kann die volle POSIX-Kompatibilität mit Hilfe des „cygwin"-Projektes erreicht werden. Es enthält auch die Entwicklungswerkzeuge wie die GNU Compiler Collection (GCC).

- Die Quelltexte müssen mit der POSIX-Umgebung und den Standard-Bibliotheken der jeweiligen Programmiersprachen übersetzbar sein. Die Einbindung weiterer Bibliotheken sollte nicht nötig sein.

- Als Übersetzer wurde für die C und C++ Teile des Quelltextes der g++ aus der GCC, in der Version 3.0 oder höher ausgewählt. Für die Ada 95-Quelltexte wurde GNAT, in der Version 3.13 oder höher, gewählt. Diese Übersetzer sind frei verfügbar, unterstützen eine Vielzahl an Plattformen und setzen die jeweiligen Sprachstandards sehr gut um. Auf den Rechnern der Abteilung Programmiersprachen und Übersetzerbau war keine untereinander kompatible Kombination eines C++ und eines Ada Übersetzers verfügbar. Deshalb wurde für Solaris die GCC in der Version 3.2 mit C++ und GNAT eingesetzt.

  Alle hier eingesetzten Übersetzer bieten Erweiterungen der Programmiersprachen jenseits des jeweiligen Sprachstandards an. Für das cafe++ Projekt wird jedoch nur der jeweilige Sprachstandard genutzt, so dass die Erweiterungen nicht eingesetzt werden dürfen. Damit soll die Portabilität des Quelltextes erhöht werden.

- Als Editor für den Quelltext von cafe++ wird SciTE ab Version 1.48 empfohlen. Mit diesem Editor können Bereiche, die mit

```
//{ Kommentar
...
//} Kommentar
```

gekennzeichnet sind, auf eine Zeile zusammengefaltet werden. Dieses Merkmal wurde im Quelltext von cafe++ ausgiebig benutzt.

4.2.3 Systemarchitektur

T.K.

Bedingt durch die Vorgaben bezüglich bestehender Software kann die cafe++ Implementierung grob in drei große Komponenten aufgeteilt werden: dem C++ Front-End von EDG, dem in dieser Diplomarbeit implementierten Zwischenstück `CafePP` und der IML-Implementierung, welche vom Generator aus der Bauhaus Tool-Suite automatisch erzeugt wird. Abbildung 4.1 veranschaulicht diese grobe Aufteilung des Systems.

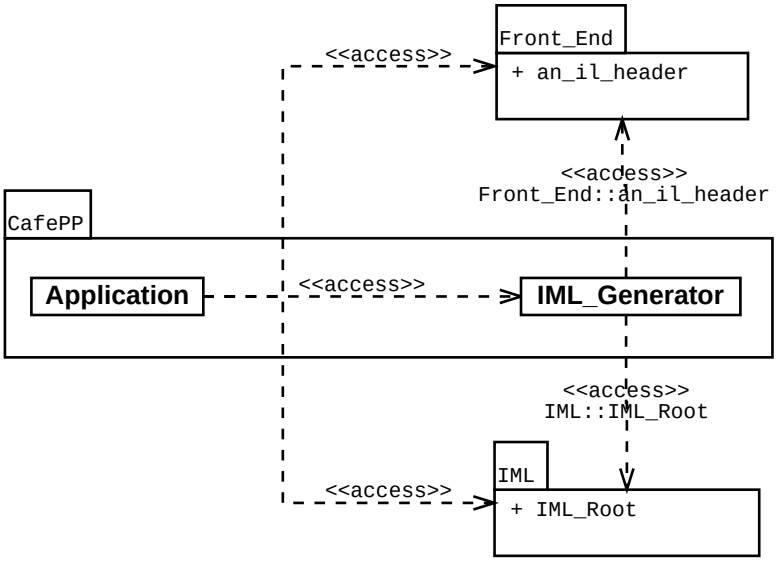


Abbildung 4.1: Systemarchitektur von cafe++

Die Systemarchitektur wurde objektorientiert entworfen. Damit wird das cafe++ Programm gestartet, indem eine globale Instanz der Applikationsklasse erzeugt wird; im Falle von cafe++ heißt diese `CafePP::Application`. Die Applikationsklasse ist dann verantwortlich für den weiteren Ablauf der Applikation. Nach der Initialisierung und dem Interpretieren der Aufrufparameter wird das EDG-Front-End angestoßen und analysiert die Übersetzungseinheit. War die Analyse erfolgreich, d.h. es wurden keine Fehler im Quelltext der Übersetzungseinheit entdeckt, wird der vom Front-End erzeugte IL-Graph dem IML-Übersetzer übergeben, welcher in der Klasse `IML_Generator` implementiert ist. Dieser traversiert den IL-Graphen und baut parallel dazu den IML-Graphen auf. Da die Übersetzung bei einem komplett gegebenen

IL-Graphen immer erfolgreich sein muss, kann die Applikation das Ergebnis der Übersetzung nun in eine externe Datei, die IML-Datei, schreiben. Damit ist der Übersetzungsvorgang beendet.

4.2.3.1 Untermodule des Moduls `CafePP`

S.S.

Das Modul `CafePP` verbindet das Front-End von EDG mit den automatisch erzeugten IML-Klassen von Bauhaus. Es wurde von den Autoren dieser Diplomarbeit entwickelt. Bis auf einige Methodendeklarationen und Definitionen mit leerem Rumpf (Stubs) wurde der Quelltext von Hand geschrieben. Das Modul ist in mehrere Untermodule aufgeteilt, über die hier eine Übersicht gegeben wird.

Application Das Modul `Application` enthält das Hauptprogramm und die Klasse `CafePP::Application`. Dort werden alle nötigen Schritte zur Übersetzung von C++ nach IML eingeleitet. Die wichtigsten Schritte sind:

1. Aufbau des IL-Graphen durch das EDG-Front-End;
2. IL-Vorbereitung durch das Modul IL-Prepare;
3. Generieren des IML-Graphen durch das Modul IML_Generator;

IL_Prepare Dieses Modul führt kleine Änderungen am IL-Graphen durch. Die Änderungen werden in Kapitel 4.2.3.3 erklärt.

IML_Generator Dieses Modul übersetzt den vorbereiteten IL-Graphen nach IML. Dazu benutzt es die C++ Schnittstelle der automatisch erzeugten IML-Klassen. Der Entwurf dieses Moduls ist in Kapitel 4.2.3.4 enthalten.

IML_Comfort ist ein Hilfsmodul für das `IML_Generator` Modul. Es dient der Vereinfachung des Zugriffs auf die IML-Klassen.

IL_Comfort ist ein Hilfsmodul für das `IML_Generator` Modul. Es dient der Vereinfachung des Zugriffs auf den IL-Graphen.

Wie diese Module auf Quelltextdateien verteilt sind, ist in Kapitel 4.3.3 zu finden.

4.2.3.2 IL-Graph für die ganze Übersetzungseinheit

T.K.

Das EDG-Front-End sieht für die Weitergabe des IL-Graphen an das Back-End, im Falle von cafe++ der IML-Übersetzer, mehrere Vorgehensweisen vor. Zum einen besteht die Möglichkeit, den IL-Graphen in eine Datei zwischenzuspeichern, welche vom Back-End dann eingelesen werden kann. Zum anderen kann der IL-Graph auch direkt im Speicher an das Back-End übergeben werden. Für die Implementierung des IML-Übersetzers wurde letztere Möglichkeit vorgezogen, da sie schneller ist, eine schlankere Implementierung ermöglicht und außer einer aufwändigeren Übersetzungsprozedur für die cafe++ Implementierung keine weiteren Nachteile hat.

Eine weitere Konfigurationseinstellung des Front-Ends, die die Übergabe des IL-Graphen an das Back-End regelt, entscheidet, ob der IL-Graph als

Ganzes oder inkrementell übergeben werden soll. Im Falle einer inkrementellen Übersetzung wird der IL-Graph Funktion für Funktion an das Back-End überreicht, was während der Übersetzung Speicher spart, den Übersetzungsvorgang jedoch auch komplizierter macht. Als zweite Option kann auch der gesamte IL-Graph auf einmal übergeben werden, was die Verarbeitung des IL-Graphen vereinfacht.

Für die cafe++ Implementierung wurde die zweite Option bevorzugt, eine spätere Erweiterung könnte jedoch auch die erste Option nutzen, um den Speicherbedarf des Übersetzers zu verringern. Dann muss jedoch eine Möglichkeit gefunden werden, die IL-Vorbereitung zu vermeiden oder so zu modifizieren, dass keine Zeiger eingefügt werden, die von einem Verarbeitungsteil in einen anderen zeigen.

4.2.3.3 IL-Vorbereitung

S.S.

„IL-Vorbereitung" ist der Name einer Phase bei der Übersetzung von C++ nach IML. Diese Phase wird in dem Modul zu IL-Vorbereitung, `IL_Prepare`, implementiert. Die IL-Vorbereitung wird vor dem eigentlichen Übersetzungsvorgang in die IML durchgeführt. Während des Übersetzungsvorgangs wird die IL nicht mehr verändert.

Bei der IL-Vorbereitung werden bestimmte Zeiger in der IL gesetzt, die vorher den Wert Null hatten. Diese Zeiger sind `a_routine_type_supplement::assoc_routine` und `a_source_correspondence::source_sequence_entry`.

`a_routine_type_supplement::assoc_routine` ist in der unveränderten IL nicht gesetzt für Typen von Funktionen, zu denen kein Funktionsrumpf angegeben wurde. Ein entsprechender Knoten existiert aber in der IL. Die in diesem Knoten enthaltene Information wird beim Generieren von TC_Routine-Knoten (und allen Knoten aus Unterklassen von TC_Routine) benötigt. Die Information wird zum Setzen der Attribute Artificial, Is_Inline, Is_Template, Is_Virtual und Its_Class von TC_Routine-Knoten, beim Erzeugen der Parameterdeklarationen und zur Bestimmung der genauen TC_Routine-Unterklasse des zu erzeugenden Knoten benötigt.

Bei Typen von Funktionszeigern existiert kein entsprechender Knoten, so dass dort `a_routine_type_supplement::assoc_routine` nicht durch die IL-Vorbereitung gesetzt werden kann. Die benötigte Information muss dann allein aus der Tatsache, dass es sich um den Typ eines Funktionszeigers handelt, geschlossen werden.

Dieser Zeiger ist in der unveränderten IL manchmal nicht gesetzt, weil die Beschreibung des Typs von mehreren Knoten für Funktionen geteilt werden kann. Bei der IL-Vorbereitung wird eine beliebige Beschreibung einer Funktionsdeklaration mit dem Typ verknüpft. Die Informationen, die aus `assoc_routine` gewonnen werden, sind für Funktionsdeklarationen mit gleichem Typ gleich. Darum ist die Wahl eines beliebigen Repräsentanten möglich.

`a_source_correspondence::source_sequence_entry` ist in der unveränderten IL nicht gesetzt für Deklarations-Knoten von Funktionen oder Variablen, die zum Sichtbarkeitsbereich der ganzen Übersetzungseinheit gehören, die jedoch nur innerhalb von Funktionen deklariert wurden (z.B. „`extern`"-Deklarationen). Diese Information wird in der Methode `IML_Generator::get_ST_Key` benötigt.

Dieser Zeiger ist in der unveränderten IL manchmal nicht gesetzt, weil er in IL-Speicherbereiche von Funktionen zeigt. Wenn der IL-Graph, um Speicher zu sparen, Funktion für Funktion an das Back-End übergeben wird, sind solche Zeiger nicht erlaubt. In diesem Fall müsste eine alternative Implementierung von `IML_Generator::get_ST_Key` gefunden werden, die ohne diese Art der IL-Vorbereitung auskommt.

Für die IL-Vorbereitung wird das Besuchermuster für die IL verwendet (siehe Kapitel 3.2.1.2). Da keine Information vom Besuch eines Knotens zum Besuch eines anderen Knotens weitergegeben werden muss, ist die Anwendung des Besuchermusters besonders einfach. Die Änderung des IL-Graphen beeinflussen den Traversierungsalgorithmus nicht negativ. Es wurden schon vor dem Einfügen der neuen Kanten alle Knoten erreicht. Ein mehrfaches Besuchen von Knoten wird durch Markierungen verhindert.

4.2.3.4 Generieren des IML-Graphen

S.S.

Ist der IL-Graph für eine Übersetzungseinheit generiert, kann dieser mit dem IML-Übersetzer (Klasse `IML_Generator`) in den entsprechenden IML-Graphen übersetzt werden. Dieser Übersetzungsvorgang läuft im `IML_Generator` nach einem festgelegten, einfachen Verfahren ab und ist wie folgt organisiert:

- Generell wird der IL-Graph direkt traversiert und Schritt für Schritt in den IML-Graphen übersetzt. Der IML-Graph entsteht dabei parallel zum IL-Graph, so dass der IL-Graph durch die Übersetzung nicht verändert wird. Dieses Vorgehen ist möglich, da beide Datenstrukturen, der IL-Graph und der IML-Graph, sich in ihrem Aufbau am Quelltext orientieren. Zusätzlich enthält der IL-Graph sehr viele Kanten, die für das Traversieren genutzt werden können.

- Das Traversieren des IL-Graphen wird anhand der Struktur des IML-Graphen durchgeführt. Jeder IML-Knoten wird währenddessen durch eine Methode der Klasse `IML_Generator` erzeugt. Alle diese Methoden haben die Signatur

  ```
  generate_<Name des IML-Knoten>(<IL-Teilgraph>);
  ```

  d.h. jede dieser Methoden trägt den Namen des zugehörigen IML-Knotens als Postfix und gibt eine neue Instanz eines IML-Knotens zurück.

Welcher IML-Knoten von einer Methode erzeugt werden soll, wird anhand des übergebenen Parameters festgelegt. Z.B. bekommt die Methode `generate_OC_Local()` den IL-Teilgraph einer lokalen Variablendefinition übergeben, dessen IML-Darstellung sie dann zurückgibt. Die IL-Teilgraphen werden durch IL-Knoten repräsentiert, von denen aus alle relevanten Knoten durch Traversieren von Kanten erreichbar sind.

Die Methoden `generate_XXX` können in beliebiger Reihenfolge aufgerufen werden. Methoden, die HPG-Knoten (HPGNode) generieren, geben immer eine neue, vollständig initialisierte Instanz eines IML-Knotens zurück. Sie dürfen deshalb nicht mehrmals für den gleichen IL-Teilgraph aufgerufen werden. Die Vermeidung mehrfacher Aufrufe ist möglich, da der HPG eine Baumstruktur hat.

Methoden, die Knoten vom Typ SymNode generieren, geben bei mehrfachem Aufruf mit dem gleichen IL-Teilgraphen immer die gleiche Instanz eines IML-Knotens zurück. Diese Instanz ist nicht vollständig initialisiert, falls die Methode sich während der Initialisierung der Instanz indirekt rekursiv aufruft. Der rekursive Aufruf ist erlaubt. Deshalb kann bei SymNodes, die durch eine `generate_XXX`-Methode erzeugt wurden, nicht davon ausgegangen werden, dass alle Attribute schon korrekt gesetzt sind. Falls ein Attribut eines SymNodes ausgelesen wird, muss durch Inspektion sichergestellt werden, dass es schon gesetzt wurde, dass also der rekursive Aufruf im Fall dieses Knotens nicht geschieht.

Intern in den Methoden zur Generierung von SymNodes ist der Ablauf immer in drei Teile unterteilt. Abbildung 4.2 auf der nächsten Seite veranschaulicht das interne Vorgehen als Flussdiagramm.

1. *Prüfung auf wiederholten Aufruf*
   Die Prüfung erfolgt mit Hilfe einer Symboltabelle (abgekürzt als S.T. in der Abbildung), die IL-Knoten auf IML-Knoten abbildet. Details zum Entwurf dieser Symboltabelle können in Kapitel 4.2.3.5 nachgelesen werden. Direkt nach dem Anlegen des IML-Knotens wird der Knoten in der Symboltabelle registriert. Die Registrierung erfolgt mit dem IL-Teilgraphen, zu dem der Knoten gehört, als Registrierungs-Schlüssel.

2. *Erzeugen des IML-Knotens*
   Wenn die Prüfung auf wiederholten Aufruf ergab, dass der gewünschte IML-Knoten noch nicht erzeugt wurde, wird er erzeugt. Dazu wird ein Konstruktor aus der C++ Schnittstelle zu den IML-Klassen aufgerufen. Beim Aufruf des Konstruktors müssen Werte für einige Attribute des zu erzeugenden IML-Knoten angegeben werden. Soweit möglich, werden diese Werte vor dem Aufruf des Konstruktors generiert (siehe Teil 3). Für Werte, die angegeben werden müssen, aber nicht vor dem Aufruf des Konstruktors generiert werden können, werden Null-Werte angegeben. Diese Werte werden später überschrieben.

3. *Attributwerte generieren*
   Attributwerte können einfache Werte, z.B. Zahlen oder Verweise auf andere IML-Knoten sein. Sie werden oft durch Aufruf einer weiteren `generate_XXX`-Methode generiert. Um Endlosrekursionen zu vermeiden,

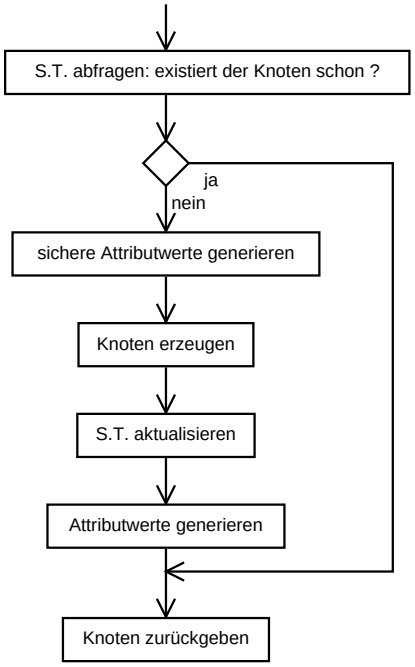


Abbildung 4.2: Generierung der IML-Knoten

werden `generate_XXX`-Methoden nie vor dem Erzeugen des Knotens und der Registrierung bei der Symboltabelle aufgerufen. Eine Ausnahme bilden `generate_XXX_S`-Methoden. Der Suffix „`_S`" gibt an, dass die Methode sicher („secure") ist. Das heißt, sie ruft selbst keine `generate_XXX`-Methode auf.

Werte, die mit Hilfe von `generate_XXX_S`-Methoden oder direkt aus der IL generiert werden können, werden schon vor dem Erzeugen des IML-Knotens generiert („sichere Attributwerte generieren" in der Abbildung). Sie können dann bei der Erzeugung als Konstruktorparameter gesetzt werden. Werte, die durch normale `generate_XXX`-Methoden generiert werden, werden nachträglich gesetzt.

Die Strukturierung und die Konventionen für die Methoden zur Generierung des IML-Graphen machen die Gliederung des Quelltextes des Übersetzers sehr einfach. Zu jedem IML-Knoten existiert genau eine Methode in der Klasse `IML_Generator`, die die gesamte Verarbeitung dafür implementiert. Diese Methode kann überladen sein. Sollte sich die Spezifikation der IML ändern, indem z.B. ein IML-Knoten ein weiteres Attribut bekommt oder ein gänzlich neuer IML-Knoten eingeführt wird, ist aus der Spezifikation klar zu ersehen, welche Teile des Quelltextes vom `IML_Generator` zu ändern sind. Die genaue Festlegung des Ablaufs innerhalb der Methoden soll weiterhin helfen, Fehler in der Implementierung zu vermeiden. In Abbildung 4.3 wird die `IML_Generator`-Klasse veranschaulicht.

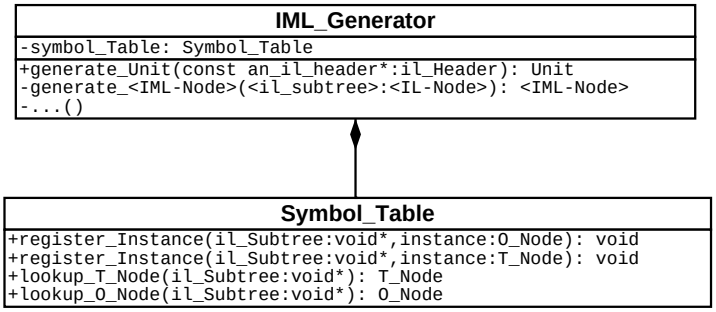


Abbildung 4.3: Klassen für die IML-Übersetzung

Entwurfsentscheidungen Am Anfang dieses Kapitel wurde das Ergebnis eines Teils des Entwurfs beschrieben. An dieser Stelle werden jetzt Alternativen dazu gezeigt. Die Gründe, warum gegen diese Alternativen entschieden wurde, werden auch angegeben.

(1) Andere Aufteilung der Aufgabe Die Aufgabe des Übersetzens der IL in die IML hätte anders aufgeteilt werden können: In Methoden (oder Funktionen) zum Abarbeiten von IL-Knoten anstatt in Methoden zum Generieren von IML-Knoten. Dabei hätte das Besuchermuster für den IL-Graphen benutzt werden können (siehe Kapitel 3.2.1.2).

Nachteil 1 Die Kopplung ist größer, weil viel aufgesammelte Information weitergereicht werden muss, bis letztendlich in einer der Funktionen ein fertiger IML-Knoten entsteht. Wenn das Besuchermuster verwendet wird, muss diese Information sogar an einer auch für andere Besucher zugänglichen Stelle abgelegt werden. Die Information kann nicht als Parameter von einem Besucher an einen bestimmten anderen weitergegeben werden.

Nachteil 2 Die Gefahr, dass Teile der IML nicht generiert werden, ist größer. Beispielsweise kann das Setzen einiger Attribute vergessen werden.

(2) Erzeugen völlig uninitialisierter IML-Knoten Anstatt möglichst viele Attribute eines IML-Knotens schon vor seiner Erzeugung zu bestimmen, hätte der Knoten völlig „leer" erzeugt werden können. Die Erzeugung von leeren IML-Knoten kann durch Aufruf des Konstruktors mit Nullwerten geschehen. Um die Angabe der Nullwerte zu vermeiden, könnte vom Generator der IML-Klassen für jede Klasse ein Konstruktor ohne Parameter erzeugt werden.

Nachteil 1 Die Gefahr, dass Attribute nicht gesetzt werden, ist größer. Wenn Attribute schon bei der Erzeugung initialisiert werden müssen, kann der Übersetzer das prüfen. Das ist insbesondere beim Hinzufügen neuer Attribute in der IML-Spezifikation wichtig.

Nachteil 2 Nicht gesetzte Attribute können auch nicht ausgelesen werden. Attribute, die schon beim Erzeugen des IML-Knotens richtig gesetzt werden, können auch bei SymNodes gefahrlos ausgelesen werden. Das oben beschriebene Problem mit den rekursiven Aufrufen wirkt sich bei diesen Attributen nicht aus.

Attribute mit Mengen- oder Listentyp können in der automatisch generierten C++ Schnittstelle der IML-Klassen nicht als Konstruktorparameter angegeben werden. Diese Attribute sind also beim Erzeugen zwangsweise uninitialisiert.

(3) „set_XXX_Attributes"-Methoden Attribute von IML-Knoten könnten von Methoden mit Name „set_XXX_Attributes" gesetzt werden, anstatt sie direkt in generate_XXX zu setzen. Dabei steht „XXX" wieder für den Namen eines IML-Knotentyps. Die Attribute von Basisklassen („BBB") würden von der set_BBB_Attributes-Methode der Basisklasse gesetzt werden. Dadurch reicht es beim Implementieren von set_XXX_Attributes aus, nur die Spezifikation der IML-Klasse XXX zu kennen. Die korrekte Initialisierung aller Attribute der direkten und aller indirekten Basisklassen wäre durch Aufruf von set_BBB_Attributes sichergestellt.

Nachteil Wenn *alle* Attribute so gesetzt werden, heißt das, dass völlig uninitialisierte IML-Knoten erzeugt werden (siehe oben).

Bei Attributen mit Mengen- oder Listentyp, die nicht beim Erzeugen initialisiert werden *können*, wurden set_XXX_Attributes-Methoden eingesetzt, falls es sinnvoll war.

(4) Keine Attribute auslesen Anstatt ein Attribut eines IML-Knoten beim Generieren des Knoten zu setzen und nach der Rückkehr der generierenden Methode das Attribut auszulesen, könnte der Wert des Attributs an beiden Stellen direkt aus der IL berechnet werden. Zum Berechnen des Wertes aus der IL sollte eine Hilfsfunktion benutzt werden, damit bei einer Änderung der Berechnungsvorschrift nur die Hilfsfunktion betrachtet werden muss.

Wenn nie Attribute ausgelesen werden, besteht keine Gefahr, dass ein Attribut ausgelesen wird, das erst später initialisiert wird.

Nachteil Den Wert erneut zu berechnen ist möglicherweise weniger effizient als den Wert auszulesen.

Falls der Nachteil mit der Effizienz nicht ausschlaggebend ist, kann diese Alternative später umgesetzt werden. Attribute werden momentan nur an sehr wenigen Stellen ausgelesen. Diese Stellen sind leicht zu lokalisieren, da die Funktionen der IML-C++ Schnittstelle zum Auslesen von Attributen einheitlich benannt sind.

4.2.3.5 Behandlung der IML-Symboltabelle

T.K.

Der im vorherigen Kapitel beschriebene Ablauf für die Generierung des IML-Graphen enthält die Problemstellung, dass die Methoden erkennen müssen,

ob sie bereits die IML-Repräsentation für einen IL-Teilbaum erzeugt haben oder nicht. Wie bereits erwähnt, tritt dieser Fall nur für die Symboltabelle der IML auf, also für alle IML-Knoten, die von der Klasse SymNode abgeleitet sind. Um diese Problemstellung zu behandeln sind zwei Lösungen denkbar:

1. Alle IL-Knoten haben einen sog. Entry-Präfix Eintrag [6, Kapitel 5.29], der erweitert werden kann. Eine einfache Lösung des Problems würde darin bestehen, diesen Entry-Präfix Eintrag um einen Zeiger auf den dazugehörigen IML-Knoten zu erweitern. Dieser Zeiger würde anfangs eine Markierung enthalten, die anzeigt, dass der IL-Knoten noch nicht übersetzt wurde. Nach der Übersetzung des IL-Knotens würde der eingefügte Zeiger einfach auf den IML-Knoten zeigen, dessen Repräsentation er darstellt.

   Vorteilhaft an dieser Vorgehensweise ist die einfache Behandlung und die schnelle Verarbeitung zur Laufzeit. Nachteilig ist, dass der Quelltext des Front-Ends geändert werden muss.

   Weiterhin entsteht ein Konflikt, wenn aus einem IL-Knoten zwei IML-Knoten erzeugt werden müssen. Dies ist z.B. der Fall bei einer Funktionsdeklaration, bei der ein IL-Knoten in zwei IML-Knoten übersetzt wird, nämlich ein OC_Routine-Knoten und ein TC_Routine-Knoten. In einer solchen Situation ist nicht mehr klar, auf welchen der beiden IML-Knoten der IL-Konten verweist. Die Erweiterung des Entry-Präfix Eintrags um zwei Zeiger, einen vom Typ OC_Node und einen vom Typ TC_Node, kann diese Zweideutigkeit jedoch einfach auflösen.

2. Die zweite Möglichkeit, das Problem anzugehen, besteht darin, eine eigenständige Abbildungsmöglichkeit von IL- auf IML-Knoten zu schaffen. Diese kann beispielsweise mittels einer Hash-Tabelle realisiert werden. Der Konflikt, wie er bei der ersten Lösung entsteht, kann hier umgangen werden, indem die Abbildung durch IML-Knotentypen verfeinert wird. D.h. es muss eine eindeutige Abbildung geschaffen werden von einem IL-Knoten und einem IML-Knotentyp zu einer IML-Knoteninstanz. Da diese Abbildung unabhängig von den IL-Datenstrukturen gespeichert werden kann, ist auch keine Änderung des Front-End-Quelltextes notwendig. Nachteilig wirkt sich jedoch die größere Implementierung und die geringere Geschwindigkeit des Übersetzungsvorgangs aus.

   Die Abbildung von IL- auf IML-Knoten könnte anstatt mit einer Hash-Tabelle auch mit einem einfachen Array realisiert werden. Das Array kann mit einer eindeutigen Nummer aus dem Entry-Präfix Eintrag indiziert werden. Diese eindeutige Nummer ist schon im Entry-Präfix Eintrag enthalten, der Quelltext des Front-Ends muss also nicht geändert werden. Der Speicherbedarf des Arrays ist aber vermutlich höher als der einer Hash-Tabelle oder eines Suchbaums, weil nur zu wenigen IL-Knoten ein Symboltabelleneintrag benötigt wird.

Für die Realisierung in dieser Diplomarbeit wurde die zweite Lösung ausgewählt. Umgesetzt wird die Abbildung von IL- nach IML-Knoten in der Klasse

`Symbol_Table`. Die Schnittstelle der Klasse wurde dabei aber so ausgelegt, dass die Implementierung der `Symbol_Table`-Klasse auch problemlos die erste Lösung realisieren kann, ohne dass der übrige Quelltext geändert werden müsste. Abbildung 4.3 veranschaulicht die Klasse als UML-Diagramm.

Symboltabellen mit IL-Knoten als Schlüssel werden auch an anderen Stellen eingesetzt, an denen IL-Knoten annotiert werden müssen. Durch diese „scaffolding" genannte Technik konnte auch in diesen Fällen eine Modifikation des EDG-Quelltextes vermieden werden.

4.2.4 Erweiterte Namen

S.S.

Das Erzeugen von erweiterten Namen wird vom EDG-Front-End nicht direkt unterstützt. Das ebenfalls mit dem Front-End mitgelieferte sog. „IL-lowering" enthält jedoch eine entsprechende Funktionalität, die genutzt werden kann (siehe Kapitel 3.2.1.8). Mit Hilfe des IL-lowering kann der IL-Graph eines C++ Programmes umgeformt werden in einen IL-Graphen, der nur noch C Knoten enthält. Diese Funktion ist gedacht, um es Übersetzerherstellern zu erleichtern, schnell C++ Übersetzer herzustellen.

Da für die IML-Darstellung von C++ Programmen erweiterte Namen notwendig sind (siehe Kapitel 2.3.2), sollte diese bereits vorhandene Funktionalität hierfür genutzt werden. Die komplette IL-lowering Phase kann jedoch nicht benutzt werden, da dadurch wichtige Informationen verloren gehen würden.

Das Erzeugen der erweiterten Namen während des IL-lowerings geschieht durch Aufruf der Funktionen `do_all_name_mangling()` und `do_final_name_mangling()`. Die beiden Funktionen werden in `fe_wrapup.c` aufgerufen, aber leider nur, wenn die Einstellung `DO_IL_LOWERING = TRUE` ist. Wenn nur `NEED_NAME_MANGLING = TRUE` ist, wird keine Namenserweiterung durchgeführt.

Es ist nicht möglich, die beiden Funktionen zur Erzeugung erweiterter Namen innerhalb des Back-Ends aufzurufen, denn bei C++ Quelltext mit Templates geschieht sonst ein unerlaubter Speicherzugriff (`SIGSEGV`). Auf eine Support-Anfrage bei EDG hin wurde dazu geraten, den Quelltext in `fe_wrapup.c` so zu modifizieren, dass die beiden Funktionen dort aufgerufen werden.

Änderungen am Quelltext von EDG sollten möglichst vermieden werden, um neue Versionen des Front-Ends einfacher integrieren zu können. Trotzdem wurde der Rat des EDG-Supports befolgt und der Quelltext in `fe_wrapup.c` geändert. Die Änderung ist so klein, dass sie an einer neuen Version des Front-Ends relativ einfach durchzuführen sein sollte. Sie ist in Kapitel 4.3.4.2 dokumentiert.

4.3 Realisierung

S.S.

In diesem Kapitel werden verschiedene Aspekte der Implementierungsphase behandelt. Diese Phase hat den Entwurf als Grundlage. Beschrieben wird das Vorgehen bei der Realisierung sowie einzelne Aspekte der Realisierung selbst, bei denen die Umsetzung des Entwurfs nicht trivial war.

4.3.1 Qualität des Quelltextes

S.S.

Um eine hohe Qualität des Quelltextes zu erreichen, wurden folgende Maßnahmen ergriffen:

- Ein Styleguide für C++ Quelltext wurde eingesetzt. Dazu wurde der Styleguide der Abteilung Programmiersprachen und Übersetzerbau für Ada Quelltexte übernommen und für C++ Quelltexte adaptiert.
- Fragen, die nicht mit Hilfe des Styleguides beantwortet werden konnten, wurden durch Absprache zwischen den beiden Autoren geklärt. Dadurch wurden Inkonsistenzen zwischen Quelltextteilen der beiden Autoren vermieden.
- Um Fehler aufzudecken, wurde getestet (siehe Kapitel 5). Fehler, die durch den Test gefunden wurden, wurden dokumentiert. Sie wurden behoben, soweit das in der begrenzten Zeit möglich war.
- Für die Implementierung wurde (fast) ausreichend viel Zeit eingeplant.
- Bei der Erstellung des Entwurfs wurde Wert auf eine hohe Qualität gelegt.

Diese Maßnahmen waren erfolgreich. Die Qualität des Quelltextes lässt sich daran erkennen, dass er wenig Fehler enthält und gut wartbar ist. Die Ergebnisse des Tests geben Grund zur Annahme, dass beides der Fall ist. Es wurden relativ wenig Fehler gefunden und die Behebung von Fehlern war in den meisten Fällen einfach. „Einfach" heißt hier, dass zur Behebung eines Fehlers nur ein kleiner Bereich des Quelltextes betrachtet werden musste und dass dieser Bereich schnell zu lokalisieren war.

4.3.2 Buildprozess

S.S.

Der Buildprozess enthält alle Schritte, die notwendig sind, um den Quelltext von cafe++ in ein ausführbares Programm zu übersetzen. Das sind im einzelnen:

1. Anpassen des EDG-Quelltextes (siehe Kapitel 4.3.4.2).
2. Erzeugen von EDG-Fehlermeldungsinformationen und Übersetzen der EDG-Front-End-Quelltexte mit `gcc`.
3. Übersetzen der IML-Bibliothek mit `gnatmake` und Binden der einzelnen Übersetzungseinheiten mit `gnatbind`. Dieser Schritt entfällt, falls eine bereits übersetzte IML-Bibliothek verwendet wird.
4. Übersetzen der cafe++ Quelltexte mit `g++`.
5. Zusammenlinken der einzelnen Übersetzungseinheiten mit `gnatlink`.

Der Buildprozess wird durch das Werkzeug `make` automatisiert. Alle oben beschriebenen Schritte werden in einem einzigen Lauf von `make` durchgeführt. Es gibt also nur ein `Makefile`. Die Abhängigkeiten zwischen den Schnittstellen (Header mit Endung `.h`) und Übersetzungseinheiten werden automatisch bestimmt. Das gilt auch für die Übersetzungseinheiten des EDG-Front-Ends.

Bei der Kombination von Ada und C++ Quelltext waren die Erfahrungen aus dem jafe-Project [19, 20] von Markus Knauß sehr hilfreich. Bei jafe wird jedoch `gnatgcc` direkt aus dem Makefile heraus aufgerufen, während bei cafe++ zum Übersetzen der Ada Übersetzungseinheiten `gnatmake` verwendet wird. Die Verwendung von `gnatmake` vereinfacht den Buildprozess, da `gnatmake` Abhängigkeiten zwischen Ada Übersetzungseinheiten selbstständig bestimmt.

4.3.3 Organisation der Dateien

S.S.

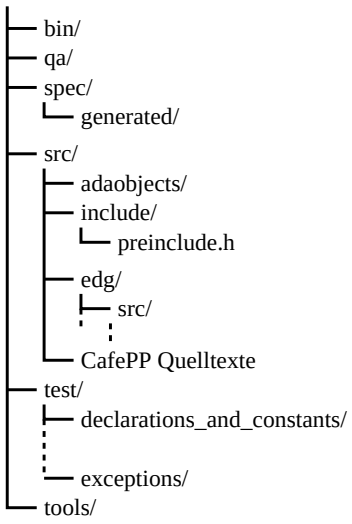


Abbildung 4.4: Verzeichnisstruktur von cafe++

Die während dieser Diplomarbeit erstellten Dokumente werden vom CVS-Versionsverwaltungssystem der Abteilung Programmiersprachen und Übersetzerbau verwaltet. Sie sind für alle Mitglieder der Abteilung zugänglich. Aus rechtlichen Gründen konnte der EDG-Quelltext nicht für alle Mitglieder der Abteilung zugänglich in das Versionsverwaltungssystem übernommen werden – der Quelltext darf nur Personen zugänglich sein, die eine Erklärung bezüglich des Copyrights von EDG unterschrieben haben. Um eine ausführbare Version von cafe++ zu erzeugen, muss der EDG-Quelltext in dem Verzeichnis `edg` des cafe++ Verzeichnisbaums verfügbar sein. Ein symbolischer Verweis (Unix: symbolic link) reicht aus.

Die Schnittstelle von Klassen ist in je einer Datei mit dem Namen der Klasse und der Endung „`.h`" enthalten. Die Implementierung ist in je einer Datei mit gleichem Namen, aber der Endung „`.cpp`" enthalten. Auch bei den Hilfsmodulen, die nicht durch eine Klasse repräsentiert werden, wird der Name

des Moduls für die Dateinamen verwendet. Alle Quelltextdateien des Moduls `CafePP` befinden sich im Verzeichnis `src`. Alle Quelltextdateinamen sind in Kleinbuchstaben geschrieben.

Das Hauptprogramm von cafe++ ist in der Datei `cafe++.cpp` enthalten. Diese Datei ist eine kopierte und modifizierte Version der Datei `cfe.c` von EDG (siehe Kapitel 3.2.2) und gehört zum Modul `Application` (siehe Kapitel 4.2.3).

Die für den Buildprozess (Kapitel 4.3.2) und die Bestimmung der Testüberdeckung nötigen Dateien befinden sich ebenfalls im Verzeichnis `src`. Ein Unterverzeichnis `src/adaobjects` wird vom Buildprozess angelegt, wenn die IML-Klassen durch den Buildprozess übersetzt werden. Es nimmt verschiedene, bei der Übersetzung der IML-Klassen erzeugte Dateien auf.

Die Datei `preinclude.h` wird vom Test im Unterverzeichnis `src/include` erwartet. Sie wird für die Übersetzung von Quelltext benötigt, der Funktionen mit Ellipsen definiert oder Arrays enthält, die durch Destruktoraufrufe aufgeräumt werden müssen.

4.3.4 Änderungen/Konfiguration des EDG-Front-Ends

S.S.

Die für die korrekte Funktion von cafe++ relevanten Konfigurationsoptionen sind in der Datei `defines.h` gesetzt. Dort ist auch kommentiert, *warum* die jeweilige Option so gesetzt wurde. Die Datei `defines.h` ist nicht im EDG-Quellverzeichnis, sondern im cafe++-Quellverzeichnis enthalten. Sie wird über den Include-Pfad vom Übersetzer gefunden. Systemabhängige Konfigurationsoptionen werden in der Datei `system_dependent_defines.h`, die von `defines.h` über `#include` eingebunden wird, gesetzt. Diese Datei muß vor dem Übersetzen von cafe++ für jede Plattform neu erstellt werden. Die Einstellungen aus `defines.h.linux` bzw. `defines.h.solaris`, die von EDG mitgeliefert werden, sind für Linux bzw. Solaris ausreichend.

Allgemeine Informationen über die Konfiguration des Front-Ends sind im Kapitel 3.2.3 enthalten. In Kapitel 4.3.4.1 wird die spezielle Konfiguration des EDG-Front-Ends für cafe++ beschrieben. In Kapitel 4.3.4.2 geht es um Änderungen des EDG-Quelltextes, die zusätzlich zur Konfiguration notwendig waren.

4.3.4.1 Konfiguration durch Definition von Makros

S.S.

Das EDG-Front-End wird durch Definition von Präprozessormakros konfiguriert. Dazu wird die Konfigurationsdatei `defines.h` von den EDG-Quelltexten mit Hilfe des C Präprozessors eingebunden. Sie muss von den Benutzern des Front-Ends erstellt werden. Für cafe++ wurde diese Datei bereits von den Autoren dieser Diplomarbeit erstellt. Sie enthält die für cafe++ nötigen, aber von der Plattform unabhängigen Einstellungen.

Folgende Einstellungen werden in `defines.h` vorgenommen:

- Einstellungen aus `system_dependent_defines.h` werden übernommen.

- Das Front-End wird so eingestellt, dass es mit einem ANSI-C kompatiblen Übersetzer übersetzt werden kann.
- Für Einstellungen, die an der Kommandozeile verändert werden können, werden für cafe++ keine speziellen Standardwerte durch Makrodefinitionen eingestellt. Die Standardwert-Einstellungen von EDG werden übernommen.
- Erweiterungen der IL für Fortran und für C und C++ Dialekte werden deaktiviert. Die in dieser Diplomarbeit entwickelte Version von cafe++ unterstützt also nur C++ Quelltext nach ISO/IEC 14882:1998 [14]. Falls später mehr unterstützt werden soll, muss die Unterstützung hier aktiviert und im Modul `CafePP` implementiert werden.

  Die Unterstützung neuer, von anderen Übersetzern nicht immer unterstützter Merkmale von C++ nach ISO/IEC 14882:1998 wird aktiviert.
- Beispiel Back-Ends von EDG werden deaktiviert.
- Das Front-End wird so eingestellt, dass die IL komplett im Speicher an das Back-End übergeben wird. Die IL wird nicht in einer Datei zwischengespeichert oder Funktion für Funktion abgearbeitet.
- Das Front-End wird so eingestellt, dass möglichst viel Information über den Quelltext in der IL erhalten bleibt. Für Übersetzer, die Maschinensprache erzeugen, ist diese Information nicht nötig. cafe++ braucht diese Information, um Quelltextnähe in der erzeugten IML zu erreichen.

  Dazu gehört auch, dass die IL-lowering Phase deaktiviert wird.

  Das Front-End wird so eingestellt, dass nicht instantiierte Templates als IL-Teilgraphen dargestellt werden. Diese Teilgraphen werden bei EDG „prototype instantiations" genannt. Das Front-End wird *nicht* so eingestellt, dass Templates nur in instantiierter Form dargestellt werden oder dass die Darstellung von nicht-instantiierten Templates als Zeichenkette erfolgt.

4.3.4.2 Änderungen am EDG-Quelltext

S.S.

In diesem Kapitel wird beschrieben, welche Änderungen am EDG-Quelltext vorgenommen wurden, so dass sie an neueren Versionen des EDG-Quelltextes wieder durchgeführt werden können, falls das nötig sein sollte. Die Gründe für die Änderung werden in Kapitel 4.2.4 beschrieben.

In der Datei `fe_wrapup.c` sind zwei Stellen der Form

```
#if DO_IL_LOWERING
...
    if (il_lowering_needed()) {
    ...
        do_XXX_name_mangling();
...
#endif
```

enthalten. An der einen Stelle im Quelltext ist XXX = „all", an der anderen ist XXX = „final". Beide Stellen wurden komplett auskommentiert und ersetzt durch:

```
if (total_errors == 0) {
    do_XXX_name_mangling();
} /* if */
```

Der Teil „`total_errors == 0`" ist, zusammen mit einigen weiteren Bedingungen, in der Funktion `il_lowering_needed()` enthalten. Wie oben zu sehen ist, wurde der Aufruf dieser Funktion auskommentiert und durch die Bedingung „`total_errors == 0`" ersetzt. Alle anderen Bedingungen für `il_lowering_needed()` sind nur bei Anwendung der vollständigen IL-lowering Phase sinnvoll.

Der EDG-Quelltext ist aus rechtlichen Gründen nicht an der gleichen Stelle abgelegt, wie der Quelltext von cafe++ (siehe Kapitel 4.3.3). Deshalb wird er nur in unveränderter Form zur Verfügung gestellt. Die notwendige Änderung wird jedoch automatisch vom Buildprozess durchgeführt (siehe Kapitel 4.3.2). Dazu wird das Werkzeug „`patch`" benutzt. Die automatische Änderung durch das Werkzeug `patch` funktioniert möglicherweise nur bei der in dieser Diplomarbeit eingesetzten Version des EDG-Quelltextes.

4.3.5 `switch`-Anweisung

S.S.

Die `switch`-Anweisung in C und C++ besteht aus einem Kopf „`switch(` *Ausdruck*`)`" und einer darauf folgenden Anweisung, dem Rumpf. Der Rumpf ist üblicherweise eine zusammengesetzte Anweisung und enthält Sprungzielmarkierungen (labels) der Form „`case` *Konstante*`:`" und „`default:`". Beim Erreichen des `switch`-Kopfes wird der darin enthaltene Ausdruck ausgewertet und je nach Ergebnis zu einer der Sprungzielmarkierungen gesprungen.

Genau so wird die `switch`-Anweisung in der IML auch modelliert: Die übliche zusammengesetzte Anweisung des Rumpfes wird als Statement_Sequence, der Kopf als ein oder mehrere bedingte Sprünge modelliert, die je nach Wert des Ausdrucks an eine Stelle des Rumpfes springen.

In der IL wird die `switch`-Anweisung dagegen als eine Auswahl zwischen mehreren bedingt auszuführenden Anweisungen (Zweigen) dargestellt. Das entspricht den `case`-Anweisungen aus Sprachen wie Pascal oder Ada. Falls der Kontrollfluss vom Ende eines Zweigs zum Anfang des nächsten Zweigs weitergeht, weil nicht – wie üblich – am Ende des ersten Zweigs eine `break`-Anweisung steht, wird vom Front-End ein künstlicher unbedingter Sprung (goto) zum Anfang des nächsten Zweigs eingefügt. `break`-Anweisungen am Ende von Zweigen werden in der IL nicht explizit dargestellt.

In Abbildung 4.5 auf der nächsten Seite und Abbildung 4.6 auf Seite 137 sind die verschiedenen Modellierungen in der IML bzw. IL als UML-Diagramme dargestellt. Beide Diagramme beziehen sich auf dieses Beispiel einer `switch`-Anweisung:

```
switch (...) {
case 1:
    ...
    // fall through
case 2:
case 3:
    ...
    break;
default:
    ...
}
```

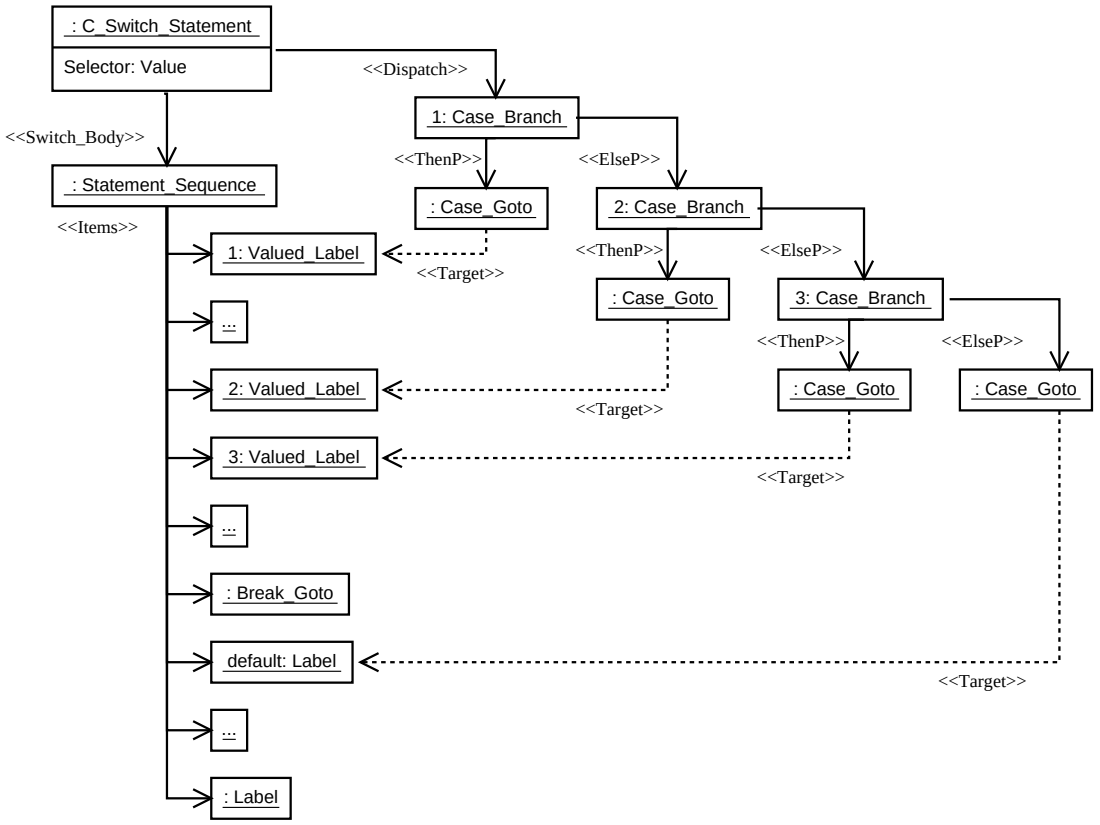


Abbildung 4.5: Darstellung der `switch`-Anweisung in der IML

Um die Darstellung einer `switch`-Anweisung in der IL in die Darstellung in der IML zu übersetzen, muss Folgendes geschehen:

- Die einzelnen Zweige der IL-Darstellung müssen zu einer zusammengesetzten Anweisung (Statement_Sequence) mit Sprungzielmarkierungen aneinandergereiht werden.
- In der IL implizit dargestellte `break`-Anweisungen müssen dabei am Ende der Zweige eingefügt werden.

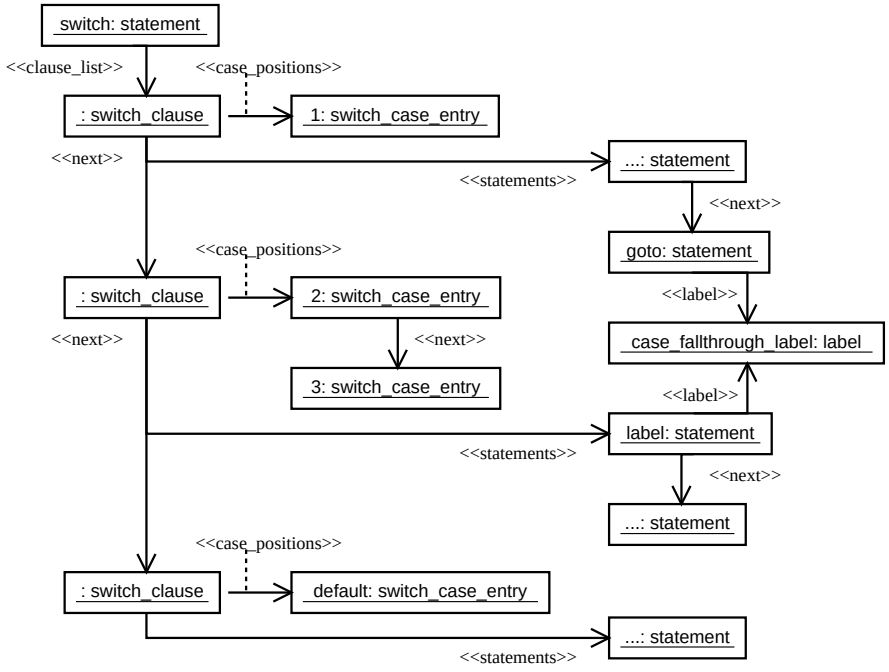


Abbildung 4.6: Darstellung der `switch`-Anweisung in der IL

- Die künstlichen unbedingten Sprunganweisungen der IL vom Ende eines Zweigs an den Anfang des nächsten dürfen nicht übernommen werden.

Falls am Anfang des Rumpfes Anweisungen stehen, die in keinem Zweig enthalten sind, verweist das Attribut „`body_statement`" der IL-Darstellung der `switch`-Anweisung auf diese Anweisungen. Dieser Fall ist äußerst selten. Ein Beispiel für diesen Fall wäre

```
switch (...) {
named_Label:
    ...
case 1:
    ...
}
```

Hier wurde ein benanntes Sprungziel `named_Label` benutzt, damit die Anweisungen vor dem ersten Sprungziel der `switch`-Anweisung überhaupt erreicht werden können. Das kann mit einem unbedingten Sprung „`goto named_Label;`" geschehen.

4.3.6 Deklarationen in Funktionsrümpfen

T.K.

Der IL-Graph sieht über die so genannten „source sequences" eine Möglichkeit vor, die IL-Knoten in der Reihenfolge zu traversieren, die der Reihenfolge im

Quelltext entspricht. Über diese Information ist es möglich, auch die genaue Position von Deklarationen innerhalb von Funktionen zu bestimmen, welche für die Darstellung in der IML wichtig ist. Leider hat diese Vorgehensweise zwei gravierende Nachteile:

1. Die „source sequence"-Informationen sind in künstlichen Funktionsrümpfen nicht vorhanden. Künstliche Funktionsrümpfe können z.B. implizite Copy-Konstruktoren und Destruktoren sein, die nicht im Quelltext deklariert werden müssen.

2. Nicht jede Deklaration ist über die „source sequence"-Information erreichbar. Wenn beispielsweise eine Klasse definiert und gleichzeitig instantiiert wird, ist die Instantiierung von den source sequences aus nicht sichtbar: „`class C {} c_Instance;`"

Abgesehen von der „source sequence" bietet das EDG-Front-End in der IL die Möglichkeit, über die so genannten „statements" auf die Anweisungen innerhalb von Funktionsrümpfen zuzugreifen. Über die statements ist es ebenfalls möglich, auf die Variablendeklarationen im Quelltext zuzugreifen. Es fehlen aber die Typdeklarationen.

Die für cafe++ implementierte Vorgehensweise sieht eine Kombination von mehreren Informationen vor, um alle Deklarationen in den Funktionsrümpfen in der IML darstellen zu können:

- Lokale Variablendeklarationen werden über die „variables"-Liste der IL ermittelt. Diese Liste steht auch für temporäre Variablen und in künstlich vom Front-End erzeugten Funktionen zur Verfügung. Nachteilig ist hier jedoch, dass die genaue Position der Deklarationen verloren geht. Deswegen kann für alle Variablendeklarationen zwar die genaue Reihenfolge angegeben werden, für jede Variablendeklaration beginnt die Lebenszeit aber gleich zu Beginn des umschließenden Blocks, auch wenn diese erst später im Block deklariert wurde. Die Initialisierung der Variablen geschieht jedoch mittels der „statements" und somit an der genauen Position, so dass die Semantik der IML-Darstellung im Endeffekt korrekt bleibt.

- Lokale Typdeklarationen werden über die „source sequences" ermittelt, da sie in künstlichen Funktionsrümpfen nicht vorkommen.

4.3.7 Initialisierung von Objekten

S.S.

Die Modellierung der Initialisierung von Objekten bereitete bei der Implementierung einige Schwierigkeiten. Der Grund dafür ist, dass Objekte in C++ nicht nur durch Zuweisung, sondern auch durch Konstruktoraufrufe initialisiert werden können.

Besonders zeitaufwändig zu implementieren waren:

- *Kopien von Objekten, die bitweise erfolgen können*
  Diese werden in der IL durch „bitwise_copy" gekennzeichnet. Im Gegensatz zu jeder anderen Art der Initialisierung ist hier in der IL die

Quelle, von der kopiert wird, nicht direkt verfügbar. Sie muss aus dem Kontext gewonnen werden.

- *Kopien von Funktionsrückgabewerten mit Hilfe eines Copy-Konstruktors*
  In der IL ist hier kein direkter Verweis auf den Copy-Konstruktor enthalten. Er muss aus den Methoden des Funktionsrückgabetyps herausgesucht werden.

- *Initialisierung von Arrays mit Konstruktoren*
  Eine künstliche `for`-Schleife muss erzeugt werden, die für jedes Element des Arrays einen Konstruktor aufruft. Ein Zeiger auf das jeweilige Element muss dem Konstruktor als `this`-Zeiger übergeben werden.

4.3.8 Zerstörung von Objekten

T.K.

Für die Zerstörung von Objekten wird die „object lifetime"-Information verwendet, die bereits in Kapitel 3.2.1.4 vorgestellt wurde. Besonders hilfreich sind dabei die „end of lifetime"-Zeiger, die in der IL bereitgestellt werden [6, Kapitel 5.18 „Object Lifetime"].

Ein Manko der „end of lifetime"-Zeiger ist, dass zwar die Zerstörungen an sich angezeigt werden, die bei bestimmten C++ Anweisungen ausgeführt werden müssen, aber teilweise die Information fehlt, welches Objekt zerstört werden muss. Aus diesem Grund wurde eine Tabelle eingeführt, welche die zu einer Zerstörung gehörenden Objekte enthält. Die Tabelle ist in der Variablen `destructible_Entity_Table` in der Klasse `IML_Generator` enthalten und wird beim Generieren den entsprechenden Variablendeklarationen gefüllt.

4.4 Bekannte Einschränkungen

S.S.

Bedingt durch den Systementwurf und die eingebundene Software, insbesondere das C++ Front-End, ergeben sich in der Realisierung von cafe++ Einschränkungen gegenüber den Anforderungen. Diese sollen in diesem Kapitel genannt werden.

- Speicher, der für die Darstellung der IML-Repräsentation der Übersetzungseinheit angefordert wird, wird nach dem Gebrauch nicht mehr freigegeben. Diese Einschränkung entsteht durch die Eigenschaft des vom Bauhaus-Generator erzeugten Quelltextes, der zwar Funktionen für das Anlegen von IML-Objekten vorsieht, Funktionen zum Löschen von Objekten jedoch fehlen. Der Entwurf von cafe++ verlässt sich in diesem Fall auf das Betriebssystem, das nach Beenden des cafe++ Übersetzers die Verantwortung übernimmt, den durch cafe++ belegten Speicher wieder freizugeben. Ist das Betriebssystem dazu in der Lage, ergeben sich keine weiteren Einschränkungen durch die fehlenden Funktionen.

- Das Attribut Storage_Class von Knoten für zweitrangige Deklarationen wird nicht gesetzt, da das EDG-Front-End diese Information nicht bereitstellt. Dadurch geht Quelltextnähe verloren. Die semantischen Aspekte

der „Storage Class" werden jedoch vollständig erfasst. Sie werden in der IML durch Listen wie Provided_Definitions, Unresolved_Declarations, Global_Lifetime_Definitions und Local_Lifetime_Definitions dargestellt.

- Die Visibility von zweitrangigen Deklarationen ist die der „gültigen" Deklaration.
- Es wird nicht zwischen reinen C `struct` Typen und C++ `struct` Typen unterschieden (siehe Kapitel 2.3.4.4).
- Instantiierungen von Template-Parametern können nicht mit vertretbarem Aufwand von normalen Typen (bzw. Konstanten) unterschieden werden. Beispiel:

```
1 template<class Param> class Beispiel {
2 public:
3     Param attr_1;
4     int attr_2;
5 };
6 int main() {
7     Beispiel<int> usage;
8 }
```

  Die Typen von „`usage.attr1`" und „`usage.attr2`" unterscheiden sich in der IL nicht. Um einen Unterschied festzustellen müsste man in der IL von der Instantiierung „`Beispiel<int>`" zum Template „`Beispiel<Param>`" gehen und dort „`attr_1`" und „`attr_2`" suchen. Allerdings tritt dieses Problem nicht nur bei Klassenattributen auf, sondern bei jeder Verwendung eines Typs oder einer Konstante im Sichtbarkeitsbereich der Template-Deklaration (also zwischen Zeile 1 und Zeile 5). Bei jeder Verwendung eines Typs oder einer Konstanten alle umschließenden Template-Sichtbarkeitsbereiche durchzugehen und im zugehörigen Template nach der Verwendung zu suchen wäre zu viel Aufwand, vermutlich mehr Aufwand, als die Kopie zur Template-Instantiierung selbst durchzuführen.

  Deshalb wurde entschieden, dass bei der Verwendung eines Templateparameters in der Instantiierung kein Templateparameter-Knoten erzeugt wird, der auf den aktuellen Parameter zeigt. Statt dessen wird der aktuelle Parameter direkt eingesetzt. Dies widerspricht der Spezifikation in Kapitel 2.3.10.4.

- In der IL können Typumwandlungen der Form „`static_cast<Typ>`", „`const_cast<Typ>`" und „`(Typ)`" im allgemeinen nicht unterschieden werden. Nur innerhalb von Templates werden die verschiedenen Typumwandlungen verschieden dargestellt. cafe++ erzeugt nur in den Fällen, in denen in der IL verschiedene Darstellungen benutzt werden, verschiedene IML-Knotentypen.

  Typumwandlungen der Form „`const_cast<Typ>`" könnten daran erkannt werden, dass nur ein `const`-Typqualifizierer entfernt wird. Wenn

im Quelltext ein `const`-Typqualifizierer durch eine herkömmliche Typumwandlung „`(Typ)`" entfernt wird, würde das jedoch ebenfalls als „`const_cast<Typ>`" erkannt werden.

- Trotz der eingestellten Front-End-Konfigurationsoption `RECORD_CONSTANT_EXPRESSIONS_IN_IL == TRUE`, die konstante Ausdrücke in der IL erhalten soll, werden manche konstante Ausdrücke von EDG optimiert. Beispiel:

```
true && false; // ok
0 && 123456;   // wird optimiert zu bool(0)
98765 && 0;    // ok: bool(98765) && bool(0)
i && j;        // ok: bool(i) && bool(j)
```

  cafe++ kann nicht mehr erzeugen als in der IL enthalten ist. Es geht also Quelltextnähe verloren, wenn vom Front-End etwas wegoptimiert wurde.

- L-Values werden in der IL durch ihre Adressen dargestellt. cafe++ versucht zu erkennen, wann vom Front-End Dereferenzierungs-Operatoren entfernt oder Adress-Operatoren künstlich eingefügt wurden. Das ist nicht immer möglich, z.B. kann „`(*s).x`" nicht von „`s->x`" unterschieden werden.

- Der in C++ eingebaute `wchar_t`-Typ wird momentan durch einen speziellen `TC_Int`-Knoten modelliert. Damit dies durch Knoten des Typs `TCPP_WChar` geschieht, muss die entsprechende Option im EDG-Front-End umgestellt werden.

4.5 Umfang und Eigenschaften

T.K. und S.S.

Zum Umfang der Realisierung wurden Messungen des Quelltextumfangs und des Zeitaufwands durchgeführt. Der Zeitaufwand für die Implementierung und für andere Teile der Diplomarbeit wird in Kapitel 7.1.1.2 dokumentiert und mit dem vorher geschätzten Zeitaufwand verglichen.

In Tabelle 4.1 auf der nächsten Seite ist der Umfang der Implementierung von cafe++ in Quelltextzeilen aufgeführt. Insgesamt benötigt die Implementierung von cafe++ mehr als eine halbe Million Zeilen C und C++ Quelltexte. Mit 325.000 Zeilen C Quelltext nimmt das EDG-Front-End den größten Anteil davon in Anspruch. Ebenfalls sehr umfangreich ist die generierte Implementierung der IML-Datenstrukturen mit über 155.000 Zeilen Ada 95- und C Quelltext. Die von den Autoren verfasste Implementierung zu cafe++ ist mit 21.000 Zeilen C++ Quelltext in Zeile zwei der Tabelle angegeben. Erwartungsgemäß groß ist dabei die Klasse `IML_Generator` mit über 18.000 Quelltextzeilen, was 88% der erstellten Implementierung ausmacht.

In Tabelle 4.2 auf der nächsten Seite sind Ergebnisse einer Messung, wie viele IML-Knoten pro Quelltextzeile in einer zu analysierenden Übersetzungseinheit erzeugt werden. Dabei wird cafe++ mit cafe verglichen.

Modul	LOC	NCLOC
Klasse `IML_Generator`	18506	11257
Modul `CafePP` (inkl. `IML_Generator`)	21099	12269
EDG Front-End	325343	203725
IML-Klassen	155397	145732
Summe	520345	372983

Tabelle 4.1: Quelltextzeilen der cafe++ Implementierung

Testfall	NCLOC	IML-Knoten	$\frac{\text{IML-Knoten}}{\text{NCLOC}}$
Empty	0	9	∞
Hello World	14	58	4,1
RegEx	524	4243	8,1
Stresstest	61248	560387	9

Tabelle 4.2: Verhältnis zwischen Quelltextzeilen und Anzahl erzeugter IML-Knoten

Die folgenden Tabellen zeigen die Ergebnisse von Zeit- und Speicherbedarfsmessungen, die mit verschiedenen Übersetzern durchgeführt wurden. Alle Messungen wurden in einer Referenzumgebung mit den folgenden Daten durchgeführt, um vergleichbar zu sein.

1. Das Referenzsystem war ein Rechner mit Linux-Betriebssystem, 1GHz Intel Pentium-III Prozessor und 265MB Hauptspeicher.
2. Die eingesetzten Versionen der Übersetzer waren wie folgt:
   (a) cafe: CVS-Stand vom 15.09.2002;
   (b) cafe++: CVS-Stand vom 06.02.2003 (dritter Testdurchlauf);
   (c) gcc: Version 3.2;

Übersetzer	Datei	IML-Knoten	$\frac{\text{IML-Knoten}}{\text{NCLOC}}$
cafe++	Empty (500)	9	∞
cafe	NCLOC: 0	6	∞
cafe++	il_read.c (100)	2835	2,7
cafe	NCLOC: 1033	3169	3,1
cafe++	statements.c (10)	49146	3,5
cafe	NCLOC: 13983	51811	3,7
cafe++	il.c (10)	81308	4,4
cafe	NCLOC: 18534	86867	4,7

Tabelle 4.3: Vergleich von cafe++ und cafe

cafe und cafe++ wurden mit den Optionen „-d" (Erzeugung von Debug-Code) übersetzt, wobei keine Optimierung des Nativcodes durchgeführt wurde. Der gcc wurde dagegen mit der Option „-O2" (Optimierung des Nativcodes, Stufe 2) übersetzt.

3. Während der Zeitmessungen wurden keine Hintergrundprozesse ausgeführt. Die benötigte Zeit wurde mit Hilfe des GNU „time"-Kommandos ermittelt, Benutzer- und Systemzeit wurden addiert. Alle Zeitmessungen wurden mehrmals in Folge ausgeführt, um einen Einfluss des Referenzsystem-Zustands auf die Messergebnisse zu minimieren. Der angegebene Zeitverbrauch beziffert dabei das arithmetische Mittel aller Messungen. Die durchgeführte Anzahl von Messungen ist jeweils in Klammern hinter der Übersetzungseinheit angegeben.

4. Die Speicherbedarfsmessungen geben jeweils den maximalen Speicher an, der durch den Prozess während der Übersetzung belegt wurde. Gemessen wurde hier mittels des GNU „top"-Kommandos. Der zuletzt angezeigte Wert wurde verwendet, da der Speicherbedarf mit der Zeit monoton anstieg. Im Falle von cafe++ wurde zusätzlich noch der GNU Debugger eingesetzt, um den Speicherbedarf während bestimmter Phasen der Übersetzung zu messen.

5. Die C++ Übersetzungseinheiten entstammen der Testumgebung von cafe++ und werden in Kapitel 5 näher beschrieben. Die „Hello World"-Übersetzungseinheit ist zusätzlich in Anhang B zu finden.
   Die C Übersetzungseinheiten entstammen dem Quelltext des EDG-Front-Ends.

Die Tabellen 4.4 und 4.5 zeigen jeweils die Ausführungszeit und den Speicherbedarf von cafe++ beim Verarbeiten von C++ Übersetzungseinheiten. Um aufzuzeigen, wie viel Quelltextzeilen pro Sekunde von cafe++ verarbeitet werden können, wurde auch das Verhältnis angegeben. Ebenso wird beim Speicherbedarf angegeben, wie viel Speicher pro Quelltextzeile gebraucht wird. Bei den bereinigten Verhältnissen wurde jeweils der Sockelbetrag aus dem „Empty"-Testfall abgezogen. Die „bereinigt"-Spalten beziehen sich jeweils auf die Spalte links davon. Der „Empty"-Testfall ist ein Testfall ohne eine einzige Anweisung.

Übersetzungseinheit	NCLOC	Zeit	$\frac{\text{NCLOC}}{\text{s}}$	bereinigt
Empty (500)	0	0,049s	0	-
Hello World (500)	14	0,054s	260	2800
RegEx (500)	524	0,302s	1735	2071
Stresstest (5)	61248	33,272s	1840	1843

Tabelle 4.4: Ausführungszeit von cafe++

An der Entwicklung der bereinigten Verhältnisse ist erkennbar, dass der Zeit- und Speicherbedarf etwas weniger als linear mit der Größe der Übersetzungseinheit ansteigt.

Übersetzungseinheit	NCLOC	Speicher	$\frac{\text{Speicher}}{\text{NCLOC}}$	bereinigt
Empty (500)	0	4012kB	∞	-
Hello World (500)	14	4220kB	301429Bytes	15214 Bytes
RegEx (500)	524	5564kB	10618Bytes	3033 Bytes
Stresstest (5)	61248	130140kB	2125Bytes	2109 Bytes

Tabelle 4.5: Speicherbedarf von cafe++

In den Tabellen 4.6 und 4.7 ist der Zeit- und Speicherbedarf nochmals aufgeschlüsselt in die Phasen „Front-End" und „Back-End". Die Zeit für das Front-End ist dabei die Zeit, die verbraucht wird, bis der IL-Graph vollständig durch das EDG-Front-End erzeugt ist. Die Back-End Zeit ist die Zeit für das Generieren und Speichern des IML-Graphen. Wie in der prozentualen Angabe des Back-End Anteils zu sehen ist, ist das EDG-Front-End sowohl in der Ausführungszeit, als auch im Speicherbedarf effizienter, wenn die Übersetzungseinheiten größer werden. Abbildung 4.7 auf der nächsten Seite zeigt, dass sich die Größe der Übersetzungseinheit auf den Speicherbedarf relativ zum Front-End mehr auswirkt als auf den Zeitverbrauch.

Übersetzungseinheit	NCLOC	Front-End	Back-End	Anteil Back-End
Empty (500)	0	0,046s	0,003s	6,1%
Hello World (500)	14	0,047s	0,007s	13,0%
RegEx (500)	524	0,077s	0,225s	74,5%
Stresstest (5)	61248	5,156s	28,116s	84,5%

Tabelle 4.6: Ausführungszeit der cafe++ Phasen

Übersetzungseinheit	NCLOC	Front-End	Back-End	Anteil Back-End
Empty (500)	0	3732kB	280kB	7,0%
Hello World (500)	14	3908kB	312kB	7,4%
RegEx (500)	524	4496kB	1068kB	19,2%
Stresstest (5)	61248	46540kB	83600kB	64,2%

Tabelle 4.7: Speicherbedarf der cafe++ Phasen

In der Tabelle 4.8 auf der nächsten Seite wurde weiterhin die Zeit- und Speichereffizienz von cafe++ mit der von cafe und gcc, dem C Übersetzer aus dem GCC verglichen. Da cafe auf C Übersetzungseinheiten beschränkt ist, wurden hier auch nur solche eingesetzt. Teilweise fehlen bei den beiden kleineren Übersetzungseinheiten die Speicherangaben, da sie nicht zuverlässig ermittelt werden konnten. Hier ist erkennbar, dass cafe++ im Gegensatz zu cafe sowohl in der Zeit- als auch in der Speichereffizienz besser abschneidet. Gleichzeitig ist cafe++ aber auch in beiden Kriterien dem gcc unterlegen.

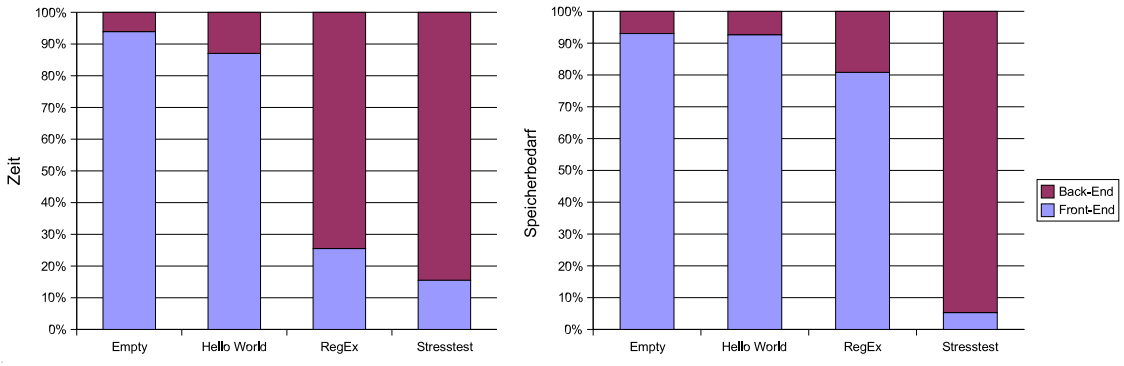


Abbildung 4.7: Ausführungszeit und Speicherbedarf der cafe++ Phasen, prozentual

Übersetzer	Datei	Zeit	$\frac{\text{NCLOC}}{\text{s}}$	bereinigt	Speicher	$\frac{\text{Speicher}}{\text{NCLOC}}$
cafe++	Empty (500)	0,049s	0	-	4012kB	∞
cafe	NCLOC: 0	0,042s	0	-	-	-
gcc		0,111s	0	-	-	-
cafe++	il_read.c (100)	0,366s	2822	3259	5704kB	5,65kB
cafe	NCLOC: 1033	1,059s	975	1016	-	-
gcc		0,141s	7326	34433	-	-
cafe++	statements.c (10)	4,418s	3165	3200	19436kB	1,39kB
cafe	NCLOC: 13983	17,468s	800	802	48448kB	3,47kB
gcc		1,326s	10545	11509	9152kB	0,66kB
cafe++	il.c (10)	6,508s	2848	2869	26496kB	1,46kB
cafe	NCLOC: 18534	26,928s	688	689	70220kB	3,88kB
gcc		3,141s	5901	6116	10644kB	0,59kB

Tabelle 4.8: Vergleich von cafe++, cafe und gcc

In Abbildung 4.8 auf der nächsten Seite werden die Ergebnisse der Geschwindigkeitsmessungen zusammengefasst. Dabei fällt auf, dass C++ Übersetzungseinheiten von cafe++ langsamer übersetzt wurden, als C Übersetzungseinheiten.

Die Ergebnisse der Speicherbedarfsmessungen sind in Abbildung 4.9 zusammengefasst. Bei den kleineren Übersetzungseinheiten RegEx und il_read wirkt sich der Sockelbetrag des Speichers, der immer benötigt wird, stärker auf den Verbrauch pro Quelltextzeile aus. Der relative Speicherbedarf beim Stresstest ist höher als bei den C Übersetzungseinheiten, obwohl der Sockelbetrag hier am wenigsten Einfluss hat. Der Grund dafür sind möglicherweise die Templates, die im Stresstest verwendet werden. Templates, die im Quelltext nur einmal definiert werden, können während der Übersetzung mehrfach im Speicher instantiiert werden.

Wenn in der Abbildung 4.9 Balken fehlen, liegt das nicht an zu kleinen Werten, sondern daran, dass die Werte nicht existieren. Das gilt auch für den

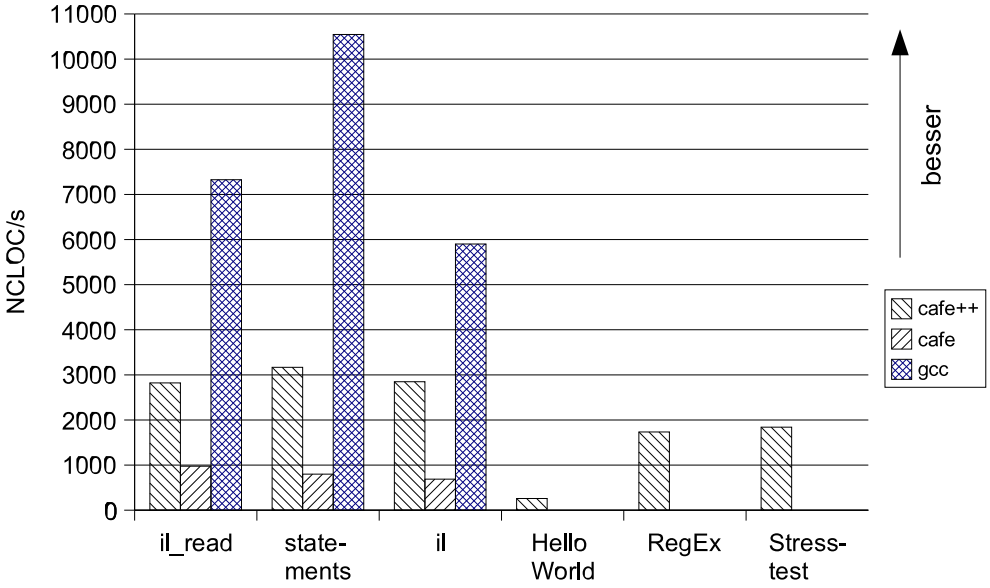


Abbildung 4.8: Vergleich der Geschwindigkeit von cafe++, cafe und gcc

Speicherbedarf von cafe und gcc für il_read. Der Speicherbedarf pro Quelltextzeile wurde für „Hello World" nicht dargestellt, da bei nur 14 Quelltextzeilen der Wert so hoch ist, dass die Darstellung zu unübersichtlich würde.

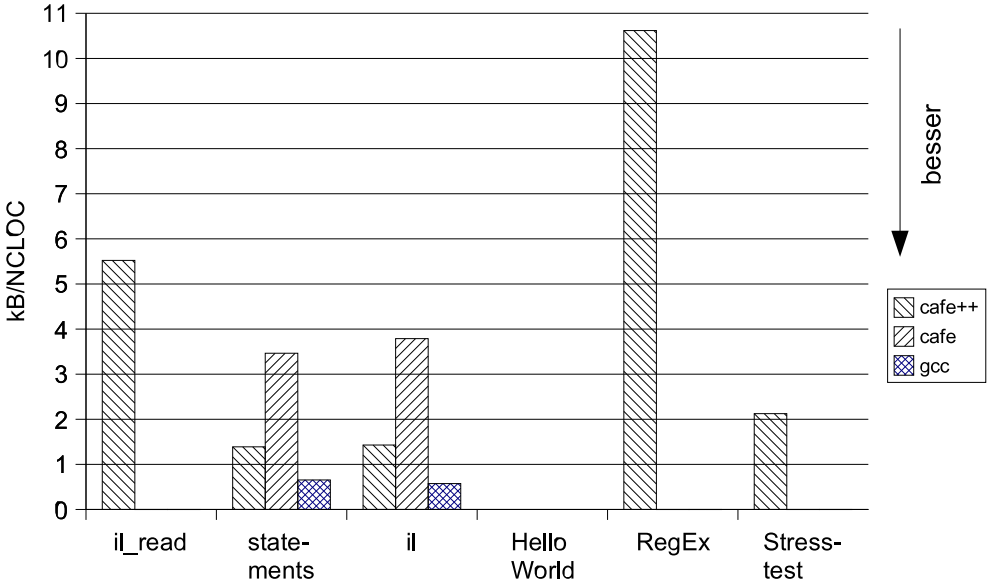


Abbildung 4.9: Vergleich des Speicherbedarfs von cafe++, cafe und gcc

Kapitel 5

Test

T.K.

In dieser Diplomarbeit wurde das Testen angewendet, um die Produktqualität zu sichern. Eine Verifikation von cafe++ wurde als weniger vorteilhaft angesehen. Die Durchführung eines umfangreichen Tests war zudem eine Anforderung in der Aufgabenstellung der Diplomarbeit [21].

Prüfling des Tests war die Implementierung von cafe++. Getestet wurde auf Fehler gegenüber der Spezifikation der erweiterten IML für C, C++ und Java, die bereits in den Kapiteln 2.3 und 2.4 beschrieben wurde. Ziel des Tests war es, sicherzustellen, dass die von cafe++ erzeugte IML-Repräsentation von C++ Übersetzungseinheiten korrekt ist für alle gültigen Eingabedaten. Testabbruchkriterium war der für die reine Durchführung des Tests eingeplante Aufwand von 16 Manntagen. Dieser Zeitrahmen schließt zwar die Zeit für die Fehlerbehebung, nicht aber die Zeit für die Vorbereitung und die Dokumentation des Tests mit ein.

In den folgenden Unterkapiteln wird zuerst die Vorgehensweise beschrieben, die beim Testen angewendet wurde, um im Anschluss daraufhin die Realisierung und die Ergebnisse des Tests zu präsentieren.

5.1 Vorgehensweise

T.K.

Die primäre Testmethode bei cafe++ war die Durchführung eines Black-Box-Tests, der im folgenden Unterkapitel beschrieben wird. Weiterhin wurde unterstützend der Quelltext herangezogen, was im darauf folgenden Kapitel 5.1.2 näher erläutert wird.

5.1.1 Black-Box-Test

T.K.

Der Black-Box-Test wurde als die geeignete Testmethode für das cafe++ Projekt ausgewählt. Gegenüber anderen Methoden bietet der Black-Box-Test den Vorteil, dass er allein auf der Spezifikation aufbauend erstellt werden kann. Damit ist es möglich, den Test völlig unabhängig von den anderen Tätigkeiten im Projektablauf vorzubereiten und damit auch Tätigkeiten zu parallelisieren. Vor allem aber ist der Test unabhängig von der Realisierung des Projektes, so dass z.B. auch die im Entwurf enthaltenen Fehler vom Test aufgedeckt werden können. Dadurch, dass cafe++ auf bestehenden Quelltexten aufbaut, näm-

lich dem C++ Front-End von EDG und dem IML-Generator von Bauhaus, verstärkt sich dieser Vorteil weiter, da Fehler in den Protokollen der Schnittstellen eher durch einen Black-Box-Test aufgedeckt werden können. Somit kann der Black-Box-Test hier wahrscheinlich die meisten Fehler bei vorgegebenem Testaufwand aufdecken.

Die Auswahl der Eingabedaten für den Black-Box-Test richtete sich nach den Eigenschaften der Aufgabenstellung. Der Fokus wurde also mehr auf die Überprüfung der korrekten Übersetzung gerichtet als auf die semantische und syntaktische Analyse von C++ Quelltexten, wie dem Auflösen von Mehrdeutigkeiten in Quelltexten. Diese Aufgabe wird bereits vom EDG-Front-End übernommen. Programmbedingt bestehen die Eingabedaten von cafe++ lediglich aus C++ Übersetzungseinheiten. Weitere Eingabedaten, wie z.B. Kommandozeilenschalter, können außer Acht gelassen werden, da sie keinen gravierenden Einfluss auf die IML-Repräsentation haben.

Um einen möglichst guten Test fahren zu können, wurden folgende Annahmen bezüglich den Programmeigenschaften gemacht, die alle von den für cafe++ geltenden Rahmenbedingungen abgeleitet wurden.

1. Zu beachten ist, dass cafe++ für ein Programm dieser Größe relativ wenig interne Zustände hat. Es erhält vom Front-End den vollständig aufgebauten AST überreicht, welcher sich während der Laufzeit des cafe++ Übersetzerteils nicht mehr gravierend ändert. Den größten Teil des internen Zustandsraumes macht die IML-Repräsentation des zu analysierenden Programms aus. Dieser Umstand erleichtert den Test, da keine komplizierten Eingabesequenzen getestet werden müssen, womit die Testfälle aus relativ kleinen Übersetzungseinheiten bestehen können.

2. Im Gegensatz dazu ist die Komplexität der Eingabedaten sehr hoch, da cafe++ für jedes korrekte C++ Programm die entsprechende IML-Repräsentation generieren können muss. C++ hat dabei recht viele Sprachelemente, die miteinander kombiniert oder ineinander verschachtelt werden können. Aus diesem Grund wurde es angestrebt, möglichst alle Ausprägungen von Sprachelementen mindestens einmal in den Testfällen vertreten zu haben, um so möglichst alle Äquivalenzklassen abzudecken [23].

5.1.2 Weitere Testmethoden

T.K.

Nicht durchgeführt wurde beim Testen ein Glass-Box-Test. Aufgrund der bereits im vorherigen Kapitel genannten Argumente wurde ein Black-Box-Test dem Glass-Box-Test vorgezogen. Trotzdem wurden Elemente des Glass-Box-Tests mit in den Black-Box-Test integriert, um die Effizienz des Tests zu erhöhen und um eine Aussage über die Qualität des Tests zu erhalten.

Der Quelltext von cafe++ wurde instrumentiert, um Vor- und Nachbedingungen von Funktionen und weitere zwingende Bedingungen innerhalb von Funktionen zu prüfen. Diese eingebauten Prüfungen wurden mittels `assert`-Anweisungen aus der C++ Standardbibliothek realisiert und werden dadurch dynamisch, d.h. zur Laufzeit, überprüft. Während des Black-Box-Tests sind

diese zusätzlichen Prüfungen eingeschaltet und jedes Fehlschlagen einer solchen Prüfung bedeutet auch einen Fehler in der Implementierung, d.h. einen positiven Testfall. Mit Hilfe dieser zusätzlichen Prüfungen sollen in erster Linie Fehler der Implementierung selbst, wie z.B. fehlerhafte Schnittstellenprotokolle, möglichst früh identifiziert werden. Die `assert`-Anweisungen wurden manuell von den Autoren während der Implementierung eingefügt. Für getestete Versionen der Software können alle diese Prüfungen ohne Seiteneffekte auch ausgeschaltet werden. Insgesamt enthält die Implementierung 887 `assert`-Anweisungen.

Der zweite Zweck, für den der Quelltext beim Testen herangezogen wurde, war das Ermitteln des Überdeckungsgrades. Um eine Aussage über die Qualität des durchgeführten Tests machen zu können, wurde die Anweisungsüberdeckung als Überdeckungsmaß herangezogen. Das Überdeckungsmaß wurde während des Black-Box-Tests mittels des gcov-Werkzeugs aus der GNU Compiler Collection ermittelt. Sehr interessant wäre an dieser Stelle noch eine Aussage über die Verzweigungsüberdeckung gewesen. Leider ist das gcov-Werkzeug nicht fähig, dieses Überdeckungsmaß zu ermitteln und ein anderes Werkzeug hierfür war zu vertretbaren Kosten nicht verfügbar.

Als Letztes wurde noch ein Stresstest mit einer sehr großen Übersetzungseinheit durchgeführt. Der Quelltext für den Stresstest wurde dem Jikes-Projekt entnommen, welches bereits beim jafe-Projekt zum Einsatz kam. Dabei wurden alle Übersetzungseinheiten (inklusive aller Header-Dateien) des Jikes-Übersetzers in eine einzelne Übersetzungseinheit integriert, die 85367 LOC / 61239 NCLOC umfasst. Der Jikes-Übersetzer ist bis auf wenige Ausnahmen in ISO C++ implementiert und verwendet ausgiebig Sprachelemente wie Ausnahmen, Templates und die RTTI. Der Stresstest hat allerdings die große Einschränkung, dass das Resultat des Testfalls wegen der Größe des Quelltextes nicht mehr auf Korrektheit geprüft werden kann. Die einzige Prüfung, die in diesem Fall durchgeführt wird, sind die dynamischen Prüfungen mittels der `assert`-Anweisungen und das objektive Prüfkriterium, ob die Übersetzung überhaupt von cafe++ durchgeführt werden konnte oder nicht. Da der Aufwand für diesen Test relativ gering ausfiel, wurde er trotz seiner eingeschränkten Aussagekraft als sinnvoll betrachtet.

Weitere Testmethoden wurden aus Zeitgründen nicht mehr in dieser Diplomarbeit umgesetzt. Ein erfolgversprechender Ansatz wäre sicherlich auch ein regelbasierter Test gewesen. In einem solchen Testverfahren würden Regeln dazu eingesetzt werden, die Korrektheit eines IML-Graphen zu prüfen. Beispielsweise könnte eine solche Regel, hier natürlichsprachig formuliert, lauten: „Jeder C++ Datentyp darf nur maximal einmal als Typ in der Symboltabelle modelliert sein." Außer direkt auf den Graphen zu arbeiten, könnten Regeln auch auf Metriken angewendet werden. Zusammen mit weiteren Informationen über den konkreten Testfall könnten auch Regeln wie z.B. „Dieser Testfall enthält genau drei Klassendefinitionen" aufgestellt werden.

Der entscheidende Vorteil eines regelbasierten Testverfahrens liegt klar darin, dass einmal formulierte Regeln vollständig automatisch auf jedem IML-Graphen angewendet werden können. Die einzelnen Regeln wären zudem voneinander unabhängig und die Regelmenge damit leicht erweiterbar.

Nachteilig an diesem Testverfahren ist der Aufwand für die Realisierung. Für eine solche Realisierung wäre es eventuell nötig, eine Notation zu kreieren, in der die Regeln formuliert werden können. Ein Werkzeug müsste dann diese Regeln verarbeiten und auf die IML-Graphen anwenden. Wegen des begrenzten zeitlichen Rahmens wurde ein solcher regelbasierter Test nicht umgesetzt. Jedoch ist es auch mit Hilfe der angewendeten `assert`-Anweisungen möglich, zumindest einfache Regeln zu formulieren. Von dieser Möglichkeit wurde, wie bereits angedeutet, großzügig Gebrauch gemacht.

5.2 Realisierung

T.K.

In diesem Kapitel wird die Realisierung des Tests von cafe++ erläutert. Es werden die Testumgebung, die Eingabedaten, die Durchführung und das Testwerkzeug beschrieben.

5.2.1 Vorbereitung des Black-Box-Tests

T.K.

Direkt nach der Fertigstellung der erweiterten IML-Spezifikation für C, C++ und Java wurde der Test vorbereitet. Wie bereits beschrieben, bestand dieser hauptsächlich aus einem Black-Box-Test. Die Testfälle wurden parallel und manuell von den beiden Autoren der Diplomarbeit erstellt. Die Auswahl orientierte sich dabei am C++ Standardwerk „Die C++ Programmiersprache" von Bjarne Stroustrup [31], es wurde jedoch ergänzend der C++ Standard [14] herangezogen.

Entstanden sind auf diese Weise insgesamt 314 Testfälle, die nach Themengebieten gegliedert sind. Jeder Testfall ist minimal gestaltet, so dass nur der jeweils relevante Aspekt getestet wird und möglichst wenig weitere IML-Knoten entstehen. Einzelne Testfälle prüfen jedoch auch mehrere verwandte Aspekte, wenn dadurch der Test vereinfacht wurde. Zusammen haben diese Testfälle insgesamt 8248 LOC / 2676 NCLOC, was im Durchschnitt 26,3 LOC bzw. 8,5 NCLOC pro Testfall entspricht. 109 der 314 Testfälle beschränken sich überdies auf den C Sprachumfang, so dass sie auch mit cafe übersetzbar sind.

Anfangs wurde die Möglichkeit in Betracht gezogen, bereits vorhandene C++ Testfälle aus anderen Projekten zu verwenden. Eine Möglichkeit hierfür wären die Testfälle der C++ Übersetzers aus der GCC gewesen. Leider hat sich herausgestellt, dass zum einen kein Satz von Testfällen öffentlich verfügbar war, der den kompletten Sprachumfang erfasste. Zum anderen waren die vorhandenen Testfälle oft auf die syntaktische und semantische Analyse von Quelltexten zugeschnitten und damit für den Test von cafe++ eher ungeeignet.

Zusätzlich zu diesen regulären Testfällen wurden weitere Testfälle ausgewählt, um die Grenzfälle der Implementierung zu testen. Konkret waren dies ein Testfall mit einer leeren Übersetzungseinheit, d.h. ohne eine Anweisung, und eine in sich abgeschlossene Übersetzungseinheit einer realen Software. Für den letzteren Fall wurde der Algorithmus für die Behandlung von regulären Ausdrücken des Scintilla-Projektes (`http://www.scintilla.org`) verwendet mit 955 LOC / 524 NCLOC Größe. Dies sollte einer üblichen Größe entsprechen und gleichzeitig immer noch vollständig überprüfbar sein.

5.2.2 Wegwerftests während der Implementierung

T.K.

Während der Implementierung konnten die bereits vorhandenen Testfälle für einfache Wegwerftests verwendet werden. War ein Autor gerade mit der Implementierung eines Teilsystems beschäftigt, konnten einfach die dazu passenden Testfälle ausgewählt und übersetzt werden. Mit dem iml2html-Werkzeug aus dem Bauhaus-Projekt wurden dann aus den IML-Dateien lesbare HTML-Dateien generiert, die dann manuell vom Autor kontrolliert wurden. Das iml-dump-Werkzeug wäre für diese Aufgabe ebenfalls geeignet gewesen, es konnte zum Zeitpunkt der Implementierung von cafe++ jedoch keine Knotenmengen (Sets) ausgeben und war damit nur bedingt einsetzbar.

5.2.3 Systematischer Test

T.K.

Am Ende der Implementierung wurde ein systematischer Test durchgeführt. Die Testumgebung wurde dazu vollständig automatisiert, was möglich ist, da der cafe++ Übersetzer nur eine sehr schmale Benutzerschnittstelle hat und diese vollständig von der Kommandozeile aus bedienbar ist.

Die Eingabedaten des Tests bestanden aus den bereits beschriebenen Testfällen für den Black-Box-Test, den Grenzfällen und dem Stresstest. Daraus wurden die Ist-Resultate durch eine Übersetzung mit cafe++ erzeugt. Das Bereitstellen von Soll-Resultaten war beim ersten Systemtest leider nicht möglich gewesen. Auch das Formulieren von Korrektheitskriterien war nicht praktikabel zu machen. Deswegen wurde für den Systemtest dieselbe Strategie wie auch im jafe-Projekt verwendet [19], wobei die erzeugten Ist-Resultate manuell von den Autoren auf Korrektheit gegenüber der IML-Spezifikation kontrolliert wurden. Das Testergebnis wurde dann im Testprotokoll festgehalten. Im Falle eines negativen Testergebnisses wurde das Soll-Resultat der Testumgebung hinzugefügt und konnte so in folgenden Testläufen als Regressionstest verwendet werden. Waren erst einmal Soll-Resultate für die Testfälle vorhanden, konnte nun eine vollständig automatisierte Testumgebung konstruiert werden. Diese setzt einen Regressionstest um, der völlig ohne manuelle Tätigkeiten auskommt. Ein weiterer Vorteil dieser Vorgehensweise ist, dass nicht der Autor der Implementierung den Test durchführen muss, was allgemein beim Testen als sehr fragwürdig angesehen wird.

Eine weitere Möglichkeit, wenigstens für die in C formulierten Testfälle Soll-Resultate zu erhalten, wäre eine Übersetzung mit cafe gewesen. Leider war auch das nicht möglich, da sich die IML-Repräsentation von cafe und cafe++ auch bei gleichen Testfällen unterscheidet. Beispielsweise verfügt die IML-Repräsentation von C Quelltexten bei cafe++ im Gegensatz zu cafe über ein gesetztes Namensbereich-Attribut und zudem tragen die Einträge in der Symboltabelle der IML erweiterte Namen. Die von cafe erzeugten IML-Dateien waren aber trotzdem als Vorgabe für die manuelle Kontrolle der Ist-Resultate sehr nützlich.

5.2.4 Testwerkzeug

T.K.

Die Testumgebung wurde realisiert mit Hilfe eines Makefiles und hat folgende Eigenschaften:

- Die Durchführung eines Systemtests ist vollständig automatisiert.
- Die Überprüfung auf Korrektheit wird durch einen Regressionstest realisiert. Der Vergleich von Ist- und Soll-Resultaten ist sicher, da er mit dem diff-Werkzeug bytegenau auf den IML-Dateien durchgeführt wird.
- Ein Testlauf mit nur partiell vorhandenen Soll-Resultaten ist möglich.
- Im Falle eines positiven Testergebnisses liegen Ist- und Soll-Resultate in Form von HTML-Dateien vor und können vom Wartungsingenieur ausgewertet werden.
- Die Verwaltung der Dateien übernimmt die Testumgebung. Die Dateien sind dabei wie folgt organisiert:
  - Die Eingabedaten liegen mit dem Namen *Testfallname*.cpp vor.
  - Ist-Resultate haben die Dateinamen *Testfallname*.iml in der Testumgebung. Die HTML-Darstellung davon wird im Unterverzeichnis *Testfallname*.html angelegt.
  - Soll-Resultate besitzen die Dateinamen *Testfallname*.nominal.iml. Die HTML-Darstellung davon wird im Unterverzeichnis *Testfallname*.nominal.html angelegt.
  - Die Überprüfung, ob ein Testfall ausschließlich reinen C Quelltext enthält, geschieht ebenfalls automatisch mittels des C Übersetzers des Systems. In dem Fall, dass die Übersetzungseinheit nur C Quelltext enthält, wird eine *Testfallname*.c Datei erzeugt und mit cafe in die IML-Repräsentation umgewandelt, welche den Namen *Testfallname*.cafe.iml trägt. Die HTML-Darstellung davon wird im Unterverzeichnis *Testfallname*.cafe.html angelegt.
- Ein Testprotokoll wird ebenfalls automatisch erzeugt, welches u.a. folgende Angaben enthält:
  - Zeitpunkt des Testlaufs
  - Resultate in Form einer Liste der Testfälle mit dem Testergebnis und zusätzlichen Hinweisen zum Testlauf, falls der Test positiv war.
  - Eine Zusammenfassung des Testlaufs mit Angaben bezüglich der Menge an Eingabedaten, Soll-Resultaten, dem Testergebnis und der Anweisungsüberdeckung.

5.3 Ergebnisse

T.K.

Für den systematischen Test von cafe++ wurden drei Testdurchläufe ausgeführt. Ziel des ersten Testdurchlaufs war es, alle Fehler zu identifizieren, die gegen die eingebauten `assert`-Anweisungen verstießen. Auf diese Weise wurden 19 Fehler aufgedeckt, welche im Anschluss komplett behoben wurden.

Im zweiten Testdurchlauf wurden daraufhin die erzeugten IML-Dateien manuell auf ihre Korrektheit gegenüber der IML-Spezifikation geprüft. Der Testdurchlauf offenbarte insgesamt 53 Fehler und Mängel. Die Fehler wurden daraufhin nach deren Wichtigkeit bewertet, bevor die wichtigsten davon behoben wurden.

Der dritte und letzte Testdurchlauf bestätigte eine verbleibende Anzahl von 25 Fehlern. Diese verteilen sich auf 11 Mängel, 4 Fehler bei der Darstellung von Templates und 10 sonstige Fehler. Im Regressionstest führen die 25 Fehler bei 63 von 314 Testfällen zu einem positiven Testergebnis.

Während des Tests wurden weiterhin zwei Metriken erhoben. Die bereits erwähnte Anweisungsüberdeckung soll die Qualität des Tests ermitteln. Sie betrug beim hier durchgeführten dritten Testdurchlauf 92,98%, was bereits für eine gute Auswahl von Testfällen spricht. Ebenfalls für die Qualität der Testfälle spricht, dass sich die Anweisungsüberdeckung durch Aufnahme des Stresstests nicht weiter erhöht hat.

Die zweite erhobene Metrik war die IML-Knotentypüberdeckung. Diese gibt eine Aussage darüber, welche IML-Knotentypen durch die Testfälle abgedeckt wurden. Diese Zahl wurde mit Hilfe des imlstat-Werkzeugs ermittelt. Aus der Knotentypüberdeckung kann jedoch keine direkte Aussage über die Qualität des Tests erzeugt werden, denn die Knotentypüberdeckung schließt auch die Fehler in der Implementierung mit ein. Aus diesem Grund sollen im Folgenden die IML-Knotentypen aufgeführt werden, die nicht durch den Test erzeugt wurden, zusammen mit einer Begründung.

- Checked_Cast, Const_Cast und Static_Cast:
  Diese Typkonvertierungen sind in der IL nicht von C Typkonvertierungen unterscheidbar, außer sie treten innerhalb von Templates auf. Für das Auftreten innerhalb von Templates fehlen momentan noch die Testfälle.

- Assert, Fill_Zero_Shift_Right, Synchronized_Sequence, Java_Interface, O_Interface, T_Interface, T_Interface_Name und TJ_Array:
  Knoten dieser Typen werden nur aus Java Quelltexten generiert.

- OC_Struct, OC_Union, TC_Struct, TC_Union, TC_Struct_Name und TC_Union_Name:
  Knoten dieser Typen werden nur aus C Quelltexten generiert (siehe hierzu Kapitel 2.3.4.4).

- Join_Phi, Link_In_Definition und Link_Out_Use:
  Knoten für den Kontroll- und Datenfluss dürfen nicht durch cafe++ erzeugt werden.

- System:
  Nur der IML-Linker erzeugt Knoten dieses Typs.

- TCPP_WChar:
  Der in C++ eingebaute `wchar_t`-Typ wird momentan durch einen speziellen TC_Int-Knoten modelliert. Damit dies durch Knoten des Typs TCPP_WChar geschieht, muss die entsprechende Option im EDG-Front-End umgestellt werden. Dieser Makel wurde aus Zeitgründen nicht mehr behoben.

- Array_LR_Conversion, TC_Completed_Array und TC_Derived_Pointer:
  Die momentane Implementierung unterstützt die Erzeugung dieser Knoten nicht.

Insgesamt kann der bei cafe++ eingesetzte Test als erfolgreich angesehen werden. Eine große Anzahl von Fehlern wurde entdeckt, und die aufgestellten Metriken lassen auf eine gute Qualität des Tests schließen. Das Beheben der verbliebenen gefundenen Fehler wird sicherlich eine der nächsten Tätigkeiten sein, die an der Implementierung von cafe++ durchgeführt werden.

Kapitel 6

cafe++ Bedienungsanleitung

S.S.

In diesem Kapitel wird erklärt, wie mit Hilfe von cafe++ IML-Dateien aus C++ Quelltexten erzeugt werden können. Es werden nur IML-Dateien für einzelne Übersetzungseinheiten erzeugt. Zur weiteren Verarbeitung sind andere Werkzeuge des Bauhaus-Projekts nötig. Dazu gehört der IML-Linker „imllink", mit dem IML-Dateien mehrerer Übersetzungseinheiten verbunden werden können.

Es wird davon ausgegangen, dass eine ausführbare Version der cafe++ Software vorhanden ist. Diese hat den Dateienamen „cafe++". Die Erzeugung einer ausführbaren Version aus dem Quelltext wird in Kapitel 4.3.2 beschrieben. Falls Sie keinen Zugang zu einer ausführbaren Version oder zum Quelltext haben, wenden Sie sich bitte an einen der Ansprechpartner des Bauhaus-Projekts, die auf der Internetseite [16] angegeben sind.

cafe++ ist ein Kommandozeilenwerkzeug. Das heißt, es kann von der Kommandozeile, aus Shellscripten, aus Makefiles usw. aufgerufen werden. Wie die ausführbare Datei vom System gefunden wird, ist plattformabhängig.

In einigen wenigen Fällen ist es beim Einsatz von Templates notwendig, ein beim EDG-Front-End mitgeliefertes Shellscript dem cafe++ Übersetzer vorzuschalten. Beispielsweise muss dies geschehen, wenn das `export`-Schlüsselwort zum Exportieren von Templates eingesetzt wird. Der Name des Shellscripts ist „eccp" und dessen genaue Beschreibung ist in [6, Kapitel 2] nachzulesen.

Parameter

Beim Aufruf können Kommandozeilenparameter angegeben werden. Davon erkennt cafe++ einige selbst und gibt die anderen an das integrierte EDG-Front-End weiter. [6, Kapitel 2] enthält die Beschreibung der Kommandozeilenparameter, die vom EDG-Front-End akzeptiert werden.

Die Aufrufsyntax kann folgendermaßen zusammengefasst werden:

`cafe++` [*EDG-Front-End-Parameter*]
[**`--dump-il`** *Dateiname*]
[**`--dump-orig-il`** *Dateiname*]
`-o` *IML-Dateiname Quelltextdateiname*

Ein Kommandozeilenparameter, der unbedingt angegeben werden muss, ist der Name der Eingabedatei. Er muss *nach* allen anderen Parametern des Front-Ends angegeben werden. Der Dateiname wird direkt angegeben, ohne einleitenden Strich und Parameternamen.

Alle anderen Parameter werden durch einen oder zwei Striche eingeleitet. Darauf folgt der Name des Parameters und bei manchen Parametern ein Wert, durch Leerzeichen vom Namen getrennt. Das entspricht den üblichen Konventionen von Unix, wie sie beispielsweise auch für die Übersetzer der GCC gelten.

Die von cafe++ selbst erkannten, nicht in [6, Kapitel 2] beschriebenen, Parameter sind:

`--dump-il` ***Dateiname*** Wenn dieser Parameter angegeben wird, wird ein IL-dump (siehe Kapitel 3.2.1.3) erzeugt. Er wird in der Datei mit dem angegebenen Namen gespeichert.

`--dump-orig-il` ***Dateiname*** Wie `--dump-il`, es wird jedoch ein Dump des IL-Graphen vor der IL-Vorbereitungsphase (siehe Kapitel 4.2.3.3) erzeugt.

`-o` ***Dateiname*** Mit diesem Parameter wird angegeben, in welche Datei der IML-Graph gespeichert werden soll. Dieser Parameter muss unbedingt angegeben werden.

Zu cafe++ gehört auch die Datei „preinclude.h" (siehe Kapitel 4.3.3). Diese Datei muss mit dem Parameter `--preinclude` angegeben werden, da ansonsten einige Quelltexte nicht übersetzt werden können.

Falls von dem zu übersetzenden Programm System-Header-Dateien eingebunden werden, können die normalen System-Header-Dateien verwendet werden, die auch von anderen C++ Übersetzern verwendet werden. Einige System-Header-Dateien werden auch mit dem EDG-Front-End mitgeliefert. Diese können verwendet werden, falls die normalen System-Header-Dateien nicht-standardkonforme Sprachelemente enthalten.

Damit Header-Dateien gefunden werden, müssen die Suchpfade mit dem Parameter `--sys_include` oder `-I` angegeben werden (siehe auch [6, Kapitel 2]).

Beispiel

Ein einfaches Beispiel für die Übersetzung der C++ Übersetzungseinheit „hello_world.cpp" mit cafe++ in die IML-Repräsentation „hello_world.iml" unter Unix würde folgendermaßen aussehen:

```
cafe++ \
    --sys_include cafepp/src/edg/include \
    --preinclude cafepp/src/include/preinclude.h \
    -o hello_world.iml hello_world.cpp
```

Kapitel 7

Zusammenfassung und Ausblick

T.K.

In diesem Kapitel soll im ersten Teil der Projektverlauf in einer Zusammenfassung darstellt werden, bevor die Ergebnisse der Diplomarbeit kurz zusammengefasst und bewertet werden. Im zweiten Teil wird ein Ausblick auf die mögliche Zukunft von cafe++ gegeben.

7.1 Zusammenfassung

7.1.1 Projektverlauf

T.K.

Das Projekt „Erweiterung und Generierung der Zwischendarstellung IML für C++ Programme" wurde in sieben Phasen geplant und durchgeführt:

- Auswahl eines Front-Ends,
- Projektplanung,
- Erweiterung der IML,
- Entwurf,
- Implementierung,
- Test und
- Bericht.

Um ein besseres Bild vom Projektverlauf zu vermitteln, sollen die sieben Phasen des Projekts im Folgenden näher erläutert werden. Im Anschluss daran wird auf den benötigten Aufwand für das Projekt eingegangen, wobei auch die Abweichung des Soll-Zeitplans vom Ist-Zeitplan diskutiert wird. Zuletzt soll noch kurz auf die Aufgabenverteilung eingegangen werden.

7.1.1.1 Projektphasen

T.K.

Die hier aufgeführten Phasen des Projekts wurden grob in der aufgeführten Reihenfolge durchgeführt. Teilweise überschnitten sich jedoch die Phasen, worauf bei der Beschreibung zur jeweiligen Phase hingewiesen wird. Die Dokumentation der einzelnen Ergebnisse war eine phasenübergreifende Tätigkeit und wird aus diesem Grund nicht gesondert aufgeführt.

Auswahl eines Front-Ends In der ersten Phase des Projekts wurde ein Vorprojekt durchgeführt. Es galt dabei herauszufinden, ob und welche Front-Ends für die Realisierung des Projekts in Frage kommen würden. Die Auswahl des Front-Ends wurde in Form einer Evaluierung durchgeführt. Letztendlich war das Ergebnis, dass das Projekt mit Hilfe der Front-Ends von EDG und der GCC realisierbar sein würde. Die Entscheidung fiel zu Gunsten des EDG-Front-Ends aus, welches dann für die spätere Realisierung eingesetzt wurde. Eine genaue Beschreibung des Auswahlprozesses ist in Kapitel 3.1 zu finden.

Ein weiteres Ergebnis dieser Voruntersuchung war, dass das Projekt zwar durchführbar wäre, der benötigte Aufwand jedoch den Rahmen einer Diplomarbeit für eine Person bei weitem übersteigen würde. Für die Lösung dieses Problems kamen zwei Möglichkeiten in Frage. Zum einen konnte der C++ Sprachumfang für den zu entwickelnden Übersetzer gekürzt werden. Zum anderen konnte die Diplomarbeit für zwei Bewerber ausgeschrieben werden. Wie eingangs erwähnt, wurde die Diplomarbeit für zwei Bewerber ausgeschrieben und von diesen auch durchgeführt.

Projektplanung Als das Ergebnis des Projektplans feststand wurde ein detaillierter Projektplan erstellt. Eine gründliche Planung der Tätigkeiten wurde für ein Projekt dieser Größe als sehr wichtig angesehen, vor allem, da es galt, die Arbeit von zwei Autoren zu koordinieren. Eine weitere Schwierigkeit bei der Durchführung ist auch, dass die Rahmenbedingungen von Diplomarbeiten, ganz besonders der feststehende Endtermin, strikt einzuhalten sind.

Zu den zentralen Bestandteilen des Projektplans gehören der Zeitplan und die Risikoanalyse. Im Zeitplan wird das Vorgehen zusammen mit einer Aufwandsschätzung in einen zeitlichen Ablauf gebracht. Dieser enthält Arbeitspakete, die genau definieren,

- welche Tätigkeiten
- von welchem Autor
- in welchem Zeitraum und mit wie viel zeitlichen Aufwand
- und mit welchen Endergebnissen

durchgeführt werden. Jedes Arbeitspaket wird entweder durch einen internen oder einen externen Meilenstein abgeschlossen. Bei den externen Meilensteinen, von denen in diesem Projekt sieben Stück definiert wurden, war eine Abnahme durch den Betreuer notwendig. Interne Meilensteine wurden insgesamt 14 Stück definiert. Im nächsten Kapitel wird die ursprüngliche Planung der tatsächlichen entgegengestellt.

Das Ziel der Risikoanalyse war es, Risiken zu erkennen, die den Projekterfolg gefährden können. Zu jedem Risiko wurden die Indikatoren, die Konsequenzen im Falle des Eintretens, mögliche Gegenmaßnahmen und die Wahrscheinlichkeit des Eintretens ausgearbeitet. Damit sollte es möglich sein, eintretende Risiken früh zu erkennen, um schnell und definiert darauf reagieren zu können. Um eventuelle Risiken während des Projektverlaufs abfangen zu können, wurden zwei zeitliche Puffer mit einem Gesamtumfang von zwei Wochen in die Zeitplanung mit aufgenommen.

Erweiterung der IML Die Erweiterung der IML kann in drei Teilschritte aufgeteilt werden. Die drei Teilschritte wurden in sequenzieller Reihenfolge direkt nach dem Erstellen des Projektplans durchgeführt. Im ersten Schritt war eine Einarbeitung in die IML nötig. Hier mussten die Konzepte und Details der aktuellen IML für C und der erweiterten IML für C und Java erarbeitet werden. War dies geschehen, so konnten im nächsten Schritt Konzepte für die Erweiterung aufgestellt werden, um im letzten Schritt die gemeinsame IML für C, C++ und Java zu spezifizieren. Die Ergebnisse dieser Phase sind im Kapitel 2 nieder geschrieben.

Entwurf Die Entwurfsphase begann wiederum mit einer Einarbeitung, dieses Mal in das C++ Front-End von EDG. Das dabei erlangte Wissen war zwingend nötig für die Erstellung eines Entwurfs, da dieser stark vom Front-End, insbesondere dessen Zwischendarstellung, bestimmt wird. Der grundlegende Entwurf ist im Systementwurf festgehalten. Dieser legt die Konzeption fest und diskutiert Entwurfsfragen. Kapitel 4 in dieser Ausarbeitung widmet sich dem Systementwurf. Auf den Systementwurf aufbauend wurde daraufhin ein Feinentwurf erstellt. Hierfür wurden Quelltextdateien mit Stubs der Schnittstellenfunktionen erzeugt.

Implementierung Die Implementierungsphase folgte direkt auf die Entwurfsphase. In dieser Zeit wurde der Übersetzer realisiert, der C++ nach IML transformiert. Zu Beginn wurde ein Buildprozess ausgearbeitet und das Grundgerüst der Applikation erstellt. In der folgenden Zeit wurde daraufhin die Funktionalität der Implementierung Schritt für Schritt fertig gestellt. Weitere Informationen zur Implementierung sind in den Kapiteln 4.3 bis 4.5 zu finden.

Test Die Testphase begann bereits nach der Spezifikation der erweiterten IML. Es wurde eine Teststrategie festgelegt und die Testdaten wurden erzeugt. Außerdem wurde eine Testumgebung geschaffen und die Testprotokolle wurden vorbereitet.

Der zweite Teil der Testphase begann nach der Implementierung. Während der Testdurchführung wurden drei systematische Testdurchläufe mit anschließender Fehlerbehebung gemacht. Die vollständige Testphase ist in Kapitel 5 dokumentiert.

Bericht Die letzte Phase des Projektes widmete sich der Erstellung des Abschlussberichts. Im vorliegenden Abschlussbericht werden im Wesentlichen die Ergebnisse der vorhergehenden Phasen zusammengetragen.

T.K.

7.1.1.2 Aufwand und Verteilung

Während des gesamten Projektverlaufs wurde der benötigte Aufwand genau erfasst, um die Genauigkeit des Projektplans überprüfen zu können. In die Aufwandsangaben wurden die Zeit für Pausen und Zeiten für An- und Abfahrt zur Arbeitsstelle nicht mit aufgenommen. Der tatsächliche Aufwand für die Diplomarbeit ist also deutlich größer.

In Tabelle 7.1 ist die Aufwandsverteilung aufgeführt, wie sie zum einen geplant war und wie sie sich tatsächlich ergeben hat. Alle prozentualen Angaben beziehen sich auf den Soll-Aufwandsteil von 1563h. Der Puffer wurde nicht mit eingerechnet.

	Aufwandsanteil Soll (Stunden / %)	Aufwandsanteil Ist (Stunden / %)
Auswahl Front-End	60h / 3,8%	115h / 7,4%
Projektplanung	50h / 3,2%	104h / 6,7%
Erweiterung IML	246h / 15,7%	266h / 17,0%
Entwurf	185h / 11,8%	102h / 6,5%
Implementierung	548h / 35,1%	654h / 41,8%
Test	258h / 16,5%	245h / 15,7%
Bericht	216h / 13,8%	430h / 27,5%
Summe	1563h / 100,0%	1916h / 122,6%

Tabelle 7.1: Soll- und Ist-Aufwandsverteilung

Deutlich sichtbar ist hier, dass die Implementierung mehr Aufwand erfordert hat als dafür eingeplant war. Dass diese Diplomarbeit sehr implementierungslastig werden würde, war bereits zu Beginn abzusehen. Aus diesem Grund wurde mit 35,1% mehr als ein Drittel der Zeit der Diplomarbeit für die Implementierung eingeplant. Trotzdem wurde auch diese Schätzung übertroffen. Die Gründe dafür lagen hauptsächlich bei der Implementierung von:

- Initialisierung und Zerstörung (siehe Kapitel 4.3.7 und Kapitel 4.3.8),
- Lebenszeit von Objekten (siehe Kapitel 3.2.1.4) und
- Deklarationen innerhalb von Funktionen (siehe 4.3.6).

Für diese Punkte wurde in der detaillierten Zeitplanung der Implementierung viel zu wenig Zeit eingeplant. Die Gründe für den unerwarteten Aufwand liegen in der inhärenten Komplexität der angegebenen Punkte in der Programmiersprache C++. Im Gegensatz dazu ging jedoch die Implementierung der Operatoren schneller vonstatten, als dies in der Planung vorgesehen war.

Die eingeplanten zwei Wochen Puffer wurden komplett für die Implementierung aufgebraucht. An dieser Stelle erwies sich die Risikoanalyse als erfolgreich, da eine aufwändigere Implementierung ein eingeplantes Risiko darstellte. Von den weiteren eingeplanten Risiken ist zum Glück keines eingetreten.

Der detaillierte Zeitplan sah für die Implementierungsphase eine genaue Planung der Teilschritte bis auf die Ebene der einzelnen Sprachelemente von

C++ vor, die übersetzt werden sollten. In der Realität hat sich jedoch gezeigt, dass eine solch genaue Planung nicht einzuhalten war. Während der Implementierungsphase wurde die Reihenfolge der Teilschritte stark geändert, da es den Autoren als vorteilhaft erschien. Dies lag daran, dass die Planung sich an den Sprachelementen von C++ orientierte, wobei sich die Teilschritte der Implementierung viel mehr an der IML orientierten.

Abgesehen von diesen Abweichungen von der Planung wurde der Zeitplan vollständig eingehalten. Damit lagen auch alle Ergebnisse zum Zeitpunkt der externen Meilensteine vor. Wie aus der Ist-Aufwandsverteilung in Tabelle 7.1 auf der vorherigen Seite zu sehen ist, war das aber nur durch erhöhten Aufwand der Autoren möglich. Wie bereits erwähnt wurde zwar der Zeitplan eingehalten, aber der dafür eingeplante Aufwand wurde in praktisch allen Phasen des Projekts überschritten.

Während der einzelnen Phasen arbeiteten beide Autoren stets parallel. Dadurch ist es nicht verwunderlich, dass auch die Aufwandsverteilung auf die beiden Autoren sehr gleichmäßig ausfiel, wie in Abbildung Abbildung 7.1 zu sehen ist.

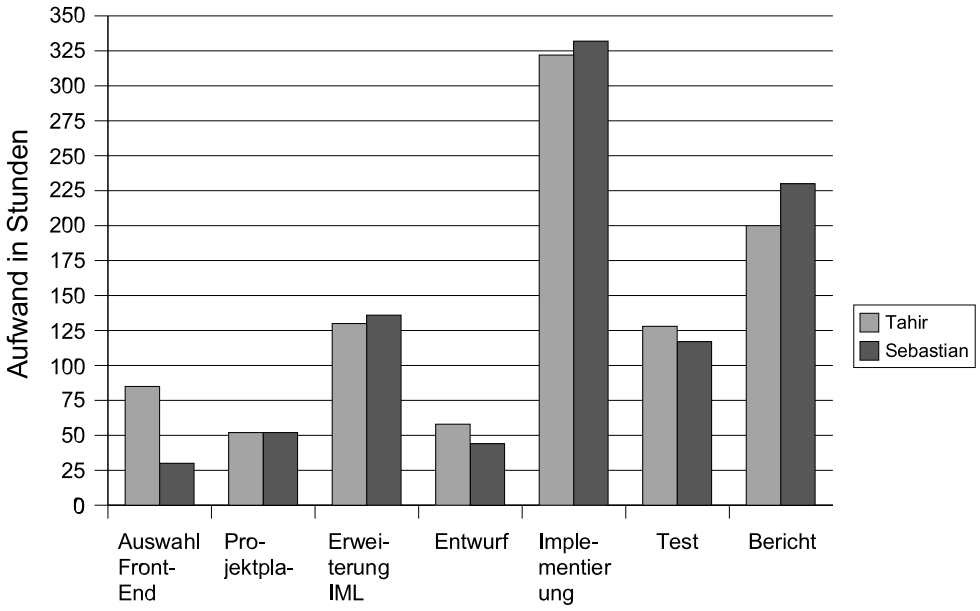


Abbildung 7.1: Ist-Aufwandsverteilung der beiden Autoren

Für die Implementierungsphase kann weiterhin die Aufgabenverteilung genauer angegeben werden, da hier größere Teile von jeweils einem Autoren alleine realisiert wurden:

- T_Nodes: Tahir
- O_Nodes: Sebastian
- Prog_Units: Sebastian
- Statement_Sequence: Tahir

- Initialisierung: Sebastian
- Zerstörung: Tahir
- Ausnahmen: Tahir
- Funktionsaufrufe: Sebastian; virtuelle und indirekte: Tahir
- Operatoren und Ausdrücke: Sebastian
- Kontrollstrukturen: Sebastian

Aufgaben, die größere Risiken enthielten, wurden zuerst bearbeitet, falls sie nicht von der Bearbeitung einer anderen Aufgabe abhängig waren. Die Aufteilung zwischen den Autoren entstand dadurch, dass ein Autor, sobald er eine Aufgabe gelöst hatte, die nächste Aufgabe in Angriff nahm.

7.1.2 Ergebnisse

7.1.2.1 Bewertung der Ergebnisse

T.K.

Die Ergebnisse der Diplomarbeit sollen in diesem Kapitel noch einmal zusammengefasst und bewertet werden. Das erste Ergebnis dieser Diplomarbeit ist die erweiterte IML mit den folgenden Eigenschaften:

- Die erstellte IML-Spezifikation ist mächtig genug, um die Programmiersprachen C, C++ und Java darzustellen. Dadurch ist die primäre Aufgabenstellung dieser Diplomarbeit erfüllt.
- Die IML verfügt weiterhin über eine vereinheitlichte Darstellung über verschiedene Programmiersprachen hinweg, die gleichzeitig quellennah und analysierbar ist. Dadurch wurden die Grundkonzepte der IML auch mit der C++ Erweiterung weitergeführt.
- Die C++ Erweiterung wurde im Sinne der bestehenden Konzepte der C und Java Modellierung durchgeführt. Beispielsweise werden Templates nicht nur generisch dargestellt, sondern zusätzlich noch expandiert. Die notwendigen Änderungen an den bestehenden Quelltexten von Bauhaus, insbesondere den Analysen, sollten damit gering ausfallen.

Alle Anforderungen bezüglich der IML-Erweiterung sollten damit erfüllt sein.

Das zweite Ergebnis der Diplomarbeit ist der entwickelte Übersetzer, der folgende Eigenschaften besitzt:

- Der Übersetzer, cafe++ genannt, baut auf einem bestehenden C++ Front-End, dem von EDG, auf. Mit dem EDG-Front-End wurde die beste zur Verfügung stehende Lösung gewählt. Das Front-End ist besonders standardkonform, effizient und besitzt allgemein eine hohe Produktqualität.

- Die notwendigen Änderungen am Front-End umfassen nur wenige Zeilen im Quelltext und werden automatisch durchgeführt. Die Einbindung des Front-Ends wurde im Buildprozess von cafe++ ebenfalls automatisiert. Die Integration zukünftiger Versionen des Front-Ends benötigt durch die geringen Änderungen am Front-End nur einen minimalen Aufwand.

- Die Implementierung von cafe++ baut auf dem IML-Generator auf und ist leicht in die Bauhaus-Infrastruktur integrierbar. Der gesamte Übersetzungsvorgang mit cafe++ ist außerdem effizient, wie die Laufzeitmessungen bestätigt haben.

Weiterhin wurde bei der Durchführung eine hohe Qualität des Prozesses und des Produkts erreicht. Das Erreichen dieser Ziele wurde vor allem durch folgende Maßnahmen gefördert:

- Eine genaue Planung des gesamten Projekts sollte die Prozessqualität sichern. Tatsächlich konnte das Projekt nach den Wasserfallmodell durchgeführt werden. Diese Vorgehensweise wurde durch die früh feststehenden Anforderungen und durch eine disziplinierte Durchführung ermöglicht.

- Gegenseitige Reviews der Autoren und externe Reviews dienten der Qualitätssicherung von Produkt und Dokumentation.

- Ein guter Entwurf, eine defensive Programmierung und die Verwendung eines Styleguides sollte die Qualität der Implementierung im Voraus sicherstellen. Der Entwurf erwies sich als tragfähig genug und musste also nach dessen Erstellung nicht mehr geändert werden.

- Ein ausführlicher und systematischer Test half, die tatsächliche Qualität der Implementierung festzustellen und diese zu verbessern.

Wie der erfolgreiche Test zeigte, enthält die Implementierung noch einige Fehler. Trotzdem kann die Implementierung als erfolgreich angesehen werden, vor allem im Hinblick auf die begrenzte Zeit, die dafür zur Verfügung stand.

7.1.2.2 Erfahrungen

S.S.

Diese Diplomarbeit hatte nicht nur die bereits genannten Ergebnisse als Folge, sondern gab den Autoren zusätzlich die Möglichkeit, Erfahrungen zu sammeln. Primär wurde natürlich das Wissen der Autoren in den Gebieten

- Übersetzerbau,

- insbesondere bei Zwischendarstellungen von Programmen,

- den Programmiersprachen C, C++ und Java und

- Projektmanagement

stark erweitert.

Außerdem wurde diese Diplomarbeit von zwei Autoren erstellt. Im Nachhinein erwies sich diese Entscheidung als sehr vorteilhaft. Ein einzelner Autor hätte, auch bei gekürzter Aufgabenstellung, niemals die hier erreichte Sorgfalt bei der Durchführung erreichen können.

Ungünstig wirkte sich die Aufgabenstellung für zwei Autoren aus, da sich die Aufgabenstellung nicht vollständig auftrennen lies, so dass auch zwei Autoren parallel daran arbeiten konnten. So wurde das Projekt praktisch im Team von den beiden Autoren durchgeführt, was bereits bei der Besprechung der Aufwandsverteilung angesprochen wurde. Dieses Vorgehen spiegelt sich auch in der Aufteilung dieses Berichts, der ebenfalls in Teamarbeit erstellt wurde.

Andererseits lief die Teamarbeit sehr erfolgreich. Die Autoren konnten sich in ihren Fähigkeiten oft ergänzen, was dem Projekt zugute kam. Alle Tätigkeiten waren auch stets in dem Maße parallelisierbar, dass keine bremsenden Abhängigkeiten entstanden sind. Nicht zuletzt wirkte sich auch die Teamarbeit sehr positiv auf das Arbeitsklima aus.

7.2 Ausblick

S.S.

In diesem Kapitel wird versucht, die zukünftige Entwicklung und Anwendung von cafe++ und dem Bauhaus-Projekt vorauszusehen und teilweise zu planen. Eine zeitliche Übersicht über die geplanten Aktivitäten bezüglich cafe++ gibt Abbildung 7.2.

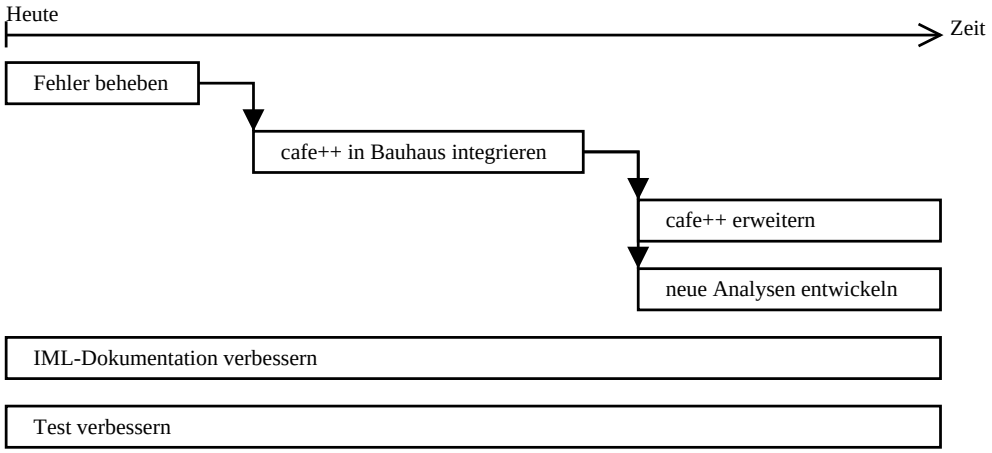


Abbildung 7.2: Geplante Zukunft von cafe++

Zuerst müssen aus Zeitmangel noch nicht behobene Fehler behoben werden. Danach kann mit der Integration von cafe++ in Bauhaus begonnen werden. Sobald cafe++ mit den existierenden Bauhaus-Werkzeugen zufriedenstellend zusammenarbeitet, kann die Entwicklung weiterer Analysewerkzeuge und die Weiterentwicklung von cafe++ in Angriff genommen werden. Die Dokumentation der IML kann und soll zu jedem Zeitpunkt verbessert und mitgeführt werden, genau wie der Test, der von der IML-Spezifikation abhängt. Die

einzelnen Aufgaben werden in den folgenden Kapiteln noch näher erläutert.

Nach allen Änderungen von cafe++ sollte der in Kapitel 5 vorgestellte Regressionstest durchgeführt werden. Dadurch können unerwünschte Nebenwirkungen der Änderung aufgedeckt und die Soll-Daten bei Spezifikationsänderungen immer auf dem neuesten Stand gehalten werden.

7.2.1 Integration von cafe++ in Bauhaus

S.S.

Die im Rahmen dieser Diplomarbeit entwickelte Software cafe++ ist ein Teil des Bauhaus-Projekts. Durch Integration von cafe++ in die bestehende Werkzeugsammlung von Bauhaus sind Bauhaus-Werkzeuge in Zukunft in der Lage, C++ Programme zu analysieren.

Die Schnittstelle zwischen cafe++ und den anderen Bauhaus-Werkzeugen ist die IML. Die IML (-Spezifikation) musste für die Darstellung von C++ Programmen erweitert werden (siehe Kapitel 2). Die ursprüngliche IML wurde unabhängig davon weiterentwickelt. Die dadurch entstandenen beiden Zweige von IML-Versionen müssen zusammengeführt werden, um cafe++ in das Bauhaus-Projekt zu integrieren.

Die anderen Bauhaus-Werkzeuge wurden noch nicht an die erweiterte IML-Spezifikation angepasst. Um die Integration von cafe++ in das Bauhaus-Projekt zu vervollständigen, müssen einige der Bauhaus-Werkzeuge an die erweiterte IML angepasst werden. Die folgenden Unterkapitel behandeln den Einfluss der IML-Erweiterungen auf verschiedene Bauhaus-Werkzeuge.

7.2.1.1 Analysen

S.S.

In der Sprache C++ wurden einige Elemente der Sprache C übernommen und einige neue Sprachelemente hinzugefügt. Die Behandlung dieser beiden Arten von Sprachelementen wird im Folgenden getrennt beschrieben, da die Bauhaus-Werkzeuge C Sprachelemente schon vor der IML-Erweiterung behandeln konnten.

Behandlung von C Sprachelementen Die Darstellung von C Sprachelementen hat sich in der erweiterten IML nicht wesentlich geändert. Die meisten C Sprachelemente können deshalb von bestehenden Analysewerkzeugen behandelt werden, wie vor der IML-Erweiterung. Es gibt jedoch folgende Ausnahmen, bei denen einige Analysen auch für C Sprachelemente angepasst werden müssen:

- Bei der IML-Klasse Unconditional_Branch zur Darstellung von unbedingten Sprüngen wurde das Attribut End_Lifetime verändert (siehe Kapitel 2.4.13). Werkzeuge, die die durch unbedingte Sprünge ausgelöste Zerstörung von Variablen betrachten, müssen angepasst werden.

- Das Attribut Name der IML-Klasse SymNode wurde entfernt und durch die Attribute Mangled_Name und Unmangled_Name ersetzt (siehe Kapitel 2.3.2). Werkzeuge, die das Attribut Name verwenden, müssen angepasst werden. Die Anpassung ist einfach, sobald klar ist, ob die Ver-

wendung von Name durch den eindeutigen Mangled_Name oder den quelltextnahen Unmangled_Name ersetzt werden soll.

- Das `struct` Sprachelement kann von cafe++ nicht als C Sprachelement erkannt werden (siehe Kapitel 2.3.4.4). Analysen, die das `struct` Sprachelement auch in C++ Quelltexten behandeln sollen, müssen deshalb angepasst werden.

- Das Ergebnis von Zuweisungen mit den Operatoren „=", „+=" usw., den Präfixoperatoren „++", „--" und dem „?"-Operator kann in C++ ein L-Value sein. Im folgenden Beispiel wird die Zuweisung „i = 5" als L-Value verwendet:

  ```
  int i;
  (i = 5) = 42; // => i == 42
  ```

  In C war das nicht möglich und wird deshalb von einigen Analysewerkzeugen möglicherweise nicht korrekt behandelt. Diese Werkzeuge müssen angepasst werden.

Behandlung von C++ Sprachelementen Die wichtigsten Erweiterungen von C++ gegenüber C sind Klassen, Ausnahmen und Templates. Analysewerkzeuge, die Klassen oder Ausnahmen in die Analyse mit einbeziehen sollen, müssen dafür angepasst werden. Das gilt selbstverständlich auch für Templates, wenn sie im Zusammenhang mit Klassen auftreten. Template-Funktionen können von manchen Analysewerkzeugen unverändert behandelt werden (siehe unten).

Außer Klassen, Ausnahmen und Templates wurde C++ noch um einige andere Sprachelemente erweitert. Diese Erweiterungen werden in der IML jedoch so dargestellt, dass eine Anpassung der Analysewerkzeuge in den meisten Fällen nicht nötig ist. Diese Sprachmittel und ihre Auswirkungen auf Bauhaus-Werkzeuge werden hier beschrieben:

Einzeilenkommentare mit „//" Kommentare werden in der IML und damit von allen Bauhaus-Werkzeugen ignoriert.

Variablendeklarationen innerhalb Blöcke Variablendeklarationen und -initialisierungen sind in C nur am Anfang eines Blocks erlaubt, in C++ jedoch an jeder Stelle innerhalb eines Blocks. Analysen, die die Initialisierung von Variablen nicht vor jeder anderen Anweisung eines Blocks erwarten, können unverändert weiterverwendet werden.

Das gleiche gilt für Variablendefinitionen im Kopf einer Kontrollstruktur.

Überladene Funktionen werden in der IML als verschiedene Funktionen mit dem gleichen Unmangled_Name dargestellt. Viele Analysen behandeln überladene Funktionen deshalb automatisch richtig, sofern Name mit Unmangled_Name ersetzt wird.

Falls eine überladene Funktion als *eine* Funktion betrachtet werden soll, kann dazu das Attribut Mangled_Name verwendet werden.

Überladene Operatoren werden in der IML wie normale Funktionen dargestellt, wobei das Attribut Unmangled_Name den Namen des Operators enthält, z.B. „`operator+`". Viele Analysen behandeln überladene Funktionen deshalb automatisch richtig.

inline-Funktionen können wie normale Funktionen behandelt werden. Eine Anpassung von Bauhaus-Werkzeugen ist nicht notwendig.

Templatefunktionen werden von Analysen, die die Semantik von Programmen analysieren, automatisch richtig behandelt. Die unveränderten Analysen behandeln die Instantiierung des Templates richtig. Die Darstellung des eigentlichen Templates, das keine direkte Auswirkung auf die Semantik des Programms hat, ist nur über den The_Template-Zeiger erreichbar. Sie wird von Analysen, die keine The_Template-Zeiger verfolgen, ignoriert. Das Attribut The_Template ist neu, deshalb wird es von keiner bestehenden Analyse ausgewertet.

`new/delete` Die Anwendung des `new`- oder `delete`-Operators auf nicht-Klassen-Datentypen wird von unveränderten Werkzeugen wie ein normaler Funktionsaufruf behandelt.

`bool` Der Datentyp `bool` existierte schon in der C IML, obwohl er in C Quelltext nie explizit vorkommt. Eine Anpassung von Werkzeugen ist deshalb nicht notwendig.

`namespace` Namensbereiche wirken sich nur auf die Namenserweiterung aus. Analysewerkzeuge werden nicht beeinflusst.

Referenzen werden von unveränderten Analysewerkzeugen wie Zeiger behandelt.

7.2.1.2 Linker

S.S.

Der IML-Linker ist ein Bauhaus-Werkzeug, welches separat erzeugte IML-Dateien einzelner Übersetzungseinheiten verbindet (siehe Kapitel 2.1.1.4). Der Linker verbindet Knoten vom Typ OC_Entity durch neue Kanten und verwendet dabei das Attribut Name zur Identifikation dieser Knoten. Das Attribut Name existiert in der für C++ erweiterten IML nicht mehr. Der IML-Linker muss so angepasst werden, dass das Attribut Mangled_Name anstatt dem Attribut Name verwendet wird. Dabei muss beachtet werden, dass erweiterte Namen Sonderzeichen wie „@" oder Leerzeichen enthalten können.

Inline-Funktionsdefinitionen, die in mehreren Übersetzungseinheiten auftreten, führen zu einer Warnung des Linkers. Diese Warnung sollte (nur für inline-Funktionen) unterdrückt werden.

Statische Klassenattribute werden vom Linker gleich behandelt wie statische Variablen in Funktionen und wie globale Variablen. Der Linker muss hierfür nicht angepasst werden.

Der Linker behandelt nur Variablen und Funktionen. Deklarationen von Klassen, Strukturen, exportierten Templates usw. werden nicht gelinkt. Deshalb werden diese Deklarationen im IML-Graph nicht miteinander verbunden

sein, wenn sie in verschiedenen Übersetzungseinheiten vorkommen. Das entspricht der Modellierung des `struct`-Sprachelements in der IML für C und es entspricht auch dem C++ Standard [14, Abschnitt (basic.def.odr)-5]:

> „There can be more than one definition of a class type (_class_), enumeration type (_dcl.enum_), inline function with external linkage (_dcl.fct.spec_), class template (_temp_), non-static function template (_temp.fct_), static data member of a class template (_temp.static_), member function template (_temp.mem.func_), or template specialization for which some template parameters are not specified (_temp.spec_, _temp.class.spec_) in a program provided that each definition appears in a different translation unit, and provided the definitions satisfy the following requirements..."

Falls sich herrausstellt, dass diese Modellierung für ein Analysewerkzeug nicht ausreicht, muss der Linker angepasst werden.

7.2.1.3 cafe und jafe

S.S.

Die Bauhaus-Werkzeuge cafe und jafe übersetzen C bzw. Java Quelltext in die IML. Die in Kapitel 7.2.1.1 angeführten Auswirkungen der IML-Erweiterung auf die Darstellung von C Sprachelementen betreffen zum Teil auch diese beiden Werkzeuge. Sie müssen so angepasst werden, dass sie der erweiterten IML-Spezifikation entsprechende IML-Graphen erzeugen.

Die IML-Erweiterungen für Klassen und für Ausnahmen, die schon bei der Erweiterung der IML für Java Programme durchgeführt wurde, wurde bei der Erweiterung für C++ Programme nochmals geringfügig verändert. jafe muss an diese Änderungen angepasst werden.

Eine dieser Änderungen war, dass das Throw_Statement als Unconditional_Branch eingeordnet wurde. Deshalb besitzt es jetzt das Attribut End_Lifetime, das in Zukunft auch von jafe gesetzt werden muss.

Während der Erweiterung der IML für Java wurde davon ausgegangen, dass IML-Knoten vom Typ O_Node Objekte beschreiben. Bei der Einarbeitung in die IML durch Befragen verschiedener Mitarbeiter zu Beginn dieser Diplomarbeit stellte sich jedoch heraus, dass IML-Knoten vom Typ O_Node Deklarationen beschreiben. Deshalb wurde die bei der Java Erweiterung eingeführte Klasse O_Instance wieder entfernt und einige Attribute zwischen Class, O_Class und T_Class und ihren Unter- oder Oberklassen verschoben. Außerdem wurde die Modellierung der Objekterzeugung verändert. Auch an diese Änderungen muss jafe angepasst werden.

7.2.1.4 iml2html

S.S.

Das Werkzeug iml2html wird fast vollständig mit Hilfe des IML-Generators automatisch aus der IML-Spezifikation erzeugt. Deshalb wurde es an die meisten IML-Erweiterungen automatisch angepasst.

Die von iml2html erzeugte HTML-Darstellung von IML-Graphen enthält Kurzbeschreibungen von IML-Knoten. Die Erzeugung dieser Kurzbeschreibungen ist direkt implementiert, sie wird nicht durch die IML-Spezifikation

gesteuert. Ein Teil der Kurzbeschreibung ist das Attribut Name, das bei der IML-Erweiterung entfernt wurde. Anstatt Name sollte das Attribut Unmangled_Name in Zukunft in der Kurzbeschreibung verwendet werden.

In C wird nicht zwischen Zeigern und Referenzen unterschieden, aber in C++. Deshalb wurde bei der IML-Erweiterung der Knotentyp T_Reference eingeführt. Die iml2html-Kurzbeschreibung von Knoten vom Typ T_Reference ist gleich, wie die von Knoten vom Typ TC_Pointer – sie enthält einen Stern „*". Das ist korrekt, da Knoten vom Typ T_Reference gleichzeitig vom Typ TC_Pointer sind. Es wäre jedoch schöner, wenn anstatt des Sterns das Und-Zeichen „&" verwendet werden würde, da Referenz-Typen in C++ durch „&" anstatt durch „*" gekennzeichnet werden.

7.2.2 Weiterentwicklung von cafe++

S.S.

Auch wenn alle Fehler behoben sind, muss cafe++ gewartet und weiterentwickelt werden. Der Bedarf dafür entsteht einerseits, wenn sich die Anforderungen ändern, beispielsweise weil eine neue Version des C++ Standards oder des EDG-Front-Ends unterstützt werden soll, oder weil sich die IML-Spezifikation ändert. Andererseits kann die Implementierung von cafe++ noch verbessert werden.

Eine mögliche Änderung der IML-Spezifikation betrifft die Modellierung der Vererbungsbeziehung zwischen Klassen durch Extends_Relation. Die Vererbungsbeziehung könnte anzeigen, ob sie virtuell ist oder nicht. „virtuell" heißt, dass bei mehrfacher Abstammung von einer Basisklasse nur eine Instanz jedes Basisklassen-Attributs existiert (siehe [14, Abschnitt (class.mi)]).

Die Implementierung von cafe++ kann weiterhin verbessert werden, indem bestehende Einschränkungen behoben werden. Beispielsweise können Heuristiken verwendet werden, um zwischen verschiedenen Arten von Typumwandlungen zu unterscheiden (siehe Kapitel 4.4).

Auch die Leistungsfähigkeit von cafe++ kann weiter erhöht werden. Dazu gehört z.B. die Speichereffizienz, die durch funktionenweises Abarbeiten der IL erhöht werden kann (siehe Kapitel 4.2.3.2). Zusätzliche Leistungsmerkmale, die in cafe++ realisiert werden können, werden in den folgenden beiden Kapiteln und in Kapitel 3.2.3 beschrieben. Durch die globale Analyse mehrerer Übersetzungseinheiten in einem Lauf von cafe++ könnte auch das in Kapitel 2.3.4.4 behandelte Problem mit der Erzeugung von TC_Struct-Knoten besser gelöst werden.

7.2.3 Unterstützung weiterer Sprachdialekte

S.S.

Das EDG-Front-End kann so konfiguriert werden, dass einige Dialekte von C++, darunter die GNU-Erweiterungen und der Dialekt der Firma Microsoft, unterstützt werden können. Wenn diese Unterstützungen im Front-End aktiviert werden, muss auch das Back-End entsprechend angepasst werden, da der IL-Graph dann neue Elemente enthält. Eine zukünftige Erweiterung von cafe++ könnte die Verarbeitung dieser Elemente im IL-Graphen und dadurch die Unterstützung der entsprechenden C++ Dialekte sein.

7.2.4 Unterstützung weiterer Sprachen

S.S.

Das EDG-Front-End kann neben C++ Quelltexten auch C und Fortran Quelltexte in einen IL-Graphen übersetzen. Dabei werden im Fall von C mehrere Dialekte unterstützt, darunter auch C89 und C99 [12, 13]. Besonders interessant sind dabei C99 und Fortran, da das Bauhaus-Werkzeug cafe nur C89 unterstützt.

Wie die Unterstützung weiterer C++ Dialekte wirkt sich selbstverständlich auch die Unterstützung weiterer Sprachen auf den IL-Graphen aus. Deshalb ist auch hier eine Erweiterung von cafe++ notwendig, falls in Zukunft andere Sprachen mit cafe++ in die IML übersetzt werden sollen.

Die Firma EDG bietet nicht nur ein C/C++/Fortran-Front-End, sondern inzwischen auch ein separates Java-Front-End an. Die Zwischendarstellung, die von diesem Front-End erzeugt wird könnte durch ein ähnliches Back-End wie das von cafe++ nach IML übersetzt werden. Möglicherweise können dabei Teile von cafe++ wiederverwendet werden. Allerdings ist die Entwicklung eines weiteren, auf dem EDG Java-Front-End basierenden IML-Front-Ends vermutlich aufwändiger als die Wartung von jafe und hat deshalb wenig Sinn.

7.2.5 Verbesserung der IML-Dokumentation

S.S.

Die Einarbeitung in die IML ist zur Zeit relativ zeitaufwändig. Die Informationen müssen aus verschiedenen, teilweise veralteten Diplom- und Studienarbeiten, aus der IML-Spezifikation und durch Fragen an Mitarbeiter zusammengesammelt werden. Das einzige Dokument über die IML, das gewartet wird, ist die IML-Spezifikation. Die IML-Spezifikation gibt jedoch nur Auskunft über Detailfragen. Es sollte auch ein Dokument geben, das eine Einführung in die IML enthält und gewartet wird.

7.2.6 Verbesserung des Tests

S.S.

Der Test von cafe++ hat noch einige Schwächen. Dazu gehört, dass die Testdaten einige Teile des Quelltextes und einige Arten von IML-Knoten noch nicht erfassen (siehe Kapitel 5.3). Mit Hilfe der während des Tests ermittelten Quelltext- und IML-Knotenüberdeckung kann der Test weiter verbessert werden, indem neue Testfälle hinzugefügt werden.

Automatischer Vergleich von Soll- und Ist-Daten Der größte Nachteil des Regressionstests ist jedoch, dass beim Regressionstest Soll- und Ist-Daten manuell verglichen werden müssen, falls die entsprechenden Dateien nicht exakt übereinstimmen. Die IML-Dateien zweier gleicher IML-Graphen stimmen jedoch nicht immer exakt überein. Der exakte Inhalt von IML-Dateien ist z.B. abhängig von den Adressen, die die Knoten im Speicher haben. Deshalb sollte der Vergleich von Soll- und Ist-Daten von einem IML-Graphenvergleichswerkzeug durchgeführt werden. Dieses Werkzeug sollte, wenn möglich, nicht nur bestimmen, *ob* sich zwei Graphen unterscheiden, sondern auch *worin* sie sich unterscheiden.

Hauptprogramm

```
compare(rootNode1, rootNode2)
```

Vergleichsfunktion

```
bool compare(node1, node2) {
    // number ist mit 0 initialisiert.
    // Mindestens ein Knoten schon nummeriert
    // und verschieden => fertig.
    if (node1.number != node2.number)
        return false;
    // Beide schon nummeriert und gleich
    // => fertig.
    if (node1.number != 0)
        return true;
    if (node1.typ != node2.typ)
        return false;
    typ = node1.typ; // == node2.typ
    for (attribute in typ) {
        if (attribute is no set or list) {
            if (not vergleich(node1.attribute,
                              node2.attribute))
                return false;
        }
        else if (attribute is list) {
            for (element1 in node1.attribute,
                 element2 in node2.attribute) {
                if (not vergleich(element1,
                                  element2))
                    return false;
            }
        }
        else { // set
            if (nicht gleichlang)
                return false;
            // (*):
            if (keine passende Permutation)
                return false;
        }
    }
    // Vergleich ergab Gleichheit.
    node1.number = node2.number
                 = new_Unique_Number();
    return true;
}
```

Algorithmus 1: Vergleich von IML-Graphen

Ein Vergleichsalgorithmus, der in diesem Werkzeug verwendet werden könnte, ist in „Algorithmus 1" in Pseudocode dargestellt. Der Algorithmus würde in Linearzeit arbeiten, wenn nicht an der mit (*) gekennzeichneten Stelle mehrere Permutationen von Mengenelementen überprüft werden müssten.

Um nur eine Permutation überprüfen zu müssen, könnte man die beiden Mengen sortieren. Leider gibt es kein sinnvolles, eindeutiges Sortierkriterium. Es gibt einige nicht-eindeutige Sortierkriterien, z.B. die bisher vergebenen eindeutigen Nummern oder die Quelltextposition. Eine Sortierung danach könnte zur Beschleunigung des Algorithmus verwendet werden – es müssten dann nur noch die nicht-eindeutig sortierten Teilmengen permutiert werden. Um die Anzahl nummerierter Knoten in der Menge zu erhöhen, und damit die Anzahl der Rest-Permutationen zu verkleinern, können alle durch nicht-Mengen-Kanten erreichbaren Knoten in einem vorgeschalteten Durchlauf nummeriert werden.

Es muss beachtet werden, dass während der Überprüfung der Permutationen keine Nummerierung aufgrund nicht-passender Permutationen durchgeführt werden darf.

Regelbasierter Test Der Test könnte auch dadurch verbessert werden, dass ein regelbasierter Test, wie er in Kapitel 5.1.2 vorgeschlagen wird, durchgeführt wird. In dem regelbasierten Test würde die Korrektheit von Ist-Daten überprüft werden, ohne sie mit Soll-Daten zu vergleichen.

7.2.7 Andere Verbesserungen von Bauhaus

S.S.

Verbessertes imldump Das Werkzeug imldump sollte so erweitert werden, dass es die komplette im IML-Graphen enthaltene Information ausgibt. Dabei sollte es im Gegensatz zu iml2html vollständig automatisch aus der Spezifikation erzeugt werden. Die erzeugte Textdarstellung könnte dann beispielsweise zum Vergleich von Regressionstestdaten verwendet werden, solange kein Graphenvergleichswerkzeug existiert (siehe auch Kapitel 7.2.6).

Eine weitere Anwendung des verbesserten imldump-Werkzeugs kann die Verwendung der erzeugten Textdarstellung als Eingabe für einfache Programme sein, die von der Version der IML-Spezifikation unabhängig sind. Beispielsweise könnte ein Programm mit Hilfe des Werkzeugs `dot` kleine IML-Graphen darstellen, so wie das schon bei kleinen IL-Graphen möglich ist (siehe Kapitel 3.2.1.3).

Möglicherweise wird diese Art der Anwendung jedoch durch ein zur Zeit laufendes Studienprojekt obsolet [17]. In diesem Studienprojekt wird ein Werkzeug zur grafischen Darstellung von IML-Graphen entwickelt. Mit Hilfe des dafür entwickelten Reflection-API können dann auch andere Werkzeuge von der IML-Spezifikation unabhängig entwickelt werden, ohne den IML-Generator jedesmal erweitern zu müssen. Dadurch wird ein verbessertes imldump-Werkzeug einerseits weniger wichtig, andererseits aber auch einfacher realisierbar.

Vereinfachung der IML-SymNode-Hierarchie In der IML-Knotentyphierarchie werden zur Zeit zwei Arten von SymNode-Knoten unterschieden: Knoten des Typs O_Node und des Typs T_Node. O_Node-Knoten repräsentieren Deklarationen und T_Node-Knoten repräsentieren Typen. Deklarationen von benutzerdefinierten Typen werden durch Knoten vom Typ OC_User_Type_Declaration, eine Unterklasse von O_Node, repräsentiert.

Um die IML-Hierarchie und die Modellierung von Typen in der IML einfacher und leichter verständlich zu machen, könnte man sich für fundamentale Typen künstliche Deklarationen vorstellen, also Deklarationen, die in der Programmiersprache eingebaut sind. Etwas ähnliches gibt es bei C++ schon bei Konstruktoren. Der C++ Sprachstandard [14, Abschnitt (special)] schreibt vor, dass vom Übersetzer in bestimmten Fällen Deklarationen oder sogar Definitionen von Konstruktoren erzeugt werden müssen.

Sobald auch für fundamentale Datentypen Deklarationen vorhanden sind, können alle Typen durch Deklarationen dargestellt werden. Das ergibt Sinn, weil ein Typ durch seine Definition vollständig beschrieben wird und eine Definition gleichzeitig eine Deklaration ist. Die Darstellung von Typen wäre dann auch konsistenter zur Darstellung von Objekten, die ja auch durch ihre Deklaration vertreten werden. Abbildung 7.3 gibt eine Übersicht über die veränderte IML-Hierarchie.

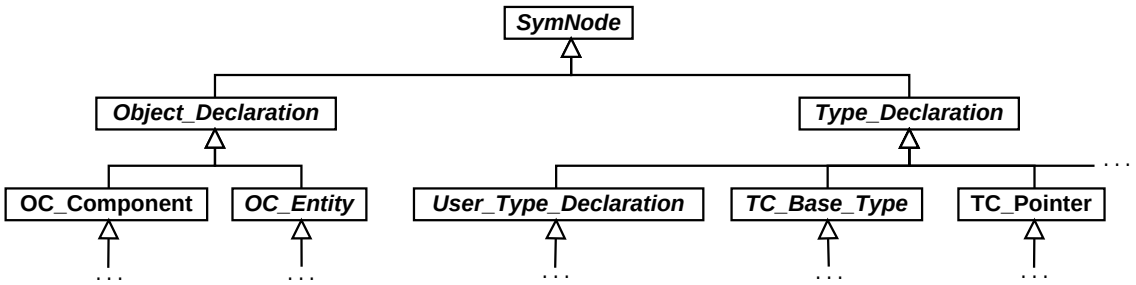


Abbildung 7.3: Vereinfachte IML-Hierarchie bei SymNodes

Der Unterschied zur bisherigen IML-Hierarchie besteht darin, dass OC_User_Type_Declaration und die Klassen zur Beschreibung benutzerdefinierter Typen verschmolzen wurden zu der Klasse User_Type_Declaration. Zusätzlich wurde T_Node in Type_Declaration umbenannt, um deutlich zu machen, dass alle Typen durch ihre Deklarationen dargestellt werden.

Die User_Type_Declaration-Klassen übernehmen jetzt die Aufgabe der früheren OC_User_Type_Declaration-Klassen und deren parallel-Klassen unter T_Node. Deshalb müssen sie die Attribute beider Teile übernehmen.

Die Benutzung eines Typs könnte als eine implizite Deklaration des Typs angesehen werden, wodurch die TC_User_Type_Name-Klassen nicht mehr benötigt werden. Diese implizite Deklaration würde dann auf die Definition des Typs verweisen, genau wie jede andere Deklaration auch auf ihre Definition verweist. Ein Nachteil dieser Modellierung ist, dass die Darstellung der Benutzung eines benutzerdefinierten Typs mehr Speicher benötigt als bisher. Falls das vermieden werden soll, werden weiterhin TC_User_Type_Name-

Knoten benötigt, die nur eine Quelltextposition und einen Verweis auf die eigentliche Typdeklaration enthalten. Noch speichersparender wäre der Verweis auf irgendeine Deklaration beim Benutzen eines Typs. Das ist bisher nicht möglich, da Typen und Typdeklarationen verschieden dargestellt werden – durch T_Nodes und O_Nodes. Falls jedoch bei der Benutzung eines Typs direkt auf eine bestehende Deklaration verwiesen wird, ohne einen Knoten dazwischenzuschalten, kann keine Quelltextposition mehr gespeichert werden.

7.2.8 Neue Analysen

S.S.

Durch die Erweiterung der IML für C++ Programme kann in IML-Graphen zu C++ Quelltexten mehr Information über den Entwurf des zu analysierenden Programms dargestellt werden als das bei C Quelltext möglich war. Vorschläge, wie neue Analysewerkzeuge aus dieser Information Nutzen ziehen können, werden weiter unten in diesem Kapitel gemacht.

Die Vorschläge aus [19, Kapitel 7.1.4 „Unterstützung des Software-Reengineering“] sind bei der Analyse von C++ Programmen genausogut umsetzbar wie bei der Analyse von Java Programmen. Weitere vorstellbare Analysen werden in den folgenden Absätzen beschrieben.

Auswirkungen von Nachrichten an Objekte (Methodenaufrufe) C++ unterstützt die Objektorientierte Programmierung (OOP). Bei der OOP werden Nachrichten an Objekte gesandt. Um ein Programm zu verstehen, wäre es hilfreich, die Auswirkungen einer Nachricht an ein Objekt zu kennen. Beispielsweise wäre es interessant, ob die Nachricht weitere Nachrichten an andere Objekte auslöst oder ob nur das eine Objekt betroffen ist. Falls weitere Nachrichten an andere Objekte ausgelöst werden, könnte auch die Menge aller solcher Objekte bestimmt werden.

Auch die Frage, ob eine Nachricht den Zustand eines Objekts verändert, könnte durch eine Analyse beantwortet werden. Falls eine Methode als konstant deklariert ist, kann man sicher sein, dass das Objekt nicht verändert wird. Das kann ausgenutzt werden, um eine entsprechende Analyse effizienter zu machen. Eine Methode, die nicht als konstant deklariert wurde, muss aber den Zustand des Objektes nicht verändern. Es kann sein, dass der Programmierer vergessen hat, die Methode als konstant zu deklarieren. Auch bei einer Methode, die nicht konstant deklariert werden darf, weil mit Hilfe des Rückgabewertes schreibend auf das Objekt zugegriffen werden kann, möchte man möglicherweise wissen, ob die Methode selbst das Objekt verändert.

Alle diese Analysen können nicht nur auf Methoden durchgeführt werden, sondern auch auf normalen Funktionen, die ein Objekt als Parameter bekommen. Wenn der Parameter als Zeiger oder Referenz auf ein konstantes Objekt deklariert ist, entspricht das einer als konstant deklarierten Methode – da zeigt der this-Zeiger auf ein konstantes Objekt.

Aufrufe virtueller Methoden Beim Aufruf virtueller Methoden findet im Normalfall eine Bindung zur Laufzeit statt, das heißt, erst beim Methodenaufruf zur Laufzeit wird festgelegt, welche Methode wirklich ausgeführt wird.

Eine Analyse könnte versuchen, die Menge der Methoden, die potentiell aufgerufen werden können, einzuschränken. Falls sich dabei herausstellt, dass nur eine einzige Methode aufgerufen werden kann, liegt möglicherweise ein Entwurfsfehler vor oder die Methode sollte nicht virtuell sein.

Bei virtuellen Methodenaufrufen kann die Menge der Methoden einfacher eingeschränkt werden als bei Funktionsaufrufen über Funktionszeiger. Es können nur die verschiedenen Versionen der Methode in allen Ober- und Unterklassen der Klasse des Objekts betroffen sein.

Klonerkennung Bei der Klonerkennung dürfen Instantiierungen von Templates nicht als Klone voneinander erkannt werden – sie wurden mit Absicht automatisch aus dem Template „geklont", um echte Klone im Quelltext zu vermeiden. Falls jedoch eine normale Funktion das Gleiche macht wie die Instantiierung eines Templates, sollte diese normale Funktion als Klon erkannt werden.

Mit Hilfe der IML-Erweiterung für Templates könnte das Ergebnis der Klonerkennung in einem neuen IML-Graphen mit Templates dargestellt werden. Dadurch könnten nachfolgende Analysen das Ergebnis der Klonerkennung verwenden.

Literaturverzeichnis

[1] Ada Core Technologies, Inc.: *GNAT, The GNU Ada 95 Compiler*, Document revision level 1.1.2.2, GNAT Version 3.16w. Free Software Foundation 2002

[2] AT&T Labs-Research: *dot; Graphviz - open source graph drawing software*. AT&T: http://www.research.att.com/sw/tools/graphviz/

[3] Balzert, Helmut : *Lehrbuch der Software-Technik: Software-Entwicklung (Band 1)*, 1. Auflage. Heidelberg; Berlin; Oxford: Spektrum Akademischer Verlag 1996

[4] Dean, Thomas R.; Malton, Andrew J; Holt, Ric: *Union Schemas as a Basis for a C++ Extractor*. Proceedings of the Eighth Working Conference on Reverse Engineering (WCRE'01), 2001.

[5] Devanbu, Prem; Eaves, L. E.: *The GEN++ Manual*, Draft. AT&T 1994.

[6] Edison Design Group: *C++ Front End Internal Documentation*, Version 3.0, Proprietary Information of Edison Design Group 2002
Das zweite Kapitel „External Interface" ist auch über
http://www.edg.com verfügbar.

[7] Eisenbarth, Thomas; Koschke, Rainer; Plödereder, Erhard; Girard Jean-Francois; Würthner Martin: *Projekt Bauhaus: Interaktive und inkrementelle Wiedergewinnung von SW-Architekturen*. Workshop Software-Reengineering. Bad Honnef, Universität Koblenz-Landau: Fachberichte Informatik, Nr. 7-99, S. 17-S. 26 (1999)

[8] Ferenc, Rudolf; Beszédes, Árpád: *Data Exchange with the Columbus Schema for C++*. Sixth European Conference on Software Maintenance and Reengineering. Budapest (Ungarn), 2002

[9] Frühauf K.; Ludewig J.; Sandmayr H.: *Software-Projektmanagement und -Qualitätssicherung*, 3. Auflage. Zürich: vdf Hochschulverlag 1999

[10] Goldberg, Adele; Robinson, David: *Smalltalk-80 The Language and its implementation*. Xerox Palo Alto Research Center: Addison-Wesley 1983

[11] IEEE: *IEEE Std 1003.1-2001 (POSIX)*. 2001

[12] International Standard: *Programming Languages - C*. ANSI/ISO/IEC 9899:1990

[13] International Standard: *Programming Languages - C*. ANSI/ISO/IEC 9899:1999

[14] International Standard: *Programming Languages - C++*, ANSI/ISO/IEC 14882:1998(E) in ASC X3, first edition. American National Standards Institute: September 1998, ISO/IEC JTC 1

[15] Internetseite: *Richtlinien für Studien-, Diplomarbeiten und Fachstudien*. http://www.informatik.uni-stuttgart.de/fakultaet/studienberatung/richtlinien/richtlinien.html

[16] Internetseite: *Projekt Bauhaus*. http://www.bauhaus-stuttgart.de/

[17] Internetseite: *Studienprojekt A: IML-Browser*. http://www.informatik.uni-stuttgart.de/ifi/ps/Lehre/IML_Browser/

[18] Kern, Achim: *RFG-Generator für Ada 95 Source-Code*. Studienarbeit Nr. 1832, Universität Stuttgart: Abteilung Programmiersprachen und Übersetzerbau 2002

[19] Knauß, Markus: *Erweiterung und Generierung der Zwischendarstellung IML für Java-Programme*. Diplomarbeit Nr. 2006, Universität Stuttgart: Abteilung Programmiersprachen und Übersetzerbau 2002

[20] Knauß, Markus: *Tests for the interfacing facility of Ada 95 to C++*. Universität Stuttgart: Abteilung Programmiersprachen und Übersetzerbau, CVS:bauhaus/Tools/jafe/src/interfacing_ada95_cpp 30.09.2002

[21] Koschke, Rainer: *Diplomarbeit: Erweiterung und Generierung der Zwischendarstellung IML für C++-Programme*. Universität Stuttgart: Abteilung Programmiersprachen und Übersetzerbau 2002

[22] Malloy, Brian A. et al: *Testing C++ Compilers for ISO Language Conformance*. Dr. Dobbs Journal. Juni 2002

[23] Myers, Glenford J.: *The Art of Software Testing*. New York: John Wiley and Sons: 1979

[24] N.N.: *Ada 95 Quality and Style: Guideline for Professional Programmers*. Software Productivity Consortium: Oktober 1995

[25] N.N.: *GNU Compiler Collection (GCC) Internals*, GCC 3.2. Free Software Foundation 2002

[26] N.N.: *Prüfungsordnung der Universität Stuttgart für den Diplomstudiengang Informatik*. Universität Stuttgart: Fakultät Informatik 2002

[27] N.N.: *Prüfungsordnung der Universität Stuttgart für den Diplomstudiengang Softwaretechnik vom 10. Juni 1997 mit eingearbeiteter Revision zum 1. Oktober 2000*. Universität Stuttgart: Fakultät Informatik 2000

[28] N.N.: *Setup and User's Guide to Columbus/CAN*, Version 3.1. FrontEndART Ltd 2002

[29] Object Modelling Group (OMG): *Unified Modelling Language Specification*, Version 1.4, September 2001. http://www.omg.org/uml/

[30] Rohrbach, Jürgen: *Erweiterung und Generierung einer Zwischendarstellung für C-Programme.* Studienarbeit Nr. 1662, Universität Stuttgart: Abteilung Programmiersprachen und Übersetzerbau 1998

[31] Stroustrup, Bjarne: *Die C++ Programmiersprache*, 2. Auflage, Addison Wesley, 1992

[32] Würthner, Martin: *Entwurf und Implementierung einer Interndarstellung für die Analyse von Ada Programmen*. Studienarbeit Nr. 1567, Universität Stuttgart: Abteilung Programmiersprachen und Übersetzerbau 1996

Anhang A

Glossar

S.S.

In diesem Kapitel werden alle besonderen Begriffe, die in dem Projekt von Bedeutung sind, erklärt und, wenn möglich, definiert. Die Begriffe werden in Deutsch und Englisch angegeben, weil diese Ausarbeitung deutsch, der Quelltext der Software und dessen Kommentare englisch sind.

Jeder Begriff wird in einem Absatz erklärt. Die deutsche und die englische Version sind durch Schrägstrich getrennt fett gedruckt. Falls der Begriff in einer Sprache fehlt, wird in der deutschen und englischen Dokumentation derselbe Begriff verwendet. Verschiedene Bedeutungen sind durchnummeriert, die wichtigeren zuerst. Der Kontext der Bedeutung steht in Klammern an erster Stelle. Verweise auf andere Begriffe sind *kursiv* und vorzugsweise deutsch.

Die Begriffe sind alphabetisch (lexikalisch) sortiert (nach dem deutschen Begriff, falls vorhanden).

Abstrakter Syntaxbaum / abstract syntax tree (AST) (Übersetzerbau) Ein abstrakter Syntaxbaum ist eine hierarchische Darstellung der syntaktischen Dekomposition eines Programms, die von den unnötigen Details der zugrunde liegenden Grammatik der Programmiersprache abstrahiert.

Attribut / attribute siehe *OO*.

anlegen / allocate siehe *Ressourcenverwaltung*.

aufräumen / finalize siehe *Ressourcenverwaltung*.

Ausnahmen / exceptions (C++) Sprachelement von C++ für die Behandlung von Fehlern.

Back-End / back-end (Übersetzerbau) Generator der Ausgabe.

Bindekonvention / linkage (C++) Aufrufkonvention von Funktionen. Die Bindekonvention ist relevant, wenn Übersetzungseinheiten verschiedener Programmiersprachen zu einem Programm zusammengebunden werden. Bei einer C++ Funktion kann mit „`extern "C"`“ angegeben werden, dass dieselbe Bindekonvention wie für C Funktionen verwendet werden soll.

- / Black-Box-Test (Test) Beim Black-Box-Test wird der Quelltext der Software für den Test nicht verwendet.

Definition / definition (C++) Belegung eines Namens mit einer Bedeutung. Eine Definition ist gleichzeitig eine *Deklaration*.

Deklaration / declaration (C++) Bekanntmachung eines Namens.

Element / member (C++, OO) *Attribut* oder *Methode* einer *Klasse*.

erweiterter Name / mangled name (Übersetzerbau) Der eindeutige Name einer Funktion oder Variable, der beim Zusammenlinken von Übersetzungseinheiten zur Identifikation von Funktionen und Variablen verwendet wird. Er enthält zusätzlich zu dem in der Deklaration angegebenen Namen weitere Teile (siehe Kapitel 3.2.1.8).

erzeugen / create siehe *Ressourcenverwaltung*.

freigeben / free siehe *Ressourcenverwaltung*.

Front-End / front-end 1. (Übersetzerbau, cafe++) Analyse-Subsystem. Führt die syntaktische und semantische Analyse von Quelltext durch und gibt das Ergebnis an ein Back-End weiter.
2. (Software) Subsystem der Schnittstelle zur Umgebung, üblicherweise zum Benutzer. Dies umfasst im weitesten Sinne auch Bedeutung 1.

Funktion / function (C/C++) Synonym für Unterprogramm oder Routine. In anderen Programmiersprachen sind Funktionen Unterprogramme, die Werte zurückgeben. In C/C++ hat jedes Unterprogramm einen Rückgabetyp, deshalb wird jedes Unterprogramm Funktion genannt.

- / Glass-Box-Test (Test) Beim Glass-Box-Test wird der Quelltext der Software für den Test mit herangezogen.

Halde / heap (Informatik) Speicherbereich, der in beliebiger Reihenfolge dynamisch Daten aufnehmen und wieder freigeben kann.

Header / header (C/C++) Auch: Header-Datei. Eine Datei, die Deklarationen, Makros und anderen C++ Quelltext enthält und mit `#include` in mehrere Übersetzungseinheiten eingebunden werden kann. Die Namenserweiterung der Dateien ist üblicherweise „`.h`".

Hierarchischer Programmgraph (HPG) / Hierarchical Program Graph (IML) Ein Teilgraph des *IML-Graphen*, der Baumform hat.

- / IL (EDG) Intermediate Language. Eine Zwischendarstellung für Quelltexte der Firma EDG. Auch: IL-Graph. Der IL-Graph ist ein AST mit zusätzlichen semantischen Kanten.

- / IML (Bauhaus) Intermediate Language. Eine Zwischendarstellung für Quelltexte im Bauhaus-Projekt (siehe Kapitel 2.1.1).

IML-Analysen / IML analyses (Bauhaus) Analysewerkzeuge, die auf bereits erzeugten *IML-Graphen* arbeiten.

IML-Graph / IML graph (Bauhaus) Der IML-Graph ist die *IML*-Darstellung eines Programmes. Der Graph enthält eine hierarchische Darstellung des Programmes, einen Deklarations- und einen Typteil.

IML-Klasse / IML class (IML) Wird synonym verwendet mit IML-Knotentyp, insbesondere wenn die Vererbungsbeziehungen von IML-Klassen behandelt werden.

initialisieren / initialize siehe *Ressourcenverwaltung*.

Instantiierung / instantiation 1. (Klassen-) Erzeugen eines Objekts.
2. (Template-) Erzeugen einer Klasse oder Funktion.

Ist-Daten / actual data (Software-Test) Tatsächliche ausgabedaten eines getesteten Programms.

Klasse / class siehe OO.

Knoten / Node (IML) Knoten des *IML-Graphen.*

Knotentyphierarchie / hierarchy of classes of nodes (IML) Klassenhierarchie der Typen von Knoten des *IML-Graphen*.

K&R / K&R Kerninghan und Ritchie entwickelten die Programmiersprache B weiter zu C. Die erste Version von C ist nach ihnen benannt. Weitere, ANSI-standardisierte Versionen werden C89 und C99 genannt, nach dem Jahr ihrer Standardisierung [12, 13].

LOC, NCLOC (Software-Metriken) LOC (lines of code) ist ein Größenmaß für Quelltexte und bezeichnet die Anzahl der Quelltextzeilen. NCLOC (non-commented lines of code) bezeichnet die Anzahl von Quelltextzeilen, die nicht leer und nicht ausschließlich Kommentare darstellen.

Merkmal / feature (Software) Ausgeprägte Eigenschaft eines Programms.

Methode / method (OO) Auch Nachricht/message. Mit einem Objekt verbundene ausführbare Einheit.

Modellierung / - (cafe++) Darstellung von C++ Sprachkonstrukten in IML-Graphen

Namensbereich / namespace (C++) Explizit oder implizit definierter Sichtbarkeitsbereich für Deklarationen aller Art. In der Java Programmiersprache existiert mit den sog. Packages ein ähnliches Konzept für explizit definierte Namensbereiche.

Objektorientierung (OO) / object orientation (Informatik) Ein Paradigma bei der Softwareentwicklung.

Primäre Deklaration / primary declaration (IL) siehe Kapitel 3.2.1.7.

Qualitätssicherung / quality assurance (QA) (Softwareentwicklung) Maßnahmen zur Sicherung der Produktqualität.

Ressourcenverwaltung / resource management (Softwareentwicklung) Die Aufgabe der Ressourcenverwaltung ist es, sicherzustellen, dass der Lebenszyklus von Ressourcen korrekt durchlaufen wird: *anlegen → initialisieren → aufräumen → freigeben.* Das *Erzeugen* schließt das *Anlegen* und *Initialisieren* mit ein. Die *Zerstörung* umfasst dagegen die Schritte *Aufräumen* und *Freigeben*. Falls die Ressource ein Objekt im Speicher ist, ist das Anlegen des Objekts gleichbedeutend mit der Reservierung des Speichers und die Freigabe des Objekts ist gleichbedeutend mit der Freigabe des Speichers.

- / Ressource Flow Graph (RFG) (Bauhaus) Eine weitere Zwischendarstellung des Bauhaus-Projekts, neben der *IML*. Der RFG gibt eine grobe Übersicht über ein System, während die IML den Quelltext detailliert beschreibt.

- / RTTI (C++) Run-Time Type Information, Typinformation zur Laufzeit

- / scaffolding Bau eines Gerüsts um eine Datenstruktur mit Hilfe zusätzlicher Datenstrukturen.

Template / template (C++) Heißt in anderen Programmiersprachen generischer Typ bzw. generisches Unterprogramm.

- / Shellscript (Unix) Datei, die von der Unix-Shell abgearbeitet wird, wie wenn der Inhalt der Datei direkt in der Kommandozeile eingegeben worden wäre.

Sichtbarkeitsbereich / scope (Quelltext) Bereich, in dem die *Deklaration* eines Namens gültig ist.

Signatur / signature (C++) Name und Typliste einer *Funktion* oder *Methode*. Die Typliste enthält die Typen der Parameter. In anderen Programmiersprachen kann die Typliste auch den Rückgabetyp enthalten.

Soll-Daten / nominal data (Software-Test) Gewünschte Ausgabedaten eines zu testenden Programms.

Standardwert / default (value) 1. (C++ Funktionen, C++ Templates) Wert, der bei der Deklaration eines formalen Parameters angegeben wird und der als aktueller Parameter verwendet wird, wenn kein anderer Wert beim Aufruf bzw. bei der Instantiierung angegeben wird.
2. (EDG-Front-End Aufruf) Wert, der bei der Konfiguration des Front-Ends angegeben wird und der verwendet wird, wenn kein anderer Wert auf der Kommandozeile angegeben wird.

Struktur / struct (C++) Benutzerdefinierte Datenstruktur mit benannten Attributen. Wird in anderen Programmiersprachen „Record" genannt.

Stumpf / stub (Quelltext) Stubs sind vollständig definierte, aber nicht implementierte *Funktionen* und *Methoden*. Die *Definition* umfasst den Namen, alle Parameter und den Rückgabewert einer Funktion und deren Beschreibung als Kommentar. Die Implementierung ist jedoch nur soweit vorhanden, dass der Stub vom *Übersetzer* problemlos verarbeitet werden kann.

Übersetzer / compiler (Informatik) Programm zum Übersetzen einer formalen Sprache in eine andere formale Sprache. Üblicherweise wird ein Programm, formuliert in einer höheren Programmiersprache, in eine Maschinensprache übersetzt.

Übersetzungseinheit / translation unit (C++) Kleinster Teil eines Programms, der durch einen Übersetzungsvorgang übersetzt werden kann. Üblicherweise eine Datei mit der Endung „`.c`" und die davon eingebundenen *Header-Dateien*.

unwahr / false (Logik) Boolescher Wahrheitswert, oft auch „0" genannt. Gegenteil von „wahr".

Variante / union (C++) Benutzerdefinierte Datenstruktur mit benannten Attributen. Im Gegensatz zur *Struktur* ist jedoch zu jedem Zeitpunkt maximal eins der Attribute gültig. Dadurch kann der Speicher für die Attribute mehrfach verwendet werden.

wahr / true (Logik) Boolescher Wahrheitswert, oft auch „1" genannt. Gegenteil von „unwahr".

Werkzeug / tool (Informatik) Software, die beim Erledigen einer Aufgabe hilft.

zerstören / destroy siehe *Ressourcenverwaltung*.

Zweitrangige Deklaration / secondary declaration (IL) siehe Kapitel 3.2.1.7.

Anhang B

Beispiel „Hello World”

S.S.

Zur Veranschaulichung der IML soll in diesem Anhang ein Beispiel für eine vollständige IML-Darstellung gegeben werden. Das Beispielprogramm wurde bewusst kurz gewählt, um den Graphen klein zu halten. Das Beispielprogramm lautet wie folgt:

```
class World {
public:
    World();
    virtual void hello() const = 0;
private:
    World(const World& other);
    const World& operator=(const World& other);
};

class Sub_World: public World {
public:
    virtual void hello() const;
};
```

In den Abbildungen B.1 und B.2 ist der IML-Graph zu einem Beispielprogramm als UML-Diagramm dargestellt. Es sind alle Knoten und die wichtigsten Kanten enthalten. Die Kanten für die Attribute Unresolved_Declarations, Symbol_Table, Declaration_Table und Its_Class wurden ausgelassen. Die Kanten wurden aus Platzgründen nicht beschriftet.

Um die fehlenden Kantenbeschriftungen auszugleichen, werden im Folgenden einige Kanten erklärt. Wenn in Klammern „:IML-Klasse” steht, bedeutet das, dass der beschriebene Knoten zu der Klasse (oder einer Unterklasse) gehört.

Von den Deklarations-Knoten von Funktionen (:OC_Routine) gehen Its_Type-Kanten nach rechts zum Typ der Funktion (:TC_Routine). Return_Type Kanten gehen von der Deklaration und vom Typ der Funktion zuerst nach unten, dann nach links. Params-Kanten gehen von der Deklaration und vom Typ der Funktion zuerst nach unten, dann nach rechts.

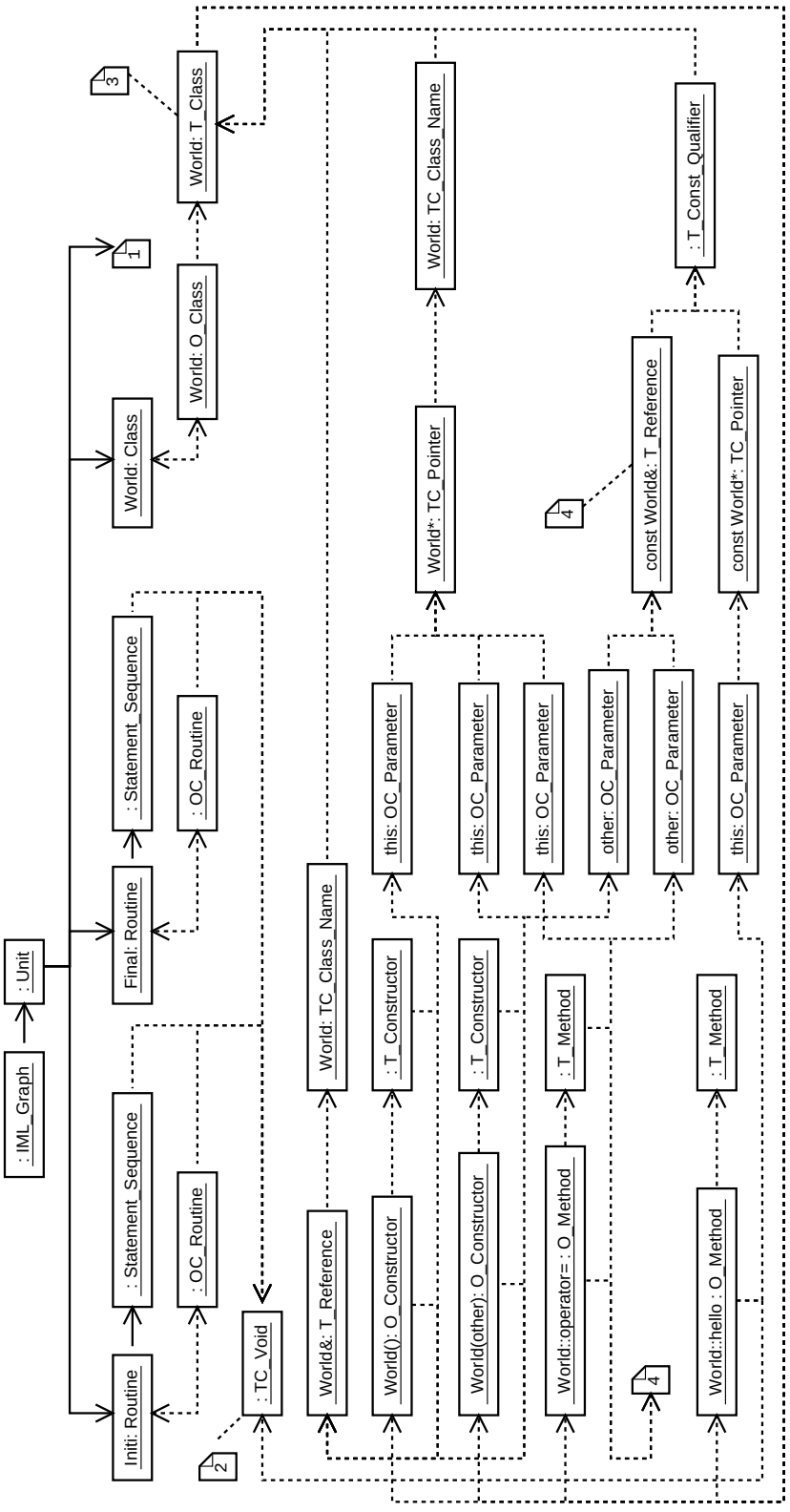


Abbildung B.1: Teil 1 des IML-Graphen zum Hello-World Programm

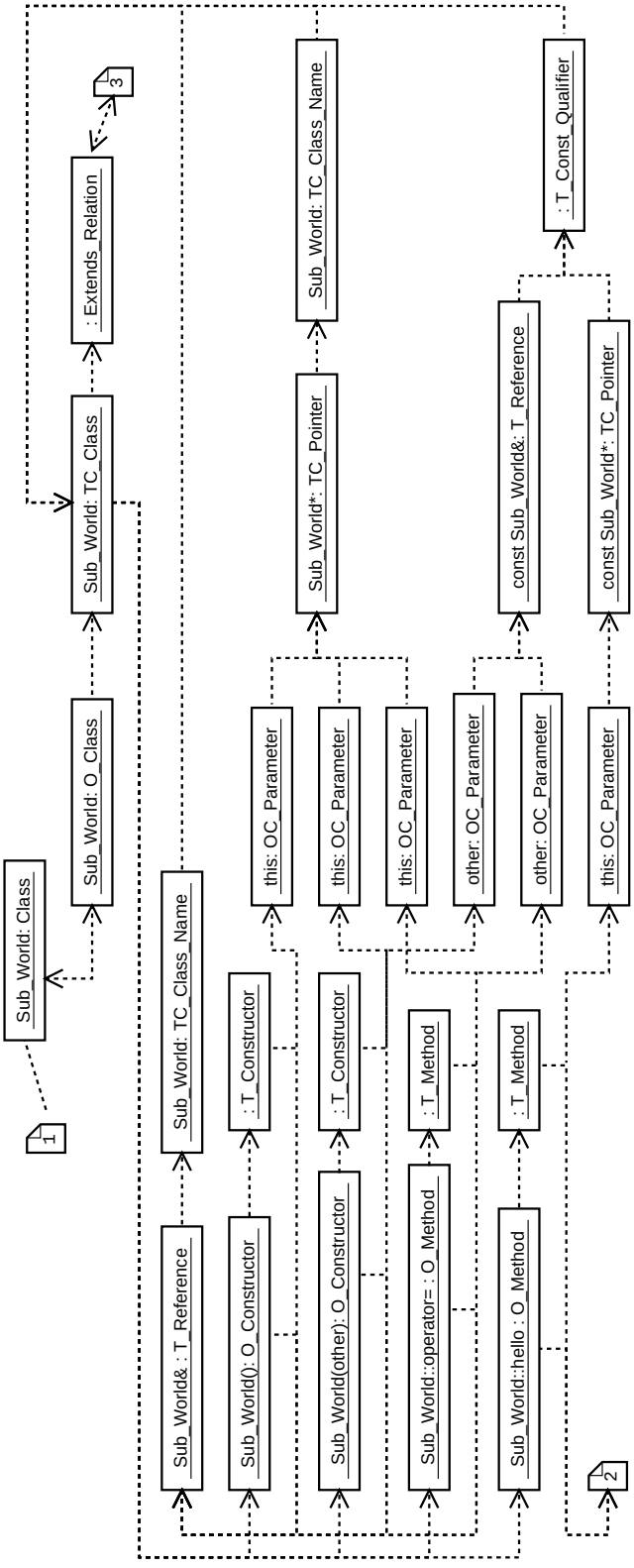


Abbildung B.2: Teil 2 des IML-Graphen zum Hello-World Programm

Von Klassen-Typen (:T_Class) gehen verschiedenartige Kanten zu Methodendeklarationen (:OC_Routine). Zur Deklaration der Methode `hello` (:O_Method) geht eine Methods- und eine Virtual_Call_Table-Kante, es ist jedoch nur eine Kante sichtbar. Zu der Deklaration des Zuweisungsoperators (:O_Method) geht nur eine Methods-Kante. Zu den Deklarationen der Konstruktoren (:O_Constructor) gehen Constructors-Kanten.

Anhang C

Erklärung

Hiermit versichern wir, diese Arbeit selbständig verfasst und nur die angegebenen Quellen benutzt zu haben.

Tahir Karaca

Sebastian Setzer