

Studiengang: Informatik
Prüfer: Prof. Erhard Plödereder
Betreuer: Dr. Rainer Koschke

begonnen am: 12. Dezember 2002

beendet am: 3. Juli 2003

CR-Klassifikation: D.2.6, F.3.3, D.1.5

Studienarbeit Nr. 1879

Einbindung einer Skriptsprache für Gravis

Florian Festi

ISTE

Zusammenfassung

Für das Reengineering-Werkzeug Gravis soll eine Benutzerschnittstelle entwickelt werden, mit der Analysen automatisiert und erstellt werden können. Dazu wurde eine Skriptsprache — Python — ausgewählt. Es wird ein Binding nach Python vorgenommen und der Pythoninterpreter in Gravis eingebettet. Hauptaufgabe dabei ist es, die in Ada geschriebene RFG-Bibliothek nach Python zu exportieren. Dazu muss jedoch zuerst ein Binding nach C realisiert werden. Zudem wird über der RFG-Bibliothek eine benutzerfreundliche Schnittstelle definiert.

Inhaltsverzeichnis

1	Einleitung	4
1.1	Hintergrund	4
1.2	Aufgabenstellung	4
2	Lösungsansätze	4
2.1	Graphische Benutzerschnittstelle	4
2.2	Implementation einer Programmiersprache	5
2.3	Einbinden einer existierenden Programmiersprache	5
3	Analyse der zu exportierenden Codeteile	5
3.1	Statistik	5
3.2	Datentypen	6
3.3	Packages	7
3.4	Architektur	8
3.5	Grobentwurf der Schnittstelle	10
3.6	Einbindung in Gravis	11
4	Kriterien für die Auswahl der Programmiersprache	11
4.1	Paradigma	12
4.2	Garbage Collection	12
4.3	Skripting	12
4.4	Look and Feel	13
4.5	Verbreitung	13
4.6	Lesbarkeit und Erlernbarkeit	14
4.7	Erweiterbarkeit und Einbettbarkeit	15
4.8	Plattformunabhängigkeit	15
4.9	Gtk Binding	15
4.10	Lizenz	15
5	Analyse verschiedener Programmiersprachen	15
5.1	Basic	16
5.2	Java	16
5.3	Perl	16
5.4	Php	17
5.5	Python	17
5.6	Ruby	18
5.7	Tcl	18
5.8	Auswahl	19
6	Die Python-Schnittstelle	19
6.1	RFG	21
6.2	View	22
6.3	Prädikate	23
6.4	Edge und Node	23
6.5	Edge_Attribute und Node_Attribute	24
6.6	Node_Set und Edge_Set	24
6.7	Gravis	25
6.8	Analysen	26

6.9	Beispiele	26
7	Binding nach C	27
7.1	Prozeduren und Funktionen	27
	7.1.1 Overloading	28
	7.1.2 Pakete	28
7.2	Callbacks	28
7.3	Datentypen	29
7.4	Binden von Adabinärdateien	30
7.5	Dispatching	30
	7.5.1 Dispatching in Ada	31
	7.5.2 Dispatching in C++	31
	7.5.3 Dispatching in Python	32
7.6	Generics	32
	7.6.1 Generisch implementierte Funktionen	32
	7.6.2 Generisch implementierte Pakete	33
	7.6.3 Generische Schnittstellen	34
	7.6.4 Funktionen mit generischen Funktionsparametern	34
7.7	Der C-Binding Generator	35
8	Binding nach Python	37
8.1	SWIG	38
8.2	Das SWIG Python Backend	39
8.3	Python/C API	39
	8.3.1 Speicherverwaltung	39
8.4	Shadow Classes	40
8.5	Die %extend Direktive	40
8.6	Typemaps	41
	8.6.1 Typemaps im RFG-Binding	42
8.7	.RFG und .View	43
8.8	Callbacks	44
9	Die Implementierung	45
9.1	Ada Wrapper	46
9.2	Ada-C Binding	46
9.3	SWIG Python-Bindung	46
10	Verlauf der Arbeit	47
11	Fazit und Ausblick	47
11.1	Implementierung	47
11.2	Programmiersprache	48
11.3	Weiterführende Arbeiten	48
11.4	Die RFG-Bibliothek	48

1 Einleitung

1.1 Hintergrund

Bauhaus ist ein Projekt zur Analyse von C-Quellcode. Ein wesentliches Zwischenergebnis dieser Analyse ist der RFG — der Resource Flow Graph. Er besteht aus Knoten, die z.B. Typen, Funktionen, Parameter und Konstanten darstellen, und Kanten, die die Beziehungen wie z.B. "ist vom Typ", "ruft auf" usw. darstellen. Im RFG werden nur statische Aspekte des analysierten Programms repräsentiert.

Gravis ist ein Programm aus der Bauhaus-Suite, mit dem RFGs betrachtet und weiter analysiert werden können. Zu Beginn der Arbeit erlaubte es die händische Manipulation von Knoten und Kanten und die Durchführung ausgewählter Analysen. Da die Zahl der Knoten und Kanten für größere Projekte im Bereich der Tausenden liegt, sind diese Interaktionsmöglichkeiten natürlich ungenügend. Deshalb soll im Rahmen dieser Studienarbeit eine Schnittstelle entwickelt werden, mit der der Benutzer von Gravis selbst neue Analysen erstellen und den RFG manipulieren kann.

1.2 Aufgabenstellung

In dieser Studienarbeit soll eine vorhandene Skriptsprache in Gravis eingebettet werden, die es dem Benutzer ermöglicht eigene Programme zur Manipulation der Graphen in Gravis zu schreiben. Die Funktionalität unserer Graphenbibliothek soll für diese Skriptsprache exportiert werden. Die vom Benutzer geschriebenen Skripte sollen in Gravis aufgerufen werden können. Sofern in den Skripten nicht auf Funktionalität von Gravis Bezug genommen wird (mit anderen Worten, nur Elemente der Graphen-Bibliothek werden verwendet), sollen die Skripte auch in einem Batch-Betrieb ablauffähig sein. In den Skripten muss auch auf die in Gravis aktuell selektierten oder markierten Knoten und Kanten Bezug genommen werden können. Sofern realisierbar, sollen in den Skripten auch Interaktionselemente von Gravis angestossen werden können (z.B. File-Selektor, Filter für Knoten/Kanten etc.).

Die Bewertung existierender Skriptsprachen und die Auswahl einer geeigneten Skriptsprache ist Teil der Aufgabenstellung.

2 Lösungsansätze

2.1 Graphische Benutzerschnittstelle

Die erste Idee, die als Lösungsansatz im Raum stand, war die Analysen und weitere Operatoren analog zur Darstellung des RFGs graphisch darzustellen und aus ihnen per Drag-and-Drop kompliziertere Analysen zusammenstellen zu können. Ähnliche Benutzerschnittstellen werden für Datenvisualisierungstools verwendet und da Gravis im wesentlichen ein Werkzeug zur Visualisierung von RFGs ist, lag dieser Ansatz nahe. Bei näherer Betrachtung zeigten sich aber rasch einige Probleme und Schwachpunkte: Der RFG besteht aus einer größeren Zahl an Datentypen, so dass die Operatoren typisierte Ein- und Ausgänge benötigen, was sowohl die Zahl der Operatoren in die Höhe treibt, als auch das beliebige Kombinieren von Operatoren zunichte macht. Zudem war zweifelhaft,

dass es einen Satz an Operatoren gibt, der den benötigten Funktionsumfang abdeckt, ohne dabei eine komplette Programmiersprache zu emulieren oder dem Benutzer die Möglichkeit zu bieten, eigene Operatoren in einer zusätzlich zur Verfügung zu stellenden Programmiersprache zu implementieren. Aufgrund des daraus resultierenden Risikos und der Tatsache, dass solche Benutzerschnittstellen nur an wenigen Stellen erfolgreich eingesetzt werden, haben wir beschlossen, dem Benutzer gleich eine komplette Programmiersprache an die Hand zu geben.

2.2 Implementation einer Programmiersprache

Nun gab es die Möglichkeit, eine geeignete Sprache selbst zu entwerfen und zu implementieren oder auf eine existierende Sprache zurückzugreifen. Eine Eigenentwicklung hätte den Vorteil einer höheren Flexibilität und damit einer besseren Integration der RFG-Funktionalität in die Sprache. Als Nachteile schlagen ganz klar der höhere Implementierungsaufwand zu buche. Daraus folgt direkt als weiterer Nachteil, dass sich bei einer eigenentwickelten Sprache nur ein Minimalumfang an Sprachfeatures implementieren lässt. Dazu kommt noch, dass sich für diese Aufgabe eigentlich eine prozedural/objektorientierte Sprache anbietet, für die die Entwicklung eines Interpreters — im Vergleich zu einer funktionalen Sprache — besonders schwierig ist. Zudem würde dieser Interpreter als Teil des Bauhausprojekts weiter gepflegt werden müssen.

2.3 Einbinden einer existierenden Programmiersprache

Aber auch die Verwendung einer existierenden Sprache ist nicht ganz ohne Nebenwirkungen: Zwar gibt es für eine ganze Reihe von Sprachen Binding-Generatoren und andere Tools, die die Integration von fremdem Code in die Sprache sehr erleichtern, allerdings sind nahezu alle Interpreter in C/C++ geschrieben und die entsprechenden Tools sind deshalb auch nur in der Lage, C-Code zu verarbeiten. Die RFG-Bibliothek, Gravis und die meisten anderen Teile des Bauhaus sind jedoch in Ada geschrieben. Um sie also in die jeweilige Sprache integrieren zu können, muss ersteinmal ein Binding nach C realisiert werden. Im Gespräch mit meinem Betreuer stellte sich heraus, dass ein C-Binding der RFG-Bibliothek durchaus als wünschenswertes Ergebnis angesehen würde. Da das Binding von C in eine Skriptsprache mit Hilfen von entsprechenden Tools eine relativ einfache Aufgabe ist, stellt sich die Frage, wie der Aufwand des Ada-C-Bindings im Verhältnis zur Implementierung einer eigenen Sprache steht. Erste Vorüberlegungen ergaben, dass es einige Sprachfeatures von Ada gibt, für die es keine offensichtliche Umsetzung nach C geben würde — unter anderem generische Funktionen und tagged Types. Deren Umsetzung stellen aber eher konzeptionelle Schwierigkeiten dar, als großen Implementierungsaufwand zu erzeugen.

3 Analyse der zu exportierenden Codeteile

3.1 Statistik

Als erste Annäherung an den Code der RFG-Bibliothek bietet sich die Verwendung von `wc` und `grep` an, um einen Eindruck für die Größe der RFG-Bibliothek

zu bekommen. Da diese Werte nicht aus einem Reengineering-Werkzeug stammen, sondern nur durch Stichproben verifizierte Ergebnissen von `grep` sind, können sie natürlich nur als Anhaltspunkte dienen; mehr ist hier aber auch nicht nötig.

*.ads	~ 14.000 LoC	~ 450 kB
*.adb	~ 30.000 LoC	~ 900 kB
Funktionen	~ 740	
Prozeduren	~ 850	
Typen	~ 350	
String-Parameter	~ 100	
Default-Parameter	~ 380	

Die Zahl der Typen ist aber aus mehreren Gründen zu hoch: Die privaten Typen sind sowohl im öffentlichen wie auch im privaten Bereich der Spezifikationen enthalten. Zudem gibt es für viele Typen noch zugehörige Access-all-Typen, sodass die effektive Zahl der Typen vielleicht eher bei 150 bis 200 liegt. Davon sind ein großer Teil aus den Typhierarchien `Node`, `Edge` und `Attribute.Contents`, auf die noch gesondert eingegangen wird. Dazu kommt, dass Ada die Verwendung von Typen fördert und deshalb im Vergleich zu anderen Sprachen, wie z.B. C, sehr viel mehr explizite Typen verwendet werden. Diese sind dann oft nur Untertypen primitiver Typen.

3.2 Datentypen

Der RFG besteht aus einer ganzen Reihe an Datentypen. Deren genaue Implementierung ist in [Eisenbarth98] beschrieben. Für die Zwecke dieser Arbeit kann aber meist von der Implementierung abstrahiert werden. Wesentliche Typen sind:

RFG enthält interne Datenstrukturen, die zur Verwaltung des RFG nötig sind, aber dem Benutzer der RFG-Bibliothek verborgen bleiben. Der RFG wird bei nahezu jeder Funktion bzw. Prozedur als erster Parameter mit übergeben. Der RFG ist als Access zu einem privaten Record implementiert.

Node ist die Superklasse für alle Nodetypen — also der Knoten des Graphen. Die Node-Klassenhierarchie umfasst mehrere Dutzend Klassen, von denen aber ein Teil keine weiteren Recordkomponenten besitzen. Mittelfristig soll diese Hierarchie aus dem RFG verschwinden und die Komponenten der unterschiedlichen Typen durch Attribute implementiert werden. Im Vorgriff auf diese Änderungen soll in dieser Arbeit nur diese Superklasse exportiert werden. Nodes werden im RFG fast nur über die Access all Typen `Node_Ptr` bzw. `*_Node_Ptr` angesprochen.

Edge: Die Edgetypen sind die Kanten im RFG. Für Edges gelten die obigen Aussagen über Nodes analog.

View: Da für verschiedene Algorithmen immer wieder Teilgraphen des RFGs benötigt werden, gibt es Views. Nodes und Edges können in den einzelnen Views sehr effizient sichtbar oder unsichtbar gemacht werden. Die Views sind dabei nur Integer-Indizes; die eigentliche Verwaltung findet im RFG statt. Views haben einen Namen, können schreibgeschützt werden und eine `Root_Node` haben.

View_Set ist eine Menge von Views. Implementiert sind View_Sets als Bitfeld. Viele Prozeduren in der RFG-Bibliothek operieren auf View_Sets und fast alle entweder auf View_Sets oder Views. Allerdings werden View_Sets selten wirklich als Mengen von mehreren Views verwendet. Vielmehr werden am vielen Stellen View_Sets mit nur einer View als Element verwendet, da viele Operationen nur für View_Sets implementiert sind.

Attribute: Sowohl an Nodes als auch an Edges können Attribute angehängt werden. Definiert sind in Node_Attributes und Edge_Attributes. Attribute sind eine Art dynamische Recordkomponenten. Die beiden Attributtypen sind aber nur Integer-Indizes; die eigentliche Datenhaltung findet im RFG statt.

Attribute_Contents ist die Superklasse der Objekte die einem Attribut zugewiesen werden kann. Neben einigen Spezialtypen gibt es Toggle_Attribute, Boolean_Attribute, Integer_Attribute und String_Attribute.

Node_Set und Edge_Set sind Mengen von Nodes und Edges auf denen außerhalb des RFG operiert werden kann. Außer für die klassischen Mengenoperationen werden die Sets vor allem für die Graphoperationen wie z.B. Vorgänger, Nachfolger, ... verwendet.

Neben diesen "Hauptdatentypen" gibt es noch eine Reihe wichtiger Hilfsdatentypen. Hier sind vor allem diverse Iteratoren und Enumerationen (eine effiziente Zuordnung zwischen Integer-Indizes und Objekten wie z.B. Nodes) zu nennen. Bei genauerer Betrachtung ist die Zahl der wirklich relevanten Typen also gar nicht so groß.

3.3 Packages

Die RFG-Bibliothek besteht aus fast 80 Paketen unterschiedlicher Größe. Bei der Durchsicht der Pakete und der darin enthaltenen Funktionen wurde schnell klar, dass nur ein kleiner Teil der Funktionalität benötigt werden würde. Es zeichnete sich früh ab, dass der Export von generischen Codeteilen wahrscheinlich massive Schwierigkeiten bereiten würde und diese deshalb besser zu meiden sind. Die RFG-Bibliothek bietet aber mehrere Möglichkeiten an um auf die Elemente und Attribute des RFGs zuzugreifen, so dass hier ein Satz an Funktionen reichen würde. Die wichtigsten Pakete sind hier mit Stichworten zu ihrem Inhalt aufgelistet:

Package	nicht zu exportieren	zu exportieren
RFGs		Datentypen
RFGs.Accessors	Size(RFG), Traversals	Predecessors, Successors, Outgoings...
RFGs.Accessors.*	spezielle Accessors	
RFGs.Attributes		Node- und Edge-Attribute
RFGs.Check	Konsistenttests	
RFGs.Closure	Grapherreichbarkeit (generisch)	
RFGs.Edges	Edgetypen	
RFGs.Enumeration	Node Hash	
RFGs.Graph_Components	Zusammenhangsanalyse	
RFGs.IO.Plain		RFG laden und speichern
RFGs.IO.*	weitere Formate	
RFGs.Iteration		Node- und Edge-Iteratoren
RFGs.Manipulation		Einfügen von N./E. in RFG und Views
RFGs.Nodes	Nodentypen	
RFGs.Predicates		Node- und Edge-Predicates
RFGs.Traversals	(generisch)	
RFGs.Unreachables	(generisch)	
RFGs.Vertex_Connectivity		
RFGs.Views	View-Operationen und -Konstanten	
RFGs.Views.Iteration		View-Iterator
RFGs.View.Enumeration	View ↔ Number	

3.4 Architektur

Neben den bloßen Datentypen ist die Architektur einer Bibliothek dieser Größe interessant — also welche größeren Strukturen sich erkennen lassen. Die RFG-Bibliothek folgt klassischen prozeduralen Entwurfsprinzipien, auch wenn sie zur Zeit noch starken Gebrauch von Polymorphie macht. Alle Datentypen enthalten nur Nutzdaten und sind auf größtmögliche Unabhängigkeit von den restlichen Datentypen entworfen. Accesstypen werden nur dort (als Komponente) verwendet wo es wirklich nicht anders geht, und dies wird vor dem Benutzer bestmöglichst verborgen. Die Funktionen und Prozeduren sind nebenwirkungsfrei bzw. wirken ihre Effekte nur auf die explizit übergebenen Parameter. Dies führt zu verschiedenen Nachteilen:

- Der RFG wirkt als zusammenhangslose Wolke verschiedener Datentypen. Der Benutzer ist dafür verantwortlich, die dennoch existierenden Konsistenzbedingungen einzuhalten; d.h. nur die richtigen Objekte miteinander zu verwenden.
- Die Signaturen der Funktionen/Prozeduren sind sehr sperrig. Nahezu jeder Aufruf bekommt als erster Parameter einen RFG und noch eine View oder ein View_Set übergeben.

- Die Funktionen und Prozeduren sind nach Aufgaben (Packages) und nicht nach Typen sortiert. Dies liegt nicht zuletzt an der verwendeten Sprache Ada, die zwar Polymorphie unterstützt, aber die Primitive nicht wie viele andere objektorientierte Sprachen in den Namensraum der Klasse bzw. tagged Records aufnimmt. Dies führt dazu, dass fast allen Namen der Paketname vorangestellt werden muss, was den Code noch zusätzlich aufbläht.

Hinzu kommt noch, dass sowohl View als auch View_Set als Parametertyp verwendet werden und manche Funktionalität nur für eine der beiden Typen bereit steht. Vor allem die fehlende Durchgängigkeit für View als Basistyp ist dabei ärgerlich. Dem Benutzer fehlt so ein Satz von Funktionen der klein genug ist, um ihm im Gedächtnis zu behalten, aber groß genug ist um alle wichtigen Aufgaben bewältigen zu können.

Um diese etwas abstrakten Punkte werden an diesen — durchaus nicht untypische — Beispiele aus der RFG-Bibliothek deutlicher:

```
function Predecessors
(The_RFG   : RFG;
 The_Views : View_Set;
 The_Node  : Node_Ptr)
return Node_Ptrs.Sets.Set;
```

Predecessors (aus RFGs.Accessors) ist eine von mehreren Funktionen, die den Zugriff auf benachbarte Knoten und Kanten erlaubt.

Hier ein Beispiel zur Verwendung der RFG-Bibliothek. Bei der folgenden Funktion handelt es sich um eine interne Routine zum Ändern eines Attributes (aus Atomic Component Detection: Node_Weights). Hier wird auf über vierzig Zeilen nichts anderes gemacht als ein Attribut auf Null zu setzen, falls es noch nicht gesetzt sein sollte, es auszulesen, daraus einen neuen Wert zu berechnen und diesen wieder in das Attribut zu schreiben.

```
procedure Set
(The_Node   : in out RFGs.Node_Ptr;
 Modality   :           Edge_Tags.Tag;
 Role       :           ACD.Clustering.Features.Role_Type;
 Which_Weight :           Weight_Map;
 Value      :           Float)
is
Attribute_Value : Shannon_Info_Weight;

begin

  if not RFGs.Temporary_Attributes.Is_Defined
    (The_Node,
     RFGs.Temporary_Attribute (Which_Weight))
  then

    RFGs.Temporary_Attributes.Set
      (The_Node,
```

```

        RFGs.Temporary_Attribute (Which_Weight),
        RFGs_Attribute_Types.Attribute_Contents
        (SIA.Make_Attribute
         (Value =>
          Shannon_Info_Weight'(others => (others => 0.0)))));
end if;

Attribute_Value :=
    SIA.Attribute_Contents
    (RFGs.Temporary_Attributes.Get
     (The_Node,
      RFGs.Temporary_Attribute (Which_Weight))).Value;

Attribute_Value (Edge_Type_To_Index (Modality), Role) := Value;

RFGs.Temporary_Attributes.Set
(The_Node,
 RFGs.Temporary_Attribute (Which_Weight),
 RFGs_Attribute_Types.Attribute_Contents (SIA.Make_Attribute
                                           (Value => Attribute_Value)));

end Set;

```

3.5 Grobentwurf der Schnittstelle

Aufgrund der verschiedenen Kritikpunkte an der Architektur der RFG-Bibliothek, soll die Benutzerschnittstelle einige Veränderungen gegenüber der RFG-Bibliothek aufweisen:

- Die Node- und Edge-Typen werden im Vorgriff auf die Weiterentwicklung der RFG-Bibliothek auf jeweils einen Typ komprimiert. Damit kann vorläufig auf die Recordkomponenten der Untertypen nicht zugegriffen werden, bis diese als Attribute realisiert sind. Sollte sich dieser Umbau der RFG-Bibliothek zu sehr verzögern, können einzelne Recordkomponenten über Zugriffsfunktionen, die dann eine Typprüfung durchführen müssen, zugänglich gemacht werden.
- Der Type View_Set wird abgeschafft, um die Schnittstelle zu vereinfachen und zu vereinheitlichen. Das Binding muss den von der Skriptsprache übergebenen View-Parameter bei Bedarf in ein entsprechendes View_Set umwandeln.
- Um die Signaturen der meisten Funktionen und Prozeduren auf ein erträgliches Maß zu verkürzen und zusätzlich dem Benutzer weniger Möglichkeiten zu bieten gegen Konsistenzbedingungen zu verstoßen, erhalten View, Edge, Node und Container, die diese Typen enthalten, eine Referenz auf den RFG, der sie enthält. Edge und Node und deren Container erhalten zusätzlich noch eine Referenz auf die View, aus der sie entnommen wurden. Dies erlaubt es bei vielen Funktionen den RFG- bzw. den RFG- und den View-Parameter wegzulassen. Dabei wird angenommen,

dass meist innerhalb eines RFGs und sogar innerhalb einer View gearbeitet wird. Methoden zu RFG- und View-übergreifenden Arbeiten wie z.B. dem Einfügen von Knoten aus fremden RFGs oder dem Sichtbarmachen von Knoten werden direkt in der RFG- bzw. View-Klasse angesiedelt.

- Als Indizes für den Zugriff auf Views und Attribute sollen Standardtypen verwendet werden, die vom Benutzer direkt als Parameter angegeben werden können. Hier bieten sich die Namen der Views bzw. Attribute an. Da Views und Attribute selbst Eigenschaften haben, die dynamisch verändert werden können, können diese Typen dadurch nicht eingespart werden, aber sie werden nur noch für ein spezifisches Aufgabengebiet verwendet und weite Codeteile können ohne sie auskommen.
- Objekte, die häufig als Funktionsparameter benötigt werden, sollen vom Benutzer direkt erstellt werden können. Sie sollen nicht nur deshalb Variablen zugewiesen werden müssen, um sie anschließend wieder freigeben zu können. Dies ist der Fall, wenn die verwendete Sprache Garbage-Collection unterstützt.
- Die verschiedenen Iteratoren, die zum Zugriff auf einzelne Objekte benötigt werden, sollen für den Benutzer möglichst nicht sichtbar sein, sondern er soll in natürlicher Weise über die entsprechenden Objekte iterieren können.
- Möglichst alle Funktionen sollen in Methoden von passenden Objekten umgewandelt werden, um sie so in verschiedene Gruppen mit eigenem Namensraum zu strukturieren.
- Schließlich soll die Zahl der Funktionen reduziert werden. Als einzige Zugriffsmechanismen auf Knoten und Kanten werden Iteratoren und Mengen verwendet.

Durch diesen Umbau stellt sich der RFG dem Benutzer als das dar, was er schon immer ist: Eine durch Referenzen verbundene Datenstruktur.

3.6 Einbindung in Gravis

Neben dem Entwurf der Datentypen und Befehlen in der Programmiersprache muss auch die Einbindung in Gravis konzipiert werden. Die von dem Benutzer erstellten Skripte sollen direkt in die Gravisoberfläche eingebunden werden, so dass sie wie die in Ada implementierten Analysen verwendet werden können. In dem selben Dialog (Execute Analysis) soll dem Benutzer auch eine Möglichkeit geboten werden, die Skripte zu bearbeiten und neue anzulegen. Zusätzlich sollen Module angelegt werden könne, in die der Benutzer wiederverwertbaren Code ablegen kann.

Zum Zugriff auf den RFG, die Auswahl und Markierung in Gravis müssen Funktionen bereit gestellt werden, die die entsprechenden Objekte zurückliefern.

4 Kriterien für die Auswahl der Programmiersprache

4.1 Paradigma

Dadurch dass die RFG-Bibliothek relativ stark an den Datentypen ausgerichtet ist, bietet sich eine prozedural/objektorientierte Programmiersprache an. Dieses Programmierparadigma hat sich in der Praxis weitgehend durchgesetzt und die RFG-Bibliothek ist mit Ada bereits in einer Sprache implementiert, die prozedural und polymorph ist. Eine reine Mengenalgebra ist angesichts der mit Views und Attributen doch etwas komplizierteren Struktur des RFGs und den zusätzlichen Anforderungen zur Interaktion mit Gravis ungeeignet. Funktionale Sprachen sind im akademischen Bereich zwar relativ verbreitet, aber es hat sich keine Sprache in einem größeren Bereich durchsetzen können und selbst einzelne Sprachen wie z.B. Lisp/Scheme sind in sich zersplittert. Außerdem ist das funktionale Programmierparadigma außerhalb des akademischen Bereichs und einigen GNU Anwendungen eher selten.

4.2 Garbage Collection

Für eine Programmiersprache, die Teil einer Benutzerschnittstelle ist, ist heutzutage Garbage Collection nahezu unerlässlich, da es dem Benutzer eigentlich nicht zuzumuten ist, sich um die Freigabe von verwendetem Speicher selbst zu kümmern. Dadurch dass der Benutzer keinen Code zum Verwalten von Objekten im Speicher selbst schreiben muss, werden die Programme der Benutzer kürzer und einfacher. Manche Konstrukte lassen sich ohne Garbage Collection gar nicht richtig benutzen. Ein Beispiel dafür sind die Mengentypen in der RFG-Bibliothek (`Node_Ptrs.Sets.Set` und `Edge_Ptrs.Sets.Set`). Sie unterstützen zwar Operatoren; diese können aber nicht zu komplizierteren Ausdrücken kombiniert werden, weil alle Zwischenergebnisse einer Variablen zugewiesen und dann freigegeben werden müssen. Dies ist natürlich für die Benutzerschnittstelle nicht erwünscht. Die Argumente, die gegen eine Garbage Collection natürlich angewandt werden müssen — zusätzlicher Ressourcen Verbrauch und nicht echtzeitfähiges Laufzeitverhalten — haben für diese Anwendung praktisch keine Bedeutung.

4.3 Skripting

Da es sich bei dieser Aufgabe um klassisches Skripting einer Anwendung handelt, bietet sich natürlich auch eine Skriptsprache an.

Eine Skriptsprache ist eine interpretierte, meist sogar interaktive Programmiersprache, die schwach oder dynamisch typisiert ist und eine Garbage Collection besitzt. Skriptsprachen beherrschen in der Regel einige höhere Standardtypen wie Listen oder Hashtabellen direkt, um dem Benutzer ein einfaches und meist deklarationsfreies Aufbauen von komplexen Datenstrukturen zu erlauben.

Skriptsprachen sind für diese Aufgabe deshalb interessant, weil sie traditionellerweise für kurze Programme gedacht sind und sie es dem Benutzer möglichst leicht machen wollen, sein Problem zu lösen. Die Skriptsprachen verzichten zwar auf statische Prüfungen z.T. sogar auf die statische Prüfbarkeit, vermeiden dafür aber andere Probleme wie Speicherlöcher und Dangling References. Zudem sind sie meist deutlich ausdrucksstärker, was zu kürzerem und damit leichter überschaubarem Code führt.

Bei so viel Schönfärberei soll auch nicht verschwiegen werden, dass viele Skriptsprachen ursprünglich von der UNIX-Shell abstammen und dies einigen Sprachen an ihrer nicht unbedingt klaren Syntax bis heute anzusehen ist. Zudem sieht man einigen Sprachen noch an, dass bei ihrem Entwurf an Textersetzung als Implementation gedacht wurde und deshalb Variable durch ein vorangestellte Sonderzeichen (in der Regel `$`) gekennzeichnet werden müssen.

4.4 Look and Feel

Die Sprache sollte sich in ihrer Syntax an bekannten compilierten Sprachen anlehnen. Compilierte Sprache heißt in diesem Zusammenhang natürlich C, da C die einzige Sprache ist, von der davon ausgegangen werden kann, dass sie von allen Benutzern von Gravis — einem Werkzeug zur Analyse von C-Code — beherrscht wird.

Da die Effizienz von interpretierten Sprachen stark von dem Aufwand für das Parsen des Programmcodes abhängt, versuchen einige — vor allem ältere — Sprachen, das Parsen durch spezielle Syntax zu vereinfachen und damit ein Teil der Aufgabe, die eigentlich der Parser übernehmen sollte, dem Benutzer aufzuerlegen. Da alle "großen" Skriptsprachen inzwischen aber entweder implizit oder explizit in Bytecode kompiliert werden bzw. werden können, sollten moderne Skriptsprachen eigentlich ohne solche Tricks auskommen.

4.5 Verbreitung

Eine weite Verbreitung der auszuwählenden Skriptsprache ist aus zwei Gründen wichtig: Akzeptanz und Zukunftssicherheit. Auch wenn von keiner Sprache (mit Ausnahme von C) erwartet werden kann, dass alle Benutzer von Gravis sie bereits beherrschen, so ist es wünschenswert, wenn zumindest ein größerer Teil ihrer mächtig ist. Viel wesentlicher als die Frage, wieviele Benutzer genau diese Sprache bereits beherrschen, ist die Frage wie sehr sie bereit sind, sich — zumindest ein wenig — in diese neue Sprache einzuarbeiten. Hierbei hilft es sehr, wenn diese Sprachkenntnisse wahrscheinlich in anderem Zusammenhang wiederverwendet werden können. Und diese Wahrscheinlichkeit steigt natürlich proportional mit der Verbreitung der Sprache.

Ein Hauptargument für die Verwendung einer existierenden Skriptsprache war, dass man so einen Teil der Entwicklung und Wartung auslagern kann. Dafür ist es wichtig, dass die verwendete Sprache auch auf absehbare Zeit weiterentwickelt wird. Nun sind Aussagen über die Zukunft immer schwierig, aber im Allgemeinen sterben Programmiersprachen bzw. deren Compiler/Interpreter relativ selten aus. Eine größere Zahl größerer Projekte (vor allem auch größerer Firmen) sind ein gutes Zeichen, dass uns die Sprache noch eine Weile erhalten bleibt.

So wichtig eine weite Verbreitung der verwendeten Sprache auch ist, so schwierig ist es festzulegen, was genau die Verbreitung sein soll und die so definierte Verbreitung dann auch zu messen. Da umfangreiche Feldstudien für diese Arbeit nicht angemessen sind, galt es einen einfachen Weg zu finden. Einfachste Art der Datenbeschaffung ist es, eine Suchmaschine zu verwenden. Da das einfache Suchen nach den Namen der verschiedenen Sprachen eine Menge Fehltreffer produziert, wurde in einer zweiten Suche dem jeweiligen Sprachennamen Wort "programming" hinzugefügt. (Die Suche nach "PHP" findet unter anderem alle

PHP-Skripte die auf `.php` enden und die Suche nach einzelnen Buchstaben — wie z.B. `C` — ist auch nicht erfolgreicher. Schließlich haben einige Namen wie Java, Perl und Python auch eine ursprüngliche Bedeutung.) Eine zweite Möglichkeit, die ausserdem überhaupt einmal eine Liste mit interessanten Programmiersprachen liefert, ist es in einem der Open-Source-Software-Verzeichnisse nach zu sehen. Hier bieten sich `www.sourceforge.net` und `www.freshmeat.net` an. Dort sind die Projekte unter anderem auch nach der verwendeten Programmiersprache aufgelistet. Es ist klar, dass diese Zahlen nicht allzu ernst genommen werden dürfen. Der Open-Source-Bereich ist nicht die gesamte Welt, und diese Verzeichnisse haben eine Gedächtnis, dass nur kurze Zeit in die Vergangenheit reicht. Benachteiligt werden dabei vor allem Sprachen, die hauptsächlich im industriellen Umfeld Anwendung finden und Sprachen, die inzwischen etwas aus der Mode sind, aber immer noch eine große Bedeutung haben. Wenn man sich die Zahlen bei TCL anschaut, könnte man diesen Effekt dort vermuten. Da es aber nur auf die Aussage ankommt "Die Sprache ist weit verbreitet", sollten diese Angaben für den Zweck dieser Arbeit genau genug sein.

Die Daten von `sourceforge.net` und `freshmeat.net` stammen vom 31.5.2003, die von Google vom 20.6.03. Die Spalte Google 1 gibt die Trefferzahl für die Suchen nach dem Sprachnamen und "programming" an und Google 2 die Trefferzahl für den Namen alleine. Beide Werte sind in Millionen Treffern angegeben.

Sprache	sourceforge	freshmeat	Google 1	Google 2
Basic	1430	27	3.0	38
C	10966	5150	4.4	317
C++	10678	2258	2.1	7.5
Java	8875	2101	2.5	32
Perl	4618	2629	1.9	13
PHP	6676	1846	2.4	165
Python	2419	1057	0.9	5.5
Ruby	222	107	0.16	5.7
TCL	668	338	0.4	2.9

4.6 Lesbarkeit und Erlernbarkeit

Da bei keiner Programmiersprache davon ausgegangen werden kann, dass sie von allen potentiellen Anwendern beherrscht wird, sollte die Sprache möglichst leicht zu erlernen sein. Um das Lernen wird aber wohl kein Anwender herumkommen, da sich jeder zumindest in die neue RFG-Schnittstelle wird einarbeiten müssen. Um eine Sprache zu lernen, ist es von großem Vorteil, wenn die Sprache sehr kompakt und die Sprachelemente zueinander orthogonal sind. Wichtig ist dabei auch, dass die Sprache nur gebräuchliche syntaktische Elemente verwendet und die Verwendung von z.B. ungewöhnlichen Sonderzeichen in der Syntax vermieden wird. Ausserdem sollte die Syntax wegen des Prinzips der kleinsten Verwunderung möglichst wenig unterschiedliche Konstrukte für den selben Zweck anbieten. Sehr hilfreich ist es auch, wenn die Sprache mit wenig Vorkenntnissen gelesen werden kann. Da sich jeder neue Benutzer zumindest die Beispiele zur Benutzung der RFG-Bibliothek wird anschauen müssen, können die Benutzer so schon etwas über die Sprache lernen. Wenn die Benutzer schon beim Durchlesen der Beispiele Verständnisprobleme bekommen, kann bezweifelt werden, dass sie sich gerne mit der weiteren Dokumentation beschäftigen werden.

4.7 Erweiterbarkeit und Einbettbarkeit

Die Sprache bzw. ihre Implementierung muss es natürlich zulassen, dass sie in ein Programm eingebettet wird und dort auf Programmelemente zugreift. Dabei ist natürlich nicht nur die theoretische Möglichkeit entscheidend, sondern die aktive Unterstützung von entsprechenden Werkzeugen und der Sprache selbst. Wichtigstes Kriterium ist hierbei die Existenz von guter Dokumentation zu diesem Themengebiet, die am besten zur Dokumentation der Sprache selbst gehören sollte.

Um die Einbettung zu erleichtern, ist es von Vorteil, wenn die Sprache keine externen — vor allem aber keine plattformabhängigen — Werkzeuge benötigt. Der Benutzer soll die Sprache direkt als Teil von Gravis nutzen können, ohne sich Gedanken über Zusatzpakete machen zu müssen. Der unter Unix übliche `configure;make;make install` Zyklus soll dem Benutzer erspart bleiben und damit auch die Probleme, die dabei auftreten können.

4.8 Plattformunabhängigkeit

Die verwendete Sprache muss unter den gängigen Unixsystemen (Solaris, HP-UX, Linux) und unter Windows lauffähig sein bzw. es müssen zueinander kompatible Implementierungen auf diesen Plattformen existieren. Um die Integration des Interpreters zu vereinfachen, ist es sehr vorteilhaft, wenn es eine Implementierung für alle Plattformen gibt.

4.9 Gtk Binding

Da Gravis in Gtk implementiert ist, wäre es von Vorteil, wenn für die Sprache auch ein Gtk-Binding existieren würde. Zwar ist dies für diese Arbeit nicht notwendig und es ist derzeit auch noch nicht untersucht, welche Anwendungsmöglichkeiten im Umfeld dieser Arbeit dafür existieren. Prinzipiell liegt es aber schon nahe, dass irgendwann auch aus dieser Erweiterungssprache heraus GUI-Programmierung betrieben werden könnte. Die Existenz eines Gtk-Bindings ist kein geeignetes Auswahlkriterium, da es für alle größeren Sprachen — und insbesondere für die hier betrachteten — ein Gtk-Binding gibt.

4.10 Lizenz

Der Interpreter muss natürlich unter einer Lizenz verfügbar sein, die es erlaubt, ihn kostenfrei in Gravis einzubetten und auch weiterzugeben, ohne die Lizenz vom Bauhaus zu ändern. Die Lizenz darf nicht verlangen, den Quellcode des Interpreters oder der des Bauhauses mitauszuliefern.

5 Analyse verschiedener Programmiersprachen

Bei der Durchsicht der Open-Source-Archive (siehe Seite 14), stellten sich die folgenden interpretierten Sprachen als einigermaßen weit verbreitet heraus: Java, JavaScript, Perl, PHP, Python, Ruby, Unix Shell, Visual Basic und Tcl.

5.1 Basic

Basic findet heute vor allem unter Windows in Form von Visual Basic und VB-Script Verwendung. Unter Unix hat sich Basic nie richtig durchgesetzt. Es gibt zwar auch unter Unix mehrere — auch freie — Interpreter. Von diesen hat aber keiner größere Bedeutung. Aufgrund der geringen Verbreitung im Unix-Bereich, stellt sich die Frage ob Basic oder ein Basic Dialekt inhaltlich die Anforderungen erfüllt, erst gar nicht.

5.2 Java

Java ist objektorientiert und statisch typisiert und lehnt sich stark an C++ an. Im Gegensatz zu C++ verzichtet es jedoch auf Pointerarithmetik und besitzt eine Garbage-Collection. Java wird explizit in Bytecode übersetzt. Damit ist Java keine Skriptsprache im eigentlichen Sinne. Sie ist aber sehr weit verbreitet und erfüllt viele der gestellten Anforderungen.

Leider sind die technische Randbedingung beim einbetten des Java-VM nicht optimal. Das JDK im Vergleich zu allen anderen Sprachen sehr groß ist (ca. 150 MB), so dass hier die Frage stellt, ob man es einfach so mit der Software verteilen will, oder ob man auf ein lokales JDK zurückgreift und sich damit möglicherweise Kompatibilitätsprobleme einhandelt. Sicher könnte man das JDK auch so ausdünnen, dass es eine vernünftige Größe hat, aber das ist ebenfalls unschön.

Die starke Ähnlichkeit zu C++ ist zwar klarer Pluspunkt, führt aber zu deutlich längeren Code, als bei einigen Skriptsprachen. Zudem ist für Skripte in der angestrebten Größe für den Benutzer nicht einzusehen, warum er eine Klasse definieren soll.

5.3 Perl

Perl (Practical Extraction and Report Language) ist vielleicht die Skriptsprache überhaupt. Version 1.0 wurde bereits im Dezember 1987 veröffentlicht. Ursprünglich zur Auswertung von Logdateien entwickelt ist Perl heute eine weitverbreitete Allzwecksprache mit einer Großzahl an Zusatzmodulen. Die Hauptstärke von Perl sind die integrierten regulären Ausdrücke und damit die Verarbeitung von Texten. Ein großes Anwendungsgebiet sind CGI-Skripte. Perl ist schwach typisiert, d.h. die sogenannten Skalare (String, Integer und Float) werden implizit ineinander umgewandelt; der Benutzer muss aber durch ein vor gestelltes Sonderzeichen (\$, @, %) angeben ob ein Wert ein Skalar, ein Array oder ein Hash ist. Perl erlaubt dem Benutzer an vielen Stellen die Auswahl aus verschiedenen Sprachelementen. Die Syntax ist an vielen Stellen variabel; so können z.B. if-Bedingungen sowohl vor als auch nach dem auszuführenden Block stehen. Dies ermöglicht es, für jede Aufgabe die passende Form des jeweiligen Konstrukts zu wählen und so dem Leser die Intention besser zu verdeutlichen. Dies setzt allerdings eine Programmierkultur voraus, die beim Autor und beim Leser die gleiche sein sollte.

Perl macht — vor allem im Zusammenhang mit regulären Ausdrücken — starken Gebrauch von Spezialvariablen, in denen (Teil-) Ergebnisse gespeichert werden. Wichtigste Spezialvariable ist die sogenannte Defaultvariable \$_, der Funktionsergebnisse zugewiesen werden.

Funktionen erhalten Parameter als eine Liste. Benannte Parameter gibt es nicht. Weder Typ noch Zahl der Funktionsparameter wird überprüft.

Perl beherrscht auch Objektorientierung. Diese ist als eine Verallgemeinerung des Modulkonzepts realisiert und wurde nachträglich auf die Sprache aufgesetzt, was ihr auch noch anzusehen ist.

Anhänger von Perl betonen zwar, dass man auch lesbare Perlprogramme schreiben kann, dennoch hält sich hartnäckig das Gerücht vom "executable line noise", dass jeder zufällige String ein gültiges Perlprogramm darstellt.

5.4 Php

PHP (PHP: Hypertext Preprocessor) ist eine sehr weit verbreitete Skriptsprache für serverseitige Webanwendungen, deren Syntax sich an C anlehnt. PHP kann direkt in HTML eingebettet werden. Leider scheint es keinerlei Dokumentation darüber zu geben, wie der PHP-Interpreter in Applikationen eingebettet werden kann. Dazu kommt noch, dass PHP einen sehr klaren Fokus auf Webanwendungen setzt.

5.5 Python

Python ist eine prozedurale, objektorientierte, dynamisch typisierte Skriptsprache. Die Zuweisung hat in Python Referenzsemantik. Dies wird dadurch abgemildert, dass viele Standardtypen unveränderlich sind und sich somit z.B die Arithmetik wie gewohnt verhält. Die Zuweisung ist deshalb keine Zuweisung im klassischen Sinne; vielmehr werden durch sie Objekte an einen Namen in einem bestimmten Namensraum gebunden. Die Typinformation ist Teil der Objekte (dynamische Typisierung); deshalb können an einen Namen nacheinander Objekte verschiedenen Typs gebunden werden. Funktionen und Methoden führen keine Typprüfung, wohl aber ein Prüfung auf die Anzahl der Parameter durch. Dadurch sind alle Objekte, die einen benötigten Satz an Methoden implementiert haben, zueinander kompatibel.

Die Syntax von Python ist sehr knapp gehalten. Die Statements bestehen nur einem Schlüsselwort (die for-Schleife aus zwei) dem zugehörigen Ausdruck und einem Doppelpunkt, der das Statement vom zugehörigen Anweisungsblock trennt. Neben prozeduralen und objektorientierten Sprachelementen hat Python auch ein paar funktionale Elemente wie `map()`, Lambda-Ausdrücke und Tupel als Rückgabewerte. Da die Liste in Python jedoch als dynamisches Array implementiert ist und auch das entsprechende Laufzeitverhalten hat, wird dieser Teil der Sprache relativ wenig genutzt.

Eine Besonderheit von Python ist die semantische Einrückung. Blöcke bestehen aus Anweisungen, die gleich weit eingerückt sind. Um einen neuen Block zu beginnen, wird einfach eine Stufe weiter eingerückt (der python-mode von Emacs rückt 4 Spaces ein). Offene Klammern verlängern dabei eine semantische Zeile, bis die Klammer wieder geschlossen wird. Dadurch können komplexere Ausdrücke auch auf mehrere Zeilen verteilt werden. Auf den ersten Blick wirkt das sehr seltsam; im Endeffekt schreibt man ein Pythonprogramm aber so, wie man ein Programm in einer beliebigen anderen Sprache auch schreiben und einrücken würde, nur dass man die geschweiften Klammer oder `begin`, `end` weglässt. Diese in der Tat ungewöhnliche Prinzip hat auf den ersten Blick den Vorteil, dass man eine Menge Codezeilen spart und deshalb deutlich mehr Code

auf einen Bildschirm bekommt. Der in meinen Augen viel wichtigere, aber nicht sofort einsichtige Vorteil ist, dass durch die semantische Einrückung auch die Formatierung des Codes vorgegeben wird. Zwar hat der Programmierer immernoch kleine Freiheiten, wie z.B. eine einzelne Anweisung nach einer if-Anweisung in die nächste Zeile umzubrechen oder nicht, im Großen und Ganzen sind Pythonprogramme, was die Formatierung angeht, alle ähnlich, was die Lesbarkeit sehr erhöht.

Ein Designfehler in Python ist die unterschiedliche Behandlung der eingebauten Typen und Klassen. Dieses Problem wird aber ab Version 2.2 durch eine neue Metaklasse behoben, deren Instanzen Typen sind, die die Aufgabe von Klassen übernehmen (Diese werden von SWIG bereits verwendet). Es ist zu erwarten, dass bei künftigen Versionen von Python, die alten Klassen fallen gelassen werden bzw. in ein Kompatibilitätsmodul abgeschoben werden.

Wenn man sich genauer mit Python beschäftigt, muss man den ersten Eindruck, dass Python eine kompakte und sehr ausgewogene Programmiersprache ist, die guten Programmierstiel fördert, etwas korrigieren. Dies trifft durchaus zu, wenn man Python wie eine gewöhnliche objektorientierte Programmiersprache benutzt. Da Python jedoch von Grund auf sehr dynamisch aufgebaut ist, lassen sich damit mit nur wenigen Zeilen mächtige aber auch komplexe und schwer durchschaubare Codefragmente erstellen. Man darf sich als erfahrener Pythonprogrammierer hier nicht zu sehr in Versuchung führen lassen. Man kann eben in jeder Programmiersprache schlechte Programme schreiben, Python verschiebt die "unsaubere" Programmierung nur von dem Mikro- in den Makro-Bereich.

5.6 Ruby

Ruby ist eine noch relativ junge Skriptsprache. Ihre Anfänge liegen im Jahre 1993, die erste Veröffentlichung 1995. Sie ist noch nicht so sehr verbreitet und auch nur deshalb auf der Liste zu untersuchender Sprachen, weil sie gerade in Mode kommt und einiges an Potential haben soll. Ruby ist in der Tat eine interessante Sprache. Ruby ist rein objektorientiert und stellt eine Art Kreuzung aus Python und Perl dar. Sie ist eine sehr schön designte Programmiersprache, die sich durch ein durchgängige Klassenhierarchie von Standardtypen positiv von Python abhebt. Im Gegensatz zu Perl wirken die Sprachkonstrukte nicht aufgesetzt. Viele Sprachkonstrukte sind synaktischer Zucker für Methoden der Standardklassen. Es gibt benannte Funktionsparameter. Die Designziele sind aber eher die von Perl als die von Python. Die Syntax ist ebenfalls sehr variabel, reguläre Ausdrücke sind direkt in die Sprache integriert und es existieren eine ganze Reihe an Spezialvariablen. Ruby ist eine würdige Alternative zu Perl, die viele Unzulänglichkeiten von Perl löst, aber die eigentliche Mächtigkeit beibehält. Bis Ruby aber einen solchen Umfang an Zusatzsoftware aufweisen kann wie Perl, werden sicher noch ein paar Jahre ins Land gehen.

5.7 Tcl

Tcl — vielleicht auch besser bekannt als Tcl/Tk — ist ein Urgestein unter den Skriptsprachen. Tcl ist prozedural und man sieht ihm deutlich an, dass es ursprünglich durch Stringoperationen implementiert war, auch wenn sich da inzwischen einiges geändert hat und Tcl inzwischen auch in Bytecode kompiliert

wird. Befehle in Tcl bestehen aus dem Befehlsname gefolgt von den Parametern, die durch Leerzeichen getrennt werden. \$Variablenname wird durch den Wert der Variable ersetzt. Um kompliziertere Konstrukte erzeugen zu können, gibt es drei verschiedene Sprachmechanismen, mit denen Parameter gruppiert werden können. Eckige Klammer schließen Befehle ein, deren Ergebnis an dieser Stelle eingefügt wird; sie entsprechen ungefähr einem Funktionsaufruf innerhalb einer Parameterliste in anderen Sprachen. Doppelte Hochkommata fassen mehrere Parameter zu einem zusammen; innerhalb von doppelten Hochkommata finden nur Variablenersetzungen statt. Zu guter Letzt können Parameter mittels geschweiften Klammern zusammen gefasst werden. Innerhalb geschweiften Klammern finden keinerlei Ersetzungen statt. Dies wird in Tcl ausgenutzt, um Befehle zu implementieren, die Statements wie Schleifen und if-Anweisungen entsprechen. Diese können dann die mit geschweiften Klammern eingeschlossenen Blöcke nach Belieben selbst auswerten.

Trotz dieses etwas ungewöhnlichen Sprachentwurfs ist die Syntax von Tcl erstaunlich unspektakulär. Zwar wirken die Befehle zur Zuweisung von Werten an Variablen und zur Auswertung von arithmetischen Ausdrücken anachronistisch, die Befehle, die Statements implementieren, unterscheiden sich aber kaum von anderen prozeduralen Sprachen. Tcl ist rein prozedural und unterstützt keine Objektorientierung.

5.8 Auswahl

Basic ist aufgrund der fehlenden Verbreitung unter Unix und PHP wegen der fehlenden Einbettbarkeit im Vorfeld ausgeschieden. Da Tcl keine Objektorientierung unterstützt und der Sprachentwurf doch sehr anachronistisch ist, wurde auch Tcl relativ schnell verworfen.

Perl und Ruby sind durch ihre variable Syntax und implizite Semantik durch die Spezialvariablen und Perl im besonderen durch seine schlechte Lesbarkeit als Sprache für diese Arbeit ausgeschlossen worden. Sie sind zu sehr klassische Skriptsprachen, die eine ganze Menge an Spezialkonstrukten für spezielle Zwecke besitzen, die die Sprachen für unsere Zwecke nur unnötig aufblähen.

Die zwei verbleibenden Sprachen Java und Python bilden die Brückenköpfe auf beiden Seiten der Trennlinie zwischen klassischen Programmiersprachen und Skriptsprachen. Java ist syntaktisch sehr nah an C++, hat deshalb aber nicht die Leichtigkeit einer Skriptsprache. Python ist aufgrund der Größe besser einzubetten, hat aber eine ungewöhnlichere Syntax und niedrigere Verbreitung. Die Entscheidung zwischen den beiden Sprachen ist also äußerst knapp und läuft auf die Frage heraus, wieviel Skripting man denn haben möchte. Die Entscheidung Python auszuwählen ist deshalb natürlich ein Stück weit willkürlich.

6 Die Python-Schnittstelle

Nach der Auswahl der zu verwendenden Sprache muss der Grobentwurf in diese Sprache umgesetzt werden. Die natürliche Umsetzung der Ada-RFG-Bibliothek nach Python besteht zu allererst im Umsetzen der wesentlichen Datentypen in Pythonklassen. Gemäß dem Grobentwurf werden die Funktionen den Klassen als Methoden zugeordnet. Die Schnittstelle versucht sich am Stil von Python

zu orientieren und benutzt — wo es geht — Standardmechanismen von Python wie Iteratoren, Dictionaries, Elementzugriffe und Operatoren.

In Python sind viele Sprachkonstrukte nur syntaktischer Zucker für darunter liegende Methoden. Um den den verschiedenen Typen ein "natürliches" Verhalten zu geben, werden einige dieser Spezialmethoden implementiert, um so diesen Zucker für die Schnittstelle nutzen zu können. Im einzelnen sind das:

`__getitem__(Schlüssel)` wird beim lesenden Elementzugriff mittels `Objekt[Schlüssel]` aufgerufen

`__set_item__(Schlüssel, Wert)` `Objekt[Schlüssel] = Wert`

`__del_item__(Schlüssel)` `del Objekt[Schlüssel]` entfernt einen Eintrag

`__contains__(Item)` wird bei Verwendung des Schlüsselwortes `in` aufgerufen, das eigentlich einen boolescher Operator ist.

```
if 1 in [1, 2, 3]: print "Found 1"
```

`__add__(other)` arithmetrischer Opererator +

`__sub__(other)` arithmetrischer Opererator -

`__mul__(other)` arithmetrischer Opererator *

`__div__(other)` arithmetrischer Opererator /

`__and__(other)` arithmetrischer Opererator &

`__or__(other)` arithmetrischer Opererator |

`__xor__(other)` arithmetrischer Opererator ^

Mehr zu Spezialmethoden ist im Kapitel "Special method names" des Python Referenze Manual [9] zu finden.

Ein weiteres wichtiges Konzept sind Iteratoren. Iterator-Objekte haben in Python die folgenden Methoden:

`__iter__()` gibt eine Referenz auf sich selbst zurück. Container liefern bei dieser Methode ein Iteratorobjekt zurück. Dadurch können sowohl Iteratoren als auch Containerobjekte in Schleifen verwendet werden.

`.next()` liefert das nächste Objekt, wenn es noch ein weiteres gibt; sonst wirft es eine `StopIteration` Exception.

Iteratoren können auf drei unterschiedliche Arten verwendet werden:

- Iteratoren und Containerobjekte, die eine Methode `__iter__()` besitzen, die einen Iterator zurückgibt. Können in `for`-Schleifen verwendet werden:

```
for Item in My_Iterator:  
    print Item
```

- Man kann die Methode `.next()` direkt aufrufen, muss dabei aber die `StopIteration` Exception abfangen.
- Schließlich kann man mit der Standard-Funktion `list()` den Iterator in eine Liste umwandeln. Dies ist im Zusammenhang mit dem RFG eher ungünstig, da Listen von Knoten und Kanten an keiner Stelle als Parameter erlaubt sind.

Der RFG soll aber nicht als lose Ansammlung von Objekte dargestellte werden, sondern soll für den Benutzer als eine geschlossene Datenstruktur erscheinen. Dies wird durch die Python-Attribute `.RFG` und `.View` realisiert. Diese können vom Benutzer im Prinzip verändert werden. Bei `.RFG` sollte das aber nie nötig sein und fast immer zu Fehlern führen. Bei `.View` ist es eher denkbar, dass es für bestimmte Fälle sinnvoll sein kann, ihm ein anderes Objekt zuzuweisen. Eigentlich ist die Schnittstelle aber so ausgelegt, dass dies nicht nötig ist.

Bei der Beschreibung der Schnittstelle lehnt sich an der Aufrufsyntax von Python an. Attribute werden durch einen vorangestellten Punkt und Funktionen durch runde Klammer, die bei Bedarf die Parameter einschließen, gekennzeichnet. Methoden erhalten sowohl den vorgestellten Punkt als auch die runden Klammern. Bei Methoden wird der beim Aufruf implizite Self-Parameter weggelassen. Da Python keine Syntax für die Typisierung von Parametern oder Rückgabewerten besitzt, werden die Parameter nach dem unterstützten Typ benannt. Bei mehreren möglichen Typen und bei Rückgabewerten werden die Typen in der Beschreibung angegeben.

Die meisten Klassen und Funktionen befinden sich in dem Modul `rfgs`. Die Prädikate sind in einem eigenen Modul untergebracht.

6.1 RFG

Der RFG ist die Wurzel der Datenstruktur die die Pythonschnittstelle darstellt. Der RFG selbst stellt kaum Funktionalität zur Verfügung, gibt aber Zugriff auf die Views und die Attributebeschreibungen.

`.views` ist ein Dictionary das die Views unter ihrem Namen enthält. Auf dieses Attribut sollte niemals schreibend zugegriffen werden.

`._get_item(Viewname)` aka `[Viewname]` Kürzel für `RFG.views[Viewname]`

`.new_view(Name)` erzeugt eine neue View mit dem angegebenen Namen

`.new_node(Tag)` gibt eine neue Node zurück, die bereit Teil des RFG ist. Tag ist dabei ein String der den Typ der Node angibt. Derzeit sind die Tag noch fest in die RFG-Bibliothek ein programmiert. Dies wird sich aber mit der Abschaffung der Node- und Edge-Hierarchien ändern.

`.new_edge(Tag, from, to)` gibt eine neue Edge zurück, die bereits Teil des RFG ist.

`.edge_attributes` ist ein Dictionary das Name:Edge_Attribute enthält.

`.node_attributes` ist ein Dictionary das Name:Node_Attribute enthält.

`.new_edge_attribute(Name, Type, Per_View, Unique)` erstellt ein neues Edge_Attribute und fügt es in `.edge_attributes` ein. Name und Type sind Strings und Per_View und Unique Wahrheitswerte. Einzelheiten zu Type sind bei der Beschreibung der Attribute zu finden.

`.new_node_attribute(Name, Type, Per_View, Unique)` s.o.

`.insert(Item)` fügt eine Node oder Edge in den RFG ein und liefert das eingefügte Objekt wieder zurück.

`.remove(Item)` entfernt eine Node oder Edge aus dem RFG.

`.save_plain(Filename)` speichert den RFG in die angegebene Datei.

Im Zusammenhang mit RFG ist die Funktion `load_plain(Dateiname)` zu nennen, die einen RFG aus der angegebenen Datei lädt und ein RFG-Objekt zurückliefert.

6.2 View

Die Viewobjekte ermöglichen es auf die Eigenschaften der View zuzugreifen. Im einzelnen sind das

.RFG enthält einen Verweis auf den zugehörigen RFG. Dieses Attribut sollte vom Benutzer unangetastet bleiben.

.Root_Node Wurzelknoten der View; enthält eine Node oder None.

.__contains__(Item) implementiert das Schlüsselwort `in`. Als Typ von Item sind Node und Edge zugelassen.

.make_visible(Item) Als Typ von Item sind Node, Node_Set, Edge und Edge_Set zugelassen.

.writable

.name

.isvalid() Ist die View noch gültig oder wurde sie inzwischen gelöscht?

.destroy() löscht die View.

.set() macht alle Nodes und Edges in der View sichtbar.

.unset() macht alle Nodes und Edges in der View unsichtbar.

.copy(View) übernimmt die Sichtbarkeit von Knoten und Kanten aus View.

.intersect(View) macht alle Knoten und Kanten aus View unsichtbar.

.add(View) macht alle Knoten und Kanten aus View sichtbar.

Zudem erlauben sie den Zugriff auf die Nodes und Edges die gerade sichtbar sind.

.nodes(Node_Predicate)

.edges(Edge_Predicate)

Beide Methoden liefern ein Python-Iteratorobjekt zurück. Dieses kann in for-Schleifen verwendet werden.

```
for Node in The_RFG.nodes(All_Nodes):
    print Node.get_tag()
```

Eine weitere Möglichkeit auf Knoten und Kanten zuzugreifen sind die Methoden

.node_set(Node_Predicate)

.edges_set(Edge_Predicate)

Diese liefern Node_Sets bzw. Edge_Sets zurück.

6.3 Prädikate

Es gibt zwei Typen von Prädikaten — `Node_Predicates` und `Edge_Predicates`. Prädikate sind Funktionen, die einen Node bzw. eine Edge als Parameter erhalten und einen Wahrheitswert zurück geben. Wenn Prädikate als Parameter an eine Funktion übergeben werden, dürfen hinter dem Namen keine runden Klammern mit angegeben werden, da dies ein Aufruf der Funktion bedeutet und dann nur das Funktionsergebnis übergeben würde.

Eine größere Zahl Prädikate stehen in einem gesonderten Modul (`predicates`) zur Verfügung. Dabei handelt es sich um die Prädikate aus den Paketen `RFGs.Predicates`, `RFGs.Predicates.Edges` und `RFGs.Predicates.Nodes`. Im Gegensatz zum RFG-Modul ist es hier evtl. angebracht nicht nur das Modul selbst, sondern auch die darin enthaltenen Prädikate direkt in den lokalen Namensraum zu importieren. Dadurch können die Prädikate ohne Angabe des Moduls verwendet werden.

```
import rfgs
from predicates import *
```

Im Python können Funktionen auch durch Lambda-Ausdrücke erstellt werden, die dann direkt in der Parameterliste verwendet werden können. Da Statements in Python keine Ausdrücke sind, können diese in Lambda-Ausdrücken nicht verwendet werden. Sie lassen sich aber zum Teil durch die short-circuit Operatoren `and` und `or` und funktionale Konstrukte wie `map()` und `reduce()` emulierten. Andererseits bietet es sich bei komplexeren Aufgabe vielleicht doch an, eine Funktion zu definieren, was in Python Lambda-Ausdrücken gegenüber keine Nachteile hat. Lambda-Ausdrücken haben in Python folgende Form: `lambda Parameter: Ausdruck`. Die Verwendung von Lambda-Ausdrücken als Prädikat kann z.B. so aussehen:

```
for Node in My_RFG["BASE"].nodes(lambda node: node["myattr"] > 10):
    print Node.get_tag()
```

6.4 Edge und Node

Nodes und Edges bilden den eigentlichen Graph.

.RFG Verweis auf den zugehörigen RFG

.View Die View aus der die Edge/Node entnommen wurde.

.get_tag() Gibt den "Tag" zurück. Der Tag ist ein String der den Typ des Knoten oder der Kante angibt.

Zum Zugriff auf die Attribute unterstützen Edge und Node einen Teil der Mapping-Schnittstelle, die auch von Dictionaries benutzt wird. Als Index-Schlüssel wird dabei der Attributname oder alternativ das Attributobjekt verwendet. Die Attributoperationen beziehen sich bei Attributen, die pro View gesetzt werden können, defaultmäßig auf die View aus der die Node oder Edge entnommen wurden. Diese ist in `.View` gespeichert. Um auf Attributwerte in anderen Views zuzuweisen, kann die View als zweiter Schlüssel mit angegeben werden.

__getitem__(Attributname[, View]) aka `[Attributname, View]`

.__set_item((Attributname, View), Wert) aka Objekt[Attributname, View]
 = Wert
.__del_item(Attributname[, View]) del Objekt[Attributname, View] löscht
 ein Attribute
.get(key, default) liefert den Wert des Attributes falls er gesetzt ist, **default**
 sonst. Für key kann auch (Attributname, View) angegeben werden.
.clear_all_attributes() löscht alle Attribute in dieser View

Edge kennt zusätzlich noch:

.is_directed() returns Boolean
.source() returns Node
.target() returns Node

Node

.predecessors([Edge_Predicate, Node_Predicate]) returns Node_Set
.successors([Edge_Predicate, Node_Predicate]) returns Node_Set
.neighbors([Edge_Predicate, Node_Predicate]) returns Node_Set
.outgoings([Edge_Predicate, Node_Predicate]) returns Edge_Set
.incomings([Edge_Predicate, Node_Predicate]) returns Edge_Set
.connectings(Node, [Edge_Predicate]) returns Edge_Set
.to(Node, Edge_Predicate) returns Edge_Set

6.5 Edge_Attribute und Node_Attribute

Die beiden Attribut-Typen enthalten nicht, wie man vielleicht vermuten könnte, Attributwerte, sondern nur Metainformationen.

.name()
.permanent
.is_per_view()
.get_type()

get_type gibt dabei eine String zurück der den Typ des Attributes beschreibt. Mögliche Typen sind:

"Toggle", "Boolean", "Integer", "String", "Other"

6.6 Node_Set und Edge_Set

Node_Set und Edge_Set sind Mengen von Knoten bzw. Kanten. Sie werden immer dann verwendet, wenn Zwischenergebnisse nicht direkt in den RFG eingetragen werden sollen. So sind diese Mengen z.B. die Ergebnistypen der verschiedenen Graphfunktionen wie z.B. **.successors()**. Als Mengen unterstützen sie natürlich verschiedene Mengenoperationen:

.insert(Item) returns (Item was new)

.remove(Item) returns (Item was in Set)
.remove_all()
.__len__()
.__contains__(Item) wird von dem Schlüsselwort `in` verwendet:

```

    if Node in My_Node_Set: print "Found node!"

```

.__iter__() gibt ein Iteratorobjekt zurück.
.__lt__(other) boolescher Operator `<`, echte Teilmenge von
.__le__(other) boolescher Operator `<=`, Teilmenge von
.__eq__(other) boolescher Operator `=`
.__ne__(other) boolescher Operator `!=`
.__gt__(other) boolescher Operator `>`, echte Obermenge von
.__ge__(other) boolescher Operator `>=`, Obermenge von
.__mul__(other) arithmetrischer Opererator `*`, gibt den Durchschnitt der beiden Mengen zurück
.intersect(other) destruktiver Durchschnitt
.__add__(other) arithmetrischer Opererator `+`, gibt die Vereinigung der beiden Mengen zurück
.add(other) destruktive Vereinigung
.__sub__(other) arithmetrischer Opererator `-`, gibt die Differenz der beiden Mengen zurück
.subtract(other) destruktive Differenz
.__xor__(other) arithmetrischer Opererator `^`, gibt die symmetrische Differenz (Left - Right) + (Right - Left) zurück.
.subtract_symmetric(other) destruktive symmetrische Differenz

Zusätzlich zu den Mengenoperationen unterstützt `Node_Set` noch ein paar Methoden zur Navigavion im RFG, die weitere `Node_Sets` zurück liefern:

.predecessors([Edge_Predicate, Node_Predicate])
.successors([Edge_Predicate, Node_Predicate])

Mehr ist zur Zeit in der Ada-Bibliothek noch nicht implementiert. Wenn die beiden Set-Typen aber in Zukunft mehr genutzt werden, können sie vielleicht noch etwas erweitert werden.

6.7 Gravis

Zur Interaktion mit Gravis gibt es ein eigenes Paket (`gravis`), von dem zwei verschiedene Versionen existieren. Die eine dient tatsächlich der Zusammenarbeit mit Gravis, das andere wird im Batch-Betrieb verwendet. Im Modul für den Batch-Betrieb werden die Mechanismen wie Markierung und Auswahl emuliert. `get_rfg` liefert ein RFG-Objekt, dass aus einer via Kommandozeilenparameter übergebenen Datei geladen wurde. Dadurch können Skripte, die für Gravis geschrieben wurden auch im Batch-Betrieb albaufen, wenn sie nicht auf eine sinnvolle Markierung oder Auswahl durch den Benutzer angewiesen sind.

get_parameters() gibt ein Tupel mit den Parametern zurück
get_rfg() gibt den RFG zurück
mark(Node_Set, Edge_Set, Keep_Selection, Consider_Content)
unmark(Node_Set, Edge_Set, Consider_Content)
unmark_all()
is_marked(Item)
get_marked_nodes() returns Node_Set
get_marked_edges() returns Edge_Set
set_selection(Node_Set, Edge_Set, View)
get_selected_nodes() returns Node_Set
get_selected_edges() returns Edge_Set
get_selected_view() returns View
show(View [, Windowtype]) öffnet ein Fenster das View anzeigt, wenn es
noch nicht offen ist.

6.8 Analysen

Bereits bestehende Analysen sollten für den Benutzer ebenfalls ausführbar sein. Da die Parameter für Analysen durch die Oberfläche von Gravis in relativ engen Grenzen festliegen, bestehen alle Analysen aus jeweils einer Funktion mit sehr ähnlichen Signaturen. Die Analysen sind in einem eigenen Modul (`analysis`) untergebracht. Als Beispiel sei hier eine Funktion aus `Acid.Internal.Access` angeführt:

```

function Perform
  (The_Rfg      : in Rfgs.Rfg;
   Base_View   : in Rfgs.View;
   User_View   : in Rfgs.View;
   Environment_View : in Rfgs.View;
   AC_View_Name : in String)
return Rfgs.View;

```

aus der dann `Internal.Access(Base_View, User_View, Environment_View, AC_View_Name)` wird.

6.9 Beispiele

Wenn man nun die etwas trockene Beschreibung der verschiedenen Datentypen anschau, stellt sich die Frage wie sie in Programmen angewand werden können. Deshalb sollen hier ein paar kleine Code-Schnipsel zur Illustration gezeigt werden.

```

import rfgs
from rfgs_predicates import *
RFG = rfgs.open_rfg_plain("protocol.rfg")
print RFG.views.keys()
# => ['ENVIRONMENT', 'BASE', 'CALL', 'FILE', 'DATAFLOW', 'TYPE']

```

```

RFG.new_view("RESULT")
test_view = RFG["CALL"]

for Node in RFG["BASE"].nodes(Is_Variable):
    if Node in test_view:
        RFG["RESULT"].make_visible(Node)

RFG["RESULT"].make_visible(RFG["BASE"].node_set(Is_Type))

```

7 Binding nach C

Erster Schritt in Richtung Python-Binding ist das Binding nach C. Eigentlich sollte man erwarten, dass es für eine solch elementare Aufgabe, wie das Binding von Ada nach C, Werkzeuge existieren, die zumindest halbautomatische Unterstützung bieten können. Dies ist jedoch nicht der Fall. Wenn man sich mit dem Thema beschäftigt, erhält man vielmehr den Eindruck, dass das Binding von Ada nach C — im Gegensatz zu dem umgekehrten Fall — gerade zu exotisch ist. Anders ist es kaum zu erklären, dass an den meisten Stellen ([6], [5]) die eigentlich schwierigen Problem überhaupt nicht angesprochen werden.

Da es für das Binding von Ada nach C keine Werkzeuge gibt, ist es am günstigsten hierfür den Weg des geringsten Widerstands zu wählen und die meisten Änderungen an der Schnittstelle erst in dem Binding nach Python oder sogar erst in Python selbst durchzuführen. Alle Sprach-Konstrukte sollen — so weit es geht — ohne aufwendige Wrapper und eins zu eins nach C umgesetzt werden. Das heißt die Hauptaufgabe besteht darin für den Ada-Code entsprechende C Konstrukte zu finden, durch die der Ada-Code direkt benutzt werden kann.

Ada ist eine der wenigen Sprachen, die Konstrukte zum Export und Import in andere Sprachen enthält. Mit Hilfe des Pragma Convention können Objekte wie z.B. Typen oder Funktionen kompatibel zu anderen Sprachen ausgelegt werden. Für den Import und Export von Objekten gibt es die gleichnamigen Pragmas, die das Anwenden der entsprechenden Konvention beinhalten. Zudem gibt es spezielle Pakete (Interfaces.*) mit Funktionen und Typen zur Kooperation mit verschiedenen Sprachen.

7.1 Prozeduren und Funktionen

Funktionen können einfach mit `pragma Export(Convention, Funktionsname [,externerName])` exportiert werden. Dabei wird die Aufruf-Konvention der Funktion geändert und ein zusätzliches Linksymbol mit dem externen Namen erzeugt. Wird kein externer Name angegeben, wird der Name der Funktion in Kleinbuchstaben verwendet. Um diese Funktion nun unter C nutzen zu können, muss nur noch eine passende Deklaration in eine geeignete Header-Datei eingetragen werden.

Zum Erzeugen der C-Deklaration bieten sich reguläre Ausdrücke an, da sich die Deklarationen von Funktionen und Prozeduren zwischen Ada und C kaum unterscheiden. In Ada folgt der Returntyp nach der Funktion — in C steht er davor. In Ada steht der Typ eines Parameters mit Doppelpunkt getrennt hinter

dem Parameter — in C steht er direkt davor. In Ada sind die Parameter mit Semikolon voneinander getrennt — in C mit einem Komma. In C gibt es keine Prozeduren, sondern nur Funktionen die `void` zurückgeben.

7.1.1 Overloading

Dabei gibt es aber noch zwei Probleme. Zum Einen das Overloading: Ada unterscheidet Funktionen anhand der Signatur — C nur nach dem Namen. Um dieses Problem zu umgehen, kann man entweder diese Funktionen beim exportieren aus Ada umbenennen oder C++ verwenden, das Overloading beherrscht. Die Verwendung von C++ zieht aber weitere Probleme nach sich: Overloading wird in C++ durch Mangling der Funktionssignatur in den Linknamen realisiert. Da dieser "mangled Name" von den Typnamen der Parameter abhängt, kann Ada diesen nicht automatisch erzeugen und er muss beim `pragma Export` als Parameter angegeben werden.

Python selbst beherrscht kein Overloading, da es auf die Typprüfung der Parameter komplett verzichtet (Die Zahl der Parameter wird jedoch geprüft, kann aber auch — durch eine explizit Angabe — beliebig sein). SWIG kann jedoch Wrapper erzeugen, die auf mehrere überladene Funktionen verzweigen. Da in unserem Fall die meisten Funktionen jedoch in Methoden umgewandelt werden, die in den Namensräumen ihrer Klassen untergebracht sind, ist Overloading in den meisten Fällen nicht nötig und nur zusätzlicher Overhead.

7.1.2 Pakete

Ein zweites Problemfeld, das ähnlich gelagert ist wie das Overloading, ist die Tatsache, dass C keine Pakete kennt und deshalb Funktionen in der RFG-Bibliothek existieren, die den selben Namen haben.

Da die Zahl dieser Namenskollisionen sich im Bereich von wenigen Dutzend aufhält, wurde auf die Verwendung von C++ verzichtet und diese Funktionen von Hand umbenannt. Dies ist in einigen Fällen sogar gewünscht, da dies für die `%extend` Direktive von SWIG benötigt wird.

Das Ändern der Aufruf-Konvention stellt streng genommen eine Änderung der Schnittstelle der Bibliothek dar. Dies hat aber im Normalfall keine Auswirkungen, da Ada die Aufruf-Konvention mit berücksichtigt und der Benutzer zum Aufruf dieser Funktionen nichts zusätzlich beachten muss.

7.2 Callbacks

Die RFG-Bibliothek benutzt zur Implementierung der Prädikate sogenannte Callbacks — also Funktionen, die per Funktionspointer übergeben werden. Da diese Funktionspointer ganz gewöhnliche Pointer sind, können sie in C direkt verwendet werden. Die zu exportierenden Prädikate werden wie andere Funktionen auch mit `pragma Export` versehen. Da dieses Pragma die Calling Convention zu C ändert sind diese Funktionen dann nicht mehr zu den eigentlichen Prädikattypen kompatibel. Deshalb müssen diese beiden Typen mit Hilfen von `pragma Convention` ebenfalls auf `Convention C` umgestellt werden. Dies zieht jedoch nach sich, dass alle Prädikate — auch die, die nicht exportiert werden sollen — auf diese Konvention umgestellt werden müssen. Diese Maßnahme hat

weitreichende Konsequenzen bis hin zu den verschiedenen Benutzern der RFG-Bibliothek, da dies eine merkbare Änderung der Schnittstelle darstellt.

7.3 Datentypen

Um Ada-Code mit anderen Programmiersprachen verwenden zu können gibt es in Ada das Package `Interfaces`. Für diese Arbeit sind dabei die Unterpakete `Interfaces.C` und `Interfaces.C.Strings` interessant. Dort sind C-kompatible Typen definiert. Die Benutzung dieser Typen hat jedoch den Nachteil, dass um alle Funktionen ein Wrapper gelegt werden müsste, der diese Typen dann in die in der RFG-Bibliothek verwendeten Ada-Standardtypen umwandelt. Dies ist jedoch zum Glück nicht nötig. Wenn man sich auf Gnat als Compiler festlegt — was angesichts der Tatsache, dass er der einzige ernstzunehmende und freie Ada-Compiler ist, nicht allzu schwer fällt — kann man die Äquivalenz verschiedener Typen zwischen Ada und C ausnutzen. Im wesentlichen sind das die folgenden:

- Alle Integertypen entsprechen den jeweiligen Integertypen der anderen Sprache mit gleicher Bitzahl und Vorzeichenbehaftung. Dies ist nicht weiter verwunderlich, da die Integertypen direkt vom Prozessor verarbeitet werden und somit nur ein Layout in Frage kommt.
- Die Floattypen mit der gleichen Bitzahl entsprechen sich aus eben den selben Gründen.
- Pointer entsprechen Accesstypen. `Out`, `inout` und `access` Parameter werden als Pointer implementiert. Zwar haben primitive Typen bei `inout` Parametern Call-by-Copy-Semantik, diese wird von der Funktion selbst sichergestellt, so dass das den Aufrufer nicht zu kümmern braucht.
- Records entsprechen structs mit den gleichen Komponenten. Bei tagged records müssen Platzhalter für die Tags eingefügt werden. Tagged records sind aber wegen des Dispatchings noch ein ganz anderes Problem, um das es in einem gesonderten Kapitel gehen soll.

Problematisch sind also noch Arrays und als Spezialfall davon Strings. Dabei stellt der Zugriff auf Arrays nicht einmal das eigentliche Problem dar, da die Daten wie unter C auch einfach hintereinander im Speicher liegen. Die Übergabe von Array als Funktionsparameter ist im Ada-Standard jedoch nicht festgelegt, hängt also vom Compiler ab. Bei unbeschränkten Array-Parametern muss Ada zusätzlich zum Array noch dessen Grenzen (`Dope`) mit übergeben. Da es in der RFG-Bibliothek jedoch keine Funktion gibt, die Arrays übergeben bekommt, wurde diesem Thema nicht näher untersucht. Es ist aber zu vermuten, dass Arrays als Pointer und die Arraygrenzen als implizite Integerparameter übergeben werden und es relativ leicht ist, diese in die C-Signatur mit aufzunehmen. Dies würde dann die Verantwortung die richtigen Grenzen, anzugeben auf den C-Programmierer verlagern.

Ein weiteres Problem im diesem Zusammenhang sind Strings. Strings werden in C als Pointer auf einen Folge von Chars implementiert, die mit einem Null-Char endet. Ada hingegen packt den String in ein Array der richtigen Länge und kann sich so die Null am Ende sparen. Deshalb müssen Strings, die von Ada nach C übergeben werden, auf jeden Fall in einen neuen Speicherbereich kopiert werden. Die dazu nötigen Funktionen sind in `Interface.C.Strings`

vorhanden. D.h. aber, das der Benutzer einer Ada-Funktion, die einen String zurückgibt, diesen neu allokierten Speicher wieder frei geben muss. Umgekehrt müssen auch C-Strings zu Ada-Strings konvertiert werden; dabei wird jedoch kein neuer Speicher angelegt. Dies führt dazu, dass alle Funktionen, die Strings als Parameter erhalten oder zurückgeben, einen Ada-Wrapper erhalten müssen, der die Typumwandlung vornimmt.

Mit diesen Kochrezepten kann man im Prinzip zu Ada-Typen kompatible C-Typen erstellen. Für die automatische Übersetzung ist aber noch einiges mehr erforderlich. Es gibt in Ada verschiedene Möglichkeiten Typen bzw. Subtypen von anderen Typen abzuleiten, Record-Typen zu erweitern oder Typen umzubenennen. Zudem kann eine Typdefinition effektiv auf mehrere Pakete verteilt sein. Ein Programm, das beliebige Adatypendefinitionen parsen und weiterverarbeiten kann, ist deshalb schon fast ein Ada-Frontend. Es wurde klar wurde, dass das über die Möglichkeiten meiner Arbeit hinaus gehen würde. Deshalb wurde versucht um diese Aufgabe herum zu kommen. Dazu mussten die Typen, um die es überhaupt geht, näher untersucht werden. View und die beiden Attribute-Typen sind Integers, View_Set ein 64-Bitfeld — also auch ein Integer. RFG ist ein Pointer auf eine komplexe, private Datenstruktur; da auf diese Datenstruktur von C aus nicht zugegriffen werden soll, reich hier also ein Pointer. Node und Edge und deren Untertypen sind zwar tagged Records werden aber fast ausschließlich über deren Access-all-Typen angesprochen. Da auf den direkten Zugriff auf Recordkomponenten verzichtet werden soll, kann man diese Typen ebenfalls als Pointer modellieren. Um zu verhindern, dass die einzelnen Pointer-Typen zueinander zuweisungskompatibel sind, kann man sie einfach als Pointer leere Structs implementieren. Damit ist die Typenproblematik weitgehendst gelöst. (Es gibt da natürlich noch ein paar andere Datentypen wie z.B. verschiedene Iteratoren; mit diesen kann aber ähnlich verfahren werden.)

7.4 Binden von Adabinärdateien

Ein wichtiger Teil beim Erstellen von Adaprogrammen ist das Binden. Dabei wird Initialisierungscode für das künftige Programm generiert. Werden Ada-Bibliotheken in Programme verwendet, deren Hauptprogramm nicht in Ada implementiert ist, muss sich der Programmierer selbst darum kümmern, dass dieser Initialisierungscode erzeugt und eingebunden wird. Bei GNAT gibt es dazu den Befehl `gnatbind -n`, der die betroffenen Pakete als Parameter übergeben bekommt. Er erzeugt die beiden Funktionen `adainit` und `adafinal`, die vom Hauptprogramm vor und nach der Benutzung des Ada-Codes aufgerufen werden müssen. Da Gravis selbst in Ada geschrieben ist, trifft diese Problematik eigentlich nicht zu. Will man aber die Pythonmodule auch einfach so unter dem Pythoninterpreter nutzen, so muss man die verwendeten Ada-Pakete binden und auch `adainit` und `adafinal` aufrufen.

7.5 Dispatching

Dispatching ist die Auswahl der aktuell aufzurufenden Funktion anhand des genauen Subtyps eines oder mehrerer Parameter. Dispatching ist eines der ausgewiesenen Merkmale objektorientierter Programmierung.

Zwar brauchen die großen Typhierarchien nicht exportiert werden, aber dennoch gibt es einige wenige Primitive — in erster Linie `Get_Tag` — die für eine

sinnvolle Anwendung des RFG benötigt werden.

In Ada muss Dispatching explizit durch den Programmierer durch die Verwendung eines 'CLASS Types verlangt werden. Die Primitive die mit `pragma Export` exportiert werden können, führen also kein Dispatching durch. C — als rein prozedurale Sprache — ist Dispatching völlig fremd. Deshalb gibt es keine direkt auf der Hand liegende Lösung für dieses Problem. Interessanterweise gibt es aber gleich mehrere Lösungsansätze:

7.5.1 Dispatching in Ada

Die erste Möglichkeit ist, einen Wrapper um die Primitive zu schreiben. Dieser bekommt statt dem tagged Type den zugehörigen access all Typ übergeben. Dadurch wird in diesem Wrapper vom Compiler Dispatchingcode erzeugt. Diese Wrapper müssen nur für die Superklasse, die dieses Primitiv einführt, geschrieben werden, und kann dann exportiert werden. Dies führt dann dazu, dass man in C eine Funktion erhält, die intern Dispatching vornimmt, obwohl es dieses Konzept in C eigentlich nicht gibt.

7.5.2 Dispatching in C++

Da GNAT zur GCC gehört, teilen sich der GNU Ada und der GNU C++ Compiler das selbe Backend. Dadurch sollten eigentlich C++ Klassen und Ada tagged Types gleich implementiert werden. Erzeugt man nun die Deklaration einer C++Klassenhierarchie, die den Ada tagged Record entspricht, kann man die Adatypen und die zugehörigen Funktionen unter C++ weiter verwenden und kann die Typhierarchie sogar in C++ erweitern. Dies ist sicher eine sehr Attraktive Art das Binding nach C(++) zu realisieren. Leider gibt es ein paar Einwände, die mich sehr früh dazugebracht haben, diese Möglichkeit nicht weiter zu verfolgen:

- Methoden in C++ erhalten die Instanz als impliziten ersten Parameter. In Ada gilt diese Einschränkung für Primitive nicht. Es besteht also die Gefahr, dass manche Primitive gar nicht in C++ dargestellt werden können.
- Die Typhierarchie soll eigentlich nicht exportiert und so den Umfang des Bindings stark beschränkt werden. Bei dieser Art des Bindings ist das jedoch nicht möglich.
- Das gemeinsame Backend von C++ und Adacompiler ist natürlich rein GNAT/GCC spezifisch. Dies ist aber eigentlich keine wesentliche Einschränkung, da in jedem Fall GCC spezifische Annahmen gemacht werden müssen. Die Annahme auf die Entsprechung von primitiven Typen sind aber eigentlich so fundamental, dass durchaus eine Chance besteht, dass sie auch für andere Compiler gelten. Bei dem C++ Backend ist dies aber eher unwahrscheinlich.
- Hauptgrund diese Lösung frühzeitig auszuschliessen war aber, dass es für diese Übereinstimmung keinerlei Hinweise in der Dokumentation gibt. Dies heißt, dass selbst wenn das Binding so implementierbar sein sollte, es nicht auszuschliessen ist, dass bei künftigen Compilerversionen Inkompatibilitäten entstehen.

- Zum Erzeugen der C++-Klassen müsste ein Tool entwickelt werden. Dies erschien mir ein relativ großer Aufwand zu sein, bei dem man sicher nicht darum herum kommen würde, sich mit den Details des GCC auseinanderzusetzen.
- Wenn es einfach oder auch nur mit begrenztem Aufwand zu machen ist, müsste es eigentlich ein Tool geben, das so etwas tut. Ich habe auch nach längerem suchen, nicht einmal die Spur eines solchen Tools gefunden.

7.5.3 Dispatching in Python

Python ist als objektorientierte Sprache natürlich auch in der Lage, Dispatching durchzuführen. Dazu muss die gesamte Typhierarchie bis nach Python exportiert werden. Jede Reimplementierung einer Methode muss dabei der entsprechenden Pythonklasse angehängt werden. Rückgabewerte von Funktionen müssen dabei in die exakte Klasse umgewandelt werden. Dies führt zu einem hohen Implementierungsaufwand, hat aber den Vorteil, dass die Typhierarchie samt ihrer Implementierung bis in die Pythonschnittstelle durch scheint. Dafür müsste man sich aber noch ein Tool schreiben, das die Pythonklassen generiert, da SWIG für diesen — aus Sicht eines C/C++ Programmierers sehr exotischen Fall — keine Unterstützung liefert.

Da es sich wirklich nur um 4 Primitive handelt, die für die Schnittstelle von Interesse sind und zudem die Polymorphie aus der RFG-Bibliothek verschwinden soll, habe ich beschlossen, um diese Funktionen Ada-Wrapper zu schreiben. Diese liegen, wie alle für das Binding benötigten Hilfsfunktionen, in RFGs.Binding.

7.6 Generics

Von generischer Programmierung spricht man, wenn Code durch Parameter den aktuellen Bedürfnissen angepasst werden kann. Im Gegensatz zu normalen Funktionsparametern findet die Auswertung dieser Parameter zur Compilezeit statt. Dadurch können auch Typen als Parameter verwendet oder Typen durch die Parameter mit beeinflusst werden. Im Gegensatz zur Polymorphie erlauben generische Programmteile eine statische Prüfung auf einen oder mehrere anzugebende Typen und nicht nur auf die Zugehörigkeit zu einer festen Oberklasse. Ada erlaubt generische Funktionen und generische Pakete. Als Parameter sind Werte, Typen, Funktionen und Pakete zugelassen.

Wenn es um den Export von generischen Codeteilen geht muss man verschiedene Fälle unterscheiden.

7.6.1 Generisch implementierte Funktionen

Einige Funktionen in der RFG-Bibliothek sind generisch implementiert. Das heißt, im Package Body wird eine generische Funktion instanziiert und damit eine in der Package Specification deklarierte Funktion implementiert. Hier ein Beispiel aus RFGs.Predicates.Edges :

```
function Gen_Reference_Edge is
  new Is_Edge_Of_Class (RFGs.Edges.Reference_Edge);
```

```

function Is_Reference_Edge
(The_RFG   : RFGs.RFG      := Ignore_RFG;
 The_Views : RFGs.View_Set := Ignore_Views;
 The_Edge  : RFGs.Edge_Ptr)
return Boolean renames Gen_Reference_Edge;

```

Derartige Funktionen können wie andere Funktionen auch mit pragma Export exportiert werden. Es ist lediglich darauf zu achten, dass die instanziierte Funktion auch mittels Pragma Convention die selbe Aufruf-Konvention zugewiesen bekommt — also in unserem Fall C.

7.6.2 Generisch implementierte Pakete

Generische Pakete sind eine sehr mächtige Möglichkeit ähnliche Funktionalität für verschiedene Typen oder Anwendungsgebiete zu realisieren. Der Vorteil dieser Vorgehensweise ist, dass die einzelne Instanzierung nur eine oder auf jeden Fall nur wenige Zeilen benötigt. Das führt aber dazu, dass die konkreten Funktionen nirgends im Sourcecode auftauchen. Deshalb ist es natürlich auch nicht möglich sie durch Dazuschreiben eines Pragma Exports zu exportieren. Eigentlich wäre es deshalb wünschenswert, die Export Pragmas direkt in der Spezifikation des generischen Pakets verwenden zu können. Dies ist im Prinzip auch möglich, führt aber zu den folgenden Problemen:

Pragma Export ist nur im Spezifikationsteil erlaubt. Wird ein Paket in einem Paket Rumpf instanziiert, meldet GNAT einen Fehler. Sollte das Paket also an irgend einer Stelle zur Implementierung herangezogen werden (z.B zur Implementierung eines privaten Typs), wird dieser Code durch die Änderung des generischen Pakets ungültig. Zweites Problem ist der Name der zu exportierenden Funktion. Leider werden generische Parameter vom Ada-Compiler nicht als statisch angesehen und es gibt auch keine Möglichkeit auf den Namen von übergebenen Typen, die zwangsweise statisch sein müssen, zuzugreifen. Damit kann der externen Namen nicht von der Instanz abhängig gemacht werden, sodass es bei der zweiten Instanzierung zu einem Konflikt zwischen den Linknamen kommt.

Leider einziger Ausweg aus dem Dilemma ist es neben, der Instanz des Pakets ein weiteres Paket von Hand zu erstellen, dass die gleiche Schnittstelle hat. Die Implementierung dieses Pakets besteht aber aus Rename-Anweisungen von den Funktionen der Instanz. Damit dies aber funktioniert, müssen die Funktionen des instanziierten Pakets schon die richtige Aufruf-Konvention haben. Leider reicht dazu nicht, Pragma Convention auf die Instanz anzuwenden. Dies ist zwar legal, hat aber keinerlei Auswirkungen. Pragma Convention muss auf jede Funktion im generischen Paket angewandt werden, was erstens sehr umständlich und zweitens ärgerlich ist, dass es die Schnittstelle des generischen Pakets ändert, dass vielleicht noch an anderer Stelle benutzt wird, wo diese Aufruf-Konvention weder erforderlich noch erwünscht ist.

Das Zusammenspiel von Pragma Export und generischen Paketen ist der Tiefpunkt bei dem Erstellen des Bindings von Ada nach C. Zum Glück betrifft es nur zwei Instanzen des selben Paketes (Ordered_Set, das als Node_Ptrs.Sets und Edge_Ptrs.Sets instanziiert wird). Leider schreibt der Ada95-Standard die Semantik von Pragma Export nur für Funktionen vor. Dennoch könnte man sich hier von GNAT etwas mehr Unterstützung wünschen.

7.6.3 Generische Schnittstellen

Generische Schnittstellen — also generische Funktionen oder Pakete, die vom Benutzer instanziiert werden sollen — können überhaupt nicht in eine andere Sprache exportiert werden. Da generische Objekte erst bei ihrer Instanzierung kompiliert werden, existieren die generischen Funktionen und Pakete nur im Sourcecode. Deshalb gibt es keinen ausführbaren Code, auf den eine fremde Programmiersprache zugreifen könnte. Da hilft es dieser Sprache dann auch nicht, selbst mit generischer Programmierung umgehen zu können. Die einzige Möglichkeit wäre es, das generische Objekt komplett in die fremde Sprache zu übersetzen. Obwohl in der RFG-Bibliothek auch einige generische Schnittstellen enthalten sind, stellt diese Einschränkung zum Glück kein größeres Problem dar. Dies ist aber nur dem Umstand zu verdanken, dass diese Teile der RFG-Bibliothek nur als Alternative angeboten werden und die gleiche Funktionalität auch auf andere Art implementiert werden kann.

7.6.4 Funktionen mit generischen Funktionsparametern

Eine Ausnahme dieser Regel sind generische Funktionen mit ausschließlich generischen Funktionsparametern. Da Funktionen nicht zu einer anderen Beeinflussung des erzeugten Codes verwendet werden können als die Funktion aufzurufen, ist es möglich, die generische Funktion in eine Funktion mit einem nicht generischen Funktionsparameter umzuwandeln. Dazu wird der Funktionsparameter in eine Variable geschrieben und eine Funktion, die diese Variable ausliest und die Funktion aufruft, wird als generischer Parameter für die Instanzierung der generischen Funktion genutzt. Bedingung für dieses vorgehen ist natürlich, dass die Signatur der Funktion, die als generischer Parameter dient, festliegt; dies ist in Ada jedoch immer der Fall, da die Typen der Parameter vorher festliegen müssen und wir nach Voraussetzung generische Typparameter, die die Signatur einer Funktion beeinflussen könnten, ausgeschlossen haben.

```
type Proceed_Edge is access
  procedure (The_Edge : Edge_Ptr);

-- Non generic version of RFGs.Accessors.Traverse_Outgoings
procedure Traverse_Outgoings
(The_RFG      : RFG;
 The_Views    : View_Set;
 The_Node     : Node_Ptr;
 An_Edge_Predicate : RFGs.Predicates.Edge_Predicate := null;
 A_Node_Predicate : RFGS.Predicates.Node_Predicate := null;
 Execute      : Proceed_Edge) is

  Execute_Tmp : Proceed_Edge;

  -- procedure that will call the parameter Execute
  -- via Execute_Tmp
  procedure Execute_Wrapper(The_Edge : Edge_Ptr) is
  begin
    Execute_Tmp(The_Edge);
  end Execute_Wrapper;
```

```

procedure My_Traverse_Outgoings is
  new RFGs.Accessors.Traverse_Outgoings(Execute_Wrapper);
-- Instance that will call Execute_Wrapper

begin
  Execute_Tmp := Execute;

  -- Call the instance
  My_Traverse_Outgoings(The_RFG,
                        The_Views,
                        The_Node,
                        An_Edge_Predicate,
                        A_Node_Predicate);
end Traverse_Outgoings;

```

Möglich ist diese Art des Wrapper natürlich nur, weil hier eigentlich gar keine Generizität erforderlich ist und deshalb die Probleme, die normalerweise beim Exportieren einer generischen Schnittstelle auftreten, gar nicht zum Tragen kommen. Das Ganze hat also eigentlich eine gewisse Exotik. Da Generizität aber in Ada aus mehreren Gründen als guter Stil angesehen wird, kommt auch dieser Fall relativ häufig vor und stellt in der RFG-Bibliothek einen Großteil der generischen Teile der Benutzerschnittstelle dar. Da auf diese Funktionen aber leicht verzichtet werden kann, die Benutzerschnittstelle des Binding eher kompakt gehalten werden sollte und weil Funktionsparameter beim Binding nach Python auch noch ein paar Probleme bereiten, wurden keine dieser Funktionen exportiert.

7.7 Der C-Binding Generator

Wenn die prinzipielle Umsetzung des Ada-Codes nach C klar ist, muss als nächstes über die Implementierung nachgedacht werden. Es stand eigentlich von Anfang an fest, dass das Binding nach C nicht von Hand erstellt werden soll. Dafür gibt es eine ganze Reihe guter Gründe:

- Mit einem Programm ist man damit flexibler. Der Arbeitsaufwand fällt einmal an und dannach kann man ohne Probleme weitere Funktionen exportierten ohne sich Gedanken zu machen, wie lange das wohl brauchen wird. Die ist insbesondere für künftige Erweiterungen, die nicht mehr Bestandteil dieser Arbeit sind, wichtig.
- Zudem war am Anfang auch noch eine ganze Weile nicht klar, wie genau die C-Header aussehen sollten. Gerade die %extend Direktiven von SWIG verlangt eine gesonderte Behandlung. Es war unklar, ob die Defaultwerte mit übernommen werden können oder sollen und ob nur die Parametertypen oder auch die Parameternamen in den C Headern auftauchen sollen und welche Auswirkungen das dann haben würde. Deshalb war es wichtig hier, möglichst lange flexibel zu bleiben und das konnte nur mit einer automatisierten Lösung geschehen.
- Da es keinen Compiler gibt, der die C-Header auf Korrektheit prüft, kann die Fehlersuche an dieser Stelle sehr langwierig sein. Bei einem Programm

ist die Chance höher, dass sich ein Fehler in höherem Maße auswirkt und deshalb auch leichter gefunden werden kann.

- Das Übersetzen von Ada-Spezifikationen nach C ist eine stupide Arbeit, mit der man sich nicht längere Zeit beschäftigen will. Zudem besteht die Gefahr, dass man Fehler über längere Zeit macht und diese dann nacharbeiten muss. Als Alternative kann man gleich Testcode schreiben, was ein gehöriger Aufwand darstellt.

Als Sprache für dieses Programm kamen eigentlich nur Ada, C oder Python in Betracht, um die Zahl der verwendeten Programmiersprachen/Werkzeuge nicht noch weiter zu erhöhen. Da für diese Aufgabe eigentlich ausschließlich Verarbeitung von Text erforderlich ist, bietet sich Python als Skriptsprache natürlich an. Ada ist dafür bekannt, dass seine Stringunterstützung auf Grund der starken Typisierung nicht gerade programmiererfreundlich ist. Strings in C sind ebenfalls eher umständlich. Zudem vereinfacht die Garbage-Collection und das re-Modul (Regular Expressions) von Python die ganze Sache. Anhand der Erfahrungen, die ich mit den drei Sprachen und meinen Fähigkeit in ihnen gemacht hatte, schätzte ich, dass die Implementierung in Python nur halb so lang sein würde wie eine Implementierung in Ada oder C und wahrscheinlich nur ein Drittel bis ein Fünftel so lange dauern würde. Das war mir Argument genug.

Die Implementierung besteht aus drei Teilen:

type_map.py setzt die Ada-Typen nach C um. Die Übersetzung selbst muss von Hand vorgenommen werden. Die Namen der C Typen werden in einem Python-Dictionary mit dem kleingeschriebenen Adatyp als Schlüssel gespeichert. Die eigentlichen Deklarationen befinden sich in einer externen Headerdatei, die an geeigneter Stelle eingebunden werden muss. Die Klasse `type_map` versucht nun, einen Typ in diesem Dictionary zu finden indem sie ihn in Kleinbuchstaben konvertiert und dann Stück für Stück die Paketangaben entfernt. Für besonders komplizierte Fälle — hier sind die Untertypen von `Node` und `Edge` zu nennen — kann auch ein regulärer Ausdruck verwendet werden. Zusätzlich kann `type_map` noch die Übergabemodi `out`, `inout` und `access` behandeln, indem sie einen weiteren Pointer zu dem Typ hinzufügt. Will man einen neuen Adatyp diesem System hinzufügen, muss man dessen C-Äquivalent in einen C-Header eintragen (i.d.R. in `rfgs-types.h`) und das Paar Ada-Typname:C-Typename in das Dictionary eintragen.

parser.py enthält — wie der Name schon sagt — die Routinen zum Parsen von Ada Specification Packages. Zusätzlich enthält es noch Routinen, um die erkannten Funktionsdeklarationen nach C zu übersetzen. Um das Parsing zu vereinfachen, macht `parser.py` einige Annahmen, die für den Code der RFG-Bibliothek zutreffend sind:

- Es gibt keine vollständigen, auskommentierten Funktionsdeklarationen oder Export-Pragmas. Dies erlaubt es, die Dateien nicht wirklich parsen zu müssen, sondern nur nach den Deklarationen zu suchen.
- Die Export-Pragmas befinden sich direkt hinter der Funktion, zu der sie gehören, oder hinter dem zur Funktion gehörenden Kommentar. Da es mehrfach vorkommt, dass mehrere überladene Funktionen

exportiert werden sollen und die Export-Pragmas nur den Funktionsnamen übergeben bekommen, kann hier nur auf die Reihenfolge bzw. Lage zueinander Bezug genommen werden. Ada selbst scheint da nicht sehr viel anders vorzugehen.

- Der Code ist "gutartig"; d.h. die Typen oder Pakete werden nicht umbenannt und es werden auch sonst keine ungewöhnlichen Sprachfeatures ausgenutzt.
- Die Export-Pragmas haben ein einfaches Format und die Parameter sind nur Namen und Strings und werden nicht zusammengesetzt.

Parser.py durchsucht die angegebene Datei sowohl nach Funktionsdeklarationen als auch nach Export-Pragmas und versucht dann, sie einander zuzuordnen. Der Name der künftigen C-Funktion wird dabei aus dem `External_Name` Parameter des Export-Pragmas entnommen oder ansonsten aus dem Namen der Funktion berechnet, d.h. der Name wird in Kleinbuchstaben umgewandelt. Dadurch ist es möglich, Funktionen einfach durch Angabe des externen Namens zu diesem umzubenennen.

header_generator.py ist das Hauptprogramm. Seine einzige Aufgabe ist es, eine Datei nach Zeilen mit zwei Prozentzeichen am Anfang zu durchsuchen und den dahinterstehenden Befehl auszuführen. Um die Implementierung dafür so simpel wie möglich zu halten, handelt es sich dabei einfach um eine Zeile Pythoncode, die mit `eval` ausgeführt und deren Ergebnis an dieser Stelle eingefügt wird. Gedacht ist natürlich, dass man sich dabei auf die zwei vorgesehenen Funktionen beschränkt, die aus `parser.py` importiert werden.

- `%%extended(Dateiname, Typname)` fügt Funktionsdeklarationen ein, die für die SWIG-Direktive `%extend` geeignet sind. Näheres zu SWIG und der `%extend` Direktive in den entsprechenden Kapiteln.
- `%%header(Dateiname [, RE, RE, ...])` fügt an dieser Stelle die Funktionsdeklarationen aus der angegebenen Datei ein, die *nicht* auf die übergebenen regulären Ausdrücke passen. Die regulären Ausdrücke RE sind dabei optional und können beliebig viele sein. Sie dienen dazu die Funktionen die bereits mittels `%%extended` verarbeitet werden, nicht nochmals einzufügen.

In beiden Fällen werden natürlich nur diejenigen Funktionen berücksichtigt, für die auch ein `Pragma Export` existiert.

In der Praxis kann das so aussehen:

```
%%headers("../src/rfgs.ads", "RFG.*", "new_RFG")
%extend RFG {
%%extended("../src/rfgs.ads", "RFG")
}
```

8 Binding nach Python

8.1 SWIG

Das Binding von C nach Python unterscheidet sich grundsätzlich von dem Binding von Ada nach C. Während es bei dem C-Binding im wesentlichen darum geht, eine dem Ada-Code entsprechende C-Deklaration zu finden, muss bei dem Binding nach Python jeder Parameter zwischen dem C- und dem entsprechenden Python-Typ konvertiert werden. Zudem müssen die Funktionen und auch die zusätzlichen Typen bei dem Python-Laufzeitsystem angemeldet werden. Da dies deutlich komplizierter ist als das C-Binding, stand von Anfang fest, dass dafür nur ein Binding-Generator in Frage kommt. Neben einer ganzen Reihe von Binding-Generatoren für einzelne Sprachen — durchaus auch für Python — bin ich dabei auf SWIG gestoßen. Ein Binding-Generator für eine ganze Reihe von Sprachen: Guile (GNU Scheme), Java, Mzscheme, OCAML, Perl, PHP, Python, Ruby, Tcl, Chicken (ein Scheme Compiler) und C# (erst seit kurzem)

Dies ermöglichte am Anfang der Arbeit von der Auswahl der verwendeten Skriptsprache zu abstrahieren und sich erst einmal auf die grundsätzliche Machbarkeit zu konzentrieren. Zudem hat SWIG eine ganze Reihe an weiteren Eigenschaften, die es zum optimalen Werkzeug für diese Aufgabe macht:

- SWIG wurde bereits 1995 entwickelt und ist immer noch unter aktiver Entwicklung. So wurde z.B. das C#-Backend erst während dieser Studienarbeit entwickelt. 2002 wurden allein sechs neue Versionen released.
- Die Dokumentation, die SWIG beiliegt, ist relativ umfangreich und macht einen guten Eindruck.
- SWIG kann C-Header direkt in ein Binding übersetzen. Dies ist in vielen Fällen nicht die optimale Lösung, da man in der Regel die Umsetzung in die Zielsprache noch beeinflussen will, zeigt aber welches Maß an Automatisierung SWIG bei dem Prozess ermöglicht.
- SWIG erzeugt ein Binding, das nach dem Compilieren direkt verwendet werden kann, ohne dass noch Nacharbeiten am erzeugten Quellcode nötig sind. Damit erspart man sich zum Einen das Arbeiten an erzeugtem Code zum Anderen kann man so nach Änderungen einfach durch einen Lauf des Generator das neue Binding erstellen und sitzt nicht auf dem einmal erzeugten und dann geänderten Code fest.
- SWIG beherrscht neben ANSI C auch ANSI C++ als Eingabesprache. Da am Anfang noch unklar war, ob nicht doch für die Umsetzung des einen oder anderen Sprachfeatures von Ada C++ benötigt werden würde, war die Fähigkeit, auch C++ verarbeiten zu können, sehr beruhigend.
- Zusätzlich zur hohen Automatisierung bietet tSWIG aber auch einige sehr mächtige Konzepte, um das erzeugte Binding zu beeinflussen, wie z.B. das Einfügen von C-Code oder Code der Zielsprache an unterschiedlichen Stellen, Erweitern von Recordtypen zu Klassen mit Methoden, volle Kontrolle über das Umwandeln von Typen, Umwandeln von Out-Parametern in Rückgabewerte, automatische Freigabe von Rückgabewerten, Einbindung von C-Objekten in die Speicherverwaltung der Zielsprache und vieles mehr.

Im Rahmen dieser Arbeit kann natürlich nicht im Detail auf die Funktionalität von SWIG eingegangen werden. Deshalb werden hier nur die groben Konzepte und deren Auswirkungen auf diese Arbeit beschrieben. Dem geneigten Leser sei die sehr gute Dokumentation (SWIG1.3 Development Documentation [7]) empfohlen, der die Tatsache, dass sie der Entwicklung von SWIG etwas hinterherhinkt, praktisch nie anzumerken war.

8.2 Das SWIG Python Backend

SWIG setzt die Werte primitiver C-Typen direkt in Pythonobjekte um:

C	Python
int, long	plain Integer
long long	Long Integer
float, double	Float
char	String der Länge eins
char*	String

Zusammengesetzte Typen wie Records und Klassen werden nur als typisierte Pointer exportiert und für Recordkomponenten werden Zugriffsfunktionen erzeugt.

Da die Zuweisung in Python Referenzsemantik besitzt und aufgrund deren Implementierung mittels Python-Dictionaries, können globale Variable nicht einfach in den globalen Namensraum von Python eingeblendet werden. SWIG erzeugt deshalb ein spezielles Objekt das die Variablen als Attribute enthält. Dessen Defaultname `cvar` kann natürlich geändert werden.

8.3 Python/C API

Für einige Fälle kommt man mit den von SWIG vordefinierten Möglichkeiten nicht weiter. Dann kann man direkt das Python/C API [12] verwenden, um Code zu schreiben, der direkt auf Pythonfunktionalität zurückgreift. Das Python/C API bietet unter anderem die Möglichkeit Python Standardtypen zu erzeugen und in die C-Typen zurück zu konvertieren. Ausserdem stellt es C-Funktionen zur Verfügung, die den Methoden, der Python Typen entsprechen. Darüber hinaus stellte es noch die Funktionen, die zum Einbetten des Pythoninterpreters benötigt werden, bereit.

8.3.1 Speicherverwaltung

Ein Aspekt, der bei der Benutzung der Python/C APIs beachtet werden muss, ist die Speicherverwaltung. Python verwendet zur Speicherverwaltung einen Reference-Count, der durch eine Garbage-Collection, die auch zyklische Abhängigkeiten erkennen kann, unterstützt wird. Durch diese Kombination ist es Python möglich mit in C implementierten Typen umzugehen, bei denen keine Erreichbarkeitsanalyse durchgeführt werden kann, da ihre interne Struktur dem Interpreter nicht bekannt sind. Von diesen Fähigkeiten von Python macht SWIG aber leider keinen Gebrauch, sondern verwendet einen anderen Mechanismus (Shadow Classes). Wichtig ist es in Codestücken, die das Python/C API verwenden, das Reference-Counting nicht durcheinander zu bringen und beim Speichern und Freigeben von Objektreferenzen die Makros `Py_INCREF()` und `Py_DECREF()` aufzurufen.

8.4 Shadow Classes

Dadurch, dass SWIG alle komplexen Objekte nur als Pointer exportiert, haben diese Typen keine Struktur. Deshalb erzeugt SWIG so genannte Shadow Classes. Dabei handelt es sich um gewöhnliche Pythonklassen, deren Methoden die Methoden bzw. Zugriffsfunktionen des von ihnen repräsentierten Typs aufrufen. Die Instanzen dieser Klassen erhalten einen Pointer auf das von ihnen repräsentierte Objekt als Attribut `.this`. Durch ein weiteres Attribut (`.thisown`) wird angegeben, ob das C-Objekt durch den Destruktor des Pythonobjekt mit freigegeben werden soll. Dadurch werden die C-Typen mit in die Speicherverwaltung von Python aufgenommen. Dies hat zudem den Vorteil, dass den Shadow Classes so einfach — wie sonst unter Python auch üblich — weitere Attribute angehängt werden kann.

Leider stellte sich bald heraus, dass SWIG Shadow Classes nur für komplexe Typen erzeugen kann. View und Attribute waren jedoch Integer-Typen, die aber nach meinem Entwurf eine ganze Reihe von Methoden und auch entsprechend wichtige Aufgaben in der Programmierschnittstelle bekommen sollten. Um sie dennoch als Pythonklasse implementieren zu können gab es zwei Möglichkeiten: Zum Einen kann man sie als ein Struct mit nur einer Integerkomponente darstellen, was SWIG davon überzeugt, dass es sich um einen komplexen Typ handelt. Das führt aber leider auch dazu, dass der erzeugte Wrappercode entsprechend komplex ausfällt. Die zweite Möglichkeit war, die Pythonklasse einfach selbst zu implementieren. Nachdem sich herausgestellt hatte, dass sich die `%extend` Direktive nicht mit dem Zusammenfassen von Parametern verträgt (mehr dazu in dem entsprechenden Kapitel) und ich mir zudem nicht 100%ig sicher war, dass die Variante mit dem einelementigen Structs nicht doch noch zu Komplikationen führen würde, beschloss ich die beiden Klassen selbst zu implementieren. Ein Großteil — dabei handelt es sich nur um wenige Zeilen Pythoncode — übernahm ich aus dem von SWIG erzeugten Code.

8.5 Die `%extend` Direktive

Mit Hilfe der `%extend` Direktive können Klassen aber auch Records um Methoden erweitert werden, die im C-Code eigenständige Funktionen sind. Diese werden genau wie echte Methoden oder die generierten Zugriffsfunktionen der Shadow Class hinzugefügt. Dabei muss — wie bei C++ Methoden auch — der erste Parameter, der das Objekt selbst übergibt, weggelassen werden. Er wird von SWIG bei der Generierung des Wrappercodes wieder hinzugefügt. Dies hat jedoch einen massiven Nachteil: Die C-Funktion muss eine für eine Methode gültige Signatur haben. Dabei würde es im Prinzip reichen, wenn die generierte Pythonfunktion eine gültige Methode der Shadow Class wäre. Dieses Problem tritt in unserem Fall in zwei verschiedenen Formen in Erscheinung. Zum Einen werden häufig mehrere Parameter zu einem zusammengefasst. Oft würde man gerne die Funktion der Shadow Class des zusammengefassten Parameters zuordnen, was nicht geht, da die C-Funktion eine andere Signatur hat. So soll zum Beispiel aus

```
function Successors
(The_RFG   : RFG;
 The_Views : View_Set;
 The_Node  : Node_Ptr)
```

```

    return Node_Ptrs.Sets.Set;
-- Returns all successors of 'The_Node' visible within 'The_Views'
-- and reachable by edges visible within 'The_Views'.

```

`Node.successors()` gemacht werden. `The_RFG` und `The_Views` werden durch eine dreistellige Typemap aus den Attributen des Shadow-Objekts von `Node` ausgelesen. Da die `%extend` Direktive bei Methoden des Typs `Node` immer einen Parameter vom Typ `Node*` am Anfang der Parameterliste einfügt, kann die Signatur der Adafunktion bzw. deren C-Entsprechung nicht angegeben werden.

Zum Anderen tritt dieses Problem auf, wenn das Objekt nicht durch einen einfacher Pointer übergeben wird, sondern wie z.B. bei den Ada "Konstruktoren" und "Destruktoren" als Doppelpointer. Dies wird meist durch eine Typemap korrigiert, um dem Benutzer Doppelpointertypen zu ersparen. Da SWIG hier aber eine passende C Signatur erwartet, können auch diese Funktionen nicht mit der `%extend` Direktive hinzugefügt werden.

Da die `%extend` Direktive pro Methode nur eine einfache Zeile Python Code erzeugt, ist es sehr einfach, auf sie zu verzichten und die entsprechenden Funktionen selbst der Shadow Class hinzuzufügen. Dies hat zudem den Vorteil, dass in der Pythonmethode zusätzlicher Code stehen kann.

Ein weiteres Problem, dass im Zusammenhang mit der `%extend` Direktive auftritt und in der Dokumentation leider nicht ausreichend beschrieben ist, ist, dass der Block der `%extend` Direktive direkt an den Block des erweiterten Typs angehängt wird. Das heißt, dass die `%extend` Direktive in der Umgebung dieses Typs ausgewertet wird und nicht etwa dort, wo sie in der SWIG-Datei steht. Das führt dazu, dass Typemaps, die z.B. direkt vor der `%extend` Direktive stehen, keinerlei Auswirkung auf diese haben. Um dieses Problem zu umgehen, muss die Typemap entweder vor den erweiterten Typ oder aber in die `%extend` Direktive verschoben werden.

8.6 Typemaps

Ein wichtiger Teil von SWIG sind die sogenannten Typemaps. Dabei handelt es sich um Schnipsel C-Code, die in die Wrapper kopiert werden um einzelne Parameter zu behandeln. Typemaps sind von der Zielsprache abhängig und stellen einen wesentlichen Teil der jeweiligen Backends dar. Die Typemaps für Python verwenden das Python/C API. Es gibt verschiedene Arten von Typemaps, die für verschiedene Fälle an die jeweils richtige Stelle im Wrapper eingefügt werden:

in Umwandlung von Funktionsparametern nach C

out Umwandlung des Rückgabewert in die Zielsprache

arginit belegt den C Parameter vor, damit die Typumwandlung einen initialisierten Wert vorfindet

default legt den Defaultwert fest.

check prüft den Übergebenen Wert auf Konsistenz

argout wandelt einen per Pointerparameter zurückgegebenen Wert in einen Rückgabewert der Funktion um

freearg gibt temporär erzeugte Parameter wieder frei

newfree gibt Rückgabewerte frei

memberin kopiert bereits konvertierte Werte in Arrays oder Recordkomponenten

varin Zuweisung an eine globale Variable

varout Lesen einer globalen Variablen

Beim Wrappen von Funktionen wird von SWIG nach den passendsten Typemaps gesucht. Dazu hat jede Typemap eine Signatur, die aus einem Typ oder einem Typ und einem Parameternamen besteht. Außerdem gibt es noch mehrstellige Typemaps.

Für die primitiven Typen gibt es fertige `in`, `out`, `varin`, `varout` und `memberin` Typemaps, die nur den Typ in der Signatur haben und damit auf alle Parameter dieses Typs angewandt werden, solange keine genauere Typemap existiert. Für einige andere gibt es fertige Typemaps mit speziellen Parameternamen, die dann im Einzelfall der jeweiligen Kombination aus Typ und Parameternamen zugewiesen werden können.

Hier als Beispiel zwei der Standardtypemaps für Integer-Parameter:

```
/* Convert from Python --> C */
%typemap(in) int {
    $1 = PyInt_AsLong($input);
}

/* Convert from C --> Python */
%typemap(out) int {
    $result = PyInt_FromLong($1);
}
```

Wie man hier sehen kann werden in den Typemaps Statthalter verwendet, die bei der Codeerzeugung durch die konkreten Parameter ersetzt werden.

Für weitere Details sei hier auf auf das Kapitel Typemaps der SWIG Dokumentation [7] verwiesen.

8.6.1 Typemaps im RFG-Binding

Typemaps werden in dieser Arbeit zu verschiedenen Zwecken verwendet. Einige dienen ganz klassisch der Umwandlung von Datentypen. Hier ist an erster Stelle `View` und `View_Set` zu nennen. Da `View` adaseitig als Integer implementiert ist, wird er von SWIG nicht als eigenständiger Typ mit eigener Shadow Class behandelt. Da diese jedoch erwünscht ist, wurde eine Pythonklasse `View` implementiert. Diese wird von der zugehörigen Typemap (`in`) `view` in einen Integer übersetzt, das heißt das `value` Attribut des Objekts wird ausgelesen und umgewandelt. (`in`) `view_Set` führt zusätzlich noch die Konvertierung nach `View_Set` durch.

Weitere Typkonvertierungen sind (`out`) `boolean`, bei dem die Ada Boolean Rückgabewerte der Predikate, die C-Chars entsprechen, in Python-Integers umgewandelt werden. SWIG würde Chars sonst in Strings der Länge eins umwandeln. Diese haben aber immer den Wahrheitswert 1, was hier natürlich nicht beabsichtigt ist.

Eigentlich müssten Strings nicht gesondert behandelt werden. Da aber der Ada-Wrappercode, der Ada Strings in `Char_Ptrs` umwandelt, neuen Speicher

allokiert, muss dieser vom Python-Binding wieder frei gegeben werden. Dazu gibt es im Prinzip zwei Möglichkeiten. Das kanonische Vorgehen wäre es eine `newfree` Typemap zu implementieren und diese mit der `%newobject` Direktive auf alle betroffenen Funktionen anzuwenden. Da in unserem Fall aber alle Funktionen, die einen String zurückliefern, Ada-Funktionen mit genau dem selben Problem sind, ist es einfacher, eine (out) Typemap zu schreiben, die neben der einzeiligen Typkonvertierung den Speicherplatz des Strings frei gibt.

Eine weitere Kategorie von Typemaps beschäftigt sich mit der Verkürzung der Signaturen. Dies sind alles mehrstellige Typemaps wie z.B. `(in)(RFG* The_RFG, View A_View)`, `(in)(RFG* The_RFG, Node* The_Node)` oder `(in)(RFG* The_RFG, Node* The_Node)`. In diesen Typemaps werden die Python-Attribute `.RFG` und `.View` ausgewertet und so die vorangestellten Parameter berechnet. Um duplizierten Code zu sparen, wurden eine Reihe von C-Makros entwickelt, die in den Typemaps verwendet werden. Diejenigen, die einen Parameter direkt umwandeln, heißen `CONVERT_TYPNAME` und diejenigen, die einen Wert erst noch aus einem anderen Parameter extrahieren, heißen `GET_TYPNAME`. Beide erhalten eine Quelle und ein Ziel als Parameter.

Es gibt dann noch einige andere Typemaps die sich z.B. mit der Umwandlung von Doppel- in Einzel-Pointer und um das Umwandeln von Out-Parametern in Rückgabewerte beschäftigen.

8.7 .RFG und .View

Die Attribute `.RFG` und `.View` werden wie oben beschrieben von mehrstelligen Typemaps ausgewertet, um die Parameterreduktion durchzuführen. Damit das funktionieren kann, müssen diese Attribute natürlich auch irgendwo richtig gesetzt werden. Eine Möglichkeit wäre, auch hierfür Typemaps zu verwenden. Out-Typemaps könnten für die entsprechenden Typen diese Attribute setzen. Allerdings stellt sich die Frage wo diese Typemaps die richtigen Werte dafür herbekommen sollen. Typemaps können Variablen anlegen, die für den gesamten Wrapper gelten — im Gegensatz zu den gewöhnlichen Variablen, die in einen eigenen Block eingeschlossen werden, damit sich die Typemaps der einzelnen Parameter nicht in die Quere kommen. In der SWIG-Dokumentation wird dem Benutzer aber deutlich nahe gelegt, solche wrapperglobalen Variablen nur zwischen Typemaps gleicher Signatur zu verwenden — also z.B. zwischen In- und Argout-Typemaps eines bestimmten Typs. Damit solche Konstruktionen auch mehrfach funktionieren, werden den wrapperglobalen Variablen noch die Nummer des zugehörigen Parameters angehängt. Gehören nun die Typemaps, die sich über eine wrapperglobale Variable austauschen wollen, nicht zum selben Parameter, kann auf diese Variable nur von Hand mit fest programmierter Parameternummer zugegriffen werden. Da es nur eine Out-Typemap pro Funktion gibt und die mehrstelligen Typemaps pro Funktion eigentlich auch nur einmal angewandt werden, ließe sich hier wahrscheinlich doch so tricksen, dass der Übergabemechanismus für die Funktionen, die hier exportiert werden sollen, funktioniert. Dieser Mechanismus ist dann aber sehr fragil und zerbricht bei ungewöhnlichen Signaturen. Zudem ist die Fehlersuche durch das hohe Maß an Indirektion, die der Erzeugung des Bindings auch so schon inne wohnt, schon sehr schwierig. Dieser Übergabemechanismus würde die Fehlersuche noch einmal deutlich erschweren, so dass aus Gründen der Wartbarkeit von dieser Konstruktion eher abzusehen ist.

Da viele Methoden durch die Probleme mit der `%extend` Direktive sowieso von Hand an die jeweiligen Typen angehängt werden müssen, ist es deutlich einfacher, die beiden Attribute `.RFG` und `.View` einfach von der umschließenden Python-Methode aus zu setzen. Dies ist zwar ein gewisser Aufwand pro Methode, aber angesichts der überschaubaren Zahl der Methoden und den relativ starken Schwierigkeiten bei einer Implementierung über Typemaps, kann das sicher kein Hindernis sein.

8.8 Callbacks

Ein wichtiger Teil der RFG-Bibliothek sind die Prädikate (`Node_Predicate` und `Edge_Predicate`). Sie enthalten eine Bedingung, die eine Node oder Edge erfüllen kann oder auch nicht. Prädikaten werden an verschiedenen Stellen zur Auswahl von Node und Edges verwendet und sind deshalb unverzichtbar. Verwendet werden sie als Funktionen (Callbacks), die in der Regel als Funktionsparameter übergeben werden. Aufgrund der Bedeutung der Prädikate muss es dem Benutzer der Pythonschnittstelle möglich sein, Prädikate selbst zu erstellen.

Callbacks stellen an der Schnittstelle zwischen C und Python jedoch ein gewisses Problem dar: Die von Python sichtbare Funktion ist nicht die C-Funktion sondern der sie umschließende Wrapper. Dieser erwartet natürlich Python-Typen und kann deshalb nicht als Ersatz für die C-Funktion herhalten. SWIG kennt zwei Direktiven, mit denen die C-Funktion so zugänglich gemacht werden kann, dass sie als Callback an eine (C-) Funktion übergeben werden kann: Mit `%constant` Signatur kann eine Funktion als Funktionspointer in die Zielsprache exportiert werden. Dieser kann dann aber nicht mehr direkt aufgerufen werden. Sollen Funktionen sowohl als Callback als auch als Funktion verfügbar sein, können sie mit den Direktiven `%callback` und `%nocallback` eingeschlossen werden. Der `%callback` Direktive muss dabei durch einen Parameter mitgeteilt werden, nach welcher Konvention die Funktionspointer benannt werden sollen. Dies alles erlaubt aber nicht, Pythoncode als Callbackfunktion zu nutzen; und SWIG unterstützt Pythoncallbacks auch nicht. Das dahinterstehende Problem ist schnell einsichtig: Callbacks werden als Funktionspointer übergeben; dieser wird dann direkt über einen Funktionsaufruf angesprungen. Da Python aber — wie alle interpretierten bzw. bytencodierten Sprachen — gar keinen direkt ausführbaren Code erzeugt, gibt es auch nichts, was man dem C-Code übergeben könnte. Die einzige Möglichkeit, diese Dilemma zu lösen, ist es eine C-Funktion zu schreiben, deren Adresse dann als Callback an den C-Code übergeben werden kann. Diese muss natürlich die passende Signatur haben. Sie muss dann die Parameterumwandlung vornehmen und die zugehörige Pythonfunktion aufrufen. Dieses Vorgehen hat aber das Problem, dass der Benutzer beliebig viele Funktionen anlegen kann, die Zahl dieser Callbackwrapper aber zur Compilezeit des Bindings festliegt. Theoretisch wäre es natürlich möglich, solche Wrapperfunktionen auch dynamisch zu erzeugen, aber das erfordert entweder das Nachcompilieren oder das Kopieren und Bearbeiten von Maschinencode zur Laufzeit. Beides ist sehr unschön; ersteres wegen des Laufzeitverhaltens und letzteres wegen der fehlenden Portabilität. Mit einer beschränkten Zahl an Wrapperfunktionen entstehen Einschränkungen für die Benutzung von in Python geschriebenen Callback-Funktionen, die von der verwendeten Technik abhängen:

- Einfachster Fall ist die Verwendung von nur einer Wrapperfunktion pro

Funktionstyp. Dies greift auf eine Variable zu der, dann immer die aktuelle Funktion zugewiesen wird. Dies hat zwar den Vorteil, dass beliebig viele Callbackfunktionen verwendet werden können, aber immer nur eine gleichzeitig. Da die Prädikate aber in erster Linie für die Iteratoren und damit für Schleifen benutzt werden, schränkt das die Verwendung von selbstdefinierten Prädikaten stark ein. Ausserdem müsste eigentlich noch ein Mechanismus entwickelt werden, der verhindert, dass ein zweites Prädikat gleichzeitig verwendet wird.

- Eine etwas bessere Variante ist es, eine feste aber für viele Fälle große Zahl von Funktionen bereitzustellen und die Prädikate diesen fest zuzuordnen. Dies schränkt die Zahl der insgesamt in einem Programm verwendeten selbstdefinierten Prädikate zwar ein (man könnte hier eine Zahl von vielleicht 20 vorstellen), ist aber einfach zu implementieren und sollte eigentlich ausreichen.
- Wenn man die Wrapperfunktionen nach Gebrauch wieder frei gibt, lässt sich das auf eine feste Zahl gleichzeitig verwendeter Prädikate verbessern. Dazu muss eine Shadow-Class nach dem Vorbild der SWIG Shadow-Classes selbst implementiert werden. Diese kann dann z.B. den Iteratoren als Attribut zugewiesen werden. Wird der Iterator dann von der Garbage-Collection eingesammelt, wird auch das Objekt, die das Prädikat repräsentiert, freigegeben. Dessen Destruktor gibt dann die Wrapperfunktion frei.

Es gibt also drei verschiedene Typen von Callback-Funktionen: Reine Python-Funktionen, die nach Python importierten C- bzw. Ada-Funktionen und zuletzt die nach Python importierten Funktionspointer. Die nach Python exportierten Prädikate sollen überhaupt nicht wieder als Callback verwendet werden, da das eine zweifache Umwandlung der Parameter nach sich ziehen würde. Um dem Benutzer die Unterscheidung zwischen Funktion und Funktionspointer abzunehmen, wird ein einfacher Mechanismus verwendet, der diese Funktionen ihrem Funktionspointer zuordnet. Dazu werden die Funktion-Funktionspointer-Paare in ein Python-Dictionary eingetragen, das aber unter Python nicht sichtbar ist. Dies geschieht durch eine Funktion, die im Rumpf der Prädikatmoduls aufgerufen wird. Dabei wird die Metaprogramming-Fähigkeiten von Python, alle Namen im Modul verarbeiten zu können, und die Regelmäßigkeit in der Benennung der Funktionspointer ausgenutzt. Die Umwandlung selbst erfolgt in der zugehörigen Typemap, die auch auf den Mechanismus für echte Pythonfunktionen verzweigt.

Da es sich bei den Callbacks um einen größeren eigenständigen Problemkreis handelt, ist die Implementierung in eine eigene Datei (`predicates.ii` ausgelagert, die in `rfigs.ii` eingebunden wird).

9 Die Implementierung

Die einzelnen Teile der Implementierung sind in den verschiedenen Kapiteln dieser Arbeit bereits beschrieben. Da das Zusammenspiel dieser Teile recht kompliziert ist, soll hier noch ein Überblick über das Ganze gegeben werden. Die Implementierung besteht aus drei Teilen:

9.1 Ada Wrapper

Ada-Wrapper umschließen, die Teile von Ada, die nicht direkt nach C exportiert werden können. Dabei handelt es sich um mehrere Ada-Pakete, die der RFG-Bibliothek hinzugefügt wurden:

RFGs.Node_Set und **RFGs.Edge_Set** kapseln die generisch instanziierten Pakete `Node_Ptrs.Sets` und `Edge_Ptrs.Set`

RFGs.Binding enthält Wrapper für einzelne Funktionen. Dabei handelt es sich um

- Funktionen mit String-Parametern oder String-Rückgabewerten.
- Primitive, für die Wrapper mit Dispatching-Code benötigt werden.
- Standard Ada Funktionen, die exportiert werden müssen.
- sonstige Hilfsfunktionen.

9.2 Ada-C Binding

Der C-Binding-Generator und seine Bestandteile sind in Kapitel 7.7 beschrieben. Für das Verständnis, der Arbeit ist aber wichtig wie und wo er eingesetzt wird. Das eigentliche C-Binding besteht aus einigen Header-Dateien, die mit Hilfe, der Binding-Generators erzeugt werden. Namentlich sind das `rfgs.h`, `predicates.h` und `gravis.h`. Sie werden aus Dateien mit der Endung `.hi` erzeugt, die nur wenige Zeilen lang sind. Die eigentliche Information wird aus den Ada-Spezifikationspaketen und den darin enthaltenen Export Pragmas extrahiert. Diese Header-Dateien werden eigentlich nicht benötigt, sie werden aber zu Kontrolle, bei der Erzeugung des Python-Bindings mit eingebunden.

Zu den erzeugten Header-Dateien kommt noch die Datei `rfgs-types.h`, in der die Umsetzung der RFG-Typen nach C definiert sind. Diese Datei ist von Hand erstellt.

Die eigentlich interessante Anwendung des C-Binding-Generators ist die Vorverarbeitung der SWIG-Dateien. Diese haben die Endung `.ii` und werden vom Generator in Dateien mit der Endung `.i` umgewandelt, die dann von SWIG weiterverarbeitet werden.

9.3 SWIG Python-Bindung

Das Python-Binding kann eigentlich nicht unabhängig von dem C-Binding betrachtet werden, da die Direktiven des C-Binding-Generators wesentlicher und fester Bestandteil der Dateien des Python-Bindings sind.

Die Datentypen des RFG werden in der Datei `rfgs.ii` etabliert, die dafür die Header-Datei `rfgs-types.h` einbindet. Diese Datentypen werden dann durch Funktionen erweitert, die über den C-Binding Generator eingefügt werden. Um `rfgs.ii` auf eine überschaubare Länge zu begrenzen, sind verschiedene Bereiche in eigene Dateien ausgelagert. Dies hat keine Sematische Bedeutung sondern dient rein der Übersichtlichkeit. Die Typemaps sind in `rfgs-typemap.i` untergebracht und die Behandlung der Prädikate und die damit verbundene Callback-Problematik wurde nach `predicates.i` ausgelagert.

Die Prädikat-Funktionen aus der RFG-Bibliothek befinden sich in einem eigenen Modul mit dem Namen `predicates`. Dieses wird von der Datei `predicates.ii` erzeugt, die auch nur wenige Zeilen lang ist. Die beiden weiteren Module `analysis` und `gravis` sind ebenfalls in einer jeweils eigenen Datei untergebracht.

10 Verlauf der Arbeit

Da am Anfang der Arbeit nicht sicher war, dass die Funktionalität der RFG-Bibliothek überhaupt exportiert werden kann, lag es nahe diesen kritischen Punkt zuerst zu klären. Bei ersten Untersuchungen stellte sich heraus, dass der Export von C in die meisten in Frage kommenden Skriptsprachen kein allzu großes Problem darstellen würde. In Gegensatz dazu schien es für einige Probleme bei dem Binding von Ada nach C keine absehbare Lösung zu geben. Dies traf vor allem auf polymorphe und generische Objekte zu, aber auch das Typ-System von Ada schien nicht mit vernünftigem Aufwand beherrschbar. Im Nachhinein betrachtet wurde hierbei aber versucht, Probleme zu lösen, die in dieser Allgemeinheit auf diese Arbeit garnicht zutreffen. Deshalb wurde leider erst sehr spät mit der Analyse der RFG-Bibliothek begonnen. Dazu beigetragen hat auch die Größe der Bibliothek mit fast einem halben Megabyte Spezifikations-Dateien. Dieser Respekt vor dem Umfang der Bibliothek hat sich aber als unbegründet herausgestellt. Bei der Analyse der RFG-Bibliothek konnte dann festgestellt werden, dass es möglich ist, die schwierigen Problem bei dem Binding nach C zu umgehen. Zudem stellte sich heraus, dass ein grosser Teil der Bibliothek redundant ist und für eine Benutzerschnittstelle besser weggelassen werden sollte.

Als zweites Ergebnis der Untersuchung der RFG-Bibliothek stellte sich heraus, dass diese nicht unverändert einem Benutzer angeboten werden kann. Deshalb musste eine Schnittstelle entworfen werden, die sowohl benutzerfreundlich als auch mit der RFG-Bibliothek leicht realisierbar ist. Der Entwurf war zwar relativ leicht zu erstellen, zog aber natürlich zusätzlichen Implementierungsaufwand nach sich.

Sobald klar war, wie das Binding nach C erstellt werden könnte, wurde mit der Implementation des Ada-C-Binding Generators begonnen, da sich die bisherigen Experimente, die von Hand ausgeführt worden waren, als sehr mühselig und fehleranfällig erwiesen hatten. Da die Grundfunktionalität des Generator relativ schnell implementiert war, konnte mit nur leichter Zeitverzögerung das Python-Binding begonnen werden, nachdem die Analyse der verschiedenen Skriptsprachen, die immer wieder in den bisherigen Verlauf der Arbeit eingeschoben worden war, abgeschlossen wurde. Bei dem Export der einzelnen Typen wurden beide Bindings gemeinsam erstellt, so dass auf die Implementierung von C-Test-Code verzichtet werden konnte.

Als letzter Teil wurde die Schnittstelle zu Gravis entworfen.

11 Fazit und Ausblick

11.1 Implementierung

Im Nachhinein betrachtet ist die Implementierung des Bindings erstaunlich kompakt. Dies liegt zum einen an dem hohen Maß an Indirektheit. Einige Zeilen Code durchlaufen von dem geschriebenen Buchstaben bis hin zur tatsächlichen

Ausführung bis zu einem halben Dutzend verschiedener Verarbeitungsschritte. Dies spiegelt sich schließlich auch in dem Verhältnis zwischen geschriebenem und erzeugtem Code wieder, das fast eins zu zehn beträgt.

11.2 Programmiersprache

Abschließend stellt sich die Frage, ob die Entscheidung keine eigene Programmiersprache zu entwerfen und zu implementieren richtig war. Wenn es darum geht, ob durch die Sprache Abstriche an der Intergration der RFG-Schnittstelle gemacht werden mussten, kann dies klar mit "Nein" beantwortet werden. Die Schnittstelle fügt sich sehr gut in Python ein. Was den Aufwand angeht so ist es schwierig, da natürlich kein direkter Vergleich vorgenommen werden kann. Es scheint aber so, als ob der Aufwand, der hier für die Implementierung nötig war, höchstens für eine sehr einfach Programmiersprache hätte reichen können.

11.3 Weiterführende Arbeiten

Aufgrund der relativ vielen konzeptionelle Problemen und der relativ kleinen Anzahl schließlich davon betroffener Funktionen, ließen sich einige Problem leider nicht in der Allgemeinheit lösen, die sie vielleicht verdient hätten. Lohnende Kandidaten für weitere Arbeiten — vielleicht nicht unbedingt im Zusammenhang mit Gravis oder dem Bauhaus — wären hier unter anderem eine Call-backunterstützung für SWIG. Der hier gezeigte Ansatz ist eigentlich allgemein genug, damit er auch für andere Zielsprachen übertragbar sein sollte. Auch im Bereich des Bindings von Ada nach C gäbe es noch Handlungsbedarf; sei es eine Verbesserung der Pragmas Export und Convention von GNAT oder aber auch Untersuchung der Frage, wie Ada und C++ zur Kooperation bewegt werden können. Allerdings haben sich allgemeine Lösungen an einigen Stellen wie z.B. der Umsetzung von Ada-Typen nach C als ausgesprochen schwierig erwiesen. Durch Umgehung der eigentlichen Problematik kommt man hier deutlich einfacher zu einer funktionierenden Lösung.

11.4 Die RFG-Bibliothek

Interessant für das Bauhausprojekt ist meiner Meinung nach die Frage, was sich aus der Python-Schnittstelle für die RFG-Bibliothek lernen lässt. Auch wenn ich das Anwendungsgebiet des RFG nicht wirklich überschaue, habe ich doch den Eindruck, dass es eine gewisse Unzufriedenheit mit einigen der getroffenen Designentscheidungen gibt. Da die Python-Schnittstelle leichter verändert werden kann als die Bibliothek selbst, kann sie als Prototyp für die Weiterentwicklung der Ada-Programmierschnittstelle dienen. Da einige Änderungen an der RFG-Bibliothek anstehen (Ersetzen der Typhierarchien durch dynamische Attribute), ist vielleicht jetzt ein guter Zeitpunkt über die Architektur der Bibliothek nachzudenken.

Literatur

- [Eisenbarth98] Eisenbarth, Thomas: GroupiusSE Eine Resource Flow Graph Bibliothek in Ada95 für das Speichern und Aufbereiten von Reengenerierungsinformationen
- [1] Ada 95 Reference Manual
 - [2] Ada95 Rationale
 - [3] GNAT Reference Manual, Version 3.14
 - [4] GNAT User's Guide, Version 3.14
 - [5] Knauss, Markus: Interfacing Ada95 and C++, 2002
 - [6] Comar, Cyrille; Dewar, Robert B.K.: Interfacing to other Languages using GNAT and Ada 95
 - [7] SWIG 1.3 Development Documentation, Version 1.3.18
 - [8] van Rossum, Guido; Drake, Fred L.: Python Tutorial, Release 2.2.3
 - [9] van Rossum, Guido; Drake, Fred L.: Python Reference Manual, Release 2.2.3
 - [10] van Rossum, Guido; Drake, Fred L.: Python Library Reference, Release 2.2.3
 - [11] van Rossum, Guido; Drake, Fred L.: Extending and Embedding the Python Interpreter, Release 2.2.3
 - [12] van Rossum, Guido; Drake, Fred L.: Python/C API Reference Manual, Release 2.2.3

Erklärung

Hiermit versichere ich, diese Arbeit
selbständig verfaßt und nur die
angegebenen Quellen benutzt zu haben.

(Florian Festi)